# Contents

Preface xv

Chapter 1 Using CLIM ........................................................1

1.1 Conceptual Overview ........................................................2
1.2 Highlights of Tools and Techniques .................................3
1.3 How CLIM Helps You Achieve a Portable User Interface ....4
1.4 What Is CLIM? ..................................................................6
  1.4.1 The Core of CLIM .......................................................6
     1.4.1.1 Application Frames ..............................................6
     1.4.1.2 Panes ....................................................................6
     1.4.1.3 Sheets ...................................................................8
     1.4.1.4 Enabling Input and Output ......................................8
     1.4.1.5 Graphics ...............................................................9
     1.4.1.6 Text .....................................................................9
     1.4.1.7 Events .................................................................9
     1.4.1.8 Mediums .............................................................9
  1.4.2 CLIM Facilities ............................................................10
     1.4.2.1 Look and Feel .......................................................10
     1.4.2.2 Controlling Look and Feel ......................................10
     1.4.2.3 Changing the appearance on Windows ..................11
     1.4.2.4 Changing the appearance on X11/Motif .................11
     1.4.2.5 Streams ...............................................................11
     1.4.2.6 Extended Input and Output ....................................12
     1.4.2.7 Presentations .......................................................12
     1.4.2.8 Command Loop ..................................................13
  1.4.3 Summary ....................................................................14
1.5 Loading CLIM .................................................................15
1.6 Testing Code Examples ....................................................15
1.7 The CLIM demos .............................................................16

Chapter 2 Drawing Graphics .............................................19

2.1 Conceptual Overview of Drawing Graphics .......................20
2.1.1 Drawing Functions and Options ..........................................................20
2.1.2 The Drawing Plane .............................................................................20
2.1.3 Coordinates ........................................................................................21
2.1.4 Mediums, Sheets, and Streams ............................................................22
2.2 Examples of Using CLIM Drawing Functions ..........................................23
2.3 CLIM Drawing Functions ........................................................................24
  2.3.1 Arguments ........................................................................................24
  2.3.2 Compound Drawing Functions ............................................................30
  2.3.3 Patterns and Stencils ..........................................................................31
  2.3.4 Pixmaps ............................................................................................33
2.4 Graphics Protocols ..................................................................................36
  2.4.1 Arguments ........................................................................................36
  2.4.2 General Behavior of Drawing Functions ............................................36
  2.4.3 Medium-Specific Drawing Functions ................................................37
2.5 General Geometric Objects in CLIM ....................................................38
  2.5.1 Regions in CLIM ................................................................................38
    2.5.1.1 Region Predicates in CLIM ..........................................................41
    2.5.1.2 Composition of CLIM Regions .....................................................42
  2.5.2 CLIM Point Objects ..........................................................................46
  2.5.3 Polygons and Polylines in CLIM .........................................................47
    2.5.3.1 Constructors for CLIM Polygons and Polylines .........................49
    2.5.3.2 Accessors for CLIM Polygons and Polylines ...............................50
  2.5.4 Lines in CLIM ...................................................................................50
  2.5.5 Rectangles in CLIM ..........................................................................52
  2.5.6 Ellipses and Elliptical Arcs in CLIM ..................................................54
    2.5.6.1 Constructor Functions for Ellipses and Elliptical Arcs in CLIM ...57
    2.5.6.2 Accessors for CLIM Elliptical Objects .......................................58
  2.5.7 Bounding Rectangles .......................................................................59
    2.5.7.1 The Bounding Rectangle Protocol ..............................................62
    2.5.7.2 Bounding Rectangle Convenience Functions ...............................63

Chapter 3  The CLIM Drawing Environment ........................................65
  3.1 CLIM Mediums .....................................................................................66
  3.2 Using CLIM Drawing Options ..............................................................70
9.7.2 Constructing a Function as Part of Running an Application .....203
9.8 Application Frame Operators and Accessors .................................203
  9.8.1 CLIM Application Frame Accessors ...........................................204
  9.8.2 Operators for Running CLIM Applications ..................................207
9.9 Frame Managers ...............................................................................212
  9.9.1 Finding Frame Managers .............................................................213
  9.9.2 Frame Manager Operators ..........................................................214
9.10 Advanced Topics .............................................................................216

Chapter 10  Panes and Gadgets .........................................................219

10.1 Panes ..............................................................................................220
  10.1.1 Basic Pane Construction ............................................................221
  10.1.2 Pane Initialization Options ...........................................................222
  10.1.3 Pane Properties ........................................................................223
10.2 Layout Panes ....................................................................................224
  10.2.1 Layout Pane Options .................................................................224
  10.2.2 Layout Pane Classes .................................................................226
  10.2.3 Composite Pane Generic Functions ............................................229
  10.2.4 The Layout Protocol .................................................................230
10.3 Extended Stream Panes ...................................................................234
  10.3.1 Extended Stream Pane Options ....................................................234
  10.3.2 Extended Stream Pane Classes ....................................................237
  10.3.3 Making CLIM Extended Stream Panes .......................................238
10.4 Defining A New Pane Type: Leaf Panes .........................................239
10.5 Gadgets ............................................................................................240
  10.5.1 Abstract Gadgets ......................................................................241
    10.5.1.1 Using Gadgets .................................................................241
    10.5.1.2 Implementing Gadgets ........................................................243
  10.5.2 Basic Gadget Classes .................................................................245
  10.5.3 Abstract Gadget Classes ..............................................................252
    10.5.3.1 The Label Gadget .............................................................252
    10.5.3.2 The List-Pane and Option-Pane Gadgets ............................253
    10.5.3.3 The Menu-Button Gadget ...................................................255
    10.5.3.4 The Push-Button Gadget ....................................................255
Chapter 11 Commands .................................................................267
11.1 Introduction to CLIM Commands .............................................268
11.2 Defining Commands the Easy Way .............................................269
   11.2.1 Command Names and Command Line Names .......................270
   11.2.2 The Command-Defining Macro ...........................................270
11.3 Command Objects .....................................................................272
11.4 CLIM Command Tables ..........................................................275
11.5 CLIM Predefined Command Tables ..........................................278
11.6 Conditions Relating to CLIM Command Tables .........................279
11.7 Styles of Interaction Supported by CLIM .................................280
11.8 Command-Related Presentation Types .......................................280
11.9 The CLIM Command Processor ...............................................282
11.10 Advanced Topics ...................................................................285
   11.10.1 CLIM Command Tables ......................................................285
   11.10.2 CLIM Command Menu Interaction Style ...............................287
   11.10.3 Mouse Interaction Via Presentation Translators .......................291
   11.10.4 CLIM Command Line Interaction Style ..................................292
   11.10.5 CLIM Keystroke Interaction Style .........................................293
   11.10.6 The CLIM Command Processor ..........................................296

Chapter 12 Menus and Dialogs ....................................................299
12.1 Conceptual Overview of Menus and Dialogs ................................300
12.2 CLIM Menu Operators ............................................................300
12.3 CLIM Dialog Operators ............................................................306
12.4  Examples of Menus and Dialogs in CLIM ........................................310
12.4.1 Using accepting-values ..............................................................310
12.4.2 Using accept-values-command-button .....................................311
12.4.3 Using :resynchronize-every-pass in accepting-values ...............312
12.4.4 Using the third value from accept in accepting-values .............313
12.4.5 Using menu-choose .................................................................315
12.4.6 Using menu-choose-from-drawer .............................................317

Chapter 13  Extended Stream Output Facilities ..............319

13.1  Basic Output Streams .................................................................320
13.2  Extended Output Streams ............................................................321
13.3  The Text Cursor ........................................................................323
  13.3.1  The Text Cursor Protocol .........................................................325
  13.3.2  The Stream Text Cursor Protocol ............................................326
13.4  Text ............................................................................................327
  13.4.1  The Text Protocol .................................................................327
  13.4.2  Mixing Text and Graphics .......................................................329
  13.4.3  Wrapping Text Lines ..............................................................329
13.5  Attracting the User’s Attention ...................................................331
13.6  Buffering Output ........................................................................331
13.7  CLIM Window Stream Pane Functions .......................................332

Chapter 14  Output Recording and Redisplay ...............335

14.1  Conceptual Overview of Output Recording .................................336
14.2  CLIM Operators for Output Recording ......................................337
  14.2.1  The Basic Output Record Protocol .......................................339
  14.2.2  The Output Record “Database” Protocol ................................342
14.2.3  Types of Output Records ........................................................343
  14.2.3.1  Standard Output Record Classes ......................................343
  14.2.3.2  Graphics Displayed Output Records ..................................344
  14.2.3.3  Text Displayed Output Records .......................................345
  14.2.3.4  Top-Level Output Records ...............................................346
14.2.4  Output Recording Streams .........................................................347
  14.2.4.1  The Output Recording Stream Protocol .............................347

<Bold>xCLIM User Guide
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.2.4.2 Graphics Output Recording</td>
<td>349</td>
</tr>
<tr>
<td>14.2.4.3 Text Output Recording</td>
<td>349</td>
</tr>
<tr>
<td>14.2.4.4 Output Recording Utilities</td>
<td>350</td>
</tr>
<tr>
<td>14.3 Conceptual Overview of Incremental Redisplay</td>
<td>354</td>
</tr>
<tr>
<td>14.4 CLIM Operators for Incremental Redisplay</td>
<td>355</td>
</tr>
<tr>
<td>14.5 Using updating-output</td>
<td>357</td>
</tr>
<tr>
<td>14.6 Example of Incremental Redisplay in CLIM</td>
<td>359</td>
</tr>
<tr>
<td>Chapter 15 Extended Stream Input Facilities</td>
<td>361</td>
</tr>
<tr>
<td>15.1 Basic Input Streams</td>
<td>362</td>
</tr>
<tr>
<td>15.2 Extended Input Streams</td>
<td>363</td>
</tr>
<tr>
<td>15.2.1 The Extended Input Stream Protocol</td>
<td>364</td>
</tr>
<tr>
<td>15.2.2 Extended Input Stream Conditions</td>
<td>368</td>
</tr>
<tr>
<td>15.3 Gestures and Gesture Names</td>
<td>369</td>
</tr>
<tr>
<td>15.4 The Pointer Protocol</td>
<td>372</td>
</tr>
<tr>
<td>15.5 Pointer Tracking</td>
<td>373</td>
</tr>
<tr>
<td>Chapter 16 Input Editing and Completion Facilities</td>
<td>381</td>
</tr>
<tr>
<td>16.1 Input Editing</td>
<td>382</td>
</tr>
<tr>
<td>16.1.1 Operators for Input Editing</td>
<td>384</td>
</tr>
<tr>
<td>16.1.2 Input Editor Commands</td>
<td>385</td>
</tr>
<tr>
<td>16.2 Activation and Delimiter Gestures</td>
<td>388</td>
</tr>
<tr>
<td>16.3 Signalling Errors Inside accept Methods</td>
<td>390</td>
</tr>
<tr>
<td>16.4 Reading and Writing Tokens</td>
<td>391</td>
</tr>
<tr>
<td>16.5 Completion</td>
<td>393</td>
</tr>
<tr>
<td>16.6 Using with-accept-help: some examples</td>
<td>399</td>
</tr>
<tr>
<td>16.7 Advanced Topics</td>
<td>400</td>
</tr>
<tr>
<td>Chapter 17 Formatted Output</td>
<td>403</td>
</tr>
<tr>
<td>17.1 Formatting Tables in CLIM</td>
<td>404</td>
</tr>
</tbody>
</table>
17.1.1 Conceptual Overview of Formatting Tables .........................404
17.1.2 CLIM Operators for Formatting Tables ............................405
17.1.3 Examples of Formatting Tables ........................................411
  17.1.3.1 Formatting a Table From a List .................................411
  17.1.3.2 Formatting a Table Representing a Calendar Month ....412
  17.1.3.3 Formatting a Table With Regular Graphic Elements ......414
  17.1.3.4 Formatting a Table With Irregular Graphics in the Cells ..415
  17.1.3.5 Formatting a Table of a Sequence of Items ...............416

17.2 Formatting Graphs in CLIM .................................................418
  17.2.1 Conceptual Overview of Formatting Graphs .....................418
  17.2.2 CLIM Operators for Graph Formatting ...........................419
  17.2.3 Examples of CLIM Graph Formatting .............................422

17.3 Formatting Text in CLIM .....................................................424

17.4 Bordered Output in CLIM ..................................................426

17.5 Advanced Topics ................................................................428
  17.5.1 The Table Formatting Protocol ....................................429
    17.5.1.1 The Row and Column Formatting Protocol ................431
    17.5.1.2 The Cell Formatting Protocol .................................432
  17.5.2 The Item List Formatting Protocol ...............................433
  17.5.3 The Graph Formatting Protocol ..................................434

Chapter 18  Sheets .................................................................439
  18.1 Overview of Window Facilities ........................................440
    18.1.1 Properties of Sheets .............................................441
    18.1.2 Sheet Protocols ..................................................441

  18.2 Basic Sheet Classes .......................................................443

  18.3 Relationships Between Sheets .........................................444
    18.3.1 Sheet Relationship Functions .................................444
    18.3.2 Sheet Genealogy Classes .....................................447

  18.4 Sheet Geometry ............................................................448
    18.4.1 Sheet Geometry Functions ....................................448
    18.4.2 Sheet Geometry Classes ......................................450

  18.5 Sheet Protocols: Input ..................................................452
<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.5.1 Input Protocol Functions</td>
<td>452</td>
</tr>
<tr>
<td>18.5.2 Input Protocol Classes</td>
<td>454</td>
</tr>
<tr>
<td>18.6 Standard Device Events</td>
<td>456</td>
</tr>
<tr>
<td>18.7 Sheet Protocols: Output</td>
<td>464</td>
</tr>
<tr>
<td>18.7.1 Mediums and Output Properties</td>
<td>464</td>
</tr>
<tr>
<td>18.7.2 Output Protocol Functions</td>
<td>467</td>
</tr>
<tr>
<td>18.7.3 Output Protocol Classes</td>
<td>467</td>
</tr>
<tr>
<td>18.7.4 Associating a Medium With a Sheet</td>
<td>468</td>
</tr>
<tr>
<td>18.8 Repaint Protocol</td>
<td>470</td>
</tr>
<tr>
<td>18.8.1 Repaint Protocol Functions</td>
<td>470</td>
</tr>
<tr>
<td>18.8.2 Repaint Protocol Classes</td>
<td>471</td>
</tr>
<tr>
<td>18.9 Sheet Notification Protocol</td>
<td>472</td>
</tr>
<tr>
<td>18.9.1 Relationship to Window System Change Notifications</td>
<td>472</td>
</tr>
<tr>
<td>18.9.2 Sheet Geometry Notifications</td>
<td>472</td>
</tr>
<tr>
<td>Chapter 19 Ports, Grafts, and Mirrored Sheets</td>
<td>475</td>
</tr>
<tr>
<td>19.1 Introduction</td>
<td>476</td>
</tr>
<tr>
<td>19.2 Ports</td>
<td>477</td>
</tr>
<tr>
<td>19.3 Grafts</td>
<td>480</td>
</tr>
<tr>
<td>19.4 Mirrors and Mirrored Sheets</td>
<td>483</td>
</tr>
<tr>
<td>19.4.1 Mirror Functions</td>
<td>483</td>
</tr>
<tr>
<td>19.4.2 Internal Interfaces for Native Coordinates</td>
<td>484</td>
</tr>
<tr>
<td>Appendix AGlossary</td>
<td>489</td>
</tr>
<tr>
<td>Appendix BImplementation Specifics</td>
<td>507</td>
</tr>
<tr>
<td>B.1 Setting Up Your Packages to Use CLIM</td>
<td>507</td>
</tr>
<tr>
<td>B.2 CLIM Packages</td>
<td>507</td>
</tr>
<tr>
<td>Appendix CTThe CLIM-SYS Package</td>
<td>509</td>
</tr>
<tr>
<td>C.1 Resources</td>
<td>509</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>C.2 Multi-Processing</td>
<td>511</td>
</tr>
<tr>
<td>C.3 Locks</td>
<td>513</td>
</tr>
<tr>
<td>C.4 Multiple-Value Setf</td>
<td>514</td>
</tr>
<tr>
<td>Appendix D Common Lisp Streams</td>
<td>516</td>
</tr>
<tr>
<td>D.1 Stream Classes</td>
<td>516</td>
</tr>
<tr>
<td>D.2 Basic Stream Functions</td>
<td>518</td>
</tr>
<tr>
<td>D.3 Character Input</td>
<td>519</td>
</tr>
<tr>
<td>D.4 Character Output</td>
<td>521</td>
</tr>
<tr>
<td>D.5 Binary Streams</td>
<td>523</td>
</tr>
<tr>
<td>D.6 Hardcopy Streams in CLIM</td>
<td>523</td>
</tr>
<tr>
<td>Appendix E Windows</td>
<td>525</td>
</tr>
<tr>
<td>E.1 Window Stream Operations in CLIM</td>
<td>525</td>
</tr>
<tr>
<td>E.2 Functions for Operating on Windows Directly</td>
<td>527</td>
</tr>
<tr>
<td>Index</td>
<td>531</td>
</tr>
</tbody>
</table>
Preface

About the User Guide

The Common Lisp Interface Manager (CLIM) is a powerful Lisp-based toolkit that provides a layered set of portable facilities for constructing user interfaces. The Common Lisp Interface Manager User Guide is intended for CLIM programmers who are looking for material arranged by concept. The Guide, based on the CLIM II Specification, is a complete reference for the LispWorks version of CLIM. Each chapter of the User Guide explains a key aspect of CLIM and includes summaries of conditions, constants, functions, macros, and presentation types that pertain to the particular topic, as well as many code examples. For a detailed syntactic description of a particular CLIM construct, refer to the on-line CLIM manual pages.

Notational Conventions

The User Guide employs the following conventions to distinguish different types of text.

**construct**

Lisp and CLIM constructs, such as functions or classes.

**significant term**

Significant terms introduced for the first time. These terms appear in the glossary.

**code examples**

Computer-generated text, prompts, and messages, as well as code examples and user entries.

**KEYSTROKES**

References to keystrokes, as in META or SHIFT. Logical keystrokes are enclosed in angle brackets. Thus for `<ABORT>`, you might type `CONTROL-z`; for `<END>`, `CONTROL-]`; and for `<HELP>`, `META-?`.

**function arguments**

Arguments to functions.
specified arguments  Specific values for arguments within code examples.

unspecified arguments  Arguments within code examples for which the user must supply a value.

Menu Item  Menu items, as in Exit or File>Save or Up.

filename  Pathnames, filenames, and parts of filenames.

Please note that <release-directory> refers to the location of CLIM in the LispWorks library. <release-directory>/demo/puzzle.lisp should be interpreted as <lispworks-directory>/lib/<version-number>/clim2/demo/puzzle.lisp.

Mouse pointer gestures are capitalized, as in Left or SHIFT-Middle.
Chapter 1  Using CLIM
1.1 Conceptual Overview

The Common Lisp Interface Manager (CLIM) is a powerful Lisp-based toolkit that provides a layered set of portable facilities for constructing user interfaces. These include application building facilities; basic windowing, input, output, and graphics services; stream-oriented input and output augmented by facilities such as output recording, presentations, and context sensitive input; high-level “formatted output” facilities; command processing; and a compositional toolkit similar to those found in the X world that supports look-and-feel independence.

CLIM does not compete with the window system or toolkits of the host machine (such as Motif or OpenLook), but rather uses their services, to the extent that it makes sense, to integrate Lisp applications into the host’s window environment. For example, CLIM “windows” are mapped onto one or more host windows, and input and output operations performed on the CLIM window are ultimately carried out by the host window system.

The CLIM programmer is insulated from most of the complexities of portability. Regardless of the operating platform (that is, the combination of Lisp system, host computer, and host window environment), applications only need deal with CLIM objects and functions. CLIM makes abstractions out of many of the concepts common to all window environments. The programmer is encouraged to think in terms of these abstractions, rather than in the specific capabilities of a particular host system. For example, using CLIM, the programmer can specify the appearance of output in high-level terms and those high-level descriptions are then turned into the appropriate appearance for the given host. Thus, the application has the same fundamental interface across multiple environments, although the details will differ from system to system.

CLIM provides a spectrum of user interface building options, all the way from detailed, low-level specification of “what goes where,” to high-level specifications in which the programmer leaves all of the details up to CLIM. This allows CLIM to balance ease of use on the one hand and versatility on the other. By using high-level facilities, a programmer can build portable user interfaces quickly, whereas by utilizing lower-level facilities, she can customize her programming and user interfaces according to her specific needs or requirements. For example, CLIM supports the development of applications that are independent of look and feel, as well as the portable development of toolkit libraries that define and implement a particular look and feel.
The CLIM architecture is divided into several layers, each with an explicitly-defined protocol. These protocols allow the programmer to customize or re-implement various parts of CLIM.

1.2 Highlights of Tools and Techniques

The facilities provided by CLIM include:

- **Graphics**
  CLIM offers a rich set of drawing functions, a wide variety of drawing options (such as line thickness), a sophisticated inking model, and color. CLIM provides full affine transformations, so that a drawing may be arbitrarily translated, rotated, and scaled to the extent that the underlying window system is capable of rendering such objects.

- **Windowing**
  CLIM provides a portable layer for implementing window-like objects known as sheets that are suited to support particular high-level facilities or interfaces. The windowing module of CLIM defines a uniform interface for creating and managing hierarchies of these objects. This layer also provides event management.

- **Output Recording**
  CLIM offers a facility for capturing all output done to a window. This facility provides the support for automatic window repainting and scrollable windows. In addition, this facility serves as the foundation for a variety of interesting high-level tools, including incremental redisplay.

- **Formatted Output**
  CLIM provides a set of macros and functions that enable programs to produce neatly formatted tabular and graphical displays with very little effort.

- **Context Sensitive Input**
  The presentation type facility of CLIM links textual or graphical output on a window with the underlying Lisp object that it represents, so that objects may be retrieved later by selecting their displayed representation with the pointer. This “semantic typing” of output allows the application builder to separate the semantics of the application from the appearance and interaction style.
Application Building

CLIM provides an application framework for organizing an application’s top-level user interface and command processing loops. This framework provides support for laying out application windows under arbitrary constraints, managing command menus and/or menu bars, and associating user interface gestures with application commands. Using these tools, application writers can easily and quickly construct user interfaces that can grow flexibly from prototype to delivery.

Adaptive Toolkit

CLIM provides a uniform interface to the standard compositional toolkits available in many commercial computer environments. CLIM defines abstract classes that are analogous to the gadgets or widgets of toolkits such as Motif or OpenLook. CLIM fosters look-and-feel independence by defining these gadgets in terms of their function, without respect to the details of their appearance or operation. If an application uses these gadgets, its user interface will ultimately draw upon whatever toolkit is available in the host environment. This facility lets programmers easily construct applications that automatically conform to a variety of user interface standards. In addition, a portable CLIM-based implementation of these gadgets is provided.

1.3 How CLIM Helps You Achieve a Portable User Interface

Portability is one of the features that sets CLIM apart from other interface managers.

CLIM provides a uniform interface to the standard compositional toolkits available in many environments. By defining user interfaces in terms of CLIM objects rather than by accessing windows and widgets of a given windowing system directly, you are able to achieve a highly portable interface. In addition to CLIM functionality, you may also incorporate aspects of Common Lisp and CLOS into your program. The dependencies of your application are outlined in Figure 1.
The portability of your code comes from the fact that it is written in terms of standardized packages: Common Lisp, CLOS, and CLIM. From the perspective of your application, the details of the host windowing system, host operating system, and host computer should be invisible. CLIM handles the interaction with the underlying windowing system. Figure 2 shows the elements of the host environment from which CLIM insulates your application.

CLIM shields you from the details of any one window system by making abstractions of the concepts that many window systems have in common. In using CLIM, you specify the appearance of your application’s interface in general, high-level terms. CLIM then turns your high-level description into the appearance appropriate for a given host environment.
For example, a request for a scroll bar pane would be interpreted as a request for the scroll bar widget in the current windowing system.

In some cases, you may prefer to have more explicit control over the appearance of your application. At the expense of portability, you may, at any time, bypass CLIM abstract interface objects and directly use functions provided by the underlying windowing system.

1.4 What Is CLIM?

In the first three sections you have been given a brief introduction to CLIM and some of its features. This section addresses the nature of CLIM in a more concrete and tangible fashion by defining important CLIM terms and discussing the fundamental elements of CLIM, as well as higher-level facilities that have been built from this core.

1.4.1 The Core of CLIM

1.4.1.1 Application Frames

An application frame, or simply a frame, is the locus of all application-specific knowledge. It is a physical, bordered object that is composed of smaller, individually functioning parts, called panes. The frame maintains information regarding the layout of these components, keeps track of the Lisp state variables that contain the state of the application, and optionally has an interface to the window manager.

In developing a simple application such as an on-line address book, the application frame could be divided into several units to accomplish various tasks, as you can see in Figure 1.4.1.2. One pane could be used to accept commands; another section of the screen could provide an index of names in the address book; another portion could be used to display a specific address entry. We might also choose to have a general menu and a few conveniently placed scroll bars. Each of these components of the application frame is a pane.

1.4.1.2 Panes

A pane is a window-like object that knows how to behave as a component of an application frame. That is, it supports the pane protocol operations for layout.
Panes come in many different varieties. For example, gadget panes include such things as buttons and scroll bars. Stream panes deal specifically with text. Some panes are defined only in terms of their functionality, without regard to their specific appearance or implementation. These panes are called abstract panes. The abstract definition allows various instances of the pane class to take on a platform-dependent look and feel. Panes can also be classified according to their role in pane hierarchies. Panes that can have child panes are called composite panes; those that cannot are called leaf panes. Composite panes that are in charge of spatially organizing their children are called layout panes.

Figure 3. An Example of Panes Within an Application Frame
The address book application frame shows a typical pane hierarchy. There are three instances of text panes that have associated scroll bars. For every extended stream pane, or text field, with affiliated scroll bar panes there is an “invisible” parent pane, known as a scroller pane, which does such things as control the layout of the child panes and ensure that its children are given the space they need.

The ability to address space allocation and composition concerns is the primary characteristic that sets panes apart from their superclass, sheets, to be discussed in Subsubsection 1.4.1.3, “Sheets”. Panes, therefore, understand how much screen space they want and need. For instance, the menu pane in the address book application has a static height, so that if the window is resized, the menu pane will not be scaled vertically. On the other hand, the scroller pane labeled in Figure 1.4.1.2 (the pane controlling the application pane for the address book index and the gadget panes for the two associated scroll bars) can be resized as long as the scroll bars are granted enough screen space to function, that is to say, to display the minimum graphics necessary to implement scrolling.

### 1.4.1.3 Sheets

Panes are built from more general objects called *sheets*. Sheets are the fundamental “window-like” entities that specify the areas of the screen to be used for input and output interactions. Sheets consist of, in part, a region on the screen, a coordinate system, and optionally a parent and/or child sheets. For a complete discussion on sheets, refer to Chapter 18, “Sheets”. CLIM programmers will typically not need to deal with sheets directly, but instead will use the higher-level pane objects.

### 1.4.1.4 Enabling Input and Output

A pane hierarchy must be attached to a display server so as to permit input and output. This is handled by the use of *ports* and *grafts*. A port specifies the device acting as the display server, whereas grafts are special sheets, typically representing the root window, which are directly connected to the display server. (The term *graft* is derived from the horticultural practice of grafting, in which the trunk of one tree is joined onto the rootstock of another.) Again, a CLIM application programmer will not normally deal with these objects directly. A call to *make-application-frame* automatically results in a port specification and graft instantiation. Refer to Section 9.2, “Defining CLIM Application Frames,” for details.
1.4.1.5 Graphics

Once your panes are ready to accept output, you may be interested in creating graphics. CLIM provides elementary graphic functions such as `draw-point` and `draw-circle` as well as higher-level graphic functions such as `draw-arrow` and `make-elliptical-arc` (see Section 3.2, “Using CLIM Drawing Options”). CLIM also supports region operations such as `region-intersection` and `region-difference` (see Section 2.5, “General Geometric Objects in CLIM”).

1.4.1.6 Text

The fundamental function for displaying text is `draw-text`. In addition to many of the graphic drawing options, text functions take a text-style argument that controls the font, face, and size.

1.4.1.7 Events

An `event` is a CLIM object that represents some sort of user gesture (such as moving the pointer or pressing a key on the keyboard) or that corresponds to some sort of notification from the display server. Event objects store such things as the sheet associated with the event, the x and y position of the pointer within that sheet, the key name or character corresponding to a key on the keyboard, and so forth.

1.4.1.8 Mediums

Graphical operations performed on panes must ultimately be carried out by the window system of the underlying host computer. This is accomplished primarily via communication with an underlying object called a `medium`. A medium understands how to implement CLIM graphics operations, such as `draw-line`, by calling the underlying host window system’s graphics functions. A medium also contains default drawing options, such as foreground and background colors, clipping region, transformations, line thickness, and fonts. There are different medium classes to support different windowing systems; thus, there is one medium class for the X Window System and a different one for the Macintosh Common Lisp environment.
This host-specific behavior is kept in a separate medium so that the pane classes themselves will be host-independent. Thus, when you build a new pane class, you do not have to build one version with X graphics mixed in, another one for the Mac, and so forth.

CLIM application programmers will not usually deal with mediums directly. In most cases, panes will automatically be allocated a medium upon creation, and output directed to the pane will be appropriately forwarded to the medium. In situations where efficiency is a concern, you may choose to send graphical output directly to the underlying medium. There are also situations, particularly when a pane has infrequent output, when you may wish to have many “light-weight” panes that share a medium.

### 1.4.2 CLIM Facilities

CLIM provides many higher-level facilities that are built from the fundamental CLIM elements.

#### 1.4.2.1 Look and Feel

CLIM offers a variety of tools and features for creating portable Lisp applications. One of these techniques, made possible by the adaptive toolkit, is the ability to transform the look and feel of a given application easily. Thus, an application can take on Motif characteristics when running on a Unix workstation, have a Microsoft Windows look and feel when running on that platform, or be presented in a different customized manner.

#### 1.4.2.2 Controlling Look and Feel

*Frame managers* are responsible for controlling the look and feel of an application frame. Each different kind of appearance, whether it be Motif or Microsoft Windows, is expressed by a different frame manager. CLIM provides frame managers that interface to a large number of host environments, including X Windows. There is also a “generic” frame manager that allows applications to maintain a “CLIM look and feel” across all platforms, rather than adopting the style of the underlying windowing system. Existing frame managers can be customized, or entirely new frame managers can be created to give your application the look and feel you desire.
A frame manager is responsible for interpreting the portable, window-system-independent layout specification of an application frame in the context of the look and feel supported by the frame manager. The abstract gadget panes, such as the scroll bars and buttons, will be mapped into specific pane classes that implement the gadget in terms of the native gadget of the host window system. For example, scroll-bar is mapped onto internal classes which uses the SCROLLBAR control on Microsoft Windows and the ScrollBar widget on Motif.

### 1.4.2.3 Changing the appearance on Windows

On Microsoft Windows (XP and later versions), themes can alter the appearance of a CLIM window and the elements within it. See the function win32:set-application-themed in the <Italic>LispWorks Reference Manual.

### 1.4.2.4 Changing the appearance on X11/Motif

You can change CLIM's color scheme and default fonts with X resources. Place your X resources in a file called CLIM port (that is, nine characters including the space) on your resource lookup path.

These are the default resources:

*buttonFontList:*times-bold-r-normal--14*
*labelFontList:  *times-bold-r-normal--14*
CLIM port*CLIM-menu-bar*foreground:#000080
CLIM port*CLIM-menu-bar*background:#b3e6fd
CLIM port*CLIMprogressbar.foreground:Red
CLIM port*CLIMpointerdoctext*foreground:#000080
CLIM port*CLIMpointerdoc*background:#b3e6fd
CLIM port*foreground: Black
CLIM port*background: White

By default, CLIM windows are mostly white. This specified in the last line above.

### 1.4.2.5 Streams

Because Common Lisp performs its input and output on objects called *streams*, CLIM does, too. In CLIM, streams are specialized sheets that implement the sheet and stream...
protocols. The basic stream protocols for input and output provide fundamental functionality such as reading and writing characters and flushing the output. Stream input is provided by low-level events; stream output is accomplished through low-level graphics.

1.4.2.6 Extended Input and Output

Streams in CLIM also support extended input and output protocols. The extended input stream protocol handles issues pertaining to, in part, non-character input such as mouse clicks. The extended output stream protocol addresses advanced issues such as text cursors, margins, text styles, inter-line spacing, and output recording.

Output recording is a facility CLIM offers for capturing all output done to an extended stream. This information is stored in structures called output records. Output recording is used in the implementation of scrollable windows and incremental redisplay. See Chapter 14, “Output Recording and Redisplay,” and Chapter 15, “Extended Stream Input Facilities,” for more details.

1.4.2.7 Presentations

The presentation facility extends output recording to remember the semantics of output displayed in a CLIM window. Presentations are specialized output records that remember not only output, but also the Lisp object associated with the output and the semantic type affiliated with that object. This semantic type, called the presentation type, allows display objects to be classified. Such semantic tagging allows the user to re-use existing output on the window to satisfy future input requests.

When a CLIM application is expecting input, an input context is established, which means the application is awaiting input of a certain semantic type. Presentations with an appropriate presentation type for the input context become sensitive; that is to say, clicking on them with the mouse will cause some action to happen. For instance, in the previous address book application example, when entering a new address, a user could type in an address or could specify input by clicking on any sensitive presentation. Addresses would be the only logical entry in this case, so only address presentation types will be sensitive. Nothing would happen if you clicked on a name or a phone number.

In a specific input context, when a given presentation type is valid input, all of the subclasses of this type are also acceptable. There are many cases, however, in which you
may wish to expand the list of valid presentation types for a given input context. This is possible by the use of presentation translators.

1.4.2.8 Command Loop

The outermost level of an application is an infinite interaction processing loop, similar to the Lisp read-eval-print loop, called a command loop. The arguments to commands are defined in terms of the presentation type facility, so that command arguments can be specified via keyboard or mouse input.

It is also possible to map presentation types to commands that operate on arguments of those types. Thus you can invoke commands by clicking on displayed data. For example, in the address book example, as the command loop awaits commands, any command display objects would be sensitive. By using the define-presentation-to-command-translator macro, however, many other presentation types can in effect be turned into commands. A click on a name in the index could represent the “Select Address” command. Similarly, clicking on a field in the displayed address, such as the “Number:” field, could be translated into the “Change Address Number” command, as illustrated in Figure 4.
1.4.3 Summary

The CLIM core, comprised of sheets, mediums, graphics, and input and output, serves as the foundation for higher-level functionality. CLIM itself provides many advanced capabilities that have been developed from this kernel. Presentations, streams, and gadgets are all descendants of the fundamental CLIM kernel. This resulting hierarchy of objects and functionality gives CLIM a layered structure. For instance, we notice that streams and gadgets are specialized panes that are themselves specialized sheets. Similarly, presentations are customized output records. At any point in these hierarchies, one may customize and specialize objects by making subclasses of existing objects and adding the
desired functionality. Although CLIM provides many advanced facilities, it is always possible to return to the fundamental CLIM building blocks and start creating anew.

1.5 Loading CLIM

To load CLIM into your LispWorks image, call

(require "clim").

To load the PostScript functionality, call

(require "clim-postscript")

To load the CLIM demos, call

(require "clim-demo")

See Section 1.7, “The CLIM demos”, for information about running the demos.

Note that module names are case-sensitive. For example (require "CLIM") will not work.

1.6 Testing Code Examples

These instructions assume that a CLIM image has already been built, or that CLIM has been loaded. Load CLIM via (require "clim"). See the <Italic>LispWorks Release Notes and Installation Guide for instructions on saving an image. Below

Below, '=>' represents the Listener prompt which may appear differently in your Lisp image

Load the sample file provided which contains CLIM code that defines an application frame:

> (load "<library-directory>/clim2/test/template.lisp")

Next, enter the following at the Lisp prompt:

> (run-frame-top-level
   (make-application-frame ‘test :width 400 :height 500))
To exit the application and return to the Lisp top level, left-click on the **Exit** menu item. Enter `(quit)` at the Lisp prompt to quit Lisp.

## 1.7 The CLIM demos

To load the demo software, enter the following in a listener:

```
(require "clim-demo")
```

To run it, enter:

```
(clim-demo:start-demo)
```

This creates a new window, containing a menu listing all the demos. Choose the demo you wish to see. The CLIM demos are quick sketches of possible applications which demonstrate a variety of CLIM programming techniques. They are not robust, production-quality applications with complete error checking, but they can provide you with some ideas.

The sources for all the demos are included. The test suite is a collection of examples of CLIM's capabilities. The testsuite examples are simple and succinct, so we recommend examining their sources for examples of CLIM's functionality that you may want to employ.

You can also run the demos directly, rather than using the menu, with the following function calls:

<table>
<thead>
<tr>
<th>Demo</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bicycle gearing</td>
<td>(clim-user::do-bicycle-gearing)</td>
</tr>
<tr>
<td>Custom output records</td>
<td>(clim-user::do-scigraph)</td>
</tr>
<tr>
<td>Peek</td>
<td>(clim-user::do-peek)</td>
</tr>
<tr>
<td>Browser</td>
<td>(clim-demo::do-ico)</td>
</tr>
<tr>
<td>Ico demo</td>
<td>(clim-browser::do-browser)</td>
</tr>
</tbody>
</table>

*Table 1. Function calls running each of the CLIM demos*
<table>
<thead>
<tr>
<th>Demo</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitmap editor</td>
<td>(clim-demo::do-bitmap-editor)</td>
</tr>
<tr>
<td>Graphics editor</td>
<td>(clim-graphics-editor::do-graphics-editor)</td>
</tr>
<tr>
<td>Color chooser</td>
<td>(clim-demo::do-color-chooser)</td>
</tr>
<tr>
<td>Plotting demo</td>
<td>(clim-demo::do-plot-demo)</td>
</tr>
<tr>
<td>Thinkadot</td>
<td>(clim-demo::do-thinkadot)</td>
</tr>
<tr>
<td>Address book</td>
<td>(clim-demo::do-address-book)</td>
</tr>
<tr>
<td>15 puzzle</td>
<td>(clim-demo::do-puzzle)</td>
</tr>
<tr>
<td>Flight planner</td>
<td>(clim-demo::do-flight-planner)</td>
</tr>
<tr>
<td>CAD demo</td>
<td>(clim-demo::do-cad-demo)</td>
</tr>
<tr>
<td>Graphics demos</td>
<td>(clim-demo::do-graphics-demo)</td>
</tr>
<tr>
<td>Lisp listener</td>
<td>(clim-demo::do-lisp-listener)</td>
</tr>
<tr>
<td>Test suite</td>
<td>(clim-test::do-test-suite)</td>
</tr>
</tbody>
</table>

Table 1. Function calls running each of the CLIM demos
<Bold>18CLIM User Guide
Chapter 2  Drawing Graphics
2.1 Conceptual Overview of Drawing Graphics

2.1.1 Drawing Functions and Options

CLIM offers a set of drawing functions that enable you to draw points, lines, polygons, rectangles, ellipses, circles, and text. You can affect the way the geometric objects are drawn by supplying options to the drawing functions. The drawing options specify clipping, transformation, line style, text style, ink, and other aspects of the graphic to be drawn. See Section 3.2, “Using CLIM Drawing Options”.

2.1.2 The Drawing Plane

When drawing graphics in CLIM, you imagine that they appear on a drawing plane. The drawing plane extends infinitely in four directions and has infinite resolution (no pixels). The drawing plane has no material existence and cannot be viewed directly. The drawing plane provides an idealized version of the graphics you draw. A line that you draw on the drawing plane is infinitely thin.

![Diagram](image)

**Figure 5. Rendering from Drawing Plane to Window**

Of course, you intend that the graphics should be visible to the user, so they must be presented on a real display device. CLIM transfers the graphics from the drawing plane to the window via the rendering process. Because the window lives on hardware that has physical
constraints, the rendering process is forced to compromise when it draws the graphics on
the window. The actual visual appearance of the window is only an approximation of the
idealized drawing plane.

Figure 5 shows the conceptual model of the drawing functions sending graphical output to
the drawing plane, and the graphics being transferred to a screen by rendering. The distinc-
tion between the idealized drawing plane and the real window enables you to develop pro-
grams without considering the constraints of a real window or other specific output device.
This distinction makes CLIM’s drawing model highly portable.

CLIM application programs can inquire about the constraints of a device, such as its reso-
lution and other characteristics, and modify the desired visual appearance on that basis.
This practice trades portability for a finer degree of control of the appearance on a given
device.

2.1.3 Coordinates

When producing graphic output on the drawing plane, you indicate where to place the out-
put with coordinates. Coordinates are a pair of numbers that specify the x and y placement
of a point. When a window is first created, the origin (that is, x = 0, y = 0) of the drawing
plane is positioned at the top-left corner of the window. Figure 6 shows the orientation of
the drawing plane. X extends toward the right, and Y extends downward.

![Figure 6. X and Y Axes of the Drawing Plane](image)

Each window looks into some rectangular area of its drawing plane. The specific area of
the drawing plane that is visible is determined by the window’s region and coordinate trans-
formation. As the window scrolls downward, the origin of the drawing plane moves above
the top edge of the window. Because windows can be located anywhere in the drawing
plane, it may be inconvenient to keep track of the coordinates of the drawing plane, and it can be easier to think in terms of a *local coordinate system*.

![Figure 7. Using a Local Coordinate System](image)

For example, you might want to draw some business graphics as shown in Figure 7. For these graphics, it is more natural to think in terms of the Y axis growing upwards, and to have an origin other than the origin of the drawing plane, which might be very far from where you want the graphics to appear. You can create a local coordinate system in which to produce your graphics. The way you do this is to define a transformation that informs CLIM how to map from the local coordinate system to the coordinates of the drawing plane. For more information, see *with-room-for-graphics*.

### 2.1.4 Mediums, Sheets, and Streams

Mediums, sheets, and streams are classes of primary importance in the creation of graphics in CLIM.

One of the arguments taken by drawing functions is a *medium*. A medium keeps track of device-specific information necessary for creating graphics. There are different medium classes to support different devices; thus, there is one medium class for the X Window System and a different one for the Macintosh Common Lisp environment. A medium implements the low-level graphic functions such as drawing a line or displaying a color. A medium also keeps track of its drawing environment, which includes such things as the current transformation, text style, line style, and foreground and background inks.
A **sheet** specifies the destination for the graphical output of a medium. Whereas mediums are device-specific, sheets are completely portable. Sheets are visible objects that have properties such as a position, a region, a parent, and children. Interface elements such as scrollbars and pushbuttons are subclasses of sheets. For convenience, sheets have also been made to support the graphics protocol. A graphics function call to a sheet object, however, simply results in the same graphics function call being made to the medium object.

**Streams** are specialized sheets that implement the sheet and stream protocols. A stream is thus a sheet that supports stream methods like **write-string** and keeps track of additional stream-related state information, such as current cursor position.

### 2.2 Examples of Using CLIM Drawing Functions

Figure 8 shows the result of evaluating the following forms:

```lisp
(clim:draw-rectangle* *my-sheet* 10 10 200 150 :filled nil :line-thickness 2)
(clim:draw-line* *my-sheet* 200 10 10 150)
(clim:draw-point* *my-sheet* 180 25)
(clim:draw-circle* *my-sheet* 100 75 40 :filled nil)
(clim:draw-ellipse* *my-sheet* 160 110 30 0 0 10 :filled nil)
(clim:draw-ellipse* *my-sheet* 160 110 10 0 0 30)
(clim:draw-polygon* *my-sheet* '(20 20 50 80 40 20) :filled nil)
(clim:draw-polygon* *my-sheet* '(30 90 40 110 20 110))
```

![Figure 8. Simple Use of the Drawing Functions](image-url)
2.3 CLIM Drawing Functions

Many of the drawing functions come in pairs. One function in the pair takes two arguments to specify a point by its \(x\) and \(y\) coordinates; the other function takes one argument, a point object. The function accepting coordinates of the point has a name with an asterisk (*) appended to it, and the function accepting a point object has the same name without an asterisk. For example, \texttt{draw-point} accepts a point object, and \texttt{draw-point*} accepts coordinates of a point. We expect that using the starred functions and specifying points by their coordinates will be more convenient in most cases.

Any drawing functions may create an output record that corresponds to the figure being drawn. See Chapter 15, “Extended Stream Input Facilities,” for a complete discussion of output recording. During output recording, none of these functions capture any arguments that are points, point sequences, coordinate sequences, or text strings. Line styles, text styles, transformations, and clipping regions may be captured.

The drawing functions are all specified as ordinary functions, not as generic functions. This is intended to ease the task of writing compile-time optimizations that avoid keyword argument taking, check for such things as constant drawing options, and so forth. If you need to specialize any of the drawing methods, use \texttt{define-graphics-method}.

Although the functions in this section are specified to be called on sheets, they can also be called on streams and mediums.

2.3.1 Arguments

- \texttt{point-seq} is a sequence of point objects.
- \texttt{coord-seq} is a sequence of coordinate pairs, which are real numbers. It is an error if \texttt{coord-seq} does not contain an even number of elements.
- The drawing functions take keyword arguments specifying drawing options. For information on the drawing options, see Section 3.2, “Using CLIM Drawing Options”. If you prefer to create and use point objects, see Subsection 2.5.2, “CLIM Point Objects”.

\begin{verbatim}
\textbf{draw-point} [Function]
Arguments: sheet point &key ink clipping-region transformation line-style line-thickness line-unit
\end{verbatim}

<Bold>24CLIM User Guide
**draw-point***  
[Function]  
Arguments: `sheet x y &key ink clipping-region transformation line-style line-thickness line-unit`  
Summary: These functions (structured and spread arguments, respectively) draw a single point on the sheet `sheet` at the point `point` (or the position `(x, y)`).

The unit and thickness components of the current line style (see Section 3.2, “Using CLIM Drawing Options”) affect the drawing of the point by controlling the number of pixels used to render the point on the display device.

**draw-points**  
[Function]  
Arguments: `sheet point-seq &key ink clipping-region transformation line-style line-thickness line-unit`  
**draw-points***  
[Function]  
Arguments: `sheet coord-seq &key ink clipping-region transformation line-style line-thickness line-unit`  
Summary: These functions (structured and spread arguments, respectively) draw a set of points on the sheet `sheet`.

For convenience and efficiency, these functions exist as equivalents to

```
(map nil #'(lambda (point) (draw-point sheet point)) point-seq)
```

and

```
(do ((i 0 (+ i 2)))
    ((= i (length coord-seq)))
  (draw-point* sheet (elt coord-seq i) (elt coord-seq (+ i 1))))
```

**draw-line**  
[Function]  
Arguments: `sheet point1 point2 &key ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape`  
**draw-line***  
[Function]  
Arguments: `sheet x1 y1 x2 y2 &key ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape`  
Summary: These functions (structured and spread arguments, respectively) draw a line segment on the sheet `sheet` from the point `point1` to `point2` (or from the position `(x1, y1)` to `(x2, y2)`).
The current line style (see Section 3.2, “Using CLIM Drawing Options”) affects the drawing of the line in the obvious way, except that the joint shape has no effect. Dashed lines start dashing at $point1$.

**draw-lines**

[Function]

**Arguments:**

- `sheet` (sheet)
- `point-seq` (sequence)
- `ink` (ink)
- `clipping-region` (clipping-region)
- `transformation` (transformation)
- `line-style` (line-style)
- `line-thickness` (number)
- `line-unit` (line-unit)
- `line-dashes` (number sequence)
- `line-cap-shape` (line-cap-shape)

**draw-lines**

[Function]

**Arguments:**

- `sheet` (sheet)
- `coord-seq` (sequence)
- `ink` (ink)
- `clipping-region` (clipping-region)
- `transformation` (transformation)
- `line-style` (line-style)
- `line-thickness` (number)
- `line-unit` (line-unit)
- `line-dashes` (number sequence)
- `line-cap-shape` (line-cap-shape)

**Summary:**

These functions (structured and spread arguments, respectively) draw a set of disconnected line segments. These functions are equivalent to

```
(defun draw-lines (sheet point-seq &key ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape)
  (do ((i 0 (+ i 2)))
      (= i (length point-seq))
    (draw-line sheet (elt point-seq i) (elt point-seq (1+ i)))))
```

and

```
(defun draw-lines* (sheet coord-seq &key ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape)
  (do ((i 0 (+ i 4)))
      (= i (length coord-seq))
    (draw-line* sheet
      (elt coord-seq i) (elt coord-seq (+ i 1))
      (elt coord-seq (+ i 2))
      (elt coord-seq (+ i 3))))
```

**draw-polygon**

[Function]

**Arguments:**

- `sheet` (sheet)
- `point-seq` (sequence)
- `ink` (ink)
- `filled` (t or nil)
- `closed` (t or nil)
- `clipping-region` (clipping-region)
- `transformation` (transformation)
- `line-style` (line-style)
- `line-thickness` (number)
- `line-unit` (line-unit)
- `line-dashes` (number sequence)
- `line-joint-shape` (line-joint-shape)
- `line-cap-shape` (line-cap-shape)

**draw-polygon**

[Function]

**Arguments:**

- `sheet` (sheet)
- `coord-seq` (sequence)
- `ink` (ink)
- `filled` (t or nil)
- `closed` (t or nil)
- `clipping-region` (clipping-region)
- `transformation` (transformation)
- `line-style` (line-style)
- `line-thickness` (number)
- `line-unit` (line-unit)
- `line-dashes` (number sequence)
- `line-joint-shape` (line-joint-shape)
- `line-cap-shape` (line-cap-shape)

**Summary:**

Draws a polygon or polyline on the sheet `sheet`. When `filled` is `nil`, this draws a set of connected lines; otherwise, it draws a filled polygon. If `closed` is `t` (the default) and `filled` is `nil`, it ensures that a segment is drawn that connects the ending point of the last segment to the starting point of the first segment. The current line style (see Section 3.3, “CLIM Line Styles” for details) affects the drawing of unfilled polygons in the obvious way. The cap shape affects only the “open”
vertices in the case when \textit{closed} is \texttt{nil}. Dashed lines start dashing at the starting point of the first segment, and may or may not continue dashing across vertices, depending on the window system.

If \textit{filled} is \texttt{t}, a closed polygon is drawn and filled in. In this case, \textit{closed} is assumed to be \texttt{t} as well.

\begin{verbatim}
draw-rectangle
Arguments: sheet point1 point2 &key (filled t) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape

draw-rectangle*
Arguments: sheet x1 y1 x2 y2 &key (filled t) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape
Summary: Draws either a filled or unfilled rectangle on the sheet \textit{sheet} that has its sides aligned with the coordinate axes of the native coordinate system. One corner of the rectangle is at the position \((x1, y1)\) or \textit{point1} and the opposite corner is at \((x2, y2)\) or \textit{point2}. The arguments \(x1, y1, x2,\) and \(y2\) are real numbers that are canonicalized in the same way as for \texttt{make-bounding-rectangle}. \textit{filled} is as for \texttt{draw-polygon*}.

The current line style (see Section 3.2, “Using CLIM Drawing Options”) affects the drawing of unfilled rectangles in the obvious way, except that the cap shape has no effect.

draw-rectangles
Arguments: sheet points &key ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape

draw-rectangles*
Arguments: sheet position-seq &key ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape
Summary: These functions (structured and spread arguments, respectively) draw a set of rectangles on the sheet \textit{sheet}. \textit{points} is a sequence of point objects; \textit{position-seq} is a sequence of coordinate pairs. It is an error if \textit{position-seq} does not contain an even number of elements.

Ignoring the drawing options, these functions are equivalent to:

\begin{verbatim}
(do ((i 0 (+ i 2)))
    ((= i (length points))
     (draw-rectangle sheet (elt points i) (elt points (1+ i))))
\end{verbatim}
\end{verbatim}
and

(do ((i 0 (+ i 4)))
  ((= i (length position-seq)))
  (draw-rectangle* sheet
    (elt position-seq i)
    (elt position-seq (+ i 1))
    (elt position-seq (+ i 2))
    (elt position-seq (+ i 3))))

**draw-ellipse**

Arguments: *sheet* center-*pt* radius-1-*dx* radius-1-*dy* radius-2-*dx* radius-2-*dy* &key (filled t)

*start-angle* end-*angle* ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape

**draw-ellipse***

Arguments: *sheet* center-*x* center-*y* radius-1-*dx* radius-1-*dy* radius-2-*dx* radius-2-*dy* &key (filled t) *start-angle* end-*angle* ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape

Summary: These functions (structured and spread arguments, respectively) draw an ellipse (when *filled* is t, the default) or an elliptical arc (when *filled* is nil) on the sheet *sheet*. The center of the ellipse is the point *center-pt* (or the position (*center-x*, *center-y*)).

Two vectors, (radius-1-*dx*, radius-1-*dy*) and (radius-2-*dx*, radius-2-*dy*) specify the bounding parallelogram of the ellipse as explained in Section 2.5, “General Geometric Objects in CLIM.” All of the radii are real numbers. If the two vectors are collinear, the ellipse is not well-defined and the **ellipse-not-well-defined** error will be signaled. The special case of an ellipse with its major axes aligned with the coordinate axes can be obtained by setting both radius-1-*dx* and radius-2-*dx* to 0.

*start-angle* and *end-angle* are real numbers that specify an arc rather than a complete ellipse. Angles are measured with respect to the positive x axis. The elliptical arc runs positively (counter-clockwise) from *start-angle* to *end-angle*. The default for *start-angle* is 0; the default for *end-angle* is $2\pi$.

In the case of a “filled arc” (that is, when *filled* is t and *start-angle* or *end-angle* are supplied and are not 0 and $2\pi$), the figure drawn is the “pie slice” area swept out by a line from the center of the ellipse to a point on the boundary as the boundary point moves from *start-angle* to *end-angle*.
When drawing unfilled ellipses, the current line style (see Section 3.2, “Using CLIM Drawing Options”) affects the drawing in the obvious way, except that the joint shape has no effect. Dashed elliptical arcs start dashing at \textit{start-angle}.

\textbf{draw-circle} \\
Arguments: \textit{sheet center-pt radius} \&\textit{key} (\textit{filled} \texttt{t}) \textit{start-angle end-angle ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape}

\textbf{draw-circle*} \\
Arguments: \textit{sheet center-x center-y radius} \&\textit{key} (\textit{filled} \texttt{t}) \textit{start-angle end-angle ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape}

Summary: These functions (structured and spread arguments, respectively) draw a circle (when \textit{filled} is \texttt{t}, the default) or a circular arc (when \textit{filled} is \texttt{nil}) on the sheet \textit{sheet}. The center of the circle is \textit{center-pt} or (\textit{center-x}, \textit{center-y}) and the radius is \textit{radius}. These are just special cases of \textbf{draw-ellipse} and \textbf{draw-ellipse*}. \textit{filled} is as for \textbf{draw-ellipse*}.

\textit{start-angle} and \textit{end-angle} allow the specification of an arc rather than a complete circle in the same manner as that of the ellipse functions.

The “filled arc” behavior is the same as that of an ellipse.

\textbf{draw-text} \\
Arguments: \textit{sheet string-or-char point} \&\textit{key} \textit{text-style} (\textit{start} \texttt{0}) \textit{end} \textit{(align-x :\texttt{left}) (align-y \texttt{baseline}) toward-point transform-glyphs ink clipping-region transformation text-style text-family text-face text-size}

\textbf{draw-text*} \\
Arguments: \textit{sheet string-or-char x y} \&\textit{key} \textit{text-style} (\textit{start} \texttt{0}) \textit{end} \textit{(align-x :\texttt{left}) (align-y \texttt{baseline}) toward-x toward-y transform-glyphs ink clipping-region transformation text-style text-family text-face text-size}

Summary: The text specified by \textit{string-or-char} is drawn on the sheet \textit{sheet} starting at the position specified by the point \textit{point} (or the position (\textit{x}, \textit{y})). The exact definition of “starting at” depends on \textit{align-x} and \textit{align-y}. \textit{align-x} is one of :\texttt{left}, :\texttt{center}, or :\texttt{right}. \textit{align-y} is one of :\texttt{baseline}, :\texttt{top}, :\texttt{center}, or :\texttt{bottom}. \textit{align-x} defaults to :\texttt{left} and \textit{align-y} defaults to :\texttt{baseline}; with these defaults, the first glyph is drawn with its left edge and its baseline at \textit{point}.
text-style defaults to nil, meaning that the text will be drawn using the current text style of the sheet’s medium.

start and end specify the start and end of the string, in the case where string-or-char is a string. If start is supplied, it must be an integer that is less than the length of the string. If end is supplied, it must be an integer that is less than the length of the string, but greater than or equal to start.

Normally, glyphs are drawn from left to right no matter what transformation is in effect. toward-x or toward-y (derived from toward-point in the case of draw-text) can be used to change the direction from one glyph to the next one. For example, if toward-x is less than the x position of point, then the glyphs will be drawn from right to left. If toward-y is greater than the y position of point, then the glyphs’ baselines will be positioned one above another. More precisely, the reference point in each glyph lies on a line from point to toward-point, and the spacing of each glyph is determined by packing rectangles along that line, where each rectangle is “char-width” wide and “char-height” high.

transform-glyphs is not supported in this version of CLIM.

2.3.2 Compound Drawing Functions

CLIM also provides a few compound drawing functions. The compound drawing functions could be composed by a programmer from the basic drawing functions, but are provided by CLIM because they are commonly used.

draw-arrow
[Function]
Arguments: sheet point-1 point-2 &key ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape to-head from-head head-length head-width
draw-arrow*
[Function]
Arguments: sheet x1 y1 x2 y2 &key ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape from-head to-head head-length head-width
Summary: These functions (structured and spread arguments, respectively) draw a line segment on the sheet sheet from the point point1 to point2 (or from the position (x1, y1) to (x2, y2)). If to-head is t (the default), then the “to” end of the line is capped by an arrowhead. If from-head is t (the default is nil), then the “from” end of the
line is capped by an arrowhead. The arrowhead has length \texttt{head-length} (default 10) and width \texttt{head-width} (default 5).

The current line style (see Section 3.2, “Using CLIM Drawing Options”) affects the drawing of the line portion of the arrow in the obvious way, except that the joint shape has no effect. Dashed arrows start dashing at \texttt{point1}.

\texttt{draw-oval} \quad [\text{Function}]
\begin{itemize}
\item \textbf{Arguments:} \texttt{sheet center-pt x-radius y-radius &key (filled t) ink clipping-region transformation line-style line-thickness line-dashes line-capshape}
\end{itemize}

\texttt{draw-oval*} \quad [\text{Function}]
\begin{itemize}
\item \textbf{Arguments:} \texttt{sheet center-x center-y x-radius y-radius &key (filled t) ink clipping-region transformation line-style line-thickness line-dashes line-capshape}
\end{itemize}

\textbf{Summary:} These functions (structured and spread arguments, respectively) draw a filled or unfilled oval (that is, a “race-track” shape) on the sheet \texttt{sheet}. The oval is centered on \texttt{center-pt} (or \texttt{(center-x, center-y)}). If \texttt{x-radius} or \texttt{y-radius} is 0, then a circle is drawn with the specified non-zero radius. Otherwise, a figure is drawn that is a rectangle with dimension \texttt{x-radius} by \texttt{y-radius}, with the two short sides replaced by a semicircular arc of the appropriate size.

\section{2.3.3 Patterns and Stencils}

\textit{Patterning} creates a bounded rectangular arrangement of designs, like a checkerboard. Drawing a pattern draws a different design in each rectangular cell of the pattern. To create an infinite pattern, apply \texttt{make-rectangular-tile} to a pattern.

A \textit{stencil} is a special kind of pattern that contains only opacities.

\texttt{make-pattern} \quad [\text{Function}]
\begin{itemize}
\item \textbf{Arguments:} \texttt{array inks}
\end{itemize}

\textbf{Summary:} Returns a pattern ink that has \texttt{(array-dimension array 0)} cells in the vertical direction and \texttt{(array-dimension array 1)} cells in the horizontal direction. \texttt{array} must be a two-dimensional array of non-negative integers less than the length of \texttt{inks}. \texttt{inks} must be a sequence of designs. The design in cell \texttt{(i, j)} of the resulting pattern is the \textit{n}th element of \texttt{inks}, if \textit{n} is the value of \texttt{(aref array i j)}. 

Drawing Graphics
For example, \textit{array} can be a bit-array and \textit{inks} can be a list of two inks, the ink drawn for 0 and the one drawn for 1.

Each cell of a pattern can be regarded as a hole that allows the ink in it to show through. Each cell might have a different ink in it. The portion of the ink that shows through a hole is the portion on the part of the drawing plane where the hole is located. In other words, incorporating an ink into a pattern does not change its alignment to the drawing plane, and does not apply a coordinate transformation to the design. Drawing a pattern collects the pieces of inks that show through all the holes and draws the pieces where the holes lie on the drawing plane. The pattern is completely transparent outside the area defined by the array.

This function captures its mutable inputs; the consequences of modifying those objects are unspecified.

\textit{Tiling} repeats a rectangular portion of a pattern throughout the drawing plane.

\textbf{make-rectangular-tile} \hfill [Function]
\textbf{Arguments:} \textit{pattern width height} \\
\textbf{Summary:} Returns a pattern that, when used as an ink, tiles a rectangular portion of the pattern \textit{pattern} across the entire drawing plane. The resulting pattern repeats with a period of \textit{width} horizontally and \textit{height} vertically. \textit{width} and \textit{height} must both be integers. The portion of \textit{pattern} that appears in each tile is a rectangle whose top-left corner is at (0, 0) and whose bottom-right corner is at (\textit{width}, \textit{height}). The repetition of \textit{pattern} is accomplished by applying a coordinate transformation to shift \textit{pattern} into position for each tile, and then extracting a \textit{width}-by-\textit{height} portion of that pattern.

Applying a coordinate transformation to a rectangular tile does not change the portion of the argument \textit{pattern} that appears in each tile. However, it can change the period, phase, and orientation of the repeated pattern of tiles. This is so that adjacent figures drawn using the same tile have their inks “line up.”

\textbf{draw-pattern*} \hfill [Function]
\textbf{Arguments:} \textit{sheet pattern x y} \&key \textit{clipping-region transformation} \\
\textbf{Summary:} Draws the pattern \textit{pattern} on the sheet \textit{sheet} at the position (\textit{x}, \textit{y}). \textit{pattern} is any pattern created by \textbf{make-pattern}. \textit{clipping-region} and \textit{transformation} are as for \textbf{with-drawing-options} or any of the drawing functions.
Note that transformation only affects the position at which the pattern is drawn, not the pattern itself. If you want to affect the pattern, you should explicitly call \texttt{transform-region} on the pattern.

You draw a bitmap by drawing an appropriately aligned and scaled pattern constructed from the bitmap’s bits. A 1 in the bitmap corresponds to \texttt{foreground-ink}. A 0 corresponds to \texttt{background-ink} if an opaque drawing operation is desired, or to \texttt{nowhere} if a transparent drawing operation is desired.

Drawing a (colored) raster image consists of drawing an appropriately aligned and scaled pattern constructed from the raster array and raster color map.

\texttt{draw-pattern}\* could be implemented as follows, assuming that the functions \texttt{pattern-width} and \texttt{pattern-height} return the width and height of the pattern.

\begin{verbatim}
(defun draw-pattern* (sheet pattern x y &key clipping-region transformation)
  (check-type pattern pattern)
  (let ((width (pattern-width pattern))
        (height (pattern-height pattern)))
    (if (or clipping-region transformation)
        (with-drawing-options
         (sheet
          :clipping-region clipping-region
          :transformation transformation)
         (draw-rectangle* sheet x y
                        (+ x width) (+ y height)
                        :filled t :ink pattern))
        (draw-rectangle* sheet x y (+ x width) (+ y height)
                        :filled t :ink pattern))))
\end{verbatim}

\section*{2.3.4 Pixmaps}

A \texttt{ pixmap} can be thought of as an “off-screen window,” that is, a medium that can be used for graphical output, but that is not visible on any display device. Pixmaps are provided to allow a programmer to generate a piece of output associated with some display device that can then be rapidly drawn on a real display device. For example, an electrical CAD system might generate a pixmap that corresponds to a complex, frequently-used part in a VLSI schematic, and then use \texttt{copy-from-pixmap} to draw the part as needed.

The exact representation of a pixmap is explicitly unspecified. There is no interaction between the pixmap operations and output recording; that is, displaying a pixmap on a med-
um is a pure drawing operation that affects only the display, not the output history. Some mediums may not support pixmaps; in this case, an error will be signaled.

**allocate-pixmap**

Arguments: *medium width height*

Summary: Allocates and returns a pixmap object that can be used on any medium that shares the same characteristics as *medium*. (What constitutes “shared characteristics” varies from host to host.) *medium* can be a sheet, a medium, or a stream. The resulting pixmap will be *width* units wide, *height* units high, and as deep as is necessary to store the information for the medium. The exact representation of pixmaps is explicitly unspecified. The returned value is the pixmap.

**deallocate-pixmap**

Arguments: *pixmap*

Summary: Deallocates the pixmap *pixmap*.

**pixmap-width**

Arguments: *pixmap*

**pixmap-height**

Arguments: *pixmap*

**pixmap-depth**

Arguments: *pixmap*

Summary: These functions return, respectively, the programmer-specified width, height, and depth of the pixmap *pixmap*.

**copy-to-pixmap**

Arguments: *medium medium-x medium-y width height &optional pixmap (pixmap-x 0) (pixmap-y 0)*

Summary: Copies the pixels from the medium *medium* starting at the position specified by (*medium-x*, *medium-y*) into the pixmap *pixmap* at the position specified by (*pixmap-x*, *pixmap-y*). A rectangle whose width and height is specified by *width* and *height* is copied. *medium-x* and *medium-y* are specified in user coordinates. (If *medium* is a medium or a stream, then *medium-x* and *medium-y* are transformed by the user transformation.)
If *pixmap* is not supplied, a new pixmap will be allocated. Otherwise, *pixmap* must be an object returned by `allocate-pixmap` that has the appropriate characteristics for *medium*.

The returned value is the pixmap.

### copy-from-pixmap

**Arguments:** `pixmap pixmap-x pixmap-y width height medium window-x window-y`  
**Summary:** Copies the pixels from the pixmap *pixmap* starting at the position specified by `(pixmap-x, pixmap-y)` into the medium *medium* at the position `(medium-x, medium-y)`. A rectangle whose width and height is specified by *width* and *height* is copied. *medium-x* and *medium-y* are specified in user coordinates. (If *medium* is a medium or a stream, then *medium-x* and *medium-y* are transformed by the user transformation.)

`pixmap` must be an object returned by `allocate-pixmap` that has the appropriate characteristics for *medium*.

The returned value is the pixmap. This is intended to specialize on both the *pixmap* and *medium* arguments.

### copy-area

**Arguments:** `medium from-x from-y width height to-x to-y`  
**Summary:** Copies the pixels from the medium *medium* starting at the position specified by `(from-x, from-y)` to the position `(to-x, to-y)` on the same medium. A rectangle whose width and height is specified by *width* and *height* is copied. *from-x*, *from-y*, *to-x*, and *to-y* are specified in user coordinates. (If *medium* is a medium or a stream, then *medium-x* and *medium-y* are transformed by the user transformation.)

### with-output-to-pixmap

**Arguments:** `(medium-var medium &key width height) &body body`  
**Summary:** Binds *medium-var* to a “pixmap medium” (that is, a medium that does output to a pixmap with the characteristics appropriate to the medium *medium*) and then evaluates *body* in that context. All the output done to the medium designated by *medium-var* inside of *body* is drawn on the pixmap stream. The pixmap medium supports the medium output protocol, including all of the graphics functions.

*width* and *height* are integers that give the dimensions of the pixmap. If they are omitted, the pixmap will be large enough to contain all the output done by *body*.
medium-var must be a symbol; it is not evaluated. The returned value is a pixmap that can be drawn onto medium using copy-from-pixmap.

2.4 Graphics Protocols

Every medium implements methods for the various graphical drawing generic functions. Furthermore, every sheet that supports the standard output protocol implements these methods as well; often, the sheet methods will simply call the same methods on the sheet’s medium.

2.4.1 Arguments

- All these generic functions take the same arguments as the non-generic spread function equivalents, except that the arguments that are keyword arguments in the non-generic functions are required arguments in the generic functions.
- The drawing-function-specific arguments are either x and y positions, or a sequence of x and y positions. Note that these positions will first be transformed by the medium’s current transformation, and then transformed a second time by the medium’s device transformation in order to produce the coordinates as they will actually appear on the screen.
- The ink, line style (or text style), and clipping regions arguments are optional, and default from the medium (medium-ink, medium-line-style (or medium-current-text-style), and medium-clipping-region, respectively).

2.4.2 General Behavior of Drawing Functions

Using draw-line* as an example, calling any of the drawing functions specified previously results in the following series of function calls on a non-output recording sheet:

- A program calls draw-line* on arguments sheet, x1, y1, x2, and y2, and perhaps some drawing options.
- draw-line* merges the supplied drawing options into the sheet’s medium, and then calls medium-draw-line* on the sheet. (Note that a compiler macro could detect the
case where there are no drawing options or constant drawing options, and do this at compile time.)

- medium-draw-line* on the sheet calls the same method—medium-draw-line*—on the medium.
- medium-draw-line* performs the necessary user transformations by applying the medium transformation to \(x_1, y_1, x_2,\) and \(y_2,\) and to the clipping region.

### 2.4.3 Medium-Specific Drawing Functions

All mediums and all sheets that support the standard output protocol implement methods for the following generic functions.

**medium-draw-point**

- **Arguments:** \(medium \ x \ y\)
- **Summary:** Draws a point on the medium \(medium\).

**medium-draw-points**

- **Arguments:** \(medium \ coord-seq\)
- **Summary:** Draws a set of points on the medium \(medium\).

**medium-draw-line**

- **Arguments:** \(medium \ x_1 \ y_1 \ x_2 \ y_2\)
- **Summary:** Draws a line from \((x_1, y_1)\) to \((x_2, y_2)\) on the medium \(medium\).

**medium-draw-lines**

- **Arguments:** \(medium \ coord-seq\)
- **Summary:** Draws a set of disconnected lines on the medium \(medium\).

**medium-draw-polygon**

- **Arguments:** \(medium \ coord-seq \ closed\)
- **Summary:** Draws a polygon or polyline on the medium \(medium\).

**medium-draw-rectangle**

- **Arguments:** \(medium \ x_1 \ y_1 \ x_2 \ y_2\)
- **Summary:** Draws a rectangle whose corners are at \((x_1, y_1)\) and \((x_2, y_2)\) on \(medium\).
medium-draw-ellipse*  [Generic Function]
Arguments: medium center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy start-angle end-angle
Summary: Draws a rectangle on medium. The center is at \((x, y)\). The vectors \((\text{radius-1-dx}, \text{radius-1-dy})\) and \((\text{radius-2-dx}, \text{radius-2-dy})\) specify the radii. \text{start-angle} and \text{end-angle} are real numbers that specify an arc, not a complete ellipse.

medium-draw-text*  [Generic Function]
Arguments: medium text x y (start 0) end (align-x :left) (align-y :baseline) toward-x toward-y transform-glyphs
Summary: Draws a character or a string on the medium medium. The text is drawn starting at \((x, y)\), and towards \((\text{toward-x}, \text{toward-y})\).

2.5 General Geometric Objects in CLIM

2.5.1 Regions in CLIM

A \textit{region} is an object that denotes a set of points in the plane. Regions include their boundaries; that is, they are closed. Regions have infinite resolution.

A \textit{bounded region} is a region that contains at least one point for which there exists a number, \(d\), called the region’s diameter, such that if \(p_1\) and \(p_2\) are points in the region, the distance between \(p_1\) and \(p_2\) is always less than or equal to \(d\).

An \textit{unbounded region} either contains no points or contains points arbitrarily far apart. \texttt{+nowhere+} and \texttt{+everywhere+} are examples of unbounded regions.

Another way to describe a region is to say that it maps every \((x, y)\) pair into either true or false (meaning member or not a member, respectively, of the region). Later, in Chapter 5, we will generalize a region to something called an \textit{ink} that maps every point \((x, y)\) into color and opacity values.

CLIM classifies the various types of regions in the following way. All regions are a subclass of \texttt{region}, and all bounded regions are also a subclass of either \texttt{point}, \texttt{path}, or \texttt{area}, as shown in Figure 9.
Figure 9. The Class Structure for All Regions

region [Protocol Class]
Summary: The protocol class that corresponds to a set of points. This includes both bounded and unbounded regions. This is a subclass of ink (see Chapter 5, “Drawing in Color,” for details).

If you want to create a new class that behaves like a region, it should be a subclass of region. Subclasses of region must obey the region protocol.

There is no general constructor called make-region because of the impossibility of a uniform way to specify the arguments to such a function.

regionp [Function]
Arguments: object
Summary: Returns t if object is a region; otherwise, it returns nil.

path [Protocol Class]
Summary: The protocol class path denotes bounded regions that have dimensionality 1 (that is, lines or curves). It is a subclass of region and bounding-rectangle. If
you want to create a new class that behaves like a path, it should be a subclass of path. Subclasses of path must obey the path protocol.

Constructing a path object with no length (via make-line*, for example) canonicalizes it to +nowhere+.

Some rendering models support the constructing of areas by filling a closed path. In this case, the path needs a direction associated with it. Since CLIM does not currently support the path-filling model, paths are directionless.

**pathp**

[Function]

Arguments: object
Summary: Returns t if object is a path; otherwise, it returns nil.

Note that constructing a path object with no length (by calling make-line with two coincident points, for example) canonicalizes it to +nowhere+.

**area**

[Protocol Class]

Summary: The protocol class area denotes bounded regions that have dimensionality 2 (that is, are flat surfaces). It is a subclass of region and bounding-rectangle. If you want to create a new class that behaves like an area, it should be a subclass of area. Subclasses of area must obey the area protocol.

Note that constructing an area object with no area (by calling make-rectangle with two coincident points, for example) canonicalizes it to +nowhere+.

**areap**

[Function]

Arguments: object
Summary: Returns t if object is an area; otherwise, it returns nil.

**coordinate**

[Type]

Summary: The type that represents a coordinate. All of the specific region classes and subclasses of bounding-rectangle will use this type to store their coordinates. However, the constructor functions for the region classes and for bounding rectangles accept numbers of any type and coerce them to coordinate.

The following two constants represent the regions that correspond, respectively, to all of the points on the drawing plane and to none of the points on the drawing plane.
+everywhere+  
[Constant]  
Summary: The region that includes all the points on the infinite drawing plane.

+nowhere+  
[Constant]  
Summary: The empty region (the opposite of +everywhere+).

2.5.1.1 Region Predicates in CLIM

The following generic functions comprise the region predicate protocol. All classes that are subclasses of region must either inherit or implement methods for these generic functions.

The methods for region-equal, region-contains-region-p, and region-intersects-region-p will typically specialize both the region1 and region2 arguments.

region-equal  
[Generic Function]  
Arguments: region1 region2  
Summary: Returns t if the two regions region1 and region2 contain exactly the same set of points; otherwise, it returns nil.

region-contains-region-p  
[Generic Function]  
Arguments: region1 region2  
Summary: Returns t if all points in the region region2 are members of the region region1; otherwise, it returns nil.

region-contains-position-p  
[Generic Function]  
Arguments: region x y  
Summary: Returns t if the point at (x, y) is contained in the region region; otherwise, it returns nil. Since regions in CLIM are closed, this must return t if the point at (x, y) is on the region’s boundary.

region-contains-position-p is a special case of region-contains-region-p in which the region is the point (x, y).

region-intersects-region-p  
[Generic Function]  
Arguments: region1 region2  
Summary: Returns nil if region-intersection of the two regions region1 and region2 would be +nowhere+; otherwise, it returns t.
2.5.1.2 Composition of CLIM Regions

Region composition in CLIM is the process in which two regions are combined in some way (such as union or intersection) to produce a third region.

Since all regions in CLIM are closed, region composition is not always equivalent to simple set operations. Instead, composition attempts to return an object that has the same dimensionality as one of its arguments. If this is not possible, then the result is defined to be an empty region, which is canonicalized to \texttt{+nowhere+}. (The exact details of this are specified with each function.)

Sometimes composition of regions can produce a result that is not a simple contiguous region. For example, region-union of two rectangular regions might not be rectangular. In order to support cases like this, CLIM has the concept of a region set, an object that represents one or more region objects related by some region operation, usually a union.

\textbf{region-set} [Protocol Class]

Summary: The protocol class that represents a region set; a subclass of \texttt{region} and \texttt{bounding-rectangle}.

Members of this class are immutable.

\textbf{region-set-p} [Function]

Arguments: \textit{object}

Summary: Returns \texttt{t} if \textit{object} is a region set; otherwise, it returns \texttt{nil}.

\textbf{standard-rectangle-set} [Class]

Summary: This instantiable subclass of \texttt{region-set} and \texttt{bounding-rectangle} represents the union of several axis-aligned rectangles.

\textbf{standard-region-union} [Class]

\textbf{standard-region-intersection} [Class]

\textbf{standard-region-difference} [Class]

Summary: These three instantiable classes respectively implement the union, intersection, and differences of regions.
Region sets that are composed entirely of axis-aligned rectangles must be canonicalized into either a single rectangle or a union of rectangles. Furthermore, the rectangles in the union must not overlap each other.

The following generic functions comprise the region composition protocol. All classes that are subclasses of region must implement methods for these generic functions.

The methods for region-union, region-intersection, and region-difference will typically specialize both the region1 and region2 arguments.

region-set-regions
[Generic Function]
Arguments: region &key normalize
Summary: Returns a sequence of the regions in the region set region. region can be either a region set or a “simple” region, in which case the result is simply a sequence of one element: region.

Note: This function returns objects that reveal CLIM’s internal state; do not modify these objects.

For the case of region sets that are unions of axis-aligned rectangles, the rectangles returned by region-set-regions are guaranteed not to overlap.

If normalize is supplied, it must be either :x-banding or :y-banding. If it is :x-banding and all the regions in region are axis-aligned rectangles, the result is normalized by merging adjacent rectangles with banding done in the x direction. If it is :y-banding and all the regions in region are rectangles, the result is normalized with banding done in the y direction. Normalizing a region set that is not composed entirely of axis-aligned rectangles using x- or y-banding causes CLIM to signal the region-set-not-rectangular error.

Drawing Graphics
Figure 10. Normalization of Rectangular Region Sets

**map-over-region-set-regions**

Arguments: `function region &key normalize`

Summary: Calls `function` on each region in the region set `region`. This is often more efficient than calling `region-set-regions`. `function` is a function of one argument, a region; it has dynamic extent. `region` can be either a region set or a “simple” region, in which case `function` is called once on `region` itself. `normalize` is as for `region-set-regions`.

**region-union**

Arguments: `region1 region2`

Summary: Returns a region that contains all points that are in either of the regions `region1` or `region2` (possibly with some points removed in order to satisfy the dimensionality rule). The result of `region-union` always has dimensionality that is the maximum dimensionality of `region1` and `region2`. For example, the union of a path and an area produces an area; the union of two paths is a path.

`region-union` will return either a simple region or a member of the class `standard-region-union`. 

44CLIM User Guide
This function captures its mutable inputs; the consequences of modifying those objects are unspecified.

**region-intersection**

[Generic Function]

**Arguments:** region1 region2

**Summary:** Returns a region that contains all points that are in both of the regions region1 and region2 (possibly with some points removed in order to satisfy the dimensionality rule). The result of region-intersection has dimensionality that is the minimum dimensionality of region1 and region2, or is +nowhere+. For example, the intersection of two areas is either another area or +nowhere+; the intersection of two paths is either another path or +nowhere+; the intersection of a path and an area produces the path clipped to stay inside of the area.

region-intersection will return either a simple region or a member of the class standard-region-intersection.

This function captures its mutable inputs; the consequences of modifying those objects are unspecified.

**region-difference**

[Generic Function]

**Arguments:** region1 region2

**Summary:** Returns a region that contains all points in the region region1 that are not in the region region2 (possibly plus additional boundary points to make the result closed). The result of region-difference has the same dimensionality as region1, or is +nowhere+. For example, the difference of an area and a path produces the same area; the difference of a path and an area produces the path clipped to stay outside of the area.

region-difference will return either a simple region, a region set, or a member of the class standard-region-difference.

This function captures its mutable inputs; the consequences of modifying those objects are unspecified.
2.5.2 CLIM Point Objects

A point is a mathematical point in the drawing plane that is identified by its coordinates, a pair of real numbers. Points have neither area nor length. Note that a point is not the same thing as a pixel; CLIM’s model of the drawing plane has continuous coordinates.

You can create point objects and use them as arguments to the drawing functions. Alternatively, you can use the spread versions of the drawing functions, that is, the drawing functions with stars appended to their names. For example, instead of \texttt{draw-point}, use \texttt{draw-point*}, which takes two arguments specifying a point by its coordinates. (Note that, for performance reasons, we generally recommend the use of the spread versions.)

The operations for creating and dealing with points are:

\texttt{point} \quad [\text{Protocol Class}]

\textbf{Summary:} The protocol class that corresponds to a mathematical point. This is a subclass of \texttt{region} and \texttt{bounding-rectangle}. If you want to create a new class that behaves like a point, it should be a subclass of \texttt{point}. Subclasses of \texttt{point} obey the point protocol.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{region-union-intersection-difference}
\caption{Examples of Region Union, Intersection, and Difference}
\end{figure}
**pointp**  
[Function]  
Arguments: `object`  
Summary: Returns `t` if `object` is a point; otherwise, it returns `nil`.  

**standard-point**  
[Class]  
Summary: An instantiable class that implements a point. This is a subclass of `point`. This is the class that `make-point` instantiates. Members of this class are immutable.  

**make-point**  
[Function]  
Arguments: `x` `y`  
Summary: Returns an object of class `standard-point` whose coordinates are `x` and `y`. `x` and `y` must be real numbers.  

The following generic functions comprise the point Application Programmer Interface. Only `point-position` is in the point protocol; that is, all classes that are subclasses of `point` must implement methods for `point-position`, but need not implement methods for `point-x` and `point-y`.  

**point-position**  
[Generic Function]  
Arguments: `point`  
Summary: Returns both the `x` and `y` coordinates of the point `point` as two values.  

**point-x**  
[Generic Function]  
Arguments: `point`  

**point-y**  
[Generic Function]  
Arguments: `point`  
Summary: Returns the `x` or `y` coordinate of the point `point`, respectively. CLIM will supply default methods for `point-x` and `point-y` on the protocol class `point` that are implemented by calling `point-position`.  

### 2.5.3 Polygons and Polylines in CLIM  

A **polyline** is a path that consists of one or more line segments joined consecutively at their end-points. A **line** is a polyline that has only one segment.
Polylines that have the end-point of their last line segment coincident with the start-point of their first line segment are called **closed**; this use of the term “closed” should not be confused with closed sets of points.

A **polygon** is an area bounded by a closed polyline.

If the boundary of a polygon intersects itself, the odd-even winding-rule defines the polygon: a point is inside the polygon if a ray from the point to infinity crosses the boundary an odd number of times.

Polylines and polygons are closed under affine transformations.

The classes that correspond to polylines and polygons are:

- **polyline** [Protocol Class]
  Summary: The protocol class that corresponds to a polyline. It is a subclass of **path**. If you want to create a new class that behaves like a polyline, it should be a subclass of **polyline**. Subclasses of **polyline** must obey the polyline protocol.

- **polylinep** [Function]
  Arguments: **object**
  Summary: Returns `t` if `object` is a polyline; otherwise, it returns `nil`.

- **polygon** [Class]
  Summary: The protocol class (a subclass of **area**) that corresponds to a mathematical polygon. If you want to create a new class that behaves like a polygon, it should be a subclass of **polygon**. Subclasses of **polygon** must obey the polygon protocol.

- **polygonp** [Function]
  Arguments: **object**
  Summary: Returns `t` if **object** is a polygon; otherwise, it returns `nil`.

- **standard-polyline** [Class]
  Summary: A class that implements a polyline. This is a subclass of **polyline**. This is the class that **make-polyline** and **make-polyline*** instantiate. Members of this class are immutable.

- **standard-polygon** [Class]

<Bold>48CLIM User Guide</Bold>
Summary: A class that implements a polygon. This is a subclass of polygon. This is the class that make-polygon and make-polygon* instantiate. Members of this class are immutable.

2.5.3.1 Constructors for CLIM Polygons and Polylines

The following functions can be used to create polylines and polygons:

**make-polyline**  
Arguments: point-seq &key closed

**make-polyline***  
Arguments: coord-seq &key closed
Summary: Returns an object of class standard-polyline consisting of the segments connecting each of the points in point-seq (or the points represented by the coordinate pairs in coord-seq).

If closed is t, then the segment connecting the first point and the last point is included in the polyline. The default for closed is nil.

These functions capture their mutable inputs; the consequences of modifying those objects are unspecified.

**make-polygon**  
Arguments: point-seq

**make-polygon***  
Arguments: coord-seq
Summary: Returns an object of class standard-polygon consisting of the area contained in the boundary that is specified by the segments connecting each of the points in point-seq (or the points represented by the coordinate pairs in coord-seq).

These functions capture their mutable inputs; the consequences of modifying those objects are unspecified.
2.5.3.2 Accessors for CLIM Polygons and Polylines

The following generic functions comprise the polygon and polyline protocol. All classes that are subclasses of `polygon` or `polyline` must implement methods for them. Some of the functions take an argument `polygon-or-polyline`, which may be a polygon or a polyline.

`polygon-points` [Generic Function]
Arguments: `polygon-or-polyline`
Summary: Returns a sequence of points that specify the segments in `polygon-or-polyline`. This function returns objects that reveal CLIM’s internal state; do not modify those objects.

`map-over-polygon-coordinates` [Generic Function]
Arguments: `function polygon-or-polyline`
Summary: Applies `function` to all of the coordinates of the vertices of `polygon-or-polyline`. `function` is a function of two arguments, the x and y coordinates of the vertex; it has dynamic extent.

`map-over-polygon-segments` [Generic Function]
Arguments: `function polygon-or-polyline`
Summary: Applies `function` to the segments that compose `polygon-or-polyline`. `function` is a function of four arguments, the x and y coordinates of the start of the segment, and the x and y coordinates of the end of the segment; it has dynamic extent. When `map-over-polygon-segments` is called on a closed polyline, it will call `function` on the segment that connects the last point back to the first point.

`polyline-closed` [Generic Function]
Arguments: `polyline`
Summary: Returns `t` if the polyline `polyline` is closed; otherwise, it returns `nil`.

2.5.4 Lines in CLIM

A line is a special case of a polyline having only one segment. The functions for making and dealing with lines are the following:

`line` [Protocol Class]
Summary: The protocol class that corresponds to a mathematical line segment, that is, a polyline with only a single segment. This is a subclass of `polyline`. If you want to create a new class that behaves like a line, it should be a subclass of `line`. Subclasses of `line` must obey the line protocol.

**linep**  
*Function*  
Arguments: `object`  
Summary: Returns `t` if `object` is a line; otherwise, it returns `nil`.

**standard-line**  
*Class*  
Summary: An instantiable class that implements a line segment. This is a subclass of `line`. This is the class that `make-line` and `make-line*` instantiate. Members of this class are immutable.

**make-line**  
*Function*  
Arguments: `start-point end-point`  

**make-line**  
*Function*  
Arguments: `start-x start-y end-x end-y`  
Summary: Returns an object of class `standard-line` that connects the two points `start-point` and `end-point` (or the positions `(start-x, start-y)` and `(end-x, end-y)`). These functions capture their mutable inputs; the consequences of modifying those objects are unspecified.

The following generic functions comprise the line Application Programmer Interface. Only `line-start-point*` and `line-end-point*` are in the line protocol; that is, all classes that are subclasses of `line` must implement methods for `line-start-point*` and `line-end-point*`, but need not implement methods for `line-start-point` and `line-end-point`.

**line-start-point**  
*Generic Function*  
Arguments: `line`  

**line-end-point**  
*Generic Function*  
Arguments: `line`  
Summary: Returns the starting or ending point, respectively, of the line `line` as two real numbers representing the coordinates of the point.
line-start-point
Arguments: line

line-end-point
Arguments: line
Summary: Returns the starting or ending point of the line line, respectively.

CLIM will supply default methods for line-start-point and line-end-point on the protocol class line that are implemented by calling line-start-point* and line-end-point*.

2.5.5 Rectangles in CLIM

A rectangle is a special case of a four-sided polygon whose edges are parallel to the coordinate axes. A rectangle can be specified completely by four real numbers (min-x, min-y, max-x, max-y). They are not closed under affine transformations, although they are closed under rectilinear transformations. CLIM uses rectangles extensively for various purposes, particularly in optimizations.

The functions for creating and dealing with rectangles are the following:

rectangle
Summary: The protocol class that corresponds to a mathematical rectangle, that is, a rectangular polygons whose sides are parallel to the coordinate axes. This is a subclass of polygon. If you want to create a new class that behaves like a rectangle, it should be a subclass of rectangle. Subclasses of rectangle must obey the rectangle protocol.

rectanglep
Arguments: object
Summary: Returns t if object is a rectangle; otherwise, it returns nil.

standard-rectangle
Summary: An instantiable class that implements an axis-aligned rectangle. This is a subclass of rectangle. This is the class that make-rectangle and make-rectangle* instantiate. Members of this class are immutable.
**make-rectangle**

Arguments: `point1 point2`

**make-rectangle***

Arguments: `x1 y1 x2 y2`

Summary: Returns an object of class `standard-rectangle` whose edges are parallel to the coordinate axes. One corner is at the point `point1` (or the position `(x1, y1)`) and the opposite corner is at the point `point2` (or the position `(x2, y2)`). There are no ordering constraints among `point1` and `point2` (or `x1` and `x2`, and `y1` and `y2`).

This function captures its mutable inputs; the consequences of modifying those objects are unspecified.

The following generic functions comprise the rectangle Application Programmer Interface. Only `rectangle-edges*` is in the rectangle protocol; that is, all classes that are subclasses of `rectangle` must implement methods for `rectangle-edges*`, but need not implement methods for the remaining functions.

**rectangle-edges***

Arguments: `rectangle`

Summary: Returns the coordinates of the minimum `x` and `y` and maximum `x` and `y` of the rectangle `rectangle` as four values, `min-x`, `min-y`, `max-x`, and `max-y`.

**rectangle-min-point**

Arguments: `rectangle`

**rectangle-max-point**

Arguments: `rectangle`

Summary: Returns the min point and max point of the rectangle `rectangle`, respectively. The position of a rectangle is specified by its min point.

CLIM supplies default methods for `rectangle-min-point` and `rectangle-max-point` on the protocol class `rectangle` that are implemented by calling `rectangle-edges*`.

**rectangle-min-x**

Arguments: `rectangle`
rectangle-min-y [Generic Function]
Arguments: rectangle

rectangle-max-x [Generic Function]
Arguments: rectangle

rectangle-max-y [Generic Function]
Arguments: rectangle
Summary: Returns (respectively) the minimum x and y coordinate and maximum x and y coordinate of the rectangle rectangle.

CLIM supplies default methods for these four generic functions on the protocol class rectangle that are implemented by calling rectangle-edges*.

rectangle-width [Generic Function]
Arguments: rectangle

rectangle-height [Generic Function]
Arguments: rectangle

rectangle-size [Generic Function]
Arguments: rectangle
Summary: rectangle-width returns the width of the rectangle rectangle, which is the difference between its maximum and minimum x values. rectangle-height returns the height, which is the difference between its maximum and minimum y values. rectangle-size returns two values, the width and the height.

CLIM supplies default methods for these four generic functions on the protocol class rectangle that are implemented by calling rectangle-edges*.

2.5.6 Ellipses and Elliptical Arcs in CLIM

An ellipse is an area that is the outline and interior of an ellipse. Circles are special cases of ellipses.

An elliptical arc is a path consisting of all or a portion of the outline of an ellipse. Circular arcs are special cases of elliptical arcs.
An ellipse is specified in a manner that is easy to transform, and treats all ellipses on an equal basis. An ellipse is specified by its center point and two vectors that describe a bounding parallelogram of the ellipse. The bounding parallelogram is made by adding and subtracting the vectors from the center point in the following manner:

The special case of an ellipse with its axes aligned with the coordinate axes can be obtained by setting $dx_2$ and $dy_1$ to 0, or setting $dx_1$ and $dy_2$ to 0.

Note that several different parallelograms specify the same ellipse, as shown here:

<table>
<thead>
<tr>
<th></th>
<th>$x$ coordinate</th>
<th>$y$ coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center of Ellipse</td>
<td>$x_c$</td>
<td>$y_c$</td>
</tr>
<tr>
<td>Vectors</td>
<td>$dx_1$</td>
<td>$dy_1$</td>
</tr>
<tr>
<td></td>
<td>$dx_2$</td>
<td>$dy_2$</td>
</tr>
<tr>
<td>Corners of Parallelogram</td>
<td>$x_c + dx_1 + dx_2$</td>
<td>$y_c + dy_1 + dy_2$</td>
</tr>
<tr>
<td></td>
<td>$x_c + dx_1 - dx_2$</td>
<td>$y_c + dy_1 - dy_2$</td>
</tr>
<tr>
<td></td>
<td>$x_c - dx_1 - dx_2$</td>
<td>$y_c - dy_1 - dy_2$</td>
</tr>
<tr>
<td></td>
<td>$x_c - dx_1 + dx_2$</td>
<td>$y_c - dy_1 + dy_2$</td>
</tr>
</tbody>
</table>

**Table 2. Bounding Parallelogram of an Ellipse**

The special case of an ellipse with its axes aligned with the coordinate axes can be obtained by setting $dx_2$ and $dy_1$ to 0, or setting $dx_1$ and $dy_2$ to 0.

Note that several different parallelograms specify the same ellipse, as shown here:

**Figure 12. Ellipses Specified by Parallelograms**

One parallelogram is bound to be a rectangle—the vectors will be perpendicular and correspond to the semi-axes of the ellipse.
The following classes and functions are used to represent and operate on ellipses and elliptical arcs.

**ellipse**

[Protocol Class]

Summary: The protocol class that corresponds to a mathematical ellipse. This is a subclass of `area`. If you want to create a new class that behaves like an ellipse, it should be a subclass of `ellipse`. Subclasses of `ellipse` must obey the ellipse protocol.

**ellipses**

[Function]

Arguments: `object`

Summary: Returns `t` if `object` is an ellipse; otherwise, it returns `nil`.

**standard-ellipse**

[Class]

Summary: An instantiable class that implements an ellipse. This is a subclass of `ellipse`. This is the class that `make-ellipse` and `make-ellipse*` instantiate. Members of this class are immutable.

**elliptical-arc**

[Protocol Class]

Summary: The protocol class that corresponds to a mathematical elliptical arc. This is a subclass of `path`. If you want to create a new class that behaves like an elliptical arc, it should be a subclass of `elliptical-arc`. Subclasses of `elliptical-arc` must obey the elliptical arc protocol.

**elliptical-arc-p**

[Function]

Arguments: `object`

Summary: Returns `t` if `object` is an elliptical arc; otherwise, it returns `nil`.

**standard-elliptical-arc**

[Class]

Summary: An instantiable class that implements an elliptical arc. This is a subclass of `elliptical-arc`. This is the class that `make-elliptical-arc` and `make-elliptical-arc*` instantiate. Members of this class are immutable.
2.5.6.1 Constructor Functions for Ellipses and Elliptical Arcs in CLIM

**make-ellipse**

Arguments: `center-point radius-1-dx radius-1-dy radius-2-dx radius-2-dy &key start-angle end-angle`

**make-ellipse**

Arguments: `center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy &key start-angle end-angle`

Summary: Returns an object of class `standard-ellipse`. The center of the ellipse is at the point `center-point` (or the position `(center-x, center-y)`).

Two vectors, `(radius-1-dx, radius-1-dy)` and `(radius-2-dx, radius-2-dy)` specify the bounding parallelogram of the ellipse as explained previously. All of the radii are real numbers. If the two vectors are collinear, the ellipse is not well-defined and the `ellipse-not-well-defined` error will be signaled. The special case of an ellipse with its axes aligned with the coordinate axes can be obtained by setting both `radius-1-dy` and `radius-2-dx` to 0.

If `start-angle` or `end-angle` are supplied, the ellipse is the “pie slice” area swept out by a line from the center of the ellipse to a point on the boundary as the boundary point moves from the angle `start-angle` to `end-angle`. Angles are measured counter-clockwise with respect to the positive `x` axis. If `end-angle` is supplied, the default for `start-angle` is 0; if `start-angle` is supplied, the default for `end-angle` is `2π`; if neither is supplied, then the region is a full ellipse and the angles are meaningless.

This function captures its mutable inputs; the consequences of modifying those objects are unspecified.

**make-elliptical-arc**

Arguments: `center-point radius-1-dx radius-1-dy radius-2-dx radius-2-dy &key start-angle end-angle`

**make-elliptical-arc**

Arguments: `center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy &key start-angle end-angle`

Summary: Returns an object of class `standard-elliptical-arc`. The center of the ellipse is at the point `center-point` (or the position `(center-x, center-y)`).
Two vectors, \((\text{radius-1-dx}, \text{radius-1-dy})\) and \((\text{radius-2-dx}, \text{radius-2-dy})\), specify the bounding parallelogram of the ellipse as explained previously. All of the radii are real numbers. If the two vectors are collinear, the ellipse is not well-defined and the \textit{ellipse-not-well-defined} error will be signaled. The special case of an elliptical arc with its axes aligned with the coordinate axes can be obtained by setting both \text{radius-1-dy} and \text{radius-2-dx} to 0.

If \text{start-angle} and \text{start-angle} are supplied, the arc is swept from \text{start-angle} to \text{end-angle}. Angles are measured counter-clockwise with respect to the positive \(x\) axis. If \text{end-angle} is supplied, the default for \text{start-angle} is 0; if \text{start-angle} is supplied, the default for \text{end-angle} is \(2\pi\); if neither is supplied, then the region is a closed elliptical path and the angles are meaningless.

This function captures its mutable inputs; the consequences of modifying those objects are unspecified.

### 2.5.6.2 Accessors for CLIM Elliptical Objects

The following functions apply to both ellipses and elliptical arcs. In all cases, the name \textit{elliptical-object} means that the argument may be an ellipse or an elliptical arc. These generic functions comprise the ellipse protocol. All classes that are subclasses of either \textit{ellipse} or \textit{elliptical-arc} must implement methods for these functions.

**ellipse-center-point**

[Generic Function]

Arguments: \textit{elliptical-object}

Summary: Returns the center point of \textit{elliptical-object} as two values representing the coordinate pair.

**ellipse-center-point**

[Generic Function]

Arguments: \textit{elliptical-object}

Summary: Returns the center point of \textit{elliptical-object}.

\textbf{ellipse-center-point} is part of the ellipse Application Programmer Interface, but not part of the ellipse protocol. CLIM will supply default methods for \textbf{ellipse-center-point} on the protocol classes \textit{ellipse} and \textit{elliptical-arc} that are implemented by calling \textbf{ellipse-center-point*}.

**ellipse-radii**

[Generic Function]

Arguments: \textit{elliptical-object}
Summary: Returns four values corresponding to the two radius vectors of \textit{elliptical-arc}.
These values may be canonicalized in some way, and so may not be the same as the values passed to the constructor function.

\textbf{ellipse-start-angle} \hspace{1cm} \textbf{[Generic Function]}

\textbf{Arguments:} \textit{elliptical-object}

\textbf{Summary:} Returns the start angle of \textit{elliptical-object}. If \textit{elliptical-object} is a full ellipse or closed path, then \textit{ellipse-start-angle} will return \textit{nil}; otherwise the value will be a number greater than or equal to zero, and less than \(2\pi\).

\textbf{ellipse-end-angle} \hspace{1cm} \textbf{[Generic Function]}

\textbf{Arguments:} \textit{elliptical-object}

\textbf{Summary:} Returns the end angle of \textit{elliptical-object}. If \textit{elliptical-object} is a full ellipse or closed path, then \textit{ellipse-end-angle} will return \textit{nil}; otherwise the value will be a number greater than zero, and less than or equal to \(2\pi\).

\subsection*{2.5.7 \hspace{0.5cm} Bounding Rectangles}

Every bounded region in CLIM has a \textit{derived bounding rectangle}, which is the smallest rectangle that contains every point in the region and which may contain additional points as well. Unbounded regions do not have any bounding rectangle. For example, all windows and output records have bounding rectangles whose coordinates are relative to the bounding rectangle of the parent of the window or output record.

The coordinate system in which the bounding rectangle is maintained depends on the context. For example, the coordinates of the bounding rectangle of a sheet are expressed in the sheet’s parent’s coordinate system. For output records, the coordinates of the bounding rectangle are maintained in the coordinate system of the stream with which the output record is associated.

Note that the bounding rectangle of a transformed region is not in general the same as the result of transforming the bounding rectangle of a region, as shown in Figure 13. For transformations that satisfy \textit{rectilinear-transformation-p}, the following equality holds. For all other transformations, it does not hold.
CLIM uses bounding rectangles for a variety of purposes. For example, repainting of windows is driven from the bounding rectangle of the window’s viewport, intersected with a “damage” region. The formatting engines used by formatting-table and formatting-graph operate on the bounding rectangles of the output records in the output. Bounding rectangles are also used internally by CLIM to achieve greater efficiency. For instance, when performing hit detection to see if the pointer is within the region of an output record, CLIM first checks to see if the pointer is within the bounding rectangle of the output record.

Note that the bounding rectangle for an output record may have a different size depending on the medium on which the output record is rendered. Consider the case of rendering text on different output devices; the font chosen for a particular text style may vary considerably in size from one device to another.

**bounding-rectangle** [Protocol Class]

Summary: The protocol class that represents a bounding rectangle. If you want to create a new class that behaves like a bounding rectangle, it should be a subclass of **bounding-rectangle**. Subclasses of **bounding-rectangle** must obey the bounding rectangle protocol.
Note that bounding rectangles are not a subclass of rectangle, nor even a subclass of region. This is because, in general, bounding rectangles do not obey the region protocols. However, all bounded regions and sheets that obey the bounding rectangle protocol are subclasses of bounding-rectangle.

Bounding rectangles are immutable, but since they reflect the live state of such mutable objects as sheets and output records, bounding rectangles are volatile. Therefore, programmers must not depend on the bounding rectangle associated with a mutable object remaining constant.

**bounding-rectangle-p**

*Function*

**Arguments:** object

**Summary:** Returns t if object is a bounding rectangle (that is, supports the bounding rectangle protocol); otherwise, it returns nil.

**standard-bounding-rectangle**

*Class*

**Summary:** An instantiable class that implements a bounding rectangle. This is a subclass of both bounding-rectangle and rectangle; that is, standard bounding rectangles obey the rectangle protocol.

**make-bounding-rectangle** returns an object of this class.

The representation of bounding rectangles in CLIM is chosen to be efficient. CLIM represents such rectangles by storing the coordinates of two opposing corners of the rectangle, namely, the “min point” and the “max point.” Because this representation is not sufficient to represent the result of arbitrary transformations of arbitrary rectangles, CLIM returns a polygon as the result of such a transformation. (The most general class of transformations that is guaranteed to always turn a rectangle into another rectangle is the class of transformations that satisfy rectilinear-transformation-p.)

**make-bounding-rectangle**

*Function*

**Arguments:** x1 y1 x2 y2

**Summary:** Returns an object of the class standard-bounding-rectangle with the edges specified by x1, y1, x2, and y2, which must be real numbers.

x1, y1, x2, and y2 are “canonicalized” in the following way. The min point of the rectangle has an x coordinate that is the smaller of x1 and x2 and a y coordinate that is the smaller of y1 and y2. The max point of the rectangle has an x coordinate that is the larger of x1 and x2 and a y coordinate that is the larger of y1 and y2. (Therefore, in a right-handed coordinate system the canonicalized values of...
2.5.7.1 The Bounding Rectangle Protocol

The following generic functions comprise the bounding rectangle protocol. All classes that participate in this protocol (including all subclasses of region that are bounded regions) implement a method for bounding-rectangle*.

These functions take the argument region, which must be either a bounded region (such as a line or an ellipse) or some other object that obeys the bounding rectangle protocol, such as a sheet or an output record.

bounding-rectangle*
[Generic Function]
Arguments: region
Summary: Returns the bounding rectangle of region as four real numbers specifying the x and y coordinates of the min point and the x and y coordinates of the max point of the rectangle.

The four returned values min-x, min-y, max-x, and max-y satisfy the inequalities:
min-x ≤ max-x
min-y ≤ max-y

bounding-rectangle
[Generic Function]
Arguments: region
Summary: Returns the bounding rectangle of region as an object that is a subclass of rectangle (described in Subsection 2.5.5, “Rectangles in CLIM”). Since bounding rectangles are volatile, programmers should not depend on the object returned by bounding-rectangle remaining constant.

bounding-rectangle is part of the bounding rectangle Application Programmer Interface, but is not part of the bounding rectangle protocol. CLIM supplies a default method for bounding-rectangle on the protocol class bounding-rectangle*.

x1, y1, x2, and y2 correspond to the left, top, right, and bottom edges of the rectangle, respectively.)

This function returns fresh objects that may be modified.
2.5.7.2  Bounding Rectangle Convenience Functions

The following functions are part of the bounding rectangle Application Programmer Interface, but are not part of the bounding rectangle protocol. They are provided as a convenience to programmers who wish to specialize classes that participate in the bounding rectangle protocol, but they will not complicate the task of those programmers who define their own types (such as sheet classes) that participate in this protocol.

CLIM supplies default methods for all of these generic functions on the protocol class `bounding-rectangle` that are implemented by calling `bounding-rectangle*`.

**with-bounding-rectangle**

[Macro]

Arguments: `(min-x min-y max-x max-y) region &body body`

Summary: Binds `min-x`, `min-y`, `max-x`, and `max-y` to the edges of the bounding rectangle of `region`, and then executes `body` in that context. The argument `region` must be either a bounded region (such as a line or an ellipse) or some other object that obeys the bounding rectangle protocol, such as a sheet or an output record.

The arguments `min-x`, `min-y`, `max-x`, and `max-y` are not evaluated. `body` may have zero or more declarations as its first forms.

`with-bounding-rectangle*` calls `bounding-rectangle*`.

**bounding-rectangle-position**

[Generic Function]

Arguments: `region`

Summary: Returns the position of the bounding rectangle of `region`. The position of a bounding rectangle is specified by its min point.

**bounding-rectangle-min-x**

[Generic Function]

Arguments: `region`

**bounding-rectangle-min-y**

[Generic Function]

Arguments: `region`

**bounding-rectangle-max-x**

[Generic Function]

Arguments: `region`
bounding-rectangle-max-y  [Generic Function]
Arguments: region
Summary: Returns (respectively) the x and y coordinates of the min point and the x and y coordinates of the max point of the bounding rectangle of region. The argument region must be either a bounded region or some other object that obeys the bounding rectangle protocol.

bounding-rectangle-width  [Generic Function]
Arguments: region

bounding-rectangle-height  [Generic Function]
Arguments: region

bounding-rectangle-size  [Generic Function]
Arguments: region
Summary: Returns the width, height, or size (as two values, the width and height) of the bounding rectangle of region, respectively. region must be either a bounded region or some other object that obeys the bounding rectangle protocol.

The width of a bounding rectangle is the difference between its maximum x coordinate and its minimum x coordinate. The height is the difference between the maximum y coordinate and its minimum y coordinate.
3.1  CLIM Mediums

Drawing in CLIM is done through a medium. A medium can be thought of as an object that knows how to draw on a specific device. For example, a medium translates a CLIM `draw-rectangle` call into the appropriate draw-rectangle call to the underlying graphics host. Mediums also keep track of default drawing options, such as a drawing plane, foreground and background inks, a transformation, a clipping region, a line style, and a text style. These default values are used when these function-call parameters are left otherwise unspecified. For related information, refer to Subsection 2.1.4, “Mediums, Sheets, and Streams.”

The drawing environment is dynamic. The CLIM facilities for affecting the drawing environment do so within their dynamic extent. For example, any drawing done by the user function `draw-stuff` (as well as any drawing performed by its callees) will be affected by the scaling transformation:

```
(clim:with-scaling (medium 2 1) (draw-stuff medium))
```

The medium has components that are used to keep track of the drawing environment. The drawing environment is controlled through the use of drawing options that can be provided as keyword arguments to all of the drawing functions.

Each CLIM medium contains components that correspond to the drawing options. These components provide the default values for the drawing options. When drawing functions are called and some options are unspecified, the options default to the values maintained by the medium.

CLIM provides accessors that enable you to read and write the values of these components. Also, these components are temporarily bound within a dynamic context by using `with-drawing-options`, `with-text-style`, and related forms. Using `setf` on a component while it is temporarily bound takes effect immediately but is undone when the dynamic context is exited.

The following functions read and write components of a medium related to drawing options. While these functions are defined for mediums, they can also be called on sheets that support the sheet output protocol and on streams that output to such sheets. All classes that support the medium protocol implement methods for these generic functions. Often, a sheet class that supports the output protocol will implement a “trampoline” method that passes the operation directly on to `sheet-medium` of the sheet.
**medium-foreground**

Arguments: *medium*  

**medium-background**

Arguments: *medium*  

Summary: Returns the foreground and background inks (which are designs) for the medium *medium*, respectively. The foreground ink is the default ink used when drawing. The background ink is the ink used when erasing. See Chapter 5, “Drawing in Color,” for a more complete description of designs.

Any indirect inks are resolved against the foreground and background at the time a design is rendered.

(setf medium-foreground)

Arguments: *ink* *medium*  

(setf medium-background)

Arguments: *ink* *medium*  

Summary: Sets the foreground and background ink, respectively, for the medium *medium* to *ink*. You may not set medium-foreground or medium-background to an indirect ink.

Changing the foreground or background of a sheet that supports output recording causes the contents of the stream’s viewport to be erased and redrawn using the new foreground and background.

**medium-ink**

Arguments: *medium*  

Summary: The current drawing ink for the medium *medium*, which can be any design. The drawing functions draw with the color and pattern that this specifies. See Chapter 5, “Drawing in Color,” for a more complete description of inks. The :ink drawing option temporarily changes the value of medium-ink.

(setf medium-ink)

Arguments: *ink* *medium*  

Summary: Sets the current drawing ink for the medium *medium* to *ink*. *ink* is as for medium-foreground, and may be an indirect ink as well.
medium-transformation

Arguments: medium
Summary: The current user transformation for the medium medium. This is used to transform the coordinates supplied as arguments to drawing functions to the coordinate system of the drawing plane. See Section 3.5, “The Transformations Used by CLIM,” for a complete description of transformations. The :transformation drawing option temporarily changes the value of medium-transformation.

(setf medium-transformation)

Arguments: transformation medium
Summary: Sets the current user transformation for the medium medium to the transformation transformation.

medium-clipping-region

Arguments: medium
Summary: The current clipping region for the medium medium. The drawing functions do not affect the drawing plane outside this region. The :clipping-region drawing option temporarily changes the value of medium-clipping-region.

The clipping region is expressed in user coordinates.

(setf medium-clipping-region)

Arguments: region medium
Summary: Sets the current clipping region for the medium medium to region. region must be a subclass of area.

medium-line-style

Arguments: medium
Summary: The current line style for the medium medium. The line and arc drawing functions render according to this line style. See Section 3.3, “CLIM Line Styles,” for a complete description of line styles. The :line-style drawing option temporarily changes the value of medium-line-style.

(setf medium-line-style)

Arguments: line-style medium
Summary: Sets the current line style for the medium medium to the line style line-style.
medium-default-text-style

Arguments: medium
Summary: The default text style for the medium medium. medium-default-text-style will return a fully specified text style, unlike medium-text-style, which may return a text style with null components. Any text styles that are not fully specified by the time they are used for rendering are merged against medium-default-text-style using merge-text-styles.

The default value for medium-default-text-style for any medium is *default-text-style*.

See Chapter 4 for a complete description of text styles.

(setf medium-default-text-style)

Arguments: text-style medium
Summary: Sets the default text style for the medium medium to the text style text-style. text-style must be a fully specified text style.

medium-text-style

Arguments: medium
Summary: The current text style for the medium medium. The text drawing functions, including ordinary stream output, render text as directed by this text style merged against the default text style. This controls both graphical text (such as that drawn by draw-text*) and stream text (such as that written by write-string). See Chapter 4 for a complete description of text styles. The :text-style drawing option temporarily changes the value of medium-text-style.

(setf medium-text-style)

Arguments: text-style medium
Summary: Sets the current text style for the medium medium to the text style text-style. text-style need not be a fully merged text style.

medium-current-text-style

Arguments: medium
Summary: The current, fully merged text style for the medium medium. This is the text style that will be used when drawing text output, and is the result of merging medium-text-style against medium-default-text-style.
3.2 Using CLIM Drawing Options

Drawing options control various aspects of the drawing process. You can supply drawing options in a number of ways:

- The medium (the destination for graphic output) itself has default drawing options. If a drawing option is not supplied elsewhere, the medium supplies the value. See the preceding section, “Components of CLIM Mediums.”
- You can use `with-drawing-options` to bind the drawing options of the medium temporarily. In many cases, it is convenient to use `with-drawing-options` to surround several calls to drawing functions, each using the same options.
- You can supply the drawing options as keyword arguments to the drawing functions. These override the drawing options specified by `with-drawing-options`.

In some cases, it is important to distinguish between drawing options and suboptions. Both text and lines have an option that controls the complete specification of the text and line style, and there are suboptions that can affect one aspect of the text or line style. For example, the value of the `:text-style` option is a text style object, which describes a complete text style consisting of family, face, and size. There are also suboptions called `:text-family`, `:text-face`, and `:text-size`. Each suboption specifies a single aspect of the text style, while the option specifies the entire text style. Line styles are analogous to text styles; there is a `:line-style` option and some suboptions.

In a given call to `with-drawing-options` or a drawing function, you would normally supply either the `:text-style` option or a text style suboption (or more than one suboption), but not both. If you do supply both, then the text style comes from the result of merging the suboptions with the `:text-style` option, and then merging that with the prevailing text style.

`with-drawing-options` [Macro]

Arguments: `(medium &rest drawing-options) &body body`

Summary: Binds the state of the medium designated by `medium` to correspond to the supplied drawing options, and executes the body with the new drawing options specified by `drawing-options` in effect. Each option causes binding of the corresponding component of the medium for the dynamic extent of the body. The drawing functions effectively do a `with-drawing-options` when drawing option arguments are supplied to them.
medium can be a medium, a sheet that supports the sheet output protocol, or a stream that outputs to such a sheet. The medium argument is not evaluated, and must be a symbol that is bound to a sheet or medium. If medium is t, *standard-output* is used. body may have zero or more declarations as its first forms.

with-drawing-options expands into a call to invoke-with-drawing-options, supplying a function that executes body as the continuation argument to invoke-with-drawing-options.

invoke-with-drawing-options [Generic Function]
Arguments: medium continuation &rest drawing-options
Summary: Binds the state of the medium medium to correspond to the supplied drawing options, and then calls the function continuation with the new drawing options in effect. continuation is a function of one argument, the medium; it has dynamic extent. drawing-options is a list of alternating keyword-value pairs, and must have even length. Each option in drawing-options causes binding of the corresponding component of the medium for the dynamic extent of the body.

medium can be a medium, a sheet that supports the sheet output protocol, or a stream that outputs to such a sheet. All classes that obey the medium protocol implement a method for invoke-with-drawing-options.

drawing-options can be any of the following, plus any of the suboptions for line and text styles. The default value specified for a drawing option is the value to which the corresponding component of a medium is normally initialized.

### 3.2.1 Set of CLIM Drawing Options

Drawing options can be any of the following, plus any of the line-style or text-style suboptions.

:ink [Option]
Summary: The drawing functions draw with the color and pattern that this ink specifies. The default value is +foreground-ink+. See Chapter 5, “Drawing in Color,” for a complete description of inks.

The :ink drawing option temporarily changes the value of (medium-ink medium) to ink, replacing (not combining) the previous ink.
:transformation [Option]
Summary: This transforms the coordinates used as arguments to drawing functions to the coordinate system of the drawing plane. The default value is +identity-transformation+. See Section 3.5, “The Transformations Used by CLIM,” for a complete description of transformations.

The :transformation xform drawing option temporarily changes the value of (medium-transformation medium) to:

(compose-transformations (medium-transformation medium) xform)

:clipping-region [Option]
Summary: The drawing functions do not affect the drawing plane outside this region, which must be an area. Rendering is clipped both by this clipping region and by other clipping regions associated with the mapping from the target drawing plane to the viewport that displays a portion of the drawing plane. The default is +everywhere+, or in other words, no clipping occurs in the drawing plane, only in the viewport.

The :clipping-region region drawing option temporarily changes the value of (medium-clipping-region medium) to:

(region-intersection
(transform-region
  (medium-transformation medium) region)
(medium-clipping-region medium))

If both a clipping region and a transformation are supplied in the same set of drawing options, the clipping region argument is transformed by the newly composed transformation before calling region-intersection.

;line-style [Option]
Summary: The line- and arc-drawing functions render according to this line style. The line style suboptions and default are defined in Section 3.3, “CLIM Line Styles.”

The :line-style ls drawing option temporarily changes the value of (medium-line-style medium) to ls, replacing the previous line style; the new and old line styles are not combined in any way.

If line-style suboptions are supplied, they temporarily change the value of (medium-line-style medium) to a line style constructed from the specified suboptions. Components not specified by suboptions default from the :line-style drawing option, if it is supplied, or else from the previous value of
(medium-line-style medium). That is, if both the :line-style option and line-style suboptions are supplied, the suboptions take precedence over the components of the :line-style option.

:text-style

Summary: The text drawing functions, including ordinary stream output, render text as directed by this text style merged against the default text style. The default value has all null components. See Chapter 4, “Text Styles,” for a complete description of text styles, including the text style suboptions.

The :text-style ts drawing option temporarily changes the value of (medium-text-style medium) to:

(merge-text-styles ts (medium-text-style medium))

If text-style suboptions are supplied, they temporarily change the value of (medium-text-style medium) to a text style constructed from the specified suboptions, merged with the :text-style drawing option if it is specified, and then merged with the previous value of (medium-text-style medium). That is, if both the :text-style option and text-style suboptions are supplied, the suboptions take precedence over the components of the :text-style option.

3.2.2 Using the :filled Option

Certain drawing functions can draw either an area or the outline of that area. This is controlled by the :filled keyword argument to these functions. If the value is t (the default), then the function paints the entire area. If the value is nil, then the function outlines the area under the control of the line-style drawing option.

The :filled keyword argument is not a drawing option and cannot be specified to with-drawing-options.

The following functions have a :filled keyword argument:

- draw-circle
- draw-circle*
- draw-ellipse
- draw-ellipse*
- draw-polygon
A line is a one-dimensional object. In order to be visible, however, the rendering of a line must occupy some non-zero area on the display hardware. CLIM uses a line style object to represent the advice supplied to the rendering substrate on how to perform the rendering.

It is often useful to create a line style object that represents a style you wish to use frequently, rather than continually specifying the corresponding line style suboptions.

**line-style** [Protocol Class]
Summary: The protocol class for line styles. If you want to create a new class that behaves like a line style, it should be a subclass of `line-style`. Subclasses of `line-style` must obey the line style protocol.

**line-style-p** [Function]
Arguments: `object`
Summary: Returns `t` if object is a line style; otherwise, it returns `nil`.

**standard-line-style** [Class]
Summary: An instantiable class that implements line styles. A subclass of `line-style`, this is the class that `make-line-style` instantiates. Members of this class are immutable.

**make-line-style** [Function]
Arguments: `&key unit thickness joint-shape cap-shape dashes`
Summary: Returns an object of class `standard-line-style` with the supplied characteristics. The arguments and their default values are described in Subsection 3.3, “CLIM Line Styles.”

Each of the following suboptions has a corresponding reader that can be used to extract a particular component from a line style. The following generic functions comprise the line style protocol; all subclasses of `line-style` implement methods for these generic functions.
:line-unit

line-style-unit
[Generic Function]
Arguments: line-style
Summary: Gives the unit used for measuring line thickness and dash pattern length for the line style. Possible values are as follows:

- **:normal**—thicknesses and lengths are given in a relative measure in terms of the usual or “normal” line thickness, which is the thickness of the “comfortably visible thin line,” a property of the underlying rendering substrate. (This is the default value.)
- **:point**—thicknesses and lengths are given in an absolute measure in terms of printer’s points (approximately 1/72 of an inch). This measure was chosen so that CLIM implementors who interface CLIM to an underlying rendering engine (the window system) may legitimately choose to make it render as 1 pixel on current (1992) display devices.
- **:coordinate**—the same units should be used for line thickness as are used for coordinates. In this case, the line thickness is scaled by the medium’s current transformation, whereas :normal and :point do not scale the line thickness.

:line-thickness

line-style-thickness
[Generic Function]
Arguments: line-style
Summary: The thickness, in the units indicated by line-style-unit, of the lines or arcs drawn by a drawing function. The thickness must be a real number. The default is 1, which, when combined with the default unit of :normal, means that the default line drawn is the “comfortably visible thin line.”

:line-joint-shape

line-style-joint-shape
[Generic Function]
Arguments: line-style
Summary: Specifies the shape of joints between segments of unfilled figures. The possible shapes are :miter, :bevel, :round, and :none; the default is :miter. Note that the joint shape is implemented by the host window system, so not all platforms will necessarily fully support it.
Figure 14. Line Joint Shapes

:line-cap-shape

[Option]

line-style-cap-shape

[Generic Function]

Arguments: line-style

Summary: Specifies the shape for the ends of lines and arcs drawn by a drawing function, one of :butt, :square, :round, or :no-end-point; the default is :butt. Note that the cap shape is implemented by the host window system, so not all platforms will necessarily fully support it.

Figure 15. Line Cap Shapes

:line-dashes

[Option]

line-style-dashes

[Generic Function]

Arguments: line-style

<Bold>76CLIM User Guide
Summary: Controls whether lines or arcs are drawn as dashed figures, and if so, what the dashing pattern is. Possible values are:

- **nil**—lines are drawn solid, with no dashing. This is the default.
- **t**—lines are drawn dashed, with a dash pattern that is unspecified and may vary with the rendering engine. This allows the underlying display substrate to provide a default dashed line for the programmer whose only requirement is to draw a line that is visually distinguishable from the default solid line.
- A sequence—specifies a sequence, usually a vector, controlling the dash pattern of a drawing function. It is an error if the sequence does not contain an even number of elements. The elements of the sequence are lengths (as real numbers) of individual components of the dashed line or arc. The odd elements specify the length of inked components; the even elements specify the gaps. All lengths are expressed in the units described by `line-style-unit`.

### make-contrasting-dash-patterns

[Function]

**Arguments:** `n` &optional `k`

**Summary:** If `k` is not supplied, this returns a vector of `n` dash patterns with recognizably different appearance. Elements of the vector are guaranteed to be acceptable values for `dashes`, and do not include `nil`, but their class is not otherwise specified. The vector is a fresh object that may be modified.

If `k` is supplied, it must be an integer between 0 and `n`–1 (inclusive), in which case `make-contrasting-dash-patterns` returns the `k`th dash-pattern rather than returning a vector of dash-patterns.

CLIM has at least 8 different contrasting dash patterns. If `n` is greater than 8, `make-contrasting-dash-patterns` signals an error.

### contrasting-dash-pattern-limit

[Generic Function]

**Arguments:** `port`

**Summary:** Returns the number of contrasting dash patterns that can be rendered on any medium on the port `port`. It is at least 8. All classes that obey the port protocol implement a method for this generic function.
3.4 Transformations in CLIM

One of the features of CLIM’s graphical capabilities is the use of coordinate system transformations. By using transformations, you can often write simpler graphics code because you can choose a coordinate system in which to express the graphics that simplifies the description of the drawing.

A *transformation* is an object that describes how one coordinate system is related to another. A graphic function performs its drawing in the current coordinate system of the stream. A new coordinate system is defined by describing its relationship to the old one (the transformation). The drawing can now take place in the new coordinate system. The basic concept of graphic transformations is illustrated in Figure 16.

For example, you might define the coordinates of a five-pointed star and a function to draw it.

```lisp
(defvar *star* '(0 3 2 -3 -3 1/2 3 1/2 -2 -3))

(defun draw-star (stream)
  (clim:draw-polygon* stream *star* :closed t :filled nil))
```

![Figure 16. Graphic Transformation](image)
Without any transformation, the function draws a small star centered around the origin. By applying a transformation, the same function can be used to draw a star of any size, anywhere. For example:

```
(clim:with-room-for-graphics (stream)
 (clim:with-translation (stream 100 100)
  (clim:with-scaling (stream 10)
   (draw-star stream)))
  (clim:with-translation (stream 240 110)
   (clim:with-rotation (stream -0.5)
    (clim:with-scaling (stream 12 8)
     (draw-star stream)))))
```

will draw a picture somewhat like Figure 16 on stream.

### 3.5 The Transformations Used by CLIM

The type of transformations that CLIM uses are called affine transformations. An affine transformation is a transformation that preserves straight lines. In other words, if you take a number of points that fall on a straight line and apply an affine transformation to their coordinates, the transformed coordinates will fall on a straight line in the new coordinate system. Affine transformations include translations, scalings, rotations, and reflections.

A **translation** is a transformation that preserves the length, angle, and orientation of all geometric entities.

A **rotation** is a transformation that preserves the length and angles of all geometric entities. Rotations also preserve one point and the distance of all entities from that point. You can think of that point as the “center of rotation”; it is the point around which everything rotates.

There is no single definition of a scaling transformation. Transformations that preserve all angles and multiply all lengths by the same factor (preserving the “shape” of all entities) are certainly scaling transformations. However, scaling is also used to refer to transformations that scale distances in the \(x\) direction by one amount and distances in the \(y\) direction by another amount.

A **reflection** is a transformation that preserves lengths and magnitudes of angles but changes the sign (or “handedness”) of angles. If you think of the drawing plane on a transparent sheet of paper, a reflection is a transformation that “turns the paper over.”
If we transform from one coordinate system to another, then from the second to a third coordinate system, we can regard the resulting transformation as a single transformation resulting from composing the two component transformations. It is an important and useful property of affine transformations that they are closed under composition.

Note that composition is not commutative; in general, the result of applying transformation A and then applying transformation B is not the same as applying B first, then A.

Any arbitrary transformation can be built up by composing a number of simpler transformations, but that same transformation can often be constructed by a different composition of different transformations.

Transforming a region applies a coordinate transformation to that region, thus moving its position on the drawing plane, rotating it, or scaling it. Note that this creates a new region, but it does not affect the region argument.

The user interface to transformations is the :transformation option to the drawing functions. Users can create transformations with constructors. See Subsection 3.5.1, “CLIM Transformation Constructors”. The other operators documented in this section are used by CLIM itself, and are not often needed by users.

### 3.5.1 CLIM Transformation Constructors

The following functions create transformation objects that can be used, for instance, in a call to compose-transformations. The transformation constructors do not capture any of their inputs. The constructors all create objects that are subclasses of transformation.

#### make-translation-transformation

[Function]

Arguments: translation-x translation-y

Summary: A translation is a transformation that preserves the length, angle, and orientation of all geometric entities.

`make-translation-transformation` returns a transformation that translates all points by `translation-x` in the x direction and `translation-y` in the y direction. `translation-x` and `translation-y` must be real numbers.

#### make-rotation-transformation

[Function]

Arguments: angle &optional origin

<Bold>80CLIM User Guide
**make-rotation-transformation**

*Function*

**Arguments:** `angle` &optional `origin-x` `origin-y`

**Summary:** A rotation is a transformation that preserves the length and angles of all geometric entities. Rotations also preserve one point (the origin) and the distance of all entities from that point.

`make-rotation-transformation` returns a transformation that rotates all points by `angle` (which is a real number indicating an angle in radians) around the point `origin`. If `origin` is supplied it must be a point; if not supplied, it defaults to (0, 0). `origin-x` and `origin-y` must be real numbers.

**make-scaling-transformation**

*Function*

**Arguments:** `scale-x` `scale-y` &optional `origin`

**Summary:** As discussed previously, there is no single definition of a scaling transformation.

`make-scaling-transformation` returns a transformation that multiplies the `x`-coordinate distance of every point from `origin` by `scale-x` and the `y`-coordinate distance of every point from `origin` by `scale-y`. `scale-x` and `scale-y` must be real numbers. If `origin` is supplied it must be a point; if not supplied, it defaults to (0, 0). `origin-x` and `origin-y` must be real numbers.

**make-reflection-transformation**

*Function*

**Arguments:** `point1` `point2`

**Summary:** A reflection is a transformation that preserves lengths and magnitudes of angles, but changes the sign (or “handedness”) of angles. If you think of the drawing plane on a transparent sheet of paper, a reflection is a transformation that “turns the paper over.”

`make-reflection-transformation` returns a transformation that reflects every point through the line passing through the points `point1` and `point2` (or through the positions `(x1, y1)` and `(x2, y2)` in the case of the spread version).
**make-transformation**  
*function*  
**Arguments:** \( m_{xx} \ m_{xy} \ m_{yx} \ m_{yy} \ t_x \ t_y \)  
**Summary:** Returns a general transformation whose effect is:

\[
\begin{align*}
x' &= m_{xx}x + m_{xy}y + t_x \\
y' &= m_{yx}x + m_{yy}y + t_y
\end{align*}
\]

where \( x \) and \( y \) are the coordinates of a point before the transformation and \( x' \) and \( y' \) are the coordinates of the corresponding point after.

All of the arguments to **make-transformation** must be real numbers.

**make-3-point-transformation**  
*function*  
**Arguments:** point-1 point-2 point-3 point-1-image point-2-image point-3-image  
**Summary:** Returns a transformation that takes points point-1 into point-1-image, point-2 into point-2-image, and point-3 into point-3-image. Three non-collinear points and their images under the transformation are enough to specify any affine transformation.

If point-1, point-2, and point-3 are collinear, the **transformation-underspecified** error will be signaled. If point-1-image, point-2-image, and point-3-image are collinear, the resulting transformation will be singular (that is, will have no inverse), but this is not an error.

**make-3-point-transformation**  
*function*  
**Arguments:** x1 y1 x2 y2 x3 y3 x1-image y1-image x2-image y2-image x3-image y3-image  
**Summary:** Returns a transformation that takes the points at the positions \((x1, y1)\), \((x2, y2)\), and \((x3, y3)\) into \((x1-image, y1-image)\), \((x2-image, y2-image)\), and \((x3-image, y3-image)\). Three non-collinear points and their images under the transformation are enough to specify any affine transformation.

If the positions \((x1, y1)\), \((x2, y2)\), and \((x3, y3)\) are collinear, the **transformation-underspecified** error will be signaled. If \((x1-image, y1-image)\), \((x2-image, y2-image)\), and \((x3-image, y3-image)\) are collinear, the resulting transformation will be singular, but this is not an error.

This is the spread version of **make-3-point-transformation**.
3.5.2 CLIM Transformation Protocol

**transformation**  
[Protocol Class]
Summary: The protocol class of all transformations. There are one or more subclasses of `transformation` that implement transformations, the exact names of which are explicitly unspecified. If you want to create a new class that behaves like a transformation, it should be a subclass of `transformation`. Subclasses of `transformation` obey the transformation protocol.

All of the instantiable transformation classes provided by CLIM are immutable.

**transformationp**  
[Function]
Arguments: `object`
Summary: Returns `t` if `object` is a transformation; otherwise, it returns `nil`.

**+identity-transformation+**  
[Constant]
Summary: An instance of a transformation that is guaranteed to be an identity transformation, that is, the transformation that “does nothing.”

**transformation-error**  
[Error Condition]
Summary: The class that is the superclass of the following three conditions. This class is a subclass of `error`.

**transformation-underspecified**  
[Error Condition]
Summary: The error that is signaled when `make-3-point-transformation` is given three collinear image points.

**reflection-underspecified**  
[Error Condition]
Summary: The error that is signaled when `make-reflection-transformation` is given two coincident points.

**singular-transformation**  
[Error Condition]
Summary: The error that is signaled when `invert-transformation` is called on a singular transformation, that is, a transformation that has no inverse.
3.5.3 CLIM Transformation Predicates

The following predicates are provided in order to be able to determine whether or not a transformation has a particular characteristic.

**transformation-equal**

Arguments: transformation1 transformation2

Summary: Returns t if the two transformations have equivalent effects (that is, are mathematically equal); otherwise, it returns nil.

**identity-transformation-p**

Arguments: transformation

Summary: Returns t if transformation is equal (in the sense of transformation-equal) to the identity transformation; otherwise, it returns nil.

**translation-transformation-p**

Arguments: transformation

Summary: Returns t if transformation is a pure translation, that is, a transformation that moves every point by the same distance in x and the same distance in y. Otherwise, it returns nil.

**invertible-transformation-p**

Arguments: transformation

Summary: Returns t if transformation has an inverse; otherwise, it returns nil.

**reflection-transformation-p**

Arguments: transformation

Summary: Returns t if transformation inverts the “handedness” of the coordinate system; otherwise, it returns nil. Note that this is a very inclusive category—transformations are considered reflections even if they distort, scale, or skew the coordinate system, as long as they invert the handedness.

**rigid-transformation-p**

Arguments: transformation

Summary: Returns t if transformation transforms the coordinate system as a rigid object, that is, as a combination of translations, rotations, and pure reflections. Otherwise, it returns nil.
Rigid transformations are the most general category of transformations that preserve magnitudes of all lengths and angles.

**even-scaling-transformation-p**

Arguments: `transformation`
Summary: Returns `t` if `transformation` multiplies all x-lengths and y-lengths by the same magnitude; otherwise, it returns `nil`. This includes pure reflections through vertical and horizontal lines.

**scaling-transformation-p**

Arguments: `transformation`
Summary: Returns `t` if `transformation` multiplies all x-lengths by one magnitude and all y-lengths by another magnitude; otherwise, it returns `nil`. This category includes even scalings as a subset.

**rectilinear-transformation-p**

Arguments: `transformation`
Summary: Returns `t` if `transformation` will always transform any axis-aligned rectangle into another axis-aligned rectangle; otherwise, it returns `nil`. This category includes scalings as a subset, and also includes 90 degree rotations.

Rectilinear transformations are the most general category of transformations for which the bounding rectangle of a transformed object can be found by transforming the bounding rectangle of the original object.

### 3.5.4 CLIM Transformation Functions

**compose-transformations**

Arguments: `transformation1` `transformation2`
Summary: Returns a transformation that is the mathematical composition of its arguments. Composition is in right-to-left order; that is, the resulting transformation represents the effects of applying the transformation `transformation2` followed by the transformation `transformation1`.

**invert-transformation**

Arguments: `transformation`
Summary: Returns a transformation that is the inverse of the transformation \textit{transformation}. The result of composing a transformation with its inverse is equal to the identity transformation.

If \textit{transformation} is singular, \textbf{invert-transformation} will signal the \textbf{singular-transformation} error, with a named restart that is invoked with a transformation and makes \textbf{invert-transformation} return that transformation. This is to allow a drawing application, for example, to use a generalized inverse to transform a region through a singular transformation.

Note that with finite-precision arithmetic there are several low-level conditions that might occur during the attempt to invert a singular or “almost singular” transformation. (These include computation of a zero determinant, floating-point underflow during computation of the determinant, or floating-point overflow during subsequent multiplication.) \textbf{invert-transformation} signals the \textbf{singular-transformation} error for all of these cases.

\textbf{compose-translation-with-transformation}  
\textbf{Arguments:} \textit{transformation} \textit{dx dy}  

\textbf{compose-scaling-with-transformation}  
\textbf{Arguments:} \textit{transformation} \textit{sx sy} \&optional \textit{origin}  

\textbf{compose-rotation-with-transformation}  
\textbf{Arguments:} \textit{transformation} \textit{angle} \&optional \textit{origin}  
\textbf{Summary:} These functions create a new transformation by composing the transformation \textit{transformation} with a given translation, scaling, or rotation, respectively. The order of composition is that the translation, scaling, or rotation “transformation” is first, followed by \textit{transformation}.

dx and dy are as for \textbf{make-translation-transformation}. sx and sy are as for \textbf{make-scaling-transformation}. angle and origin are as for \textbf{make-rotation-transformation}.

Note that these functions could be implemented by using the various constructors. They are provided because it is common to build up a transformation as a series of simple transformations.
compose-transformation-with-scaling [Function]
Arguments: transformation sx sy &optional origin

compose-transformation-with-rotation [Function]
Arguments: transformation angle &optional origin
Summary: These functions create a new transformation by composing a given translation, scaling, or rotation, respectively, with the transformation transformation. The order of composition is transformation first, followed by the translation, scaling, or rotation “transformation.”

dx and dy are as for make-translation-transformation. sx and sy are as for make-scaling-transformation. angle and origin are as for make-rotation-transformation.

Note that these functions could be implemented by using the various constructors and compose-transformations. They are provided because it is common to build up a transformation as a series of simple transformations.

The following three functions are no different than using with-drawing-options with the :transformation keyword argument supplied. However, they are sufficiently useful that they are provided as a convenience to programmers.

In order to preserve referential transparency, these three forms apply the translation, rotation, or scaling transformation first, then the rest of the transformation from (medium-transformation medium). That is, the following two forms would return the same transformation (assuming that the medium’s transformation in the second example is the identity transformation):

(compose-transformations (make-translation-transformation dx dy) (make-rotation-transformation angle))

(with-translation (medium dx dy) (with-rotation (medium angle) (medium-transformation medium)))

with-translation [Macro]
Arguments: (medium dx dy) &body body
Summary: Establishes a translation on the medium medium that translates by dx in the x direction and dy in the y direction, and then executes body with that transformation in effect.

dx and dy are as for make-translation-transformation.

The CLIM Drawing Environment
The *medium* argument is not evaluated, and must be a symbol that is bound to a
sheet or medium. If *medium* is t, *standard-output* is used. *body* may have
zero or more declarations as its first forms.

**with-scaling**

Arguments: *(medium sx &optional sy origin) &body body*

Summary: Establishes a scaling transformation on the medium *medium* that scales by *sx* in
the x direction and *sy* in the y direction, and then executes *body* with that trans-
formation in effect. If *sy* is not supplied, it defaults to *sx*. If *origin* is supplied, the
scaling is about that point; if it is not supplied, it defaults to (0, 0).

*sx* and *sy* are as for *make-scaling-transformation*.

The *medium* argument is not evaluated, and must be a symbol that is bound to a
sheet or medium. If *medium* is t, *standard-output* is used. *body* may have
zero or more declarations as its first forms.

**with-rotation**

Arguments: *(medium angle &optional origin) &body body*

Summary: Establishes a rotation on the medium *medium* that rotates by *angle*, and then exe-
cutes *body* with that transformation in effect. If *origin* is supplied, the rotation is
about that point; if it is not supplied, it defaults to (0, 0).

*angle* and *origin* are as for *make-rotation-transformation*.

The *medium* argument is not evaluated, and must be a symbol that is bound to a
sheet or medium. If *medium* is t, *standard-output* is used. *body* may have
zero or more declarations as its first forms.

These two functions also compose a transformation into the current transformation of a
stream, but have more complex behavior.

**with-local-coordinates**

Arguments: *(medium &optional x y) &body body*

Summary: Binds the dynamic environment to establish a local coordinate system on the
medium *medium* with the origin of the new coordinate system at the position (*x*,
*y*). The “directionality” of the coordinate system is otherwise unchanged. *x* and
*y* are real numbers, and both default to 0.
The *medium* argument is not evaluated, and must be a symbol that is bound to a sheet or medium. If *medium* is t, *standard-output* is used. *body* may have zero or more declarations as its first forms.

**with-first-quadrant-coordinates**

**Arguments:** *(medium &optional x y) &body body*

**Summary:** Binds the dynamic environment to establish a local coordinate system on the medium *medium* with the positive *x* axis extending to the right and the positive *y* axis extending upward, with the origin of the new coordinate system at the position (*x*, *y*). *x* and *y* are real numbers, and both default to 0.

The *medium* argument is not evaluated, and must be a symbol that is bound to a sheet or medium. If *medium* is t, *standard-output* is used. *body* may have zero or more declarations as its first forms.

### 3.5.5 Applying CLIM Transformations

Transforming a region applies a coordinate transformation to that region, thus moving its position on the drawing plane, rotating it, or scaling it. Note that transforming a region does not affect the *region* argument; it is free to either create a new region or return an existing (cached) region.

These generic functions are implemented for all classes of transformations. Furthermore, all subclasses of *region* and *ink* implement methods for *transform-region* and *untransform-region*. That is, methods for the following generic functions will typically specialize both the *transformation* and *region* arguments.

**transform-region**

**Arguments:** *transformation region*

**Summary:** Applies *transformation* to the region *region*, and returns the transformed region.

**untransform-region**

**Arguments:** *transformation region*

**Summary:** This is exactly equivalent to:

(\texttt{(transform-region (invert-transformation transformation) region)})

CLIM provides a default method for *untransform-region* on the *transformation* protocol class that does exactly this.
transform-position [Generic Function]

Arguments: transformation x y

Summary: Applies the transformation transformation to the point whose coordinates are the real numbers x and y, and returns two values, the transformed x coordinate and the transformed y coordinate.

transform-position is the spread version of transform-region in the case where the region is a point.

untransform-position [Generic Function]

Arguments: transformation x y

Summary: This is exactly equivalent to:

(transform-position (invert-transformation transformation) x y)

CLIM provides a default method for untransform-position on the transformation protocol class that does exactly this.

transform-distance [Generic Function]

Arguments: transformation dx dy

Summary: Applies the transformation transformation to the distance represented by the real numbers dx and dy, and returns two values, the transformed dx and the transformed dy.

A distance represents the difference between two points. It does not transform like a point.

untransform-distance [Generic Function]

Arguments: transformation dx dy

Summary: This is exactly equivalent to:

(transform-distance (invert-transformation transformation) dx dy)

CLIM provides a default method for untransform-distance on the transformation protocol class that does exactly this.

transform-rectangle* [Generic Function]

Arguments: transformation x1 y1 x2 y2

Summary: Applies the transformation transformation to the rectangle specified by the four coordinate arguments, which are real numbers. The arguments x1, y1, x2, and y2 are canonicalized in the same way as for make-bounding-rectangle. Returns
four values that specify the minimum and maximum points of the transformed rectangle in the order min-x, min-y, max-x, and max-y.

It is an error if transformation does not satisfy rectilinear-transformation-p.

transform-rectangle* is the spread version of transform-region in the case where the transformation is rectilinear and the region is a rectangle.

untransform-rectangle* [Generic Function]
Arguments: transformation x1 y1 x2 y2
Summary: This is exactly equivalent to:

(transform-rectangle* (invert-transformation transformation) x1 y1 x2 y2)

CLIM provides a default method for untransform-rectangle* on the transformation protocol class that does exactly this.
Chapter 4  Text Styles
4.1 Conceptual Overview of Text Styles

CLIM’s model for the appearance of text is that the application program should describe how the text should appear in high-level terms, and that CLIM will take care of the details of choosing a specific device font. This approach emphasizes portability.

You specify the appearance of text by giving it an abstract text style. Each CLIM medium defines a mapping between these abstract style specifications and particular device-specific fonts. At runtime, CLIM chooses an appropriate device font to represent the characters. However, some programmers may require direct access to particular device fonts. The text-style mechanism allows you to specify device fonts by name, thus trading portability for control.

A text style is a combination of three characteristics that describe how characters appear. Text style objects have components for family, face, and size:

family

Characters of the same family have a typographic integrity, so that all characters of the same family resemble one another. One of :fix, :serif, :sans-serif, or nil.

face

A modification of the family, such as bold or italic. One of :roman (meaning normal), :bold, :italic, (:bold :italic), or nil.

size

The size of the character. One of the logical sizes (:tiny, :very-small, :small, :normal, :large, :very-large, :huge, :smaller, :larger), or a real number representing the size in printer’s points, or nil.

Not all of these attributes need be specified for a given text style object. Text styles can be merged in much the same way as pathnames are merged; unspecified components in the style object (that is, components that have nil in them) may be filled in by the components of a “default” style object.

*default-text-style* [Variable]

Summary: This is the default text style used by all streams.

Note that the sizes :smaller and :larger are treated differently than the others, in that they are merged with the default text style size to produce a size that is discernibly smaller or
larger. For example, a text style size of :larger would merge with a default text size of :small to produce the resulting size :normal.

A text style object is called fully specified if none of its components is nil and the size component is not a relative size (that is, neither :smaller nor :larger).

When text is rendered on a medium, the text style is mapped to some medium-specific description of the glyphs for each character. This description is usually that medium’s concept of a font object. This mapping is mostly transparent to the application developer, but it is worth noting that not all text styles have mappings associated with them on all mediums. If the text style used does not have a mapping associated with it on the given medium, a special text style reserved for this case will be used.

*undefined-text-style* [Variable]
Summary: The text style that is used as a fallback if no mapping exists for some other text style when some text is about to be rendered on a display device (via write-string and draw-string*, for example). This text style must be fully merged, and it must have a mapping for all display devices.

4.2 CLIM Text Style Objects

It is often useful to create a text style object that represents a style you wish to use frequently, rather than continually specifying the corresponding text style suboptions.

For example, if you want to write on a stream with a particular family, face, and size, you can create a text style object using make-text-style:

```clim
(clim:with-text-style
  ((clim:make-text-style :fix :bold :large) my-stream)
  (write-string "Here is a text-style example." my-stream))
```

=> Here is a text-style example.

Note that text style objects are interned. That is, two different invocations of make-text-style with the same combination of family, face and size will result in the same (in the sense of eq) text style object. For this reason, you should not modify text style objects.
text-style [Protocol Class]
Summary: The protocol class for text styles. If you want to create a new class that behaves like a text style, it should be a subclass of text-style. Subclasses of text-style must obey the text style protocol.

text-style-p [Function]
Arguments: object
Summary: Returns t if object is a text style; otherwise, it returns nil.

standard-text-style [Class]
Summary: An instantiable class that implements text styles. It is a subclass of text-style. This is the class that make-text-style instantiates. Members of this class are immutable.

make-text-style [Function]
Arguments: family face size
Summary: Returns an object of class standard-text-style with a family of family, a face of face, and a size of size.

You can use text style suboptions to specify characteristics of a text style object. Each text style suboption has a reader function which returns the current value of that component from a text style object. The suboptions are listed as follows.

:text-family [Option]

text-style-family [Generic Function]
Arguments: text-style
Summary: Specifies the family of the text style text-style.
4.3 CLIM Text Style Functions

The following functions can be used to parse, merge, and create text-style objects, as well as to read the components of the objects.

**parse-text-style**

*Arguments*: `style-spec`  
*Summary*: Returns a text-style object. `style-spec` may be a text-style object or a device font, in which case it is returned as is, or it may be a list of the family, face, and size (that is, a “style spec”), in which case it is “parsed” and a text-style object is returned.

This function is for efficiency, since a number of common functions that take a style object as an argument can also take a style spec, in particular `draw-text`.

**merge-text-styles**

*Arguments*: `style1` `style2`  
*Summary*: Merges the text styles `style1` with `style2`; that is, returns a new text style that is the same as `style1`, except that unspecified components in `style1` are filled in from `style2`. For convenience, the two arguments may be also be style specs.

When merging the sizes of two text styles, if the size from `style1` is a relative size, the resulting size is either the next smaller or next larger size than is specified by `style2`. The ordering of sizes, from smallest to largest, is `:tiny`, `:very-small`, `:small`, `:normal`, `:large`, `:very-large`, and `:huge`.

TextStyles
Merging font faces is also possible. For example, merging bold and italic faces results in a bold-italic face. When the faces are mutually exclusive, the face specified by style1 prevails.

**text-style-components**  [Generic Function]
Arguments: text-style
Summary: Returns the components of text-style as three values (family, face, and size).

**text-style-family**  [Generic Function]
Arguments: text-style
Summary: Returns the family component of text-style.

**text-style-face**  [Generic Function]
Arguments: text-style
Summary: Returns the face component of text-style.

**text-style-size**  [Generic Function]
Arguments: text-style
Summary: Returns the size component of text-style.

**text-style-ascent**  [Generic Function]
Arguments: text-style medium
Summary: The ascent (an integer) of text-style as it would be rendered on medium medium.
Summary: The ascent of a text style is the ascent of the medium’s font corresponding to text-style. The ascent of a font is the distance between the top of the tallest character in that font and the baseline.

**text-style-descent**  [Generic Function]
Arguments: text-style medium
Summary: The descent (an integer) of text-style as it would be rendered on medium medium.

The descent of a text style is the descent of the medium’s font corresponding to text-style. The descent of a font is the distance between the baseline and the bottom of the lowest descending character (usually “y,” “q,” “p,” or “g”).
text-style-height

Arguments: text-style medium
Summary: Returns the height (an integer) of the “usual character” in text-style on medium medium.

The height of a text style is the sum of its ascent and descent.

text-style-width

Arguments: text-style medium
Summary: Returns the width (an integer) of the “usual character” in text-style on medium medium.

text-style-fixed-width-p

Arguments: text-style medium
Summary: Returns t if text-style will map to a fixed-width font on medium medium; otherwise, it returns nil.

The methods for this generic function will typically specialize both the text-style and port arguments. CLIM provides a “trampoline” for this generic function for mediums and output sheets which will simply call the method for the port.

text-size

Arguments: medium string &key text-style (start 0) end
Summary: Computes the “cursor motion” in device units that would take place if string (which may be either a string or a character) were output to the medium medium starting at the position (0, 0).

Five values are returned: the total width of the string in device units, the total height of the string in device units, the final x cursor position (which is the same as the width if there are no \Newline characters in the string), the final y cursor position (which is 0 if the string has no \Newline characters in it, and is incremented by the line height of medium for each \Newline character in the string), and the string’s baseline.

text-style specifies what text style is to be used when doing the output, and defaults to medium-merged-text-style of the medium. text-style must be a fully specified text style. start and end may be used to specify a substring of string.

Programmers needing to account for kerning or the ascent or descent of the text style should measure the size of the bounding rectangle of the text rendered on medium.
All mediums and output sheets implement a method for this generic function.

### 4.4 Text Style Binding Forms

CLIM provides several forms with which you can establish a binding of a text style or a text-style component. The extent of the binding is the dynamic extent of the particular binding form.

**with-text-style**

**Arguments:** (medium text-style) &body body

**Summary:** Binds the current text style of the medium medium to correspond to the new text style. text-style may either be a text style object or a style spec (that is, a list of a family, a face, and a size). body is executed with the new text style in effect.

The medium argument is not evaluated, and must be a symbol that is bound to a sheet or medium. If medium is t, *standard-output* is used. body may have zero or more declarations as its first forms.

with-text-style expands into a call to invoke-with-text-style and supplies a function that executes body as the continuation argument to invoke-with-text-style.

**invoke-with-text-style**

**Arguments:** medium continuation text-style

**Summary:** Binds the current text style of the medium medium to correspond to the new text style, and calls the function continuation with the new text style in effect. text-style may either be a text style object or a style spec (that is, a list of a family, a face, and a size). continuation is a function of one argument, the medium; it has dynamic extent.

medium can be a medium, a sheet that supports the sheet output protocol, or a stream that outputs to such a sheet. All classes that obey the medium protocol implement a method for invoke-with-text-style.

The following macros are “convenience” forms of with-text-style that expand into calls to invoke-with-text-style.

The medium argument of these macros is not evaluated, and must be a symbol that is bound to a sheet or medium. If medium is t, *standard-output* is used. body may have zero or more declarations as its first forms.
with-text-face
[Macro]
Arguments: (medium face) &body body
Summary: Binds the current text face of medium to correspond to the new text face face, within the body. face is one of :roman, :bold, :italic, (:bold :italic), or nil.

with-text-family
[Macro]
Arguments: (medium family) &body body
Summary: Binds the current text family of medium to correspond to the new text family family, within the body. family is one of :fix, :serif, :sans-serif, or nil.

with-text-size
[Macro]
Arguments: (medium size) &body body
Summary: Binds the current text size of medium to correspond to the new text size size, within the body.

4.5 Controlling Text Style Mappings

Text styles are mapped to fonts using the text-style-mapping function, which takes a port and a text style, and returns a font object. All ports implement methods for the following generic functions, for all classes of text style.

The objects used to represent a font mapping are unspecified and are likely to vary from port to port. For instance, a mapping might be some sort of font object on one type of port, or might simply be the name of a font on another. Part of initializing a port is to define the mappings between text styles and font names for the port’s host window system.

text-style-mapping
[Generic Function]
Arguments: port text-style
Summary: Returns the font mapping that will be used when rendering characters in the text style text-style on any medium on the port port. If there is no mapping associated with text-style on port, then some other object will be returned that corresponds to the “unmapped” text style.

(setf text-style-mapping)
[Generic Function]
Arguments: mapping port text-style
Summary: Sets the text style mapping for `port` and `text-style` to `mapping`. `port` and `text-style` are as for `text-style-mapping`. `mapping` is either a font name or a list of the form `(:style family face size)`; in the latter case, the given style is translated at runtime into the font represented by the specified style.

**make-device-font-text-style**

[Function]

**Arguments:** `display-device device-font-name`

**Summary:** Returns a text style object that will be mapped directly to the specified device font when text is output to the display device with this style. Device font styles do not merge with any other kind of style. As the specified font is device-specific, the use of this function may result in non-portable applications.

This code creates a device font text style and applies it to a string of characters.

```lisp
(let ((my-device-font (make-device-font-text-style
                         (port my-sheet)
                         (gp:make-font-description :family "courier"
                          :size 14
                          :weight :medium
                          :slant :italic))))
  (draw-text* my-sheet "This appears in the specified device font."
              10 10 :text-style my-device-font))
```

&lt;Bold&gt;102CLIM User Guide
Chapter 5  Drawing in Color
5.1 Conceptual Overview of Drawing With Color

This chapter describes the :ink drawing option and the simpler values that can be supplied for that option, such as colors.

To draw in color, you supply the :ink drawing option to CLIM’s drawing functions (see Chapter 2, “Drawing Graphics in CLIM,” for details). :ink can take as its value:

- a color
- the constant +foreground-ink+
- the constant +background-ink+
- a flipping ink

The drawing functions work by selecting a region of the drawing plane and painting it with color. The region is clipped by the current :clipping-region drawing option in effect, and is then transformed by the current :transformation drawing option (see Chapter 3, “The CLIM Drawing Environment,” for the rules controlling these options). The shape can be a graphical area (such as a rectangle or an ellipse), a path (such as a line segment or the outline of an ellipse), or the letter forms of text. Any viewports or dataports attached to this drawing plane are updated accordingly. The :ink drawing option is never affected by the :transformation drawing option nor by the sheet transformation; this ensures that stipple patterns on adjacent sheets join seamlessly.

Along with its drawing plane, a medium has a foreground and a background. The foreground is the default ink when the :ink drawing option is not specified. The background is drawn all over the drawing plane before any output is drawn. You can erase by drawing the background over the region to be erased. You can change the foreground or background at any time. This changes the contents of the drawing plane. The effect is as if everything on the drawing plane is erased, the background is drawn on the entire drawing plane, and then everything that was ever drawn (provided it was saved in the output history) is redrawn using the new foreground and background.
5.1.1 Color Objects

A color in CLIM is an object representing the intuitive definition of color: white, black, red, pale yellow, and so forth. The visual appearance of a single point is completely described by its color.

A color can be specified by three real numbers between 0 and 1 inclusive, giving the amounts of red, green, and blue. Three 0’s mean black; three 1’s mean white. A color can also be specified by three numbers giving the intensity, hue, and saturation. A totally unsaturated color (a shade of gray) can be specified by a single real number between 0 and 1, giving the amount of white.

You can obtain a color object by calling one of make-rgb-color, make-ihs-color, or make-gray-color, or by using one of the predefined colors listed in Section 5.3, “Predefined Color Names in LispWorks CLIM,” or . Specifying a color object as the :ink drawing option, the foreground, or the background causes CLIM to use that color in the appropriate drawing operations.

5.1.2 Rendering

When CLIM renders the graphics and text in the drawing plane onto a real display device, physical limitations of the display device force the visual appearance to be an approximation of the drawing plane. Colors that the hardware doesn’t support might be approximated by using a different color or by using a stipple pattern. Even primary colors such as red and green can’t be guaranteed to have distinct visual appearance on all devices, so if device independence is desired, it is best to use make-contrasting-inks (which produces designs of different appearances) rather than a fixed palette of colors.
The line style and text style respectively control the region of the display device that is colored when a path or text is rendered.

## 5.2 CLIM Operators for Drawing in Color

The following functions create colors. These functions produce objects that have equivalent effects and are indistinguishable when drawn; the only difference is in how the color components are specified. Whether these functions use the specified values exactly or approximate them because of limited color resolution is unspecified. Whether these functions create a new object or return an existing object with equivalent color component values is also unspecified.

**make-rgb-color**

*Arguments:* red green blue

*Summary:* Returns a member of the class color. The red, green, and blue arguments are real numbers between 0 and 1 (inclusive) that specify the values of the corresponding color components.

**make-ihs-color**

*Arguments:* intensity hue saturation

*Summary:* Returns a member of class color. The intensity argument is a real number between 0 and \(\sqrt{3}\) (inclusive). The hue and saturation arguments are real numbers between 0 and 1 (inclusive).

**make-gray-color**

*Arguments:* luminance

*Summary:* Returns a member of class color. luminance is a real number between 0 and 1 (inclusive). On a black-on-white display device, 0 means black, 1 means white, and the other values are shades of gray. On a white-on-black display device, 0 means white, 1 means black, and the other values are shades of gray.

**make-contrasting-inks**

*Arguments:* n &optional k

*Summary:* If k is not supplied, this returns a vector of n designs with recognizably different appearance. Elements of the vector are guaranteed to be acceptable values for the :ink argument to the drawing functions, and will not include +foreground-ink+.
+background-ink+, or nil. Their class is otherwise unspecified. The vector is a
fresh object that may be modified.

If k is supplied, it must be an integer between 0 and n \( \leq 1 \) (inclusive), in which
case make-contrasting-inks returns the \( k \)th design rather than returning a vector
of designs.

CLIM supports at least 8 different contrasting inks. If \( n \) is greater than the number
of contrasting inks, make-contrasting-inks signals an error.

The rendering of the design may be a color or a stippled pattern, depending on
whether the output medium supports color.

color-rgb
[Generic Function]
Arguments: color
Summary: Returns three values, the red, green, and blue components of the color color. The
values are real numbers between 0 and 1 (inclusive).

color-ihs
[Generic Function]
Arguments: color
Summary: Returns three values, the intensity, hue, and saturation components of the color color. The first value is a real number between 0 and \( \sqrt{3} \) (inclusive). The second and third values are real numbers between 0 and 1 (inclusive).
## 5.3 Predefined Color Names in LispWorks CLIM

The following color constants are provided in LispWorks CLIM: +black+, +white+, +red+, +blue+, +green+, +cyan+, +magenta+, and +yellow+. Other predefined colors are available through the facility of a palette. Application programs can define other colors.

## 5.4 Indirect Inks

Drawing with an *indirect ink* is the same as drawing another design named directly. For example, +foreground-ink+ is a design that draws the medium’s foreground design and is the default value of the :ink drawing option.

Indirect ink is a useful abstraction that enables your code to ignore the issue of what specific ink to use. It is also useful for output recording. For example, you can draw with +foreground-ink+, change to a different medium-foreground, and replay the output record; the replayed output will come out in the new color.

You can change the foreground or background design of a medium at any time. This changes the contents of the medium’s drawing plane. The effect is as if everything on the drawing plane is erased, the background design is drawn onto the drawing plane, and then everything that was ever drawn (provided it was saved in the output history) is drawn over again, using the medium’s new foreground and background.

If an infinite recursion is created using an indirect ink, an error is signaled when the recursion is created, when the design is used for drawing, or both. Two indirect inks have been defined:

+foreground-ink+  [Constant]
Summary: An indirect ink that uses the medium’s foreground design.

+background-ink+  [Constant]
Summary: An indirect ink that uses the medium’s background design.
5.5 Flipping Ink

Use “flipping ink” to exchange the colors of two inks. You can also use it to exchange the values of \(+\text{foreground-ink}\)+ and \(+\text{background-ink}\)\+. For an example of its use, see Subsection 5.6.1.

\[\text{+flipping-ink+}\]
[Constant]
Summary: A flipping ink that flips \(+\text{foreground-ink}\)+ and \(+\text{background-ink}\)\+.

\text{make-flipping-ink}
[Function]
Arguments: \(ink1\ i\nk2\)
Summary: Returns a design that interchanges occurrences of the two designs \(ink1\) and \(ink2\).

Drawing a flipping ink over a background changes the color in the background that would have been drawn by \(ink1\) at that point into the color that would have been drawn by \(ink2\) at that point, and vice versa. The effect on any color other than the colors determined by those two inks is unspecified; however, drawing the same figure twice using the same flipping ink is guaranteed to be an “identity” operation. If either \(ink1\) or \(ink2\) is not solid, the consequences are unspecified. The purpose of flipping is to allow the use of \((\text{xor})\) operations for temporary changes to the display.

If \(ink1\) and \(ink2\) are equivalent, the result can be \(+\text{nowhere}+\).

5.6 Examples of Simple Drawing Effects

To draw in the foreground color, use the default, or specify \(\text{:ink +\text{foreground-ink}+}\).

To erase, specify \(\text{:ink +\text{background-ink}+}\).

To draw in color, specify \(\text{:ink +green+, :ink (make-rgb-color 0.6 0.0 0.4)}\), and so forth.

To draw an opaque gray, specify \(\text{:ink (make-gray-color 0.25)}\). This will draw a shade of gray independent of the window’s foreground color. On a non-color, non-gray-scale display this will generally turn into a stipple.
To draw a stipple of little bricks, specify \texttt{:ink \ bricks}, where \texttt{bricks} is defined as:

\begin{verbatim}
(make-rectangular-tile
 (make-pattern #2a(0 0 0 0 1 0 0 0 0)
 (0 0 0 1 0 0 0 0)
 (0 0 0 1 0 0 0 0)
 (1 1 1 1 1 1 1 1)
 (0 0 0 0 0 0 0 1)
 (0 0 0 0 0 0 0 1)
 (0 0 0 0 0 0 0 1)
 (1 1 1 1 1 1 1 1))
 (list +background+ +foreground+)) 8 8)
\end{verbatim}

To draw a tiled pattern, specify \texttt{:ink (make-rectangular-tile (make-pattern array colors))}.

To draw a pixmap, use \texttt{(draw-design (make-pattern array colors) medium)}.
5.6.1 Using Flipping Ink

(defun cmd-rubberband ()
  (let ((x1 0) ; x1, y1 represents the fix point
        (y1 0)
        (x2 0) ; x2, y2 represents the point that is changing
        (y2 0)
        (mouse-button-press nil)
        ;; press to select pivot
        (stream (get-frame-pane *application-frame* 'main)))
    (tracking-pointer (stream)
      (:pointer-button-press
       (event x y)
       (setf x1 x y1 y x2 x y2 y)
       (draw-line* stream x1 y1 x2 y2
                    :ink +flipping-ink+)
       (setf mouse-button-press t))
      (:pointer-motion
       (window x y)
       (when Mouse-button-press
        ;; erase
        (draw-line* stream x1 y1 x2 y2
                    :ink +flipping-ink+)
        ;; draw
        (draw-line* stream x1 y1 x y
                    :ink +flipping-ink+)
        (setf x2 x y2 y))
      (:pointer-button-release
       (event x y)
       (cond
        ((eq mouse-button-press t)
         (return
          (list x1 y1 x2 y2)))))))

Drawing in Color
<Bold>112CLIM User Guide
Chapter 6  Presentation Types

6.1  Conceptual Overview of CLIM Presentation Types

6.1.1  User Interaction With Application Objects

In object-oriented programming systems, applications are built around internal objects that model something in the real world. For example, an application that models a university has objects representing students, professors, and courses. A CAD system for designing circuits has objects representing gates, resistors, and so on. A desktop publishing system has objects representing paragraphs, headings, and illustrations.
Application objects have to be presented to the user, and the user has to be able to interact with them. In CLIM, an interface enables the user to see visual representations of the application objects and, via these representations, operate on the application objects themselves.

A very basic part of designing a CLIM user interface is specifying how the user will interact with application objects. There are two directions of interaction: you must present application objects to the user as output, and you must accept input from the user that indicates application objects. This is done with two basic functions, `present` and `accept`, plus some related functions.

### 6.1.2 Presentations and Presentation Types

CLIM keeps track of the association between a visual representation of an object and the object itself. CLIM maintains this association in a data structure called a *presentation*. A presentation embodies three things:

- The underlying application object
- Its presentation type
- Its visual representation

![Diagram](image-url)
In other words, a presentation is a special kind of output record that remembers not only output, but the object associated with the output and the semantic type associated with that object.

A presentation type can be thought of as a CLOS class that has some additional functionality pertaining to its roles in the user interface of an application. In defining a presentation type, the application programmer defines all of the user interface components of the entity:
- Its displayed representation, textual or graphical
- Textual representation, for user input via the keyboard
- Pointer sensitivity, for user input via the pointer

In other words, the application programmer describes in one place all the information about an object necessary to display it to the user and interact with the user for object input.

### 6.1.3 Output With Its Semantics Attached

For example, a university application has a “student” application object. The user sees a visual representation of a student, which might be a textual representation, a graphical representation (such as a form with name, address, and student id number), or even an image of the face of the student. The presentation type of the student is “student”; that is, the semantic type of the object that appears on the screen is “student.” Since the type of a displayed object is known, CLIM knows which operations are appropriate to perform on the displayed object. For example, when a student is displayed, it is possible to perform operations such as send-tuition-bill or show-transcript.

### 6.1.4 Input Context

Presentations are the basis of many of the higher-level application-building tools that use accept to get input and present to display output. A command that takes arguments as input specifies the presentation type of each argument. When a call to accept is made, CLIM establishes an “input-context” based on the presentation type. This input context is used to determine which presentations will be sensitive to mouse clicks. For instance, when a user gives the send-tuition-bill command, the input context is of type “student,” so any students displayed—both those being displayed for the first time and those that have been displayed before—are sensitive. This is because presentations that have been output in previous user
interactions retain their semantics; that is, CLIM has recorded the fact that a student has been displayed and has saved this information.

6.1.5 Inheritance

CLIM presentation types are designed to use inheritance, just as CLOS classes do. For example, a university might need to model “night-student,” which is a subclass of “student.” When the input context is looking for a student, night-students are sensitive because they are represented as a subtype of student.

The set of presentation types forms a type lattice, an extension of the Common Lisp CLOS type lattice. When a new presentation type is defined as a subtype of another presentation type, it inherits all the attributes of the supertype except those explicitly overridden in the definition.

6.1.6 Presentation Translators

You can define presentation translators to make the user interface of your application more flexible. For example, suppose the input context is expecting a command. In this input context, all displayed commands are sensitive, so the user can point to one to execute it. However, suppose the user points to another kind of displayed object, such as a student. In the absence of a presentation translator, the student is not sensitive because only commands can be entered to this input context.

In the presence of a presentation translator that translates from students to commands, however, both students and commands would be sensitive. When the student is highlighted, the middle pointer button might execute the command show-transcript for that student.

6.1.7 What the Application Programmer Does

By the time you get to the point of designing the user interface, you have probably designed the rest of the application and know what the application objects are. At this point, you need to do the following:

1. Decide what types of application objects will be presented to the user as output and accepted from the user as input.
2. For each type of application object that the user will see, assign a corresponding presentation type. In many cases, this means simply using a predefined presentation type. In other cases, you need to define a new presentation type yourself. Usually the presentation type is the same as the class of the application object.

3. Use the application-building tools to specify the windows, menus, commands, and other elements of the user interface. Most of these elements will use the presentation types of your objects.

### 6.2 How to Specify a CLIM Presentation Type

This section describes how to specify a CLIM presentation type. For a complete description of CLIM presentation types, options, and parameters, see Section 6.5, “Predefined Presentation Types”.

Several CLIM operators take presentation types as arguments. You specify them using a presentation type specifier.

Most presentation type specifiers are also Common Lisp type specifiers. For example, the `boolean` presentation type is a Common Lisp type specifier. Not all presentation types are Common Lisp types, and not all Common Lisp types are presentation types (e.g., hash-tables), but there is a lot of overlap (e.g., commands, numbers, and strings).

A presentation type specifier appears in one of the following three patterns:

- `name`
- `(name parameters...)`
- `((name parameters...) options...)`

The first pattern, `name`, indicates a simple presentation type, which can be one of the predefined presentation types or a user-defined presentation type. Examples of the first pattern are:

- `integer`  A predefined presentation type
- `pathname` A predefined presentation type
- `boolean`  A predefined presentation type
student  A user-defined presentation type

The second pattern, \((name\ parameters...),\) supports parameterized presentation types, which are analogous to parameterized Common Lisp types such as \((integer\ 0\ 9)\) in method lambda lists. The function \texttt{presentation-typep} uses the parameters to check object membership in a type. Adding parameters to a presentation type specifier produces a subtype that contains some but not necessarily all of the objects that are members of the unparameterized type. Thus the parameters can turn off the sensitivity of some presentations that would otherwise be sensitive. The parameters state a restriction on the presentation type, so a parameterized presentation type is a specialization or a subset of the unparameterized presentation type of that name.

Examples of the second pattern are:

\((integer\ 0\ 10)\) \hspace{1cm} A parameterized type indicating an integer in the range of zero through ten.

\((string\ 25)\) \hspace{1cm} A parameterized type indicating a string whose length is 25.

\((member\ :yes\ :no\ :maybe)\)

\hspace{1cm} A parameterized type that can be one of the three given values: \texttt{yes}, \texttt{no}, and \texttt{maybe}.

The third pattern, \(((name\ parameters...)\ options...),\) enables you to specify options that affect the use or appearance of the presentation, but not its semantic meaning. The options are keyword/value pairs, and are defined by the presentation type. All presentation types accept the \texttt{description} option, which enables you to provide a string describing the presentation type. If provided, this option overrides the description specified in the \texttt{define-presentation-type} form, and also overrides the \texttt{describe-presentation-type} presentation method.

For example, you can use this form to specify an octal integer from 0 to 10:

\((\{integer\ 0\ 10\} :base\ 8)\)

While in theory some presentation type options may appear as an option in any presentation type specifier, currently the only such option is \texttt{description}.

Each presentation type has a name, which is usually a symbol naming the presentation type. The name can also be a CLOS class object (but not a built-in class object); this usage provides the support for anonymous CLOS classes.
Every presentation type is associated with a CLOS class. If \textit{name} is a class object or the name of a class, and that class is not a built-in class, that class is used as the associated class. Otherwise, \texttt{define-presentation-type} defines a class with the metaclass \texttt{clim:presentation-type-class} and superclasses determined by the presentation type definition. This class is not named \textit{name}, since that could interfere with built-in Common Lisp types such as \texttt{and}, \texttt{member}, and \texttt{integer}. \texttt{class-name} of this class returns a list of the form \texttt{(presentation-type name)}. \texttt{clim:presentation-type-class} is a subclass of \texttt{standard-class}.

Programmers are required to evaluate the \texttt{defclass} form first in the case when the same name is used in both a \texttt{defclass} and a \texttt{define-presentation-type}.

Every CLOS class (except for built-in classes) is a presentation type, as is its name. Unless it has been defined with \texttt{define-presentation-type}, it allows no parameters and no options.

\textit{Presentation type inheritance} is used both to inherit methods (“what parser should be used for this type?”), and to establish the semantics for the type (“what objects are sensitive in this input context?”). Inheritance of methods is the same as in CLOS and thus depends only on the type name, not on the parameters and options.

During presentation method combination, presentation type inheritance arranges to translate the parameters of a subtype into a new set of parameters for its supertype, and translates the options of the subtype into a new set of options for the supertype.

## 6.3 Using CLIM Presentation Types for Output

Presentations for program output so that the objects presented will be acceptable to input functions. Suppose, for example, you present an object, such as \texttt{5}, as a TV channel. When a command that takes a TV channel as an argument is issued or when a presentation translation function is “looking for” such a thing, the system will make that object sensitive. Also, when a command that is looking for a different kind of object (such as a highway number), the object \texttt{5} is not sensitive, because that object represents a TV channel, not a highway number.

A presentation includes not only the displayed representation itself, but also the object presented and its presentation type. When a presentation is output to a CLIM window, the object and presentation type are “remembered”—that is, the object and type of the display at a particular set of window coordinates are recorded in the window’s output history. Because

Presentation Types
this information remains available, previously presented objects are themselves available for input to functions for accepting objects.

An application can use the following operators to produce output that will be associated with a given Lisp object and declared to be of a specified presentation type. This output is saved in the window’s output history as a presentation. Specifically, the presentation remembers the output that was performed (by saving the associated output record), the Lisp object associated with the output, and the presentation type specified at output time. The object can be any Lisp object.

## 6.3.1 CLOS Operators

CLOS provides these top-level facilities for presenting output. `with-output-as-presentation` is the most general operator, and `present` and `present-to-string` support common idioms.

**with-output-as-presentation**  
[Macro]

**Arguments:**  
(stream object type &key modifier single-box allow-sensitive-inferiors record-type) &body body

**Summary:**  
The output of body to the extended output recording stream stream is used to generate a presentation whose underlying object is object and whose presentation type is type. Each invocation of this macro results in the creation of a presentation object in the stream’s output history unless output recording has been disabled or :allow-sensitive-inferiors nil was specified at a higher level, in which case the presentation object is not inserted into the history. `with-output-as-presentation` returns the presentation corresponding to the output.

The stream argument must be a symbol that is bound to an extended output stream or output recording stream. If stream is t, *standard-output* is used. body may have zero or more declarations as its first forms.

type is a presentation type specifier and may be an abbreviation.

modifier, which defaults to nil, is a function that describes how the presentation object might be modified. For example, it might be a function of one argument (the new value) that can be called in order to store a new value for object after a user somehow “edits” the presentation. modifier must have indefinite extent.

single-box is used to specify the presentation-single-box component of the resulting presentation. It can take on the values described under presentation-single-box.
When the boolean `allow-sensitive-inferiors` is `nil`, nested calls to `present` or `with-output-as-presentation` inside this one will not generate presentations. The default is `t`.

`record-type` specifies the class of the presentation output record to be created. It defaults to `standard-presentation`. This argument should only be supplied by a programmer if there is a new class of output record that supports the updating output record protocol.

All arguments of this macro are evaluated:

```
(with-output-as-presentation (stream #p"foo" 'pathname)
  (princ "FOO" stream))
```

**present**

```
[Function]
Arguments: object &optional type &key stream view modifier acceptably
  for-context-type single-box allow-sensitive-inferiors sensitive record-type
Summary: The object of presentation type `type` is presented to the extended output stream `stream` (which defaults to `*standard-output*`), using the type’s `present` method for the supplied view `view`. `type` is a presentation type specifier, and can be an abbreviation. It defaults to `(presentation-type-of object)`. The other arguments and overall behavior of `present` are as for `stream-present`.

The returned value of `present` is the presentation object that contains the output corresponding to the object.

`present` expands any presentation type abbreviations (`type` and `for-context-type`), and then calls `stream-present` on `stream`, `object`, `type`, and the remaining keyword arguments.

**stream-present**

```
[Generic Function]
Arguments: stream object type &key view modifier acceptably for-context-type single-box allow-sensitive-inferiors sensitive record-type
Summary: `stream-present` is the per-stream implementation of `present`, analogous to the relationship between `write-char` and `stream-write-char`. All extended output streams and output recording streams implement a method for `stream-present`. The default method (on `standard-extended-output-stream`) is as follows.

The object `object` of type `type` is presented to the stream `stream` by calling the type’s `present` method for the supplied view `view`. The returned value is the presentation containing the output corresponding to the object.

`type` is a presentation type specifier.

Presentation Types
view is a view object that defaults to stream-default-view of stream.

for-context-type is a presentation type specifier that is passed to the present method for type, which can use it to tailor how the object will be presented. for-context-type defaults to type.

modifier, single-box, allow-sensitive-inferiors, and record-type are the same as for with-output-as-presentation.

acceptably defaults to nil, which requests the present method to produce text designed to be read by human beings. If acceptably is t, it requests the present method to produce text that is recognized by the accept method for for-context-type. This makes no difference to most presentation types.

The boolean sensitive defaults to t. If it is nil, no presentation is produced.

The boolean sensitive defaults to t. If it is nil, no presentation is produced.

present-to-string [Function]
Arguments: object &optional type &key view acceptably for-context-type string index
Summary: Same as present inside with-output-to-string. If string is supplied, it must be a string with a fill pointer. When index is supplied, it is used as an index into string. view, acceptably, and for-context-type are as for present.

The first returned value is the string. When string is supplied, a second value is returned, the updated index.

6.3.2 Additional Functions for Operating on Presentations in CLIM

The following functions can be used to examine or modify presentations:

presentation [Protocol Class]
Summary: The protocol class that corresponds to a presentation and is a subclass of output-record. If you want to create a new class that behaves like a presentation, it should be a subclass of presentation. Subclasses of presentation obey the presentation protocol.

presentationp [Function]
Arguments: object
Summary: Returns t if and only if object is of type presentation.
**presentation-object**  
[Generic Function]  
Arguments: *presentation*  
Summary: Returns the object represented by the presentation *presentation*.

(setf presentation-object)  
[Generic Function]  
Arguments: *object presentation*  
Summary: Changes the object associated with the presentation *presentation* to *object*.

**presentation-type**  
[Generic Function]  
Arguments: *presentation*  
Summary: Returns the presentation type of the presentation *presentation*.

(setf presentation-type)  
[Generic Function]  
Arguments: *type presentation*  
Summary: Changes the type associated with the presentation *presentation* to *type*.

**presentation-single-box**  
[Generic Function]  
Arguments: *presentation*  
Summary: Returns the “single box” attribute of the presentation *presentation*, which controls how the presentation is highlighted and when it is sensitive. This will be one of four values:

- **nil** (the default)—if the pointer is pointing at a visible piece of the output that was drawn as part of the presentation, then it is considered to be pointing at the presentation. The presentation is highlighted by highlighting each visible part of the output that was drawn as part of the presentation.
- **t**—if the pointer is inside the bounding rectangle of the presentation, it is considered to be pointing at the presentation. The presentation is highlighted by drawing a thin border around the bounding rectangle.
- **:position**—like **t** for determining whether the pointer is pointing at the presentation, but like **nil** for highlighting.
- **:highlighting**—like **nil** for determining whether the pointer is pointing at the presentation, but like **t** for highlighting.

(setf presentation-single-box)  
[Generic Function]  
Arguments: *single-box presentation*  
Summary: Changes the “single box” attribute of the presentation *presentation* to *single-box*.  

Presentation Types
**presentation-modifier**  
*Generic Function*  

**Arguments:** `presentation`  

**Summary:** Returns the “modifier” associated with the presentation `presentation`. The modifier is some sort of object that describes how the presentation object might be modified. For example, it might be a function of one argument (the new value) that can be called in order to store a new value for `object` after a user somehow “edits” the presentation.

**standard-presentation**  
*Class*  

**Summary:** The output record class that represents presentations. `present` normally creates output records of this class. Members of this class are mutable.

- `:object`  
  *Initarg*  

- `:type`  
  *Initarg*  

- `:view`  
  *Initarg*  

- `:single-box`  
  *Initarg*  

- `:modifier`  
  *Initarg*  

All presentation classes must handle these five initargs, which are used to specify, respectively, the object, type, view, single-box, and modifier components of a presentation.

### 6.4 Using CLIM Presentation Types for Input

The primary means for getting input from the end user is `accept`. Characters typed in at the keyboard in response to a call to `accept` are parsed, and the application object they represent is returned to the calling function. (The parsing is done by the `accept` method for the presentation type.) Alternatively, if a presentation of the type specified by the `accept` call has previously been displayed, the user can click on it with the pointer and `accept` returns it directly (that is, no parsing is required).

Examples:
In the first call to accept, abracadabra was typed at the keyboard. In the second call to accept, the user clicked on the keyboard-entered string of the first function. In both cases, the string object "abracadabra" was returned.

Typically, not all objects are acceptable as input. Only an object of the presentation type specified in the current accept function (or one of its subtypes) can be input. In other words, the accept function establishes the current input context. For example, if the call to accept specifies an integer presentation type, only an entered or displayed integer is acceptable. Numbers displayed as integer presentations would, in this input context, be sensitive, but those displayed as part of some other kind of presentation, such as a file pathname, would not. In this manner, accept controls the input context and the sensitivity of displayed presentations.

It is possible, however, to click on a presentation of a type different from the current input context and invoke a presentation translator that would produce a type acceptable to the input context. For example, you could make a presentation of a file pathname translate to an integer—say, its length—if you want. It is very common to translate to a command that operates on a presented object. For more information on presentation translators, see Section 6.5, “Predefined Presentation Types”.

We said previously that the range of acceptable input is typically restricted, but how restricted is up to you, the programmer. Using compound presentation types like and and or, as well as other predefined or specially devised presentation types, gives you a high degree of flexibility and control over the input context.

CLIM provides the following top-level operators for accepting typed input. The most general operator is with-input-context, and accept and accept-from-string support common idioms.

Note that, in general, CLIM accept operators do not insert newlines. If you want each call to accept to appear on a new line, use terpri.

*input-context* [Variable]
Summary: The current input context. This will be a list, each element of which corresponds to a single call to `with-input-context`. The first element of the list is the context established by the most recent call to `with-input-context`, and the last element is the least recent call to `with-input-context`. This ordering of input contexts is called “nesting.”

The exact format of the elements in the list is unspecified, but will typically be a list of a presentation type and a tag that corresponds to the point in the control structure of CLIM at which the input context was established. `*input-context*` and the elements in it may have dynamic extent.

```lisp
with-input-context

Arguments: (type &key override) (&optional object-var type-var event-var options-var)
form &body pointer-cases

Summary: Establishes an input context of presentation type type; this is done by binding `*input-context*` to reflect the new input context. When the boolean override is `nil` (the default), this invocation of `with-input-context` adds its context presentation type to the current context. In this way an application can solicit more than one type of input at the same time. Alternatively, when `override` is `t`, it overrides the current input context rather than nesting inside the current input context.

Type can be a presentation type abbreviation.

After establishing the new input context, `form` is evaluated. If no pointer gestures are made by the user during the evaluation of `form`, the values of `form` are returned. Otherwise, one of the `pointer-cases` is executed (based on the presentation type of the object that was clicked on) and its value is returned. (See the descriptions of `call-presentation-menu` and `throw-highlighted-presentation`.) `pointer-cases` is constructed like a `typecase` statement clause list whose keys are presentation types; the first clause whose key satisfies the condition `(presentation-subtypep type key)` is the one that is chosen.

During the execution of one of the `pointer-cases`, `object-var` is bound to the object that was clicked on (the first returned value from the presentation translator that was invoked), `type-var` is bound to its presentation type (the second returned value from the translator), `event-var` is bound to the pointer button event that was used. `options-var` is bound to any options that a presentation translator might have returned (the third value from the translator), and will be either `nil` or a list of keyword-value pairs. `object-var, type-var, event-var,` and `options-var` must all be symbols.

type, stream, and override are evaluated, but the others are not:
accept  [Function]
Arguments: type &key stream view default default-type provide-default insert-default replace-input history prompt prompt-mode display-default query-identifier activation-gestures additional-activation-gestures delimiter-gestures additional-delimiter-gestures
Summary: Requests input of type type from the stream stream, which defaults to *query-io*. accept returns two values, the object representing the input and its presentation type. type is a presentation type specifier, and can be an abbreviation. The other arguments and overall behavior of accept are as for accept-1.

accept first expands any presentation type abbreviations (type, default-type, and history), handles the interactions between the default, default type, and presentation history, prompts the user by calling prompt-for-accept, and then calls stream-accept on stream, type, and the remaining keyword arguments.

Note: The reason accept is specified as a three-function “trampoline” is to allow close tailoring of the behavior of accept. accept itself is the function that should be called by application programmers. stream-accept exists so that CLIM implementors can specialize on a per-stream basis. (For example, the behavior of accepting-values can be implemented by creating a special class of stream that turns calls to accept into fields of a dialog.) accept-1 is provided as a convenient function for the stream-accept methods to call when they require the default behavior.

stream-accept  [Generic Function]
Arguments: stream &key view default default-type provide-default insert-default replace-input history prompt prompt-mode display-default query-identifier activation-gestures additional-activation-gestures delimiter-gestures additional-delimiter-gestures
Summary: stream-accept is the per-stream implementation of accept, analogous to the relationship between read-char and stream-read-char. All extended input streams implement a method for stream-accept. The default method (on standard-extended-input-stream) simply calls accept-1.

The arguments and overall behavior of stream-accept are as for accept-1.

Presentation Types
accept-1

Arguments: stream type &key view default default-type provide-default insert-default replace-input history prompt prompt-mode display-default query-identifier activation-gestures additional-activation-gestures delimiter-gestures additional-delimiter-gestures

Summary: Requests input of type type from the stream stream. type must be a presentation type specifier. view is a view object that defaults to stream-default-view of stream. accept-1 returns two values, the object representing the input and its presentation type. (If frame-maintain-presentation-histories is true for the current frame, then the returned object is also pushed on to the presentation history for that object.)

accept-1 establishes an input context via with-input-context, and then calls the accept presentation method for type and view. accept allows input editing when called on an interactive stream; see Section 16.1 for a discussion of input editing. The call to accept will be terminated when the accept method returns or the user clicks on a sensitive presentation. The typing of an activation and delimiter character is typically one way in which a call to an accept method is terminated.

A top-level accept satisfied by keyboard input discards the terminating keyboard gesture (which will be either a delimiter or an activation gesture). A nested call to accept leaves the terminating gesture unread.

If the user clicked on a matching presentation, accept-1 will insert the object into the input buffer by calling presentation-replace-input on the object and type returned by the presentation translator, unless either the boolean replace-input is nil or the presentation translator returned an :echo option of nil. replace-input defaults to t, but this default is overridden by the translator explicitly returning an :echo option of nil.

If default is supplied, then it and default-type are returned as values from accept-1 when the input is empty. default-type must be a presentation type specifier. If default is not supplied and provide-default is t (the default is nil), then the default is determined by taking the most recent item from the presentation type history specified by history. If insert-default is t and there is a default, the default will be inserted into the input stream by calling presentation-replace-input. It will be editable.

history must be either nil, meaning that no presentation type history will be used, or a presentation type (or abbreviation) that names a history to be used for the call to accept. history defaults to type.

prompt can be t, which prompts by describing the type, nil, which suppresses prompting, or a string, which is displayed as a prompt (via write-string). The
default is \texttt{t}, which produces \texttt{Enter a type:} in a top-level call to \texttt{accept} or \texttt{“(type)”} in a nested call to \texttt{accept}.

If the boolean \texttt{display-default} is \texttt{t}, the default is displayed (if one was supplied). If \texttt{display-default} is \texttt{nil}, the default is not displayed. \texttt{display-default} defaults to \texttt{t} if \texttt{prompt} was provided; otherwise, it defaults to \texttt{nil}.

\texttt{prompt-mode} can be \texttt{:normal} (the default) or \texttt{:raw}, which suppresses putting a colon after the prompt and/or default in a top-level \texttt{accept} and suppresses putting parentheses around the prompt and/or default in a nested \texttt{accept}.

\texttt{query-identifier} is used within \texttt{accepting-values} to identify the field within the dialog.

\texttt{activation-gestures} is a list of gesture names that will override the current activation gestures, which are stored in \texttt{*activation-gestures*}. \texttt{additional-activation-gestures} can be supplied to add activation gestures without overriding the current ones. See Subsection 16.2 for a discussion of activation gestures.

\texttt{delimiter-gestures} is a list of gesture names that will override the current delimiter gestures, which are stored in \texttt{*delimiter-gestures*}. \texttt{additional-delimiter-gestures} can be supplied to add delimiter gestures without overriding the current ones. See Subsection 16.2 for a discussion of delimiter gestures.

\texttt{accept-from-string} \hspace{1cm} \textbf{[Function]}

\textbf{Arguments:} \texttt{type string} \&\texttt{key} \texttt{view default default-type start end}

\textbf{Summary:} Like \texttt{accept}, except that the input is taken from \texttt{string}, starting at the position specified by \texttt{start} and ending at \texttt{end}. \texttt{view}, \texttt{default}, and \texttt{default-type} are as for \texttt{accept}.

\texttt{accept-from-string} returns an object and a presentation type (as in \texttt{accept}), but also returns a third value, the index at which input terminated.

\texttt{prompt-for-accept} \hspace{1cm} \textbf{[Generic Function]}

\textbf{Arguments:} \texttt{stream type view} \&\texttt{rest accept-args} \&\texttt{allow-other-keys}

\textbf{Summary:} Called by \texttt{accept} to prompt the user for input of presentation type \texttt{type} on the stream \texttt{stream} for the view \texttt{view}. \texttt{accept-args} are all of the keyword arguments supplied to \texttt{accept}. The default method (on \texttt{standard-extended-input-stream}) simply calls \texttt{prompt-for-accept-1}.
prompt-for-accept-1

Arguments: stream type &key default default-type display-default prompt prompt-mode &allow-other-keys

Summary: Prompts the user for input of presentation type type on the stream stream.

If the boolean display-default is t, then the default is displayed; otherwise it is not. When the default is being displayed, default and default-type are taken as the object and presentation type of the default to display. display-default defaults to t if prompt is non-nil; otherwise, it defaults to nil.

If prompt is nil, no prompt is displayed. If it is a string, that string is displayed as the prompt. If prompt is t (the default), the prompt is generated by calling describe-presentation-type to produce a prompt of the form Enter a type: in a top-level call to accept, or “(type)” in a nested call to accept.

prompt-mode can be :normal (the default) or :raw, which suppresses putting a colon after the prompt and/or default in a top-level accept and suppresses putting parentheses around the prompt and/or default in a nested accept.

6.5 Predefined Presentation Types

This section documents predefined CLIM presentation types, presentation type options, and parameters. For more information on how to use these presentation types, see Section 6.2, “How to Specify a CLIM Presentation Type”.

Note that any presentation type with the same name as a Common Lisp type accepts the same parameters as the Common Lisp type (and additional parameters in a few cases).

6.5.1 Basic Presentation Types

These basic presentation types correspond to the Common Lisp types of the same name.

**t**

Summary: The supertype of all other presentation types.

**nil**

Summary: The supertype of all other presentation types.
Summary: The subtype of all other presentation types. This has no printed representation, and is useful only in writing “context independent” translators, that is, translators whose to-type is nil.

null [Presentation Type]
Summary: The presentation type that represents “nothing.” The single object associated with this type is nil, and its printed representation is “None.”

boolean [Presentation Type]
Summary: The presentation type that represents t or nil. The textual representation is “Yes” and “No,” respectively.

symbol [Presentation Type]
Summary: The presentation type that represents a symbol.

keyword [Presentation Type]
Summary: The presentation type that represents a symbol in the keyword package. It is a subtype of symbol.

blank-area [Presentation Type]
Summary: The type that represents all the places in a window where there is no presentation that is applicable in the current input context. CLIM provides a single “null presentation” as the object associated with this type.

*null-presentation* [Variable]
Summary: The null presentation, which occupies all parts of a window in which there are no applicable presentations. This will have a presentation type of blank-area.

6.5.2 Numeric Presentation Types
The following presentation types represent the Common Lisp numeric types of the same name.

number [Presentation Type]
Summary: The presentation type that represents a general number. It is the supertype of all the number types described here.

**complex**

Summary: The presentation type that represents a complex number.

**rational**

Arguments: &optional low high

Summary: The presentation type that represents either a ratio or an integer between `low` and `high`. Options to this type are `base` and `radix`, which are the same as for the `integer` type.

**integer**

Arguments: &optional low high

Summary: The presentation type that represents an integer between `low` and `high`. Options to this type are `base` (default 10) and `radix` (default nil), which correspond to *print-base* and *print-radix*, respectively. It is a subtype of rational.

**ratio**

Arguments: &optional low high.

The presentation type that represents a ratio between `low` and `high`. Options to this type are `base` and `radix`, which are the same as for the integer type. It is a subtype of rational.

**float**

Arguments: &optional low high.

The presentation type that represents a floating point number between `low` and `high`.

### 6.5.3 Character and String Presentation Types

These two presentation types can be used for reading and writing characters and strings.

**character**

[Presentation Type]
Summary: The presentation type that represents a Common Lisp character object.

**string** [Presentation Type]
Arguments: &optional length
Summary: The presentation type that represents a string. If length is specified, the string must have exactly that many characters.

### 6.5.4 Pathname Presentation Types

**pathname** [Presentation Type]
Summary: The presentation type that represents a pathname.

The options are default-version, which defaults to :newest, default-type, which defaults to nil, and merge-default, which defaults to t. If merge-default is nil, accept returns the exact pathname that was entered; otherwise, accept merges against the default and default-version. If no default is supplied, it defaults to *default-pathname-defaults*. pathname has a default preprocessor that merges the options into the default.

### 6.5.5 One-Of and Some-Of Presentation Types

The “one-of” and “some-of” presentation types can be used to accept and present one or more items from a set of items. The set of items can be specified as a “rest” argument, a sequence, or an alist.

This table summarizes single (“one-of”) and multiple (“some-of”) selection presentation types. Each row represents a type of presentation. Columns contain the associated single and multiple selection presentation types.
The presentation type that selects one from a finite set of possibilities, with “completion” of partial inputs. Several types are implemented in terms of the completion type, including token-or-type, null-or-type, member, member-sequence, and member-alist.

sequence is a list or vector whose elements are the possibilities. Each possibility has a printed representation, called its name, and an internal representation, called its value. accept reads a name and returns a value. present is given a value and outputs a name.

test is a function that compares two values for equality. The default is eql.

value-key is a function that returns a value, given an element of sequence. The default is identity.

The following presentation type options are available:

- name-key is a function that returns a name as a string, given an element of sequence. The default is a function that behaves as follows:
  - string -> the string
  - null -> “NIL”
  - cons -> string of the car
  - symbol -> string-capitalise of its name
  - otherwise -> princ-to-string of it

### Table 3. One-Of and Some-Of Selection Presentation Types

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>most general</td>
<td>completion</td>
<td>subset-completion</td>
</tr>
<tr>
<td>&amp;rest elements</td>
<td>member</td>
<td>subset</td>
</tr>
<tr>
<td>sequence</td>
<td>member-sequence</td>
<td>subset-sequence</td>
</tr>
<tr>
<td>alist</td>
<td>member-alist</td>
<td>subset-alist</td>
</tr>
</tbody>
</table>

**completion**

Arguments: sequence &key test value-key

Summary: The presentation type that selects one from a finite set of possibilities, with “completion” of partial inputs. Several types are implemented in terms of the completion type, including token-or-type, null-or-type, member, member-sequence, and member-alist.
• documentation-key is a function that returns either nil or a descriptive string, given an element of sequence. The default always returns nil.
• test, value-key, name-key, and documentation-key must have indefinite extent.
• partial-completers is a possibly empty list of characters that delimit portions of a name that can be completed separately. The default is a list of one character, # \ Space.

member

Arguments: &rest elements
Summary: The presentation type that specifies one of elements. The options are the same as for completion.

member-sequence

Arguments: sequence &key test
Summary: Like member, except that the set of possibilities is the sequence sequence. The parameter test and the options are the same as for completion.

member-alist

Arguments: alist &key test
Summary: Like member, except that the set of possibilities is the alist alist. Each element of alist is either an atom, as in member-sequence, or a list whose car is the name of that possibility and whose cdr is one of the following:

• The value (which must not be a cons)
• A list of one element, the value
• A property list that can contain the following properties:
  :value—the value
  :documentation—a descriptive string

The test parameter and the options are the same as for completion except that value-key and documentation-key default to functions that support the specified alist format.

subset-completion

Arguments: sequence &key test value-key
Summary: The type that selects one or more from a finite set of possibilities, with “completion” of partial inputs. The parameters and options are the same as for completion.
tion, plus the additional options separator and echo-space, which are as for the sequence type. The subset types that follow are implemented in terms of the subset-completion type.

subset [Presentation Type Abbreviation]
Arguments: &rest elements
Summary: The presentation type that specifies a subset of elements. Values of this type are lists of zero or more values chosen from the possibilities in elements. The printed representation is the names of the elements separated by commas. The options are the same as for completion.

subset-sequence [Presentation Type Abbreviation]
Arguments: sequence &key test
Summary: Like subset, except that the set of possibilities is the sequence sequence. The parameter test and the options are the same as for completion.

subset-alist [Presentation Type Abbreviation]
Arguments: alist &key test
Summary: Like subset, except that the set of possibilities is the alist alist.

6.5.6 Sequence Presentation Types

The following two presentation types can be used to accept and present a sequence of objects.

sequence [Presentation Type]
Arguments: type
Summary: The presentation type that represents a sequence of elements of type type. type can be a presentation type abbreviation. The printed representation of a sequence type is the elements separated by commas. accept returns a list.

The options to this type are separator and echo-space. separator is used to specify a character that will act as the separator between elements of the sequence; the default is the comma character #\, . echo-space is t or nil; when it is t (the
default) a space will be automatically inserted into the input buffer when the user
types a separator character.

**sequence-enumerated**

[Presentation Type]

Arguments: \&rest types

Summary: *sequence-enumerated* is like *sequence*, except that the type of each element in
the sequence is individually specified. The elements of *types* can be presentation
type abbreviations. *accept* returns a list.

The options to this type are *separator* and *echo-space*, which are as for the
*sequence* type.

### 6.5.7 Constructor Presentation Types

**or**

[Presentation Type]

Arguments: \&rest types

Summary: The presentation type that is used to specify one of several types, for example,
(or (member :all :none) integer). The elements of *types* can be presentation
type abbreviations. *accept* returns one of the possible types as its
second value, not the original *or* presentation type specifier.

**and**

[Presentation Type]

Arguments: \&rest types

Summary: The type that is used for “multiple inheritance.” *and* is frequently used in con-
junction with *satisfies*, for example: (and integer (satisfies oddp)). The elements of *types* can be presentation type abbreviations.

The *and* type has special syntax that supports the two “predicates,” *satisfies* and
not. *satisfies* and not cannot stand alone as presentation types and cannot be first
in *types*. *not* can surround either *satisfies* or a presentation type.

The first type in *types* is the type whose methods will be used during calls to
*accept* and *present*.

### 6.5.8 Compound Presentation Types

The following compound presentation types are provided because they implement some
commom idioms.
**token-or-type** [Presentation Type Abbreviation]

Arguments: tokens type

Summary: A compound type that is used to select one of a set of special tokens, or an object of type type. tokens is anything that can be used as the sequence parameter to member-alist; typically it is a list of symbols.

**null-or-type** [Presentation Type Abbreviation]

Arguments: type

Summary: A compound type that is used to select nil, whose printed representation is the special token “None,” or an object of type type.

**type-or-string** [Presentation Type Abbreviation]

Arguments: type

Summary: A compound type that is used to select an object of type type or an arbitrary string, for example: (clim:type-or-string integer). Any input that accept cannot parse as the representation of an object of type type is returned as a string.

### 6.5.9 Command and Form Presentation Types

The command and form presentation types are complex types provided primarily for use by the top-level interactor of an application.

**expression** [Presentation Type]

Summary: The presentation type used to represent any Lisp object. The textual view of this type looks like what the standard print and read functions produce and accept. The standard print and read functions produce and accept the textual view of this type.

A separate presentation history for each instance of an application frame is maintained for the expression presentation type.

**form** [Presentation Type]
Summary: The presentation type used to represent a Lisp form. This is a subtype of expression and is equivalent to it, except that some presentation translators produce quote forms.

**command**  
Arguments: &key command-table  
Summary: The presentation type used to represent a command processor command and its arguments.  

A separate presentation history for each instance of an application frame is maintained for the command presentation type.

**command-name**  
Arguments: &key command-table  
Summary: The presentation type used to represent the name of a command processor command in the command table command-table.

**command-or-form**  
Arguments: &key command-table  
Summary: The presentation type used to represent either a Lisp form or a command processor command and its arguments.

### 6.6 Functions That Operate on CLIM Presentation Types

These are some general-purpose functions that operate on CLIM presentation types.

**describe-presentation-type**  
Arguments: type &optional stream plural-count  
Summary: Describes the presentation type type on the stream, which defaults to *standard-output*. If stream is nil, a string containing the description is returned. plural-count is either nil (that is, the description should be the singular form of the name), t (meaning that the description should the plural form of the name), or an integer greater than zero (the number of items to be described). The default is 1.

*type* can be a presentation type abbreviation.
**presentation-type-name**

Arguments: *type*

Summary: Returns the presentation type name of the presentation type specifier *type*. This function is provided as a convenience. It could be implemented as follows:

```lisp
(defun presentation-type-name (type)
  (with-presentation-type-decoded (name) type name))
```

**presentation-type-parameters**

Arguments: *type-name* &optional *env*

Summary: Returns a lambda-list of the parameters specified when the presentation type or presentation type abbreviation whose name is *type-name* was defined. *type-name* is a symbol or a class. *env* is a macro-expansion environment, as in **find-class**.

**presentation-type-options**

Arguments: *type-name* &optional *env*

Summary: Returns the list of options specified when the presentation type or presentation type abbreviation whose name is *type-name* was defined. This does not include the standard options unless the presentation-type definition mentioned them explicitly. *type-name* is a symbol or a class. *env* is a macro-expansion environment, as in **find-class**.

**presentation-typep**

Arguments: *object* type

Summary: Returns t if *object* is of the type specified by *type*, otherwise returns nil. *type* may not be a presentation type abbreviation. This is analogous to the Common Lisp **typep** function.

**with-presentation-type-decoded**

Arguments: (name-var &optional parameters-var options-var) type &body body

Summary: The specified variables are bound to the components of the presentation type specifier, the forms in body are executed, and the values of the last form are returned. name-var, if non-nil, is bound to the presentation type name. parameters-var, if non-nil, is bound to a list of the parameters. options-var, if non-nil, is bound to a list of the options. When supplied, name-var, parameters-var, and options-var must be symbols.

The name-var, parameters-var, and options-var arguments are not evaluated. body may have zero or more declarations as its first forms.
with-presentation-type-options [Macro]
Arguments: (type-name type) &body body
Summary: Variables with the same name as each option in the definition of the presentation type are bound to the option values in type, if present, or else to the defaults specified in the definition of the presentation type. The forms in body are executed in the scope of these variables and the values of the last form are returned.

The value of the form type must be a presentation type specifier whose name is type-name. The type-name and type arguments are not evaluated. body may have zero or more declarations as its first forms.

with-presentation-type-parameters [Macro]
Arguments: (type-name type) &body body
Summary: Variables with the same name as each parameter in the definition of the presentation type are bound to the parameter values in type, if present, or else to the defaults specified in the definition of the presentation type. The forms in body are executed in the scope of these variables and the values of the last form are returned.

The value of the form type must be a presentation type specifier whose name is type-name. The type-name and type arguments are not evaluated. body may have zero or more declarations as its first forms.

presentation-type-specifier-p [Function]
Arguments: object
Summary: Returns t if object is a valid presentation type specifier; otherwise, it returns nil.

presentation-type-of [Function]
Arguments: object
Summary: Returns a presentation type of which object is a member, in particular the most specific presentation type that can be conveniently computed and is likely to be useful to the programmer. This is often the class name of the class of the object.

presentation-type-of returns an expression when possible and t otherwise.

This is analogous to the Common Lisp type-of function.

presentation-subtypep [Function]
Arguments: type putative-supertype
Summary: Answers the question “is the type specified by the presentation type specifier type a subtype of the type specified by the presentation type specifier putative-supertype?” presentation-subtypep returns two values, subtypep and known-p. When known-p is t, subtypep can be either t (meaning that type is definitely a subtype of putative-supertype) or nil (meaning that type is definitely not a subtype of putative-supertype). When known-p is nil, then subtypep must also be nil; this means that the answer cannot reliably be determined.

type may not be a presentation type abbreviation.

This is analogous to the Common Lisp subtypep function.

map-over-presentation-type-supertypes
Arguments: function type
Summary: Calls the function function on the presentation type specifier type and each of its supertypes. function is called with two arguments, the name of a type and a presentation type specifier for that type with the parameters and options filled in. function has dynamic extent; its two arguments are permitted to have dynamic extent. The traversal of the type lattice is done in the order specified by the CLOS class precedence rules, and visits each type in the lattice exactly once.

presentation-type-direct-supertypes
Arguments: type
Summary: Returns a sequence of the names of all the presentation types that are direct supertypes of the presentation type specifier type, or nil if type has no supertypes. The consequences of modifying the returned sequence are unspecified.

find-presentation-type-class
Arguments: name &optional (errorp t) environment
Summary: Returns the class corresponding to the presentation type named name, which must be a symbol or a class object. errorp and environment are as for find-class.

class-presentation-type-name
Arguments: class &optional environment
Summary: Returns the presentation type name corresponding to the class class. This is the inverse of find-presentation-type-class. environment is as for find-class.

default-describe-presentation-type
Arguments: description stream plural-count

<Bold>142CLIM User Guide
Summary: Performs the default actions for describe-presentation-type, notably pluralization and prepending an indefinite article if appropriate. description is a string or a symbol, typically the :description presentation type option or the :description option to define-presentation-type. plural-count is as for describe-presentation-type.

make-presentation-type-specifier [Function]
Arguments: type-name-and-parameters &rest options
Summary: A convenient way to assemble a presentation type specifier with only non-default options included. For a full description of this function, see the end of Subsection 7.2.1, “Presentation Methods in CLIM.”
144CLIM User Guide
Chapter 7  Defining a New Presentation Type
7.1 Conceptual Overview of Defining a New Presentation Type

CLIM’s standard set of presentation types will be useful in many cases, but most applications will need customized presentation types to represent the objects modeled in the application.

In defining a presentation type, you define all the user interface components of the entity:

- A displayed representation, for example, textual or graphical
- Pointer sensitivity for user input via the pointer
- A textual representation for user input via the keyboard (optional)

In other words, in one place you provide all the information about an object necessary to display it to the user and to accept it as input from the user.

The set of presentation types forms a type lattice, an extension of the Common Lisp CLOS type lattice. When a new presentation type is defined as a subtype of another presentation type, it inherits all the attributes of the supertype except those explicitly overridden in the definition.

To define a new presentation type, you follow these steps:

1. Use the define-presentation-type macro.
   a. Name the new presentation type.
   b. Supply parameters that further restrict the type (if appropriate).
   c. Supply options that affect the appearance of the type (if appropriate).
   d. State the supertypes of this type, to make use of inheritance (if appropriate).
2. Define the CLIM presentation methods.
   a. Specify how objects are displayed with a present presentation method. (You must define a present method, unless the new presentation type inherits a method that is appropriate for it.)
   b. Specify how objects are parsed with an accept presentation method. (In most cases, you must define an accept method, unless the new presentation type inherits a method that is appropriate for it. If it will never be necessary to enter
the object by typing its representation on the keyboard, you don’t need to pro-
vide this method.)

c. Specify the type/subtype relationships of this type and its related types, if nec-
essary, with presentation-typep and presentation-subtypep presentation
methods. (You must define or inherit these methods when defining a presenta-
tion type that has parameters.)

7.1.1 CLIM Presentation Type Inheritance

Every presentation type is associated with a CLOS class. In the common case, the name of
the presentation type is a class object or the name of a class, and that class is not a
clos:built-in-class. In this case, the presentation type is associated with that CLOS class.

Otherwise, define-presentation-type defines a class with metaclass clim:presenta-
tion-type-class and superclasses determined by the presentation type definition. This class
is not named name, since that could interfere with built-in Common Lisp types such as and,
member, and integer. clos:class-name of this class returns a list (presentation-type
name). clim:presentation-type-class is a subclass of clos:standard-class.

Note: If the same name is defined with both clos:defclass (or defstruct) and
define-presentation-type, the clos:defclass (or defstruct) must be done
first.

Every CLOS class (except for built-in classes) is a presentation type, as is its name. If it has
not been defined with define-presentation-type, it allows no parameters and no options.
As in CLOS, inheriting from a built-in class does not work unless you specify the same in-
heritance that the built-in class already has; you may want to do this in order to add presen-
tation-type parameters to a built-in class.

If you define a presentation type that does not have the same name as a CLOS class, you
must define a presentation-typep presentation method for it. The function (as opposed to
the presentation method) presentation-typep uses find-class if the presentation type is
piggybacking on a CLOS type. Otherwise it depends on the user-defined presentation meth-
od.

If you define a presentation type that has parameters, you must define a presentation-sub-
typep for it. As noted previously, CLOS does not allow you to parameterize types, so you
must provide a presentation-subtype method even for presentation types based on CLOS
classes.
Note that CLIM itself depends on these methods for its own presentation-based utilities.

If your presentation type has the same name as a class, doesn’t have any parameters or options, doesn’t have a history, and doesn’t need a special description, you do not need to call `define-presentation-type`.

During method combination, presentation type inheritance is used both to inherit methods (“what parser should be used for this type?”), and to establish the semantics for the type (“what objects are sensitive in this context?”). Inheritance of methods is the same as in CLOS and thus depends only on the type name, not on the parameters and options.

Presentation type inheritance translates the parameters of the subtype into a new set of parameters for the supertype, and translates the options of the subtype into a new set of options for the supertype.

### 7.1.2 Defining an Accept for a Structure With Several Fields

The following code shows how to define an `accept` for a structure (instance) with several fields. That `accept` is then used within another similar `accept` call.

A presentation type called `ticket` is defined. The `accept` method has two recursive calls to `accept`, one to read the name of a candidate for president and another to read the name of the running mate. We provide two possible `accept` methods; in order to compare them, you will have to compile first one and then the other. The first reads the two names separated by a comma on the same line. The second reads the two names on separate lines, delimited by `RETURN`. They both do completion within the field. That is, if you do `(accept 'ticket :stream win)` with the first `accept` method, and type "Bu,Qu<RETURN>", the screen appearance will be "Bush,Quayle" and the return value will be `(BUSH QUAYLE)`.

If you use the second `accept` method and type:

```
"Cl
Go
"
```

the window will contain:

```
<Bold>148CLIM User Guide
```
"Clinton Gore"

and the return value will be (CLINTON GORE).

This example also demonstrates simple cross-field constraints by insisting that the two candidates be of the same party.

For key implementation details, read the comments in the code.

(define-presentation-type ticket ()

(setf (get 'bush 'party) 'republican)
(setf (get 'quayle 'party) 'republican)
(setf (get 'clinton 'party) 'democrat)
(setf (get 'gore 'party) 'democrat)

;; separated by comma version
(define-presentation-method accept ((type ticket) stream view &key &allow-other-keys)
  (declare (ignore view))
  (let ((president (accept '(member bush clinton) :stream stream :prompt nil
                   ;; add comma as a completing delimiter
                   :blip-characters '(#,))))
    ;; Make sure that the names were separated by a comma
    (unless (eql (read-gesture :stream stream) #,)
      (simple-parse-error "Ticket members must be separated by commas")))
  (let ((veep (accept '(member quayle gore) :stream stream :prompt nil)))
    ;; Validate party affiliations
    (unless (eql (get president 'party) (get veep 'party))
      (simple-parse-error "Ticket members must be of the same party"))
    (list president veep))))

;;; Separated by Return version
(define-presentation-method accept ((type ticket) stream view &key &allow-other-keys)
  (declare (ignore view))
  (let ((president (accept '(member bush clinton) :stream stream :prompt nil
                   ;; Remove Newline from activation characters
                   :activation-characters '())
                   ;; Add Newline as a delimiter, so that we get
                   ;; completion and move-to-next-field behavior
                   ;; when Return is typed.
                   :blip-characters '('#\Return #\Newline))))

Defining a New Presentation Type
(unless (eql (read-gesture :stream stream) #\Newline)
  (simple-parse-error
   "Ticket members must be entered on separate lines"))
(let ((veep (accept '(member quayle gore) :stream stream :prompt nil)))
  ;; Validate party affiliations
  (unless (eql (get president 'party) (get veep 'party))
    (simple-parse-error "Ticket members must be of the same party"))
  (list president veep))))

7.2 CLIM Operators for Defining New Presentation Types

define-presentation-type

Arguments: name parameters &key options inherit-from description history
parameters-are-types

Summary: Defines a presentation type whose name is the symbol or class name and whose
parameters are specified by the lambda-list parameters. These parameters are visible within inherit-from and within the methods created with define-presentation-method. For example, the parameters are used by presentation-typep and presentation-subtypep methods to refine their tests for type inclusion.

options is a list of option specifiers. It defaults to nil. An option specifier is either
a symbol or a list of the form (symbol &optional default supplied-p presentation-type accept-options), where symbol, default, and supplied-p are as in a normal lambda-list. If presentation-type and accept-options are present, they specify how to accept a new value for this option from the user. symbol can also be specified in the (keyword variable) form allowed for Common Lisp lambda lists.

symbol is a variable that is visible within inherit-from and within most of the
methods created with define-presentation-method. The keyword corresponding
to symbol can be used as an option in the third form of a presentation type specifier. An option specifier for the standard option :description is automatically added to options if an option with that keyword is not present; however, it does not produce a visible variable binding.

Unsupplied optional or keyword parameters default to * (as in deftype) if no
default is specified in parameters. Unsupplied options default to nil if no default is
specified in options.

inherit-from is a form that evaluates to a presentation type specifier for another
type from which the new type inherits. inherit-from can access the parameter
Defining a New Presentation Type

variables bound by the parameters lambda list and the option variables specified by options. If name is or names a CLOS class (other than a built-in-class), then inherit-from must specify the class’s direct superclasses (using and to specify multiple inheritance). It is useful to do this when you want to parameterize previously defined CLOS classes.

If inherit-from is unsupplied, the default behavior is that if name is or names a CLOS class, then the type inherits from the presentation type corresponding to the direct superclasses of that CLOS class (using and to specify multiple inheritance). Otherwise, the type named by name inherits from standard-class.

description is a string or nil. This should be the term for an instance for the type being defined. If it is nil or unsupplied, a description is automatically generated; it will be a “prettied up” version of the type name, for example, small-integer would become "small integer". You can also write a describe-presentation-type presentation method. description is implemented by the default describe-presentation-type method, so description only works in presentation types where that default method is not shadowed.

history can be t (the default), meaning that this type has its own history of previous inputs; nil, meaning that this type keeps no history; or the name of another presentation type whose history is shared by this type. More complex histories can be specified by writing a presentation-type-history presentation method.

If the boolean parameters-are-types is t, this means that the parameters to the presentation type are themselves presentation types. If they are not presentation types, parameters-are-types should be supplied as nil. Types such as and, or, and sequence will specify this as t.

Every presentation type must define or inherit presentation methods for accept and present if the type is going to be used for input and output. For presentation types that are only going to be used for input via the pointer, the accept need not be defined.

If a presentation type has parameters, it must define presentation methods for presentation-typep and presentation-subtypep that handle the parameters, or inherit appropriate presentation methods. In many cases it should also define presentation methods for describe-presentation-type and presentation-type-specifier-p.

There are certain restrictions on the inherit-from form, to allow it to be analyzed at compile time. The form must be a simple substitution of parameters and options into positions in a fixed framework. It cannot involve conditionals or computations that depend on valid values for the parameters or options; for
example, it cannot require parameter values to be numbers. It cannot depend on the dynamic or lexical environment. The form will be evaluated at compile time with uninterned symbols used as dummy values for the parameters and options. In the type specifier produced by evaluating the form, the type name must be a constant that names a type, the type parameters cannot derive from options of the type being defined, and the type options cannot derive from parameters of the type being defined. All presentation types mentioned must be already defined. and can be used for multiple inheritance, but or, not, and satisfies cannot be used.

None of the arguments, except inherit-from, are evaluated.

### 7.2.1 Presentation Methods in CLIM

Use **define-presentation-method** to define presentation methods.

```lisp
define-presentation-method
```

**Arguments:** name qualifiers* specialized-lambda-list &body body

**Summary:** Defines a presentation method for the function named name on the presentation type named in specialized-lambda-list.

- **specialized-lambda-list** is a CLOS specialized lambda list for the method, and its contents vary depending on what name is. qualifiers* is zero or more of the usual CLOS method qualifier symbols. **define-presentation-method** supports standard method combination (the :before, :after, and :around method qualifiers).

- **body** defines the body of the method. body may have zero or more declarations as its first forms.

All presentation methods have an argument named type that must be specialized with the name of a presentation type. The value of type is a presentation type specifier, which can be for a subtype that inherited the method.

All presentation methods except those for **presentation-subtypep** have lexical access to the parameters from the presentation type specifier. Presentation methods for the functions accept, present, describe-presentation-type, presentation-type-specifier-p, and accept-present-default also have lexical access to the options from the presentation type specifier.
Presentation methods inherit and combine in the same way as ordinary CLOS methods. However, they do not resemble ordinary CLOS methods with respect to the type argument. The parameter specializer for type is handled in a special way, and presentation method inheritance arranges the type parameters and options seen by each method.

For example, consider three types int, rrat, and num defined as follows:

```
(define-presentation-type int (low high) :inherit-from '(rrat ,high ,low))
(define-presentation-method presentation-typep :around (object (type int))
  (and (call-next-method)
       (integerp object)
       (<= low object high)))

(define-presentation-type rrat (high low) :inherit-from 'num)
(define-presentation-method presentation-typep :around (object (type rrat))
  (and (call-next-method)
       (rationalp object)
       (<= low object high)))

(define-presentation-type num ())
(define-presentation-method presentation-typep (object (type num))
  (numberp object))
```

(If the user were to evaluate the form `presentation-typep X '(int 1 5))`, then the type parameters will be (1 5) in the presentation-typep method for int, (5 1) in the method for rrat, and nil in the method for num. The value for type will be `((int 1 5))` in each of the methods.

Following are the names of the various presentation methods defined by define-presentation-method, along with the lambda-list for each method. For all of the presentation methods, the type will always be specialized. Where appropriate, view may be specialized as well. The other arguments are not usually specialized.

**accept**

Arguments: type stream view &key default default-type

Summary: This presentation method is responsible for “parsing” the representation of type for a particular view view on the stream stream. The accept method returns a sin-
The single value (the object that was “parsed”), or two values, the object and its type (a presentation type specifier). The method’s caller takes care of establishing the input context, defaulting, prompting, and input editing.

The accept method can specialize on the view argument in order to define more than one input view for the data.

Note that accept presentation methods can call the function accept recursively. In this case, the programmer should be careful to specify nil for :prompt and :display-default unless recursive prompting is really desired.

**present**

**Arguments:** object type stream view &key acceptably for-context-type

**Summary:** This presentation method is responsible for displaying the representation of object having type type for a particular view view; see the function accept.

The present method can specialize on the view argument in order to define more than one view of the data. For example, a spreadsheet program might define a presentation type for revenue, which can be displayed either as a number or a bar of a certain length in a bar graph. Typically, at least one canonical view should be defined for a presentation type.

**describe-presentation-type**

**Arguments:** type stream plural-count

**Summary:** This presentation method is responsible for textually describing the type type. stream is a stream, and will not be nil as it can be for the describe-presentation-type function.

**presentation-type-specifier-p**

**Arguments:** type

**Summary:** This presentation method is responsible for checking the validity of the parameters and options for the presentation type type. The default method returns t.

**presentation-typep**

**Arguments:** object type

**Summary:** This presentation method is called when the presentation-typep function requires type-specific knowledge. If the type name in the presentation type type is or names a CLOS class, the method is called only if object is a member of the class and type contains parameters. The method simply tests whether object is a member of the subtype specified by the parameters. For non-class types, the method is always called.
For example, the type method will not get called in `(presentation-typep 1.0 `(integer 10))` because 1.0 is not an integer. The method will get called by `(presentation-typep 10 `(integer 0 5))`.

**presentation-subtypep**

Arguments: `type putative-supertype`

Summary: This presentation method is called when the `presentation-subtypep` function requires type-specific knowledge. `presentation-subtypep` walks the type lattice (using `map-over-presentation-supertypes`) to determine whether or not the presentation type `type` is a subtype of the presentation type `putative-supertype`, without looking at the type parameters. When a supertype of `type` has been found whose name is the same as the name of `putative-supertype`, then the `subtypep` method for that type is called in order to resolve the question by looking at the type parameters (that is, if the `subtypep` method is called, `type` and `putative-supertype` are guaranteed to be the same type, differing only in their parameters). If `putative-supertype` is never found during the type walk, then `presentation-subtypep` will never call the `presentation-subtypep` presentation method for `putative-supertype`.

Unlike all other presentation methods, `presentation-subtypep` receives a `type` argument that has been translated to the presentation type for which the method is specialized; `type` is never a subtype. The method is only called if `putative-supertype` has parameters and the two presentation type specifiers do not have equal parameters. The method must return the two values that `presentation-subtypep` returns.

Since `presentation-subtypep` takes two type arguments, the parameters are not lexically available as variables in the body of a presentation method.

**map-over-presentation-type-supertypes**

Arguments: `function type`

Summary: This method is called in order to apply `function` to the superclasses of the presentation type `type`.

**accept-present-default**

Arguments: `type stream view default default-supplied-p present-p query-identifier`

Summary: This method specializes the kind of default that is to be presented to the user. It is called when `accept` turns into `present` inside `accepting-values`. The default method calls `present` or `describe-presentation-type`, depending on whether `default-supplied-p` is `t` or `nil`, respectively.
The boolean default-supplied-p will be t only in the case when the :default option was explicitly supplied in the call to accept that invoked accept-present-default.

present-p and query-identifier are arguments that are called internally by the accept-values mechanism that this method needs to handle. The form of present-p as it is handed down (internally) from accepting-values is a list of the presentation type of the accepting-values query (accept-values-choice) and the query object itself, e.g., (list 'accept-values-choice <AV-query-object>). The value of query-identifier is an internal accept-values query identifier object.

**presentation-type-history**  
Arguments: `type`  
Summary: This method returns a history object for the presentation type type, or nil if there is none.

**presentation-refined-position-test**  
Arguments: `(record presentation-type x y)`  
Summary: This method is supplied when the user wants a more precise test of whether the supplied coordinate arguments (x and y) are “contained” by the record argument. Without this test, whether or not a position is within a record is determined by simply by seeing if the position is inside the bounding-rectangle of that record.

**highlight-presentation**  
Arguments: `type record stream state`  
Summary: This method is responsible for drawing a highlighting box around the presentation record on the output recording stream stream. state will be either :highlight or :unhighlight.

See Section 7.4, “Advanced Topics,” for more in-depth material relating to defining presentation methods.

### 7.2.2 CLIM Operators for Defining Presentation Type Abbreviations

You can define an abbreviation for a presentation type for the purpose of naming a commonly used cliche. The abbreviation is simply another name for a presentation type specifier.
Exported functions that call `expand-presentation-type-abbreviation` allow abbreviations.

- `accept`
- `accept-from-string`
- `with-output-as-presentation`
- `with-input-context`
- `present`
- `describe-presentation-type`
- `presentation-type-history`
- `presentation-default-preprocessor`
- `define-presentation translator`

`define-presentation-type-abbreviation`  
**[Macro]**

**Arguments:** `name parameters equivalent-type &key options`

**Summary:** Defines a presentation type that is an abbreviation for the presentation type specifier that is the value of `equivalent-type`. Where presentation type abbreviations are allowed, `equivalent-type` and abbreviations are exactly equivalent and can be used interchangeably.

`name` must be a symbol and must not be the name of a CLOS class.

The `equivalent-type` form might be evaluated at compile time if presentation type abbreviations are expanded by compiler optimizers. Unlike `inherit-from`, `equivalent-type` can perform arbitrary computations and is not called with dummy parameter and option values. The type specifier produced by evaluating `equivalent-type` can be a real presentation type or another abbreviation. If the type specifier doesn’t include the standard option `:description`, the option is automatically copied from the abbreviation to its expansion.

Note that you cannot define any presentation methods on a presentation type abbreviation. If you need methods, use `define-presentation-type` instead.

`define-presentation-type-abbreviation` is used to name a commonly used cliche. For example, a presentation type to read an octal integer might be defined as:

---

Defining a New Presentation Type
(define-presentation-type-abbreviation octal-integer
  (&optional low high)
  '((integer ,low ,high) :base 8
    :description "octal integer"))

None of the arguments, except equivalent-type, is evaluated.

When writing presentation type abbreviations, it is sometimes useful to let CLIM include or exclude defaults for parameters and options. In some cases, you may also find it necessary to “expand” a presentation type abbreviation. The following three functions are useful in these circumstances.

expand-presentation-type-abbreviation-1
[Function]
Arguments: type &optional environment
Summary: If the presentation type specifier type is a presentation type abbreviation, or is an and, or, sequence, or sequence-enumerated that contains a presentation type abbreviation, then expand-presentation-type-abbreviation-1 expands the type abbreviation once and returns two values, the expansion and t. If type is not a presentation type abbreviation, then the values type and nil are returned. env is a macro-expansion environment, as in macroexpand.

expand-presentation-type-abbreviation
[Function]
Arguments: type &optional environment
Summary: expand-presentation-type-abbreviation is like expand-presentation-type-abbreviation-1, except that type is repeatedly expanded until all presentation type abbreviations have been removed.

make-presentation-type-specifier
[Function]
Arguments: type-name-and-parameters &rest options
Summary: A convenient way to make a presentation type specifier including only non-default options. This is only useful for abbreviation expanders, not for the :inherit-from clause of define-presentation-type. type-name-and-parameters is a presentation type specifier that must be in the form of:

(type-name parameters...)

options is a list of alternating keywords and values that are added as options to the specifier. If a value is equal to type-name’s default, that option is omitted, producing a more concise presentation type specifier.
7.3 Using Views With CLIM Presentation Types

The **present** and **accept** presentation methods can define more than one view of the data. For example, a spreadsheet program might define a presentation type for revenue, which can be displayed either as a number or as a bar of a certain length in a bar graph. These two views might be implemented by specializing the view arguments for the **textual-menu-view** class and the user-defined **bar-graph-view** class. Typically, at least one canonical view should be defined for a presentation type. For example, the **present** method for the **textual-menu-view** view should be defined if the programmer wants to allow objects of that type to be displayed textually. A more concrete example is the dialog view of the **member** presentation type, which presents the choices in a “radio push-button” style.

CLIM currently supports textual, menu, and dialog views. Operators for views of CLIM presentation types are listed as follows.

**view**

*Protocol Class*

Summary: The protocol class for view objects. If you want to create a new class that behaves like a view, it should be a subclass of **view**. Subclasses of **view** must obey the view protocol. All of the view classes are immutable.

**viewp**

*Function*

Arguments: *object*

Summary: Returns `t` if *object* is a view; otherwise, it returns `nil`.

**stream-default-view**

*Generic Function*

Arguments: *stream*

Summary: Returns the default view for the extended stream *stream*. **accept** and **present** get the default value for the **view** argument from this.

**setf stream-default-view**

*Generic Function*

Arguments: *view stream*

Summary: Changes the default view for *stream* to the view *view*.

**textual-menu-view**

*Class*

Summary: The class that represents the default view used inside **menu-choose** for frame managers that are not using a gadget-type look and feel.

Defining a New Presentation Type
textual-dialog-view  [Class]
Summary: The class that represents the default view used inside accepting-values dialogs for frame managers that are not using a gadget-type look and feel.

gadget-menu-view  [Class]
Summary: The class that represents the default view used inside menu-choose for frame managers using a gadget-type look and feel.

gadget-dialog-view  [Class]
Summary: This subclass of gadget-view represents the default view used inside accepting-values dialogs for frame managers that are using a gadget-type look and feel.

pointer-documentation-view  [Class]
Summary: The class that represents the default view that is used when computing pointer documentation.

+textual-menu-view+  [Constant]
+textual-dialog-view+  [Constant]
+gadget-menu-view+  [Constant]
+gadget-dialog-view+  [Constant]
+pointer-documentation-view+  [Constant]
Summary: These are objects of class textual-menu-view, textual-dialog-view, gadget-menu-view, gadget-dialog-view, and pointer-documentation-view, respectively.

7.4 Advanced Topics

Material in this section is advanced; most CLIM programmers can skip this section entirely.
The following constructs apply to defining presentation types, discussed in Section 7.2.

presentation-default-preprocessor  [Presentation Method]
Arguments: default type &key default-type
Summary: This method is responsible for taking the object default and coercing it to match the presentation type type (which is the type being accepted) and default-type (which is the presentation type of default). This is useful when you want to change the default gotten from the presentation type’s history so that it conforms to parameters or options in type and default-type. The method returns two values, the new object to be used as the default, and a new presentation type, which should be at least as specific as type.

define-presentation-generic-function
Arguments: generic-function-name presentation-function-name lambda-list &rest options
Summary: Defines a generic function that will be used for presentation methods. generic-function-name is a symbol that names the generic function that will be used internally by CLIM for the individual methods. presentation-function-name is a symbol that names the function that programmers will call to invoke the method, and lambda-list and options are as for defgeneric.

There are some “special” arguments in lambda-list that the presentation type system knows about. The first argument in lambda-list must be either type-key or type-class; CLIM uses this argument to implement method dispatching. The second argument may be parameters, meaning that when the method is invoked, the type parameters will be passed to it. The third argument may be options, meaning that when the method is invoked, the type options will be passed to it. Finally, an argument named type must be included in lambda-list; when the method is called, type argument will be bound to the presentation type specifier.

For example, the present presentation generic function might be defined thus:

(define-presentation-generic-function present-method present
  (type-key parameters options object type stream view
   &key acceptably for-context-type))

None of the arguments are evaluated.

define-default-presentation-method
Arguments: name qualifiers* specialized-lambda-list &body body
Summary: Like define-presentation-method, except that it is used to define a default method that will be used only if there are no more specific methods.

funcall-presentation-generic-function
Arguments: presentation-function-name &rest arguments

Defining a New Presentation Type
Summary: Calls the presentation generic function named by `presentation-function-name` on the arguments `arguments`. `arguments` must match the arguments specified by the `define-presentation-generic-function` that were used to define the presentation generic function, excluding the `type-key`, `type-class`, `parameters`, and `options` arguments, which are filled in by CLIM.

`funcall-presentation-generic-function` is analogous to `funcall`. The `presentation-function-name` argument is not evaluated.

To call the `present` presentation generic function, one might use the following:

```lisp
(funcall-presentation-generic-function
  present object presentation-type stream view)
```

`apply-presentation-generic-function` [Macro]

Arguments: `presentation-function-name` &rest `arguments`

Summary: Like `funcall-presentation-generic-function`, except that `apply-presentation-generic-function` is analogous to `apply`. The `presentation-function-name` argument is not evaluated.
Chapter 8  Presentation Translators in CLIM
8.1 Conceptual Overview of Presentation Translators

CLIM provides a mechanism for “translating” between presentation types. In other words, within an input context for type A, the translator mechanism allows a programmer to have presentations of some other type B treated as though they were objects of type A.

You can define presentation translators to make the user interface of your application more flexible. For example, suppose the input context is expecting a command. In this input context, all displayed commands are sensitive, so the user can point to one to execute it. However, suppose the user points to another kind of displayed object, such as a student. In the absence of a presentation translator, the student is not sensitive because only commands can be entered to this input context.

If you used a presentation translator that translates from students to commands, however, both students and commands would be sensitive. When the student is highlighted, the middle pointer button might execute the command show-transcript for that student.

A presentation translator defines how to translate from one presentation type to another. In the previous scenario, the input context is command. A user-defined presentation translator states how to translate from the student presentation type to the command presentation type.

The concept of translating from an arbitrary presentation type to a command is so useful that CLIM provides the special define-presentation-to-command-translator macro for this purpose. You can think of these presentation-to-command translators as a convenience for the users; users can select the command and give the argument at the same time.

Note that presentation-to-command translators make it easier to write applications that give a “direct manipulation” feel to the user.

A presentation that appears on the screen can be sensitive. This means that the presentation can be operated on directly by using the pointer. A sensitive presentation will be highlighted when the pointer is over it. (In rare cases, the highlighting of some sensitive presentations is turned off.)
Sensitivity is controlled by three factors:

- **Input context type**—a presentation type describes the type of input currently being accepted.
- **Pointer location**—the pointer is pointing at a presentation or a blank area on the screen.
- **Modifier keys** (CONTROL, META, and SHIFT)—these keys expand the space of available gestures beyond what is available from the pointer buttons.

Presentation translators link these three factors.

A presentation translator specifies the conditions under which it is applicable, a description to be displayed, and what to do when it is invoked by clicking the pointer.

A presentation is sensitive (and highlighted) if there is at least one applicable translator that could be invoked by clicking a button with the pointer at its current location and the modifier keys in their current state. If there is no applicable translator, there is no sensitivity and no highlighting.

Each presentation translator has two associated presentation types, its *from-presentation-type* and *to-presentation-type*, which are the primary factors in its applicability. Since a presentation translator translates an output presentation into an input presentation, a presentation translator is applicable if the type of the presentation at the pointer “matches” the *from-presentation-type* and the input context type “matches” the *to-presentation-type*. (We define what “match” means in the next section.) Each presentation translator is attached to a particular pointer gesture, which is a combination of a pointer button and a set of modifier keys. Clicking the pointer button while holding down the modifier keys invokes the translator.

Note that a translator produces an input presentation consisting of an object and a presentation type to satisfy the program accepting input. The result of a translator might be returned from **accept**, or it might be absorbed by a parser and provide only part of the input. An input presentation is not actually represented as an object. Instead, a translator’s body returns two values. The object is the first value. The presentation type is the second value; it defaults to the *to-presentation-type* if the body returns only one value.
8.2 Applicability of CLIM Presentation Translators

When CLIM is waiting for input (inside a `with-input-context`) it is responsible for determining what translators are applicable to which presentations in a given input context. This loop both provides feedback in the form of highlighting sensitive presentations and is responsible for calling the applicable translator when the user presses a pointer button.

`with-input-context` uses `frame-find-innermost-applicable-presentation` (via `highlight-applicable-presentation`) as its “input wait” handler, and `frame-input-context-button-press-handler` as its button press “event handler.”

Given a presentation, an input context established by `with-input-context`, and a user gesture, translator matching proceeds as follows.

The set of candidate translators is initially those translators accessible in the command table in use by the current application. For more information, see Section 11.3, “Command Objects”.

A translator “matches” if all of the following are true. Note that these tests are performed in the order listed.

- The presentation’s type is `presentation-subtypep` of the translator’s `from-presentation-type`, ignoring type parameters (for example, if `from-presentation-type` is `number` and the presentation’s type is `integer` or `float`, or if `from-presentation-type` is `(or integer string)` and presentation’s type is `integer`).
- The translator’s `to-presentation-type` is `presentation-subtypep` of the input context type, ignoring type parameters.
- The translator’s gesture either is `t` or is the same as the gesture that the user could perform with the current chord of modifier keys.
- The presentation’s object is `presentation-typep` of the translator’s `from-presentation-type`, if the `from-presentation-type` has parameters. The translator’s tester returned a non-nil value. If there is no tester, the translator behaves as though the tester always returns `t`.
- If there are parameters in the input context type and the :`tester-definitive` option is not used in the translator, the value returned by the body of the translator must be `prese-
The algorithm is somewhat more complicated in the case of nested presentations and nested input contexts. In this situation, the sensitive presentation is the smallest presentation that matches the innermost input context.

When there are several translators that match for the same gesture, the one with the highest :priority is chosen (see define-presentation-translator).

8.2.1 Input Contexts in CLIM

Roughly speaking, the current input context indicates what type of input CLIM is asking the user for. You can establish an input context in CLIM with the following constructs:

- accept
- accept-from-string
- present (with an accept inside)
- The command loop of an application
- with-input-context

The input context designates a presentation type. However, the way to accept one type of object may involve accepting other types of objects as part of the procedure. (Consider the request to accept a complex number, which is likely to involve accepting two real numbers.) Such input contexts are called nested. In the case of a nested input context, several different context presentation types can be available to match the to-presentation-types of presentation translators.

Each level of input context is established by a call to accept. The macro with-input-context also establishes a level of input context.

The most common cause of input context nesting is accepting compound objects. For example, you might define a command called Show File, which reads a sequence of pathnames. When reading the argument to the Show File command, the input context contains pathname nested inside of (sequence clim:pathname). Acceptable keyboard input is a sequence of pathnames separated by commas. A presentation translator that translates to a (sequence clim:pathname) supplies the entire argument to the command, and the command processor moves on to the next argument. A presentation translator that
translates to a pathname is also applicable. It supplies a single element of the sequence being built up, and the command processor awaits additional input for this argument, or the entry of a SPACE or RETURN to terminate the argument.

When the input context is nested, sensitivity considers only the innermost context type that has any applicable presentation translators for the currently pressed chord of modifier keys.

### 8.2.2 Nested Presentations in CLIM

Presentations can overlap on the screen, so there can be more than one presentation at the pointer location. Often when two presentations overlap, one is nested inside the other.

One cause of nesting is presentations of compound objects. For example, a sequence of pathnames has one presentation for the sequence, and another for each pathname.

When there is more than one candidate presentation at the pointer location, CLIM must decide which presentation is the sensitive one. It starts with the innermost presentation at the pointer location and works outwards through levels of nesting until a sensitive presentation is discovered. This is the innermost presentation that has any applicable presentation translators to any of the nested input context types for the currently pressed chord of modifier keys. Searching in this way ensures that a more specific presentation is sensitive. Note that nested input contexts are searched first, before nested presentations. For presentations that overlap, the most recently presented is searched first.

### 8.3 Pointer Gestures in CLIM

A **gesture** is an input action by the user, such as typing a character or clicking a pointer button. A pointer gesture refers to those gestures that involve using the pointer.

An **event** is a CLIM object that represents a gesture by the user. (The most important pointer events are those of class **pointer-button-event**.)

A **gesture name** is a symbol that names a gesture. CLIM defines the following gesture names (the corresponding gesture appears in parentheses) and their uses:
:select (left click) For the most commonly used translator on an object. For example, use the :select gesture while reading an argument to a command to use the indicated object as the argument.

:describe (middle click) For translators that produce a description of an object (such as showing the current state of an object). For example, use the :describe gesture on an object in a CAD program to display the parameters of that object.

:menu (right click) For translators that pop up a menu

:delete (SHIFT-middle click) For translators that delete an object

:edit (META-right click) For translators that edit an object

The special gesture name nil is used in translators that are not directly invokable by a pointer gesture. Such a translator can be invoked only from a menu.

The special gesture name t means that the translator is available on every gesture.

You can use define-gesture-name (see Section 15.3, “Gestures and Gesture Names”) to define your own pointer gesture name.

Note that with the exception of the define-gesture-name forms (which you can use to map gesture names to keys and buttons), the application is independent of which platform it runs on. It uses keywords to give names to gestures, rather than making references to specific pointer buttons and keyboard keys.

The following operators can be used to add or remove new pointer gesture names. See Chapter 15, “Extended Stream Input Facilities,” for details about the pointer and gestures.

add-pointer-gesture-name [Function]
Arguments: gesture-name button shifts &key (action :click) (unique t)
Summary: Adds a pointer gesture named gesture-name (a symbol) for the pointer button being clicked on the pointer while the shifts shift keys are being held down on.

remove-pointer-gesture-name [Function]
Arguments: gesture-name
Summary: Removes the pointer gesture named gesture-name.
8.4 CLIM Operators for Defining Presentation Translators

define-presentation-translator supports presentation translators in general, and define-presentation-to-command-translator supports a common idiom.

**define-presentation-translator** [Macro]

**Arguments:**
- `name` (from-type to-type command-table &key gesture tester tester-definitive documentation pointer-documentation menu priority) arglist &body body

**Summary:** Defines a presentation translator named `name` that translates from objects of type `from-type` to objects of type `to-type`. `from-type` and `to-type` are presentation type specifiers, but must not include any presentation type options. `from-type` and `to-type` may be presentation type abbreviations.

`command-table` is a command table designator. The translator created by this invocation of **define-presentation-translator** will be stored in the command table `command-table`.

`gesture` is a gesture name that names a pointer gesture (described in Section 15.3, “Gestures and Gesture Names”). The body of the translator will be run only if the translator is applicable and gesture used by the user matches the gesture name in the translator. `gesture` defaults to :select.

`tester` is either a function or a list of the form `(tester-arglist . tester-body)` where `tester-arglist` takes the same form as `arglist` and `tester-body` is the body of the tester. The tester must return either `t` or `nil`. If it returns `nil`, then the translator is definitely not applicable. If it returns `t`, then the translator might be applicable, and the body of the translator might be run (if `tester-definitive` is `nil`) in order to decide definitively whether the translator is applicable. If no tester is supplied, CLIM supplies a tester that always returns `t`.

When the boolean `tester-definitive` is `t`, the body of the translator is not run in order to decide whether the translator is applicable; that is, the tester is assumed to definitively decide whether the translator applies. The default is `nil`.

Both `documentation` and `pointer-documentation` are objects that will be used for documenting the translator. `pointer-documentation` will be used to generate documentation for the pointer documentation window; the documentation generated by `pointer-documentation` should be very brief, and computing it should be very fast and preferably not cons. `documentation` is used to generate such things as
items in the :menu-gesture menu. If the object is a string, the string itself will be used as the documentation. Otherwise, the object must be the name of a function or a list of the form (doc-arglist . doc-body) where doc-arglist takes the same form as arglist, but includes a named (keyword) stream argument as well, and doc-body is the body of the documentation function. The body of the documentation function should write the documentation to stream. The default for documentation is nil, meaning that there is no explicitly supplied documentation; in this case, CLIM is free to generate the documentation in other ways. The default for pointer-documentation is documentation.

menu must be t or nil. When it is t, the translator will be included in the :menu-gesture menu if it matches. When it is nil, the translator will not be included in the :menu-gesture menu. Other non-nil values are reserved for future extensions to allow multiple presentation translator menus.

priority is either nil (the default, which corresponds to 1) or an integer that represents the priority of the translator. When there are several translators that match for the same gesture, the one with the highest priority is chosen.

arglist, tester-arglist, and doc-arglist are argument lists that must “match” the “canonical” argument list (object &key presentation context-type frame event window x y). In order to do so, there must be a single positional argument that corresponds to the presentation’s object, and several named arguments that must match the canonical names listed previously (using string-equal to do the comparison).

In the body of the translator (or the tester), the positional object argument will be bound to the presentation’s object. The named arguments presentation will be bound to the presentation that was clicked on, context-type will be bound to the presentation type of the context that actually matched, frame will be bound to the application frame that is currently active (usually *application-frame*), event will be bound to the pointer button event that the user used, window will be bound to the window stream from which the event came, and x and y will be bound to the x and y positions within window that the pointer was at when the event occurred. The special variable *input-context* will be bound to the current input context. Note that context-type and *input-context* will have dynamic extent, so programmers should not store without first copying them.

body is the body of the translator, and is run in the context of the application. body may have zero or more declarations as its first forms. It returns either one, two, or three values. The first value is an object that must be presentation-typep of to-type. The second value is a presentation type that must be presentation-sub-
The `typep` of `to-type`. The first two returned values of `body` are in effect used as the returned values for the call to `accept` that established the matching input context.

The third value returned by `body` must either be `nil` or a list of options (as keyword-value pairs) that will be interpreted by `accept`. The only option defined so far is `:echo`, whose value must be either `t` (the default) or `nil`. If it is `t`, the object returned by the translator will be “echoed” by `accept`, which will use `presentation-replace-input` to insert the textual representation of the object into the input buffer. If it is `nil`, the object will not be echoed.

None of `define-presentation-translator`’s arguments are evaluated.

---

```lisp
define-presentation-to-command-translator
```

Arguments: `name` (from-type `command-name` `command-table` &key gesture tester documentation pointer-documentation menu priority `echo`) `arglist` &body `body`

Summary: This resembles `define-presentation-translator`, except that the `to-type` will be derived to be the command named by `command-name` in the command table `command-table`. `command-name` is the name of the command that this translator will translate to.

The `echo` option is a boolean value (the default is `t`) that indicates whether the command line should be echoed when a user invokes the translator.

The other arguments to `define-presentation-to-command-translator` are the same as for `define-presentation-translator`. Note that the tester for command translators is always assumed to be definitive, so there is no `:tester-definitive` option. The default for `pointer-documentation` is the string `command-name` with dash characters replaced by spaces, and each word capitalized (as in `add-command-to-command-table`).

The body of the translator returns a list of the arguments to the command named by `command-name`. `body` is run in the context of the application. The returned value of the body, appended to the command name, is passed to `execute-frame-command`. `body` may have zero or more declarations as its first forms.

None of this macro’s arguments are evaluated.

---

```lisp
define-presentation-action
```

Arguments: `name` (from-type `to-type` `command-table` &key gesture tester documentation pointer-documentation menu priority) `arglist` &body `body`

---

<Bold>172CLIM User Guide
Summary: **define-presentation-action** is similar to **define-presentation-translator**, except that the body of the action is not intended to return a value, but instead affects some sort of application state.

A presentation action does not satisfy a request for input the way an ordinary translator does. Instead, an action is something that happens while waiting for input. After the action has been executed, the program continues to wait for the same input that it was waiting for prior to executing the action.

The other arguments to **define-presentation-action** are the same as for **define-presentation-translator**. Note that the tester for presentation actions is always assumed to be definitive.

None of **define-presentation-action**’s arguments are evaluated.

define-drag-and-drop-translator

[Macro]

Arguments: name from-type to-type destination-type command-table &key gesture tester before-drag-tester documentation pointer-documentation menu priority feedback highlighting arglist &body body

Summary: Defines a presentation translator named name that will be run when a presentation is dragged with the mouse and dropped on top of another presentation. The presentation types of the “dragged” (from-type) and “dropped on” (to-type) presentations are used to determine which translator (destination-type) is invoked.

from-type, to-type, and destination-type are presentation type specifiers, but must not include any presentation type options. from-type, to-type and destination-type may be presentation type abbreviations.

The interaction style used by these translators is that a user points to a “from presentation” with the pointer, picks it up by pressing a pointer button matching gesture, drags the “from presentation” to a “to presentation” by moving the pointer, and then drops the “from presentation” onto the “to presentation.” The dropping might be accomplished by either releasing the pointer button or clicking again, depending on the frame manager. When the pointer button is released, the translator whose destination-type matches the presentation type of the “to presentation” is chosen. For example, dragging a file to the TrashCan on a Macintosh could be implemented by a drag and drop translator.

When the user drags a “from presentation” over potential targets, the function or list specified by tester is invoked. This tester is identical to tester for **define-presentation-translator** except that it can take two additional arguments: destination-object and destination-presentation.
When the user points at a potential “from presentation” to drag, the function or list specified by before-drag-tester is invoked. The before-drag-tester takes the same arguments as tester for define-presentation-translator.

While the pointer is being dragged, the function specified by feedback is invoked to provide feedback to the user. The function is called with eight arguments: the application frame object, the “from presentation,” the stream, the initial x and y positions of the pointer, the current x and y positions of the pointer, and a feedback state (either :highlight to draw feedback, or :unhighlight to erase it). The feedback function is called to draw some feedback the first time pointer moves, and is then called twice each time the pointer moves thereafter (once to erase the previous feedback, and then to draw the new feedback). It is called a final time to erase the last feedback when the pointer button is released. feedback defaults to frame-drag-and-drop-feedback.

When the “from presentation” is dragged over any other presentation that has a direct manipulation translator, the function specified by highlighting is invoked to highlight that object. The function is called with four arguments: the application frame object, the “to presentation” to be highlighted or unhighlighted, the stream, and a highlighting state (either :highlight or :unhighlight). highlighting defaults to frame-drag-and-drop-highlighting.

Note that it is possible for there to be more than one drag and drop translator that applies to the same from-type, to-type, and gesture. In this case, the exact translator that is chosen for use during the dragging phase is unspecified. If these translators have different feedback, highlighting, documentation, or pointer documentation, the exact behavior is unspecified.

The other arguments to define-drag-and-drop-translator are the same as for define-presentation-translator.
8.5 Examples of Defining Presentation Translators in CLIM

8.5.1 Defining a Translation from Floating Point Number to Integer

Here is an example that defines a presentation translator to accept an integer object from a float presentation. Users have the options of typing in a float or integer to the input prompt or clicking on any float or integer presentation.

```
(define-presentation-translator integer-to-float
  (integer float my-command-table
    :documentation "Integer as float"
    :gesture :select
    :tester ((object) (integerp object))
    :tester-definitive t)
  (object)
  (float object))

(clim:present most-positive-fixnum)
(clim:accept 'float)
```

8.5.2 Defining a Presentation-to-Command Translator

The following example defines the delete-file presentation-to-command translator:

```
(clim:define-presentation-to-command-translator
 delete-file
 (pathname com-delete-file my-command-table
   :documentation "Delete this file"
   :gesture :delete)
  (object)
  (list object))
```
8.5.3 Defining Presentation Translators for the Blank Area

You can also write presentation translators that apply to blank areas of the window, that is, areas where there are no presentations. Use `blank-area` as the `from-presentation-type`. There is no highlighting when such a translator is applicable, since there is no presentation to highlight. You can write presentation translators that apply in any context by supplying `nil` as the `to-presentation-type`.

When you are writing an interactive graphics routine, you will probably encounter the need to have commands available when the mouse is not over any object. To do this, you write a **translator** from the blank area.

The presentation type of the blank area is `blank-area`. You probably want the `:x` and `:y` arguments to the translator.

For example:

```lisp
(clim:define-presentation-to-command-translator
 add-circle-here
 (clim:blank-area com-add-circle my-command-table
 :documentation "Add a circle here.")
 (x y)
 '(,x ,y))
```

8.5.4 Defining a Presentation Action

Presentation actions are only rarely needed. Often a presentation-to-command translator is more appropriate. One example where actions are appropriate is when you wish to pop up a menu during command input. Here is how CLIM’s general menu action could be implemented:

```lisp
(clim:define-presentation-action
 presentation-menu
 (t nil clim:global-command-table
 :tester-definitive t :documentation "Menu"
 :menu nil :gesture :menu)
 (presentation frame window x y)
 (clim:call-presentation-menu presentation clim:*input-context*
 frame window x y :for-menu t))
```
8.6 Advanced Topics

The material in this section is advanced; most CLIM programmers can skip to the next chapter. This section discusses low-level functions for CLIM presentation translators.

Some applications may wish to deal directly with presentation translators, for example, if you are tracking the pointer yourself and wish to locate sensitive presentations, or want to generate a list of applicable translators for a menu. The following functions are useful for finding and calling presentation translators directly.

**find-presentation-translators**

**[Function]**

**Arguments:** `from-type to-type command-table`

**Summary:** Returns a list of all of the translators in the command table `command-table` that translate from `from-type` to `to-type`, without taking into account any type parameters or testers. `from-type` and `to-type` are presentation type specifiers, and must not be abbreviations. `frame` must be an application frame.

**test-presentation-translator**

**[Function]**

**Arguments:** `translator presentation context-type frame window x y &key event modifier-state for-menu`

**Summary:** Returns `t` if the translator `translator` applies to the presentation `presentation` in input context type `context-type`, otherwise returns `nil`. (There is no `from-type` argument because it is derived from `presentation`.) `x` and `y` are the `x` and `y` positions of the pointer within the window stream `window`.

`event` and `modifier-state` are a pointer button event and modifier state (see `event-modifier-key-state`), and are compared against the translator’s gesture. `event` defaults to `nil`, and `modifier-state` defaults to 0, meaning that no modifier keys are held down. Only one of `event` or `modifier-state` may be supplied.

If `for-menu` is `t`, the comparison against `event` and `modifier-state` is not done.

`presentation`, `context-type`, `frame`, `window`, `x`, `y`, and `event` are passed along to the translator’s tester if and when the tester is called.

`test-presentation-translator` matches type parameters and calls the translator’s tester. Under some circumstances, `test-presentation-translator` may also call the body of the translator to ensure that its value matches `to-type`. 

Presentation Translators in CLIM
find-applicable-translators

Arguments: presentation input-context frame window x y &key event modifier-state for-menu fastp

Summary: Returns an object that describes the translators that definitely apply to the presentation in the input context. The result is a list whose elements are each of the form (translator the-presentation context-type tag) where translator is a presentation translator, the-presentation is the presentation that the translator applies to (and which can be different from presentation due to nesting of presentations), context-type is the context type in which the translator applies, and tag is a tag used internally by CLIM. translator, the-presentation, and context-type can be passed to such functions as call-presentation-translator and document-presentation-translator.

Since input contexts can be nested, find-applicable-translators must iterate over all the contexts in input-context. window, x, and y are as for test-presentation-translator. event and modifier-state (which default to nil and the current modifier state for window, respectively) are used to further restrict the set of applicable translators. (Only one of event or modifier-state may be supplied; it is unspecified what will happen if both are supplied.)

When for-menu is non-nil, the value of for-menu is matched against the presentation’s menu specification, and only those translators that match are returned. event and modifier-state are disregarded in this case. for-menu defaults to nil.

When the boolean fastp is t, find-applicable-translators will simply return t if there are any translators. When fastp is nil (the default), the list of translators returned by find-applicable-translators must be in order of their “desirability”; that is, translators having more specific from-types and/or higher priorities must precede translators having less specific from-types and lower priorities.

The rules used for ordering the translators returned by find-applicable-translators are as follows (in order):

1. Translators with a higher “high order” priority precede translators with a lower “high order” priority. This allows programmers to set the priority of a translator in such a way that it always precedes all other translators.

2. Translators with a more specific “from type” precede translators with a less specific “from type.”

3. Translators with a higher “low order” priority precede translators with a lower “low order” priority. This allows programmers to break ties between translators that translate from the same type.
4. Translators from the current command table precede translators inherited from superior command tables.

**presentation-matches-context-type**

*Function*

**Arguments:** `presentation context-type frame window x y &key event modifier-state`

**Summary:** Returns `t` if there are any translators that translate from the presentation `presentation`'s type to the input context type `context-type`; otherwise, it returns `nil`. (There is no `from-type` argument because it is derived from `presentation`.) `frame`, `window`, `x`, `y`, `event`, and `modifier-state` are as for `test-presentation-translator`.

If there are no applicable translators, `presentation-matches-context-type` will return `nil`.

**call-presentation-translator**

*Function*

**Arguments:** `translator presentation context-type frame event window x y`

**Summary:** Calls the function that implements the body of the translator `translator` on the presentation `presentation`'s object, and passes `presentation`, `context-type`, `frame`, `event`, `window`, `x`, and `y` to the body of the translator as well.

The returned values are the same as the values returned by the body of the translator, namely, the translated object and the translated type.

**document-presentation-translator**

*Function*

**Arguments:** `translator presentation context-type frame event window x y &key stream documentation-type`

**Summary:** Computes the documentation string for the translator `translator` and outputs it to the stream `stream`, which defaults to `*standard-output*`. `presentation`, `context-type`, `frame`, `event`, `window`, `x`, and `y` are as for `test-presentation-translator`.

`documentation-type` must be either `:normal` or `:pointer`. If it is `:normal`, the usual translator documentation function is called. If it is `:pointer`, the translator’s pointer documentation is called.

**call-presentation-menu**

*Function*

**Arguments:** `presentation input-context frame window x y &key for-menu label`

**Summary:** Finds all the applicable translators for the presentation `presentation` in the input context `input-context`, creates a menu that contains all of the translators, and pops up the menu from which the user can choose a translator. After the translator is chosen, it is called with the arguments supplied to `call-presentation-menu`, and the matching input context established by `with-input-context` is terminated.

Presentation Translators in CLIM
window, x, y, and event are as for find-applicable-translators. for-menu, which defaults to t, is used to decide which of the applicable translators will go into the menu; only those translators whose :menu option matches menu will be included.

label is either a string to use as a label for the menu, or is nil (the default), meaning the menu will not be labelled.

The following functions are useful for finding an application presentation in an output history:

**find-innermost-applicable-presentation**  
[Function]  
Arguments: input-context window x y &key frame modifier-state event  
Summary: Given an input context input-context, an output recording window stream window, x and y positions x and y, returns the innermost presentation whose sensitivity region contains x and y that matches the innermost input context, using the translator matching algorithm described later. If there is no such presentation, this function will return nil.  

event and modifier-state are a pointer button event and modifier state (see event-modifier-key-state). event defaults to nil, and modifier-state defaults to the current modifier state for window. Only one of event or modifier-state may be supplied; it is unspecified what will happen if both are supplied.  

frame defaults to the current frame, *application-frame*.

The default method for frame-find-innermost-applicable-presentation will call this function.

**throw-highlighted-presentation**  
[Function]  
Arguments: presentation input-context button-press-event  
Summary: Given a presentation presentation, input context input-context, and a button press event (which contains the window, pointer, x and y position of the pointer within the window, the button pressed, and the modifier state), finds the translator that matches the innermost presentation in the innermost input context, then calls the translator to produce an object and a presentation type. Finally, the matching input context that was established by with-input-context will be terminated.

Note that it is possible that more than one translator having the same gesture may be applicable to presentation in the specified input context. In this case, the trans-
lator having the highest priority will be chosen. If there is more than one having
the same priority, it is unspecified what translator will be chosen.

**highlight-applicable-presentation**

*Function*

**Arguments:** `frame stream input-context &optional prefer-pointer-window`

**Summary:** This is the core of the “input wait” handler used by `with-input-context` on behalf of the application frame `frame`. It locates the innermost applicable presentation on `stream` in the input context `input-context`, unhilighting presentations that are not applicable and highlighting the presentation that is applicable. Typically on entry to `highlight-applicable-presentation`, `input-context` will be the value of `*input-context*` and `frame` will be the value of `*application-frame*`.

`prefer-pointer-window` is a boolean. If it is `t` (the default), CLIM will highlight the applicable presentation on the same window that the pointer is located over. Otherwise, CLIM will highlight an applicable presentation on `stream`.

**set-highlighted-presentation**

*Function*

**Arguments:** `stream presentation &optional prefer-pointer-window`

**Summary:** Highlights the presentation `presentation` on `stream`. This must call `highlight-presentation` methods if that is appropriate.

**unhighlight-highlighted-presentation**

*Function*

**Arguments:** `stream &optional prefer-pointer-window`

**Summary:** Unhighlights any highlighted presentations on `stream`.  

Presentation Translators in CLIM
Chapter 9  Defining Application Frames
9.1 Conceptual Overview of CLIM Application Frames

Application frames (or simply frames) are the central abstraction defined by CLIM for presenting an application’s user interface. Many of the other features and facilities provided by CLIM (for example, the generic command loop, gadgets, look-and-feel independence) can be conveniently accessed through the frame facility. Frames can be displayed as either top-level windows or regions embedded within the space of the user interfaces of other applications. In addition to controlling the screen real estate managed by an application, a frame keeps track of the Lisp state variables that contain the state of the application.

The contents of a frame is established by defining a hierarchy of panes. CLIM panes are interactive objects that are analogous to the windows, gadgets, or widgets of other toolkits. Application builders can compose their application’s user interface from a library of standard panes or by defining and using their own pane types. Application frames can use a number of different types of panes, including layout panes for organizing space, extended stream panes for presenting application-specific information, and gadget panes for displaying data and obtaining user input. Panes are described in greater detail in Chapter 10, “Panes and Gadgets.”

Frames are managed by special applications called frame managers. Frame managers control the realization of the look and feel of a frame. The frame manager interprets the specification of the application frame in the context of the available window system facilities, taking into account preferences expressed by the user. In addition, the frame manager takes care of attaching the pane hierarchy of an application frame to an appropriate place in a window hierarchy. The most common type of frame manager is one that allows the user to manipulate the frames of other applications. This type of application is typically called a desktop manager, or in X Windows terminology, a window manager. In many cases, the window manager will be a non-Lisp application. In these cases, the frame manager will act as a mediator between the Lisp application and the host desktop manager.

Some applications may act as frame managers that allow the frames of other applications to be displayed with their own frames. For example, a text editor might allow figures generated by a graphic editor to be edited in place by managing the graphics editor’s frame within its own frame.

Application frames provide support for a standard interaction processing loop, like the Lisp “read-eval-print” loop, called a command loop. The application programmer only has to
write the code that implements the frame-specific commands and output display functions. A key aspect of the command loop is the separation of the specification of the frame’s commands from the specification of the end-user interaction style.

The standard interaction loop consists of reading an input “sentence” (the command and all of its operands), executing the command, and updating the displayed information as appropriate.

To write an application that uses the standard interaction loop provided by CLIM, you need to:

■ Define the presentation types that correspond to the user-interface entities of the application.
■ Define the commands that correspond to the visible operations of the application, specifying the presentation types of the operands involved in each command.
■ Define the set of frames and panes needed to support the application.
■ Define the output display functions associated with each of the panes (possibly using other facilities such as the incremental redisplay).

In the case of a simple ECAD program, the programmer would first define the appropriate presentation types, such as wires, input and output signals, gates, resistors, and so forth. She would then define the program’s commands in terms of these types. For example, the “Connect” command might take two operands, one of type “component” and the other of type “wire.” The programmer may wish to specify the interaction style for invoking each command, for example, direct manipulation via translators, or selection of commands from menus. After defining an application frame that includes a CLIM stream pane, the programmer then writes the frame-specific display routine that displays the circuit layout. Now the application is ready to go. The end-user selects a command (via a menu, command-line, or whatever), the top-level loop takes care of collecting the operands for that command (via a variety of user gestures), and then the application invokes the command. The command may affect the frame’s “database” of information, which can in turn affect the output displayed by the redisplay phase.

Note that this definition of the standard interaction loop does not constrain the interaction style to be a command-line interface. The input sentence may be entered via single keystrokes, pointer input, menu selection, dialogs, or by typing full command lines.

Defining Application Frames
9.2 Defining CLIM Application Frames

**define-application-frame**  
*Macro*

**Arguments:** `name` `superclasses` `slots` `&rest` `options`

**Summary:** Defines a frame and CLOS class named by the symbol `name` that inherits from `superclasses` and has state variables specified by `slots`. `superclasses` is a list of superclasses that the new class will inherit from (as in `defclass`). When `superclasses` is `nil`, it behaves as though a superclass of `standard-application-frame` was supplied. `slots` is a list of additional slot specifiers, whose syntax is the same as the slot specifiers in `defclass`. Each instance of the frame will have slots as specified by these slot specifiers. These slots will typically hold any per-instance frame state.

`options` is a list of `defclass`-style options, and can include the usual `defclass` options, plus any of the following:

- **:pane form**, where `form` specifies the single pane in the application. The default is `nil`, meaning that there are either no panes or there are multiple panes. This is the simplest way to define a pane hierarchy. The `:pane` option cannot be used with the `:panes` and `:layouts` options. See Subsection 9.2.2 for a complete description of the `:pane` option.

- **:panes form**, where `form` is an alist that specifies names and panes of the application. The default is `nil`, meaning that there are no named panes. The `:panes` and `:pane` options are mutually exclusive. See Subsection 9.2.3 for a complete description of the `:panes` option.

- **:layouts form**, where `form` specifies the layout. The default layout is to lay out all of the named panes in horizontal strips. The `:layouts` and `:pane` options are mutually exclusive. See Subsection 9.2.3 for a complete description of the `:layouts` option.

- **:command-table name-and-options**, where `name-and-options` is a list consisting of the name of the application frame’s command table followed by some keyword-value pairs. The keywords can be `:inherit-from` or `:menu` (which are as in `define-command-table`). The default is to create a command table with the same name as the application frame.

- **:menu-bar form** is used to specify what commands will appear in a “menu bar.” It typically specifies the top-level commands of the application. `form` is either `nil`, meaning there is no menu bar; `t`, meaning that the menu from frame’s command table (from the `:command-table` option) should be used; a symbol
that names a command table, meaning that that command table’s menu should be used. The default is \texttt{t}.

- **:disabled-commands** \textit{commands}, where \textit{commands} is a list of command names that are initially disabled in the application frame.

- **:command-definer** \textit{value}, where \textit{value} either \texttt{nil}, \texttt{t}, or another symbol. When it is \texttt{nil}, no command-defining macro is defined. When it is \texttt{t}, a command-defining macro is defined, whose name is of the form \texttt{define-<name>-command}. When it is another symbol, the symbol names the command-defining macro. The default is \texttt{t}.

- **:top-level** \textit{form}, where \textit{form} is a list whose first element is the name of a function to be called to execute the top-level loop. The function must take at least one argument, the frame. The rest of the list consists of additional arguments to be passed to the function. The default function is \texttt{default-frame-top-level}.

The \textit{name}, \textit{superclasses}, and \textit{slots} arguments are not evaluated. The values of each of the options are evaluated.

\textbf{make-application-frame} \quad \texttt{[Function]}

\textbf{Arguments:} \texttt{frame-name} \ & \texttt{rest options} \ & \texttt{key pretty-name frame-manager enable state left top right bottom width height save-under frame-class} \ & \texttt{allow-other-keys}

\textbf{Summary:} Makes an instance of the application frame of type \texttt{frame-class}. If \texttt{frame-class} is not supplied, it defaults to \texttt{frame-name}.

The size options \texttt{left}, \texttt{top}, \texttt{right}, \texttt{bottom}, \texttt{width}, and \texttt{height} can be used to specify the size of the frame.

\textit{options} are passed as additional arguments to \texttt{make-instance}, after the \textit{pretty-name}, \textit{frame-manager}, \textit{enable}, \textit{state}, \textit{save-under}, \textit{frame-class}, and \textit{size} options have been removed.

If \texttt{save-under} is \texttt{t}, then the sheets used to implement the user interface of the frame will have the “save under” property, if the host window system supports it.

If \texttt{frame-manager} is provided, then the frame is adopted by the specified frame manager. If the frame is adopted and either \texttt{enable} or \texttt{state} is provided, the frame is pushed into the given state. See Section 9.9, “Frame Managers.”

Once a frame has been created, \texttt{run-frame-top-level} can be called to make the frame visible and run its top-level function.

Defining Application Frames
*application-frame*  
[Variable]
Summary: The current application frame. The global value is CLIM’s default application, which serves only as a repository for whatever internal state is needed by CLIM to operate properly. This variable is typically used in the bodies of commands to gain access to the state variables of the application frame, usually in conjunction with with-slots or slot-value.

with-application-frame  
[Macro]
Arguments: (frame) &body body
Summary: This macro provides lexical access to the “current” frame for use with the :pane, :panes, and :layouts options. frame is bound to the current frame within the context of one of those options.

frame is a symbol; it is not evaluated. body may have zero or more declarations as its first forms.

9.2.1 The Application Frame Protocol

application-frame  
[Protocol Class]
Summary: The protocol class that corresponds to an application frame. If you want to create a new class that behaves like an application frame, it should be a subclass of application-frame. Subclasses of application-frame must obey the application frame protocol.

All application frame classes are mutable.

application-frame-p  
[Function]
Arguments: object
Summary: Returns t if object is an application frame; otherwise, it returns nil.

:name  
[Initarg]

:pretty-name  
[Initarg]

:command-table  
[Initarg]

:disabled-commands  
[Initarg]
Defining Application Frames

:panes
:menu-bar
:calling-frame
:state
:properties

Summary: All subclasses of application-frame must handle these initargs, which specify, respectively, the name, pretty name, command table, initial set of disabled commands, panes, menu bar, calling frame, state, and initial properties for the frame.

standard-application-frame

Summary: The standard class that implements application frames. By default, most application frame classes will inherit from this class, unless a non-nil value for super-classes is supplied to define-application-frame.

9.2.2 Using the :pane Option

The panes of a frame can be specified in one of two different ways. If the frame has a single layout and no need of named panes, then the :pane option can be used. Otherwise, if named panes or multiple layouts are required, the :panes and :layouts options can be used. Note that the :pane option cannot be used with :panes and :layouts. It is meaningful to define frames that have no panes at all; the frame will simply serve as a repository for state and commands.

The value of the :pane option is a form that is used to create a single (albeit arbitrarily complex) pane. For example:
(vertically ()
  (tabling ()
    (horizontally ()
      (make-pane 'toggle-button)
      (make-pane 'toggle-button)
      (make-pane 'toggle-button)
      (make-pane 'text-field))
      (make-pane 'push-button :label "a button")
      (make-pane 'slider)))
    (scrolling ()
      (make-pane 'application-pane :display-function 'a-display-function))
    (scrolling ()
      (make-pane 'interactor-pane)))

9.2.3 Using the :panes and :layouts Options

If the :pane option is not used, a set of named panes can be specified with the :panes option. Optionally, :layouts can also be used to describe different layouts of the set of panes.

The value of the :panes option is an alist, each entry of which is of the form (name . body). name is a symbol that names the pane, and body specifies how to create the pane. body is either a list containing a single element that is itself a list, or a list consisting of a symbol followed by zero or more keyword-value pairs. In the first case, the body is a form exactly like the form used in the :pane option. In the second case, body is a pane abbreviation, where the initial symbol names the type of pane, and the keyword-value pairs are pane options. For gadgets, the pane type is the class name of the abstract gadget (for example, slider or push-button). For CLIM extended stream panes, the following abbreviations are defined:

- :interactor—a pane of type interactor-pane
- :application—a pane of type application-pane
- :command-menu—a pane of type command-menu-pane
- :pointer-documentation—a pane suitable for displaying pointer documentation, if the host window system does not provide this
- :title—a pane suitable for displaying the title of the application. If the host window system provides this, the title will be displayed with the window decorations supplied by the window manager, and the CLIM title pane will be omitted.
- **:accept-values**—a pane that can hold a “modeless” **accepting-values** dialog

See Chapter 10, “Panes and Gadgets,” for more information on the individual pane and gadget classes and the options they support.

An example of the use of **:panes** is:

```lisp
(:panes
  (buttons
    (horizontally ()
      (make-pane 'push-button :label "Press me")
      (make-pane 'push-button :label "Squeeze me")))
  (toggle toggle-button
    :label "Toggle me")
  (interactor :interactor
    :width 300 :height 300)
  (application :application
    :display-function 'another-display-function
    :incremental-redisplay t))
```

The value of the **:layouts** option is an alist, each entry of which is of the form (name . layout). name is a symbol that names the layout, and layout specifies the layout. layout is a form like the form used in the **:pane** option, with the extension to the syntax such that the name of a named pane can be used wherever a pane may appear. For example, assuming a frame that uses the **:panes** example, the following layouts could be specified:

```lisp
(:layouts
  (default
    (vertically ()
      button toggle
      (scrolling () application)
      interactor))
  (alternate
    (vertically ()
      (scrolling () application)
      (scrolling () interactor)
      (horizontally ()
        button toggle))))
```

---

Defining Application Frames
9.2.4 Example of the :pane Option to define-application-frame

Here is an example of how to use the :pane option of define-application-frame:

```
(define-application-frame test-frame ()
 ()
 (:pane
     (vertically ()
         (make-clim-interactor-pane
             :foreground +green+
             :background +red+)
         (make-pane 'push-button
             :label "press me"
             :background +black+
             :foreground +purple+
             :activate-callback
             #'(lambda (button)
                 (frame-exit *application-frame*))
             :text-style
             (make-text-style :serif :roman 20))))
)
```

9.2.5 Examples of the :panes and :layout Options to define-application-frame

Here are some examples of how to use the :panes and :layout options of define-application-frame to describe the appearance of your application.

We begin by showing Figure 18, an example of how CLIM supplies a default layout when you don’t explicitly specify one in your frame definition. The default layout is a single column of panes, in the order (top to bottom) that you specified them in the :panes option. Command menus are allocated only enough space to display their contents, while the remaining space is divided among the other types of panes equally.
(define-application-frame test () ()
  (:panes
   (main :application
     :incremental-redisplay NIL
     :display-function 'display-main)
   (test-menu :command-menu)
   (listener :interactor))
  (:layouts
   {:default
    (vertically () main test-menu listener))}
  (:command-table
   (test-menu
    :inherit-from (user-command-table)
    :menu
    ("EXIT" :command cmd-exit)))))

Figure 18. The Default Layout for the Graphic-Demo Example
When No Explicit :layout Is Specified

Now we take the same example as before and in Figure 19 add an explicit :layout option to the frame definition. The pane named explanation occupies the bottom sixth of the screen. The remaining five-sixths are occupied by the demo and commands panes, which lie side by side, with the command pane to the right. The commands pane is only as wide as is needed to display the command menu.
(define-application-frame graphics-demo () ()
 (:menu-bar nil)
 (:panes
  (commands :command-menu)
  (demo :application)
  (explanation :application :scroll-bars nil))
 (:layouts
  (:default (vertically ()
    (:fill (horizontally ()
      (:fill demo)
      (1/5 commands)))
    (1/6 explanation))))

Figure 19. The Layout for the Graphic-Demo Example With an Explicit :layout

Finally, here is a stripped-down version of the application frame definition for the CAD demo (in the file `<release-directory>/demo/new-cad-demo.lisp`) which implements an extremely simplistic computer-aided logic circuit design tool.

There are four panes defined for the application. The pane named title displays the string “Mini-CAD” and serves to remind the user which application is running. The pane named menu provides a menu of commands for the application. The pane named design-area is the actual “work surface” of the application on which various objects (logic gates and wires) can be manipulated. A pane named documentation is provided to inform the user
about what actions can be performed using the pointing device (typically the mouse) and is updated based on what object is currently being pointed to.

The application has two layouts, one named `main` and one named `other`. Both layouts have their panes arranged in vertical columns. At the top of both layouts is the `title` pane, which is of the smallest height necessary to display the title string “Mini-CAD.” Both layouts have the `documentation` pane at the bottom.

The two layouts differ in the arrangement of the `menu` and `design-area` panes. In the layout named `main`, the `menu` pane appears just below the `title` pane and extends for the width of the screen. Its height will be computed so as to be sufficient to hold all the items in the menu. The `design-area` pane occupies the remaining screen real estate, extending from the bottom of the `menu` pane to the top of the `documentation` pane, and is as wide as the screen.

To see the layout named `other`, enter `(setf (frame-current-layout *application-frame*) :other)`. This differs from the `main` layout in the shape of the `design-area` pane. Here the implementor of the CAD demo realized that, depending on what was being designed, either a short, wide area or a narrower but taller area might be more appropriate. The `other` layout provides the narrower, taller alternative by rearranging the `menu` and `design-area` panes to be side by side (forming a row of the two panes). The `menu` and `design-area` panes occupy the space between the bottom of the `title` pane and the top of the `documentation` pane, with the `menu` pane to the left and occupying as much width as is necessary to display all the items of the menu and the `design-area` occupying the remaining width.
Figure 20. The Two Layouts of the Mini-CAD Demo
9.2.6 Using an :accept-values Pane in a CLIM Application Frame

Frame :accept-values panes are used when you want one of the panes of your application to be in the form of an accepting-values dialog.

There are several things to remember when using an :accept-values pane in your application frame:

- For an :accept-values pane to work, your frame’s command table must inherit from the accept-values-pane command table.
- The :display-function option for an :accepting-values pane will typically be something like:

  (clim:accept-values-pane-displayer
   :displayer my-acceptor-function)

  where my-acceptor-function is a function that you write. It contains calls to accept just as they would appear inside a accepting-values for a dialog. It takes two arguments, the frame and a stream. my-acceptor-function doesn’t need to call accepting-values itself, since that is done automatically.

  See Chapter 12, “Menus and Dialogs,” especially the function accept-values-pane-displayer.

- While inside the display function for an :accept-values pane, *application-frame* is not bound to your application. Instead, it is bound to an application that implements accepting-values. Therefore, you cannot use with-frame-state-variables in the display function for an :accept-values pane. Use with-slots on the frame argument instead.

- Don’t use :display-after-commands with :accept-values panes, because the redisplay for those panes is managed at a slightly lower level for efficiency.

9.3 Initializing CLIM Application Frames

There are several ways to initialize an application frame:

1. The value of an application frame’s slot can be initialized using the :initform slot option in the slot’s specifier in the define-application-frame form. This technique is suitable if the slot’s initial value does not depend on the values of other slots, calculations based on the values of initialization arguments, or other information that
cannot be determined until after the application frame is created. See the macro clos:defclass to learn about slot-specifiers.

2. For initializations that depend on information which may not be available until the application frame has been created, an :after method can be defined for clos:initialize-instance on the application frame’s class. Note that the special variable *application-frame* is not bound to the application, since the application is not yet running. The macro with-frame-state-variables cannot be used in this context, either. You may use clos:with-slots, clos:with-accessors, or any slot readers or accessors that have been defined.

3. A :before method for run-frame-top-level on the application’s frame is probably the most versatile place to perform application frame initialization. This method will not be executed until the application starts running. *application-frame* will be bound to the application frame, and you can use with-frame-state-variables in this context.

4. If the application frame employs its own top-level function, then this function can perform initialization tasks at the beginning of its body. This top-level function may call default-frame-top-level to achieve the standard behavior for application frames.

Of course, these are only suggestions. There might be other techniques which might be more appropriate for your application. Of those listed, the :before method on run-frame-top-level is probably the best for most circumstances.

Although application frames are CLOS classes, do not use clos:make-instance to create them. To instantiate an application frame, always use make-application-frame. This function provides important initialization arguments specific to application frames that clos:make-instance does not. make-application-frame passes any keyword value pairs which it does not handle itself on to clos:make-instance, so it will respect any initialization options which you give it, just as clos:make-instance would.

Here is an example of how an application frame’s behavior might be modified by inheritance from a superclass. Suppose we wanted our application to record all the commands that were performed while it was executing, because the program is for the financial industry, where it is important to keep audit trails for all transactions. As this is a useful functionality that might be added to any of a number of different applications, we will make it into a special class that implements the desired behavior. This class can then be used as a superclass for any application that needs to keep a log of its actions.

The class has an initialization option, :pathname, which specifies the name of the log file. It also has a slot named transaction-stream whose value is a stream opened to the log file when the application is running.
Defining Application Frames

We use an `:around` method on `run-frame-top-level`, which opens a stream to the log file and stores it in the `transaction-stream` slot. `unwind-protect` is used to clear the value of the slot when the stream is closed.

```lisp
(defun clim:run-frame-top-level :around (frame)
  (with-slots (transaction-pathname transaction-stream) frame
    (with-open-file (stream transaction-pathname :direction :output)
      (unwind-protect
        (progn (setq transaction-stream stream)
               (call-next-method))
       (setq transaction-stream nil))))
```

This is where the actual logging takes place. The command loop in `default-frame-top-level` calls `execute-frame-command` to execute a command. Here we add a `:before` method that will log the command.

```lisp
(defun clim:execute-frame-command :before (frame command)
  (format (transaction-stream frame) "Command: ~a")
)
```

It is now an easy matter to alter the definition of an application to add the command logging behavior. Here is the definition of the puzzle application frame from the CLIM demos suite (from the file `<release-directory>/demo/puzzle.lisp`). We use the superclasses argument to specify that the puzzle application frame should inherit from `transaction-recording-mixin`. Because we are using the superclass argument, we must also explicitly include `application-frame`, which was included by default when the superclasses argument was empty.

```lisp
(define-application-frame puzzle
  (transaction-recording-mixin application-frame)
  (:initform (make-array '(4 4)) :accessor puzzle-puzzle)
  (:default-initargs :pathname "puzzle-log.text")
  (:panes :title :title)
  (:menu :command-menu)
  (:display :application
    :default-text-style '(:fix :bold :very-large)
    :incremental-redisplay t
    :display-function draw-puzzle))
```

Defining Application Frames
Also note the use of (:default-initargs :pathname "puzzle-log.text")
to provide a default value for the log file name if the user doesn’t specify one.

The user might run the application by executing the following:

```
(run-frame-top-level
 (make-application-frame 'puzzle
   :width 400
   :height 500
   :pathname "my-puzzle-log.text"))
```

Here the :pathname initialization argument is used to override the default name for the log file (as was specified by the :default-initargs clause in the previously defined application frame) and to use the name my-puzzle-log.text instead.

### 9.4 Accessing Slots and Components of CLIM Application Frames

A call to the `define-application-frame` macro creates a subclass of the `standard-application-frame` class. CLIM application frames are instances of these generated subclasses. You explicitly specify accessors for the slots you have designated for storing application-specific state information. The use of the accessors is as for any other CLOS instance. Other CLIM defined components of `standard-application-frame` subclass instances are accessed via documented functions. Such components include frame-panes, command-tables, the top-level window, and layouts.

### 9.5 Running a CLIM Application

You can run a CLIM application using the functions `make-application-frame` and `run-frame-top-level`. Here is a code fragment that illustrates this technique:

```
(clim:run-frame-top-level
 (clim:make-application-frame
  'frame-name))
```

`run-frame-top-level` will not return until the application exits.
Note that *application-frame* is not bound until run-frame-top-level is invoked.

For more information, see Section E.2, “Functions for Operating on Windows Directly.”

## 9.6 Exiting a CLIM Application

You can exit an application and make the window disappear by using frame-exit or disable-frame.

## 9.7 Examples of CLIM Application Frames

This section contains examples of how to use CLIM application frames.

### 9.7.1 Defining a CLIM Application Frame

Here is an example of an application frame. This frame has three slots: pathname, integer, and member. It has two panes, an :accept-values pane named avv and an :application pane named display. It uses a command table named dingus, which will automatically be defined for it (see define-command-table) and which inherits from the accept-values-pane command table so that the accept-values pane will function properly.

```lisp
(clim:define-application-frame
  dingus ()
  ((pathname :initform #p"foo")
   (integer :initform 10)
   (member :initform :one))
  (:panes
   {avv :accept-values
    :display-function '(clim:accept-values-pane-displayer
                       :displayer display-avv))
   {display :application :display-function 'draw-display
             :display-after-commands :no-clear)})

(:command-table (dingus :inherit-from (clim:accept-values-pane))) (:command-table (dingus :inherit-from (clim:accept-values-pane)))
```

The following is the display function for the display pane of the “dingus” application. It just prints out the values of the three slots defined for the application.

```lisp
```
(defmethod draw-display ((frame dingus) stream)
  (with-slots (pathname integer member) frame
    (fresh-line stream)
    (clim:present pathname 'pathname :stream stream)
    (write-string ", " stream)
    (clim:present integer 'integer :stream stream)
    (write-string ", " stream)
    (clim:present member '(member :one :two :three)
                        :stream stream)
    (write-string "." stream))))

The following is the display function for the avv pane. It invokes accept for each of the application’s slots so that the user can alter their values in the avv pane.

(defun run-dingus (root)
  (let ((dingus (clim:make-application-frame
                  'dingus :width 400 :height 400)))
    (clim:run-frame-top-level dingus)))

All this application does is allow the user to alter the values of the three application slots, pathname, integer, and member, using the avv pane. The new values will automatically be reflected in the display pane.
9.7.2 Constructing a Function as Part of Running an Application

You can supply an alternate top level (which initializes some things and then calls the regular top level) to construct a function as part of running the application. Note that when you use this technique, you can close the function over other pieces of the Lisp state that might not exist until application runtime.

```lisp
(clim:define-application-frame
different-prompts ()
 ((prompt-state ...) ...)
 (:top-level (different-prompts-top-level)) ...)

(make-application-frame different-prompts-top-level
 ((frame different-prompts) &rest options)
 (flet ((prompt (stream frame)
                (with-slots (prompt-state) frame
                         (apply
                          #'clim:default-frame-top-level frame
                          :prompt #'prompt options)))
                       ...)))
```

9.8 Application Frame Operators and Accessors

The following operators are used to define and instantiate application frames. They are discussed in detail in Section 9.2, “Defining CLIM Application Frames.”

**define-application-frame**  
[Macro]  
Arguments: name superclasses slots &rest options  
Summary: Defines an application frame. You can specify a name for the application class, the superclasses (if any), the slots of the application class, and options.

**make-application-frame**  
[Function]  
Arguments:frame-name &rest options &key pretty-name frame-manager enable state left top right bottom width height save-under frame-class &allow-other-keys  
Summary: Makes an instance of the application frame of type frame-class. If frame-class is not supplied, it defaults to frame-name.
9.8.1 CLIM Application Frame Accessors

The following functions may be used to access and modify the state of the application frame itself, such as what the currently exposed panes are, what the current layout is, what command table is being used, and so forth.

*application-frame* [Variable]
Summary: The current application frame. The value is CLIM’s default application. This variable is typically used in the bodies of commands and translators to gain access to the state variables of the application, usually in conjunction with clos:with-slots or clos:slot-value.

frame-name [Generic Function]
Arguments: frame
Summary: Returns the name of the frame frame, which is a symbol.

frame.pretty-name [Generic Function]
Arguments: frame
Summary: Returns the “pretty name” of the frame frame, which is a string.

(setf frame.pretty-name) [Generic Function]
Arguments: name frame
Summary: Sets the pretty name of the frame frame to name, which must be a string.

frame-command-table [Generic Function]
Arguments: frame
Summary: Returns the command table for the frame frame.

(setf frame-command-table) [Generic Function]
Arguments: command-table frame
Summary: Sets the command table for the frame frame to command-table. Changing the frame’s command table will redisplay the command menus (or menu bar) as needed. command-table is a command table designator.

frame-standard-input [Generic Function]
Arguments: frame

<Bold>204CLIM User Guide
Summary: Returns the stream that will be used for *standard-input* for the frame frame. The default method (on standard-application-frame) returns the first named pane of type interactor-pane that is exposed in the current layout; if there is no such pane, the value returned by frame-standard-output is used.

frame-standard-output

Arguments: frame

Summary: Returns the stream that will be used for *standard-output* for the frame frame. The default method (on standard-application-frame) returns the first named pane of type application-pane that is exposed in the current layout; if there is no such pane, it returns the first pane of type interactor-pane that is exposed in the current layout.

To redirect standard input or output, simply redefine the corresponding frame generic function. For example, the following form specifies the pane in my-frame named output-pane as the destination for standard output.

(defmethod frame-standard-output ((frame my-frame))
  (get-frame-pane frame 'output-pane))

frame-query-io

Arguments: frame

Summary: Returns the stream that will be used for *query-io* for the frame frame. The default method (on standard-application-frame) returns the value returned by frame-standard-input; if that is nil, it returns the value returned by frame-standard-output.

frame-error-output

Arguments: frame

Summary: Returns the stream that will be used for *error-output* for the frame frame. The default method (on standard-application-frame) returns the same value as frame-standard-output.

*pointer-documentation-output*

Summary: This will be bound either to nil or to a stream on which pointer documentation will be displayed.

frame-pointer-documentation-output

Arguments: frame

Defining Application Frames
Summary: Returns the stream that will be used for *pointer-documentation-output* for the frame frame. The default method (on standard-application-frame) returns the first pane of type pointer-documentation-pane. If this returns nil, no pointer documentation will be generated for this frame.

frame-calling-frame
Arguments:frame
Summary: Returns the application frame that invoked the frame frame.

frame-parent
Arguments:frame
Summary: Returns the object that acts as the parent for the frame frame. This often, but not always, returns the same value as frame-manager.

frame-panes
Arguments:frame
Summary: Returns the pane that is the top-level pane in the current layout of the frame frame's named panes. This function returns objects that reveal CLIM's internal state; do not modify those objects.

frame-current-panes
Arguments:frame
Summary: Returns a list of those named panes in the frame frame’s current layout. If there are no named panes (that is, the :pane option was used), only the single, top-level pane is returned. This function returns objects that reveal CLIM's internal state; do not modify those objects.

get-frame-pane
Arguments:frame pane-name
Summary: Returns the CLIM stream pane in the frame frame whose name is pane-name.

find-pane-named
Arguments:frame pane-name
Summary: Returns the pane in the frame frame whose name is pane-name. This can return any type of pane, not just CLIM stream panes.
Defining Application Frames

frame-top-level-sheet
Arguments:frame
Summary: Returns the sheet that is the top-level sheet for the frame frame. This is the sheet that has as its descendants all of the panes of frame.

frame-current-layout
Arguments:frame
Summary: Returns the current layout for the frame frame. The layout is named by a symbol.

(setf frame-current-layout)
Arguments:layout frame
Summary: Sets the layout of the frame frame to be the new layout specified by new-layout. layout must be a symbol that names one of the possible layouts.

Changing the layout of the frame causes a recomputation of what panes are used for the bindings of the standard stream variables (such as *standard-input*). After the new layout has been computed, the contents of each pane are displayed to the degree necessary to ensure that all output is visible.

layout-frame
Arguments:frame &optional width height
Summary: Resizes the frame and lays out the current pane hierarchy using the layout specified by frame-current-layout, according to the layout protocol described in Section 10.2.4. This function is automatically invoked on a frame when it is adopted, after its pane hierarchy has been generated.

If width and height are provided, then this function resizes the frame to the specified size. It is an error to provide just width. If no optional arguments are provided, this function resizes the frame to the preferred size of the top-level pane as determined by the space composition pass of the layout protocol.

In either case, after the frame is resized, the space allocation pass of the layout protocol is invoked on the top-level pane.

9.8.2 Operators for Running CLIM Applications

The following functions are used to start up an application frame, exit from it, and control the “read-eval-print” loop of the frame (for example, redisplay the panes of the frame, and read, execute, enable, and disable commands).
**run-frame-top-level**

**Arguments:** frame

**Summary:** Runs the top-level function for the frame frame. The default method on application-frame simply invokes the top-level function for the frame (which defaults to default-frame-top-level).

**run-frame-top-level**

**Arguments:** (frame application-frame)

**Summary:** The :around method of run-frame-top-level on the application-frame class establishes the initial dynamic bindings for the application, including (but not limited to) binding *application-frame* to frame, binding *input-context* to nil, resetting the delimiter and activation gestures, and binding *input-wait-test*, *input-wait-handler*, and *pointer-button-press-handler* to nil.

**default-frame-top-level**

**Arguments:** frame &key command-parser command-unparser partial-command-parser prompt

**Summary:** The default top-level function for application frames, this runs a “read-eval-print” loop that displays a prompt, calls read-frame-command and then execute-frame-command, and finally redisplays all the panes that need it. See Section 11.9, “The CLIM Command Processor,” for further details.

default-frame-top-level also establishes a simple restart for abort and binds the standard stream variables as follows. *standard-input* will be bound to the value returned by frame-standard-input. *standard-output* will be bound to the value returned by frame-standard-output. *query-io* will be bound to the value returned by frame-query-io. *error-output* will be bound to the value returned by frame-error-output. It is unspecified what *terminal-io*, *debug-io*, and *trace-output* will be bound to.

**prompt** is either a string to use as the prompt (defaulting to Command: ) or a function of two arguments, a stream and the frame.

command-parser, command-unparser, and partial-command-parser are the same as for read-command. command-parser defaults to command-line-command-parser if there is an interactor; otherwise, it defaults to menu-only-command-parser. command-unparser defaults to command-line-command-unparser. partial-command-parser defaults to command-line-read-remaining-arguments-for-partial-command if there is an interactor; otherwise, it defaults to menu-only-read-remaining-arguments.
Defining Application Frames

default-frame-top-level binds *command-parser*, *command-unparser*, and *partial-command-parser* to the values of command-parser, command-unparser, and partial-command-parser.

read-frame-command
Arguments: frame stream
Summary: Reads a command from the stream stream on behalf of the frame frame. The returned value is a command object.

The default method (on standard-application-frame) for read-frame-command simply calls read-command, supplying frame’s current command table as the command table.

execute-frame-command
Arguments: frame command
Summary: Executes the command command on behalf of the frame frame. command is a command object, that is, a cons of a command name and a list of the command’s arguments.

The default method (on standard-application-frame) for execute-frame-command simply applies the command-name of command to command-arguments of command.

command-enabled
Arguments: command-name frame
Summary: Returns t if the command named by command-name is presently enabled in the frame frame; otherwise, it returns nil. If command-name is not accessible to the command table being used by frame, command-enabled returns nil.

Whether or not a particular command is currently enabled is stored independently for each instance of an application frame; this status can vary between frames that share a single command table.

(setf command-enabled)
Arguments: enabled command-name frame
Summary: If enabled is nil, this disables the use of the command named by command-name while in the frame frame. If enabled is t, the use of the command is enabled. After the command has been enabled (or disabled), note-command-enabled (or note-command-disabled) is invoked on the frame manager in order to update the appearance of the interface.
If command-name is not accessible to the command table being used by frame, using setf on command-enabled does nothing.

display-command-menu

Arguments: frame stream &key command-table initial-spacing max-width max-height n-rows n-columns (cell-align-x :left) (cell-align-y :top)

Summary: Displays the menu associated with the specified command table on stream by calling display-command-table-menu. If command-table is not supplied, it defaults to (frame-command-table stream). This function is generally used as the display function for panes that contain command menus.

initial-spacing, max-width, max-height, n-rows, n-columns, cell-align-x, and cell-align-y are as for formatting-item-list.

frame-exit

Summary: The restart that is invoked when frame-exit is called.

frame-exit

Arguments: frame

Summary: Exits from the frame frame. The default method (on standard-application-frame) invokes the frame-exit restart.

panes-need-redisplay

Arguments: frame

panes-needs-redisplay

Arguments: frame pane

Summary: panes-need-redisplay indicates that all the panes in the frame frame should be redisplayed the next time around the command loop. pane-needs-redisplay causes only the pane pane within frame to be redisplayed; in this case, pane is either a pane or the name of a named pane.

redisplay-frame-pane

Arguments: frame pane &key force-p

Summary: Causes the pane pane within the frame frame to be redisplayed immediately. pane is either a pane or the name of a named pane. When the boolean force-p is t, the maximum level of redisplay is forced and the pane is displayed “from scratch.”
**redisplay-frame-panes**  
*Arguments:* frame &key force-p  
*Summary:* `redisplay-frame-panes` causes all of the panes in the frame `frame` to be redisplayed immediately by calling `redisplay-frame-pane` on each of the panes in `frame` that are visible in the current layout. When the boolean `force-p` is `t`, the maximum level of redisplay is forced and the pane is displayed “from scratch.”

**frame-replay**  
*Arguments:* frame stream &optional region  
*Summary:* Replays the contents of `stream` in the frame `frame` within the region specified by the region `region`, which defaults to the viewport of `stream`.

**notify-user**  
*Arguments:* frame message &key associated-window title documentation exit-boxes style text-style  
*Summary:* Notifies the user of some event on behalf of the frame `frame`. This function provides a look-and-feel independent way for applications to communicate messages to the user. For example, a frame manager might provide a top-level message window for each frame, or it might pop up an alert box.

`frame` is a CLIM application frame, `message` is a message string, `associated-window` is the window with which the notification will be associated, `title` is a title string to include in the notification, `documentation` is not implemented in the current version of CLIM, `exit-boxes` indicates what sort of exit boxes should appear in the notification, `style` is the style in which to display the notification, and `text-style` is the text style in which to display the notification.

**frame-manager-notify-user**  
*Arguments:* frame message &key associated-window title documentation exit-boxes style text-style  
*Summary:* The generic function used by `notify-user`. The default method on `standard-frame-manager` will display a dialog or an alert box that contains the message and has exit boxes that will allow the user to dismiss the notification.

`frame` is a CLIM application frame, `message` is a message string, `associated-window` is the window with which the notification will be associated, `title` is a title string to include in the notification, `documentation` is not implemented in the current version of CLIM, `exit-boxes` indicates what sort of exit boxes should appear in the notification, `style` is the style in which to display the notification, and `text-style` is the text style in which to display the notification.
**frame-properties**

[Generic Function]

Arguments: `frame property`

**(setf frame-properties)**

[Generic Function]

Arguments: `value frame property`

Summary: Frame properties can be used to associate frame specific data with frames without adding additional slots to the frame’s class. CLIM may use frame properties internally to store information for its own purposes.

### 9.9 Frame Managers

Frames may be *adopted* by a frame manager, which involves invoking a protocol for generating the pane hierarchy of the frame. This protocol provides for selecting pane types for abstract gadget panes based on the style requirements imposed by the frame manager. That is, the frame manager is responsible for the look and feel of a frame. Each frame manager is associated with one specific port. However, a single port may have multiple frame managers managing various frames associated with the port.

After a frame is adopted it can be in any of the three following states: *enabled*, *disabled*, or *shrunk*. An enabled frame is visible unless it is occluded by other frames or the user is browsing outside of the portion of the frame manager’s space that the frame occupies. A shrunk frame provides a cue or handle for the frame, but generally will not show the entire contents of the frame. For example, the frame may be iconified, or an item for the frame may be placed in a special suspended frame menu. A disabled frame is not visible, nor is there any user-accessible handle for enabling the frame.

Frames may also be *disowned*, which involves releasing the frame’s panes as well as all associated foreign resources.

**frame-manager**

[Protocol Class]

Summary: The protocol class that corresponds to a frame manager. If you want to create a new class that behaves like a frame manager, it should be a subclass of `frame-manager`. Subclasses of `frame-manager` must obey the frame manager protocol.

There are no advertised standard frame manager classes. Each port implements one or more frame managers that correspond to the look and feel for the port.
frame-manager-p
Arguments: object
Summary: Returns `t` if `object` is a frame manager; otherwise, it returns `nil`.

9.9.1 Finding Frame Managers

Most frames need to deal directly with frame managers only to the extent that they need to find a frame manager into which they can insert themselves. Since frames will usually be invoked by some user action that is handled by a frame manager, finding an appropriate frame manager is usually straightforward.

Some frames will support the embedding of other frames within themselves. Such frames not only use frames but also act as frame managers. In this case, the embedded frames are mostly unaware that they are nested within other frames, but only know that they are controlled by a particular frame manager.

The `find-frame-manager` function provides a flexible means for locating a frame manager to adopt an application’s frames. There are a variety of ways that the user or the application can influence where an application’s frame is adopted:

- An application can establish an application default frame manager using `with-frame-manager`. A frame’s top-level loop automatically establishes the frame’s frame manager.
- The programmer or user can influence what frame manager is found by setting `*default-frame-manager*` or `*default-server-path*`.

find-frame-manager
Arguments: &rest options &key port &allow-other-keys
Summary: Finds an appropriate frame manager that conforms to the options, including the `port` argument. Furthermore, CLIM applications may set up dynamic contexts that affect what `find-frame-manager` will return.

`port` defaults to the value returned by `find-port` applied to the remaining options.

A frame manager is found using the following rules in the order listed:

1. If a current frame manager has been established via an invocation of `with-frame-manager`, as is the case within a frame’s top-level, and that frame manager conforms to the options, it is returned. The exact definition of “conforming to the options” varies from one port to another, but it may include
such things as matching the console number, color or resolution properties, and so forth. If the options are empty, then any frame manager will conform.

1. If *default-frame-manager* is bound to a currently active frame manager and it conforms to the options, it is returned.

2. If *port* is nil, a port is found and an appropriate frame manager is constructed using *default-server-path*.

*default-frame-manager* [Variable]
Summary: This variable provides a convenient point for allowing a programmer or user to override the frame manager type that would normally be selected. Most users will not set this variable, since they can set *default-server-path* to indicate which host window system they want to use and are willing to use whatever frame manager is the default for the particular port. However, some users may want to use a frame manager that isn’t the typical frame manager. For example, a user may want to use both an OpenLook frame manager and a Motif frame manager on a single port.

with-frame-manager [Macro]
Arguments: (frame-manager) &body body
Summary: Generates a dynamic context that causes all calls to find-frame-manager to return frame-manager if the where argument passed to it conforms to frame-manager. Nested calls to with-frame-manager shadow outer contexts. body may have zero or more declarations as its first forms.

9.9.2 Frame Manager Operators

frame-manager [Generic Function]
Arguments: frame
Summary: Returns frame’s current frame manager if it is adopted; otherwise, it returns nil.

(setf frame-manager) [Generic Function]
Arguments: frame-manager frame
Summary: Changes the frame manager of frame to frame-manager. In effect, the frame is disowned from its old frame manager and is adopted into the new frame manager. Transferring a frame preserves its frame-state; for example, if the frame was previously enabled, it will be enabled in the new frame manager.
frame-manager-frames
Arguments:frame-manager
Summary: Returns a list of all the frames being managed by frame-manager. This function returns objects that reveal CLIM’s internal state; do not modify those objects.

adopt-frame
Arguments:frame-manager frame

disown-frame
Arguments:frame-manager frame
Summary: These functions insert or remove a frame from a frame manager’s control. These functions allow a frame manager to allocate and deallocate resources associated with a frame. For example, removing a frame from a frame manager that is talking to a remote server allows it to release all remote resources used by the frame.

frame-state
Arguments:frame
Summary: Returns one of :disowned, :enabled, :disabled, or :shrunk, indicating the current state of frame.

enable-frame
Arguments:frame

disable-frame
Arguments:frame

shrink-frame
Arguments:frame
Summary: These functions force a frame into the enabled, disabled, or shrunk states. A frame in the enabled state may be visible if it is not occluded or placed out of the user’s focus of attention. A disabled frame is never visible. A shrunk frame is accessible to the user for re-enabling, but may be represented in some abbreviated form, such as an icon or a menu item.

These functions call note-frame-state-changed to notify the frame manager that the state of the frame changed.
**note-frame-state-changed**  [Generic Function]

Arguments: `frame-manager frame new-state`

Summary: Notifies the frame manager `frame-manager` that the frame `frame` has changed its state to `state`.

**generate-panes**  [Generic Function]

Arguments: `frame-manager frame`

Summary: This function is invoked by a standard method of `adopt-frame`. It is the responsibility of the frame implementor to provide a method that invokes `setf` on `frame-panes` on the frame with a value of type `pane`. `define-application-frame` automatically supplies a `generate-panes` method if either the `:pane` or `:panes` option is used in the `define-application-frame`.

**find-pane-for-frame**  [Generic Function]

Arguments: `frame-manager frame`

Summary: This function is invoked by a standard method of `adopt-frame`. It must return the root pane of the frame’s layout. It is the responsibility of the frame implementor to provide a method that constructs the frame’s top-level pane. `define-application-frame` automatically supplies a method for this function if either the `:pane` or `:panes` option is used.

**note-command-enabled**  [Generic Function]

Arguments: `frame-manager frame command-name`

**note-command-disabled**  [Generic Function]

Arguments: `frame-manager frame command-name`

Summary: Notifies the frame manager `frame-manager` that the command named by `command-name` has been enabled or disabled (respectively) in the frame `frame`. The frame manager can update the appearance of the user interface as appropriate, for instance, by “graying out” a newly disabled command from a command menu or menu bar.

### 9.10 Advanced Topics

The material in this section is advanced; most CLIM programmers can skip to the next section. It describes the functions that interface application frames to the presentation type sys-
Defining Application Frames

All classes that inherit from `application-frame` must inherit or implement methods for all of these functions. See “Application Frame Operators and Accessors,” Section 9.8.

**frame-maintain-presentation-histories**

Arguments: `frame`

Summary: Returns `t` if the frame `frame` maintains histories for its presentations; otherwise, it returns `nil`. The default method (on `standard-application-frame`) returns `t` if and only if the frame has at least one interactor pane.

**frame-find-innermost-applicable-presentation**

Arguments: `frame input-context stream x y &key event`

Summary: Locates and returns the innermost applicable presentation on the window `stream` whose sensitivity region contains the point `(x, y)`, on behalf of the frame `frame` in the input context `input-context`. `event` defaults to `nil`, and is as for `find-innermost-applicable-presentation`.

The default method (on `standard-application-frame`) simply calls `find-innermost-applicable-presentation`.

**frame-input-context-button-press-handler**

Arguments: `frame stream button-press-event`

Summary: This function handles user pointer events on behalf of the frame `frame` in the input context `*input-context*`. `stream` is the window on which `button-press-event` took place.

The default implementation (on `standard-application-frame`) unhighlights any highlighted presentations, finds the applicable presentation by calling `frame-find-innermost-applicable-presentation-at-position`, and then calls `throw-highlighted-presentation` to execute the translator on that presentation that corresponds to the user’s gesture.

If `frame-input-context-button-press-handler` is called when the pointer is not over any applicable presentation, `throw-highlighted-presentation` must be called with a presentation of `*null-presentation*`.

**frame-document-highlighted-presentation**

Arguments: `frame presentation input-context window x y stream`

Summary: This function produces pointer documentation on behalf of the frame `frame` in the input context `input-context` on the window `window` at the point `(x, y)`. The documentation is displayed on the stream `stream`. 

Defining Application Frames
The default method (on `standard-application-frame`) produces documentation that corresponds to calling `document-presentation-translator` on all of the applicable translators in the input context `input-context. presentation, window, x, y, and stream` are as for `document-presentation-translator`.

Typical pointer documentation consists of a brief description of each translator that is applicable to the specified presentation in the specified input context, given the current modifier state for the window. For example, the following documentation might be produced when the pointer is pointing to the Lisp expression `(1 2 3)` when the input context is `form`:

Left: `(1 2 3)`; Middle: (DESCRIBE '(1 2 3)); Right: Menu

`frame-drag-and-drop-feedback` [Generic Function]
Arguments: `frame presentation stream initial-x initial-y new-x new-y state`
Summary: The default feedback function for translators defined by `define-drag-and-drop-translator`, which provides visual feedback during the dragging phase of such translators on behalf of the frame `frame. presentation` is the presentation being dragged on the stream `stream`. The pointing device was initially at the position specified by `initial-x` and `initial-y`, and is at the position specified by `new-x` and `new-y` when `frame-drag-and-drop-feedback` is invoked. (Both positions are supplied for “rubber-banding,” if that is the sort of desired feedback.) `state` will be either `:highlight`, meaning that the feedback should be drawn, or `:unhighlight`, meaning that the feedback should be erased.

`frame-drag-and-drop-highlighting` [Generic Function]
Arguments: `frame presentation stream state`
Summary: The default highlighting function for translators defined by `define-drag-and-drop-translator`, which is invoked when a “to object” should be highlighted during the dragging phase of such translators on behalf of the frame `frame. presentation` is the presentation over which the pointing device is located on the stream `stream`. `state` will be either `:highlight`, meaning that the highlighting for the presentation should be drawn, or `:unhighlight`, meaning that the highlighting should be erased.
Chapter 10  Panes and Gadgets
10.1 Panes

CLIM panes are similar to the gadgets or widgets of other toolkits. They can be used to compose the top-level user interface of applications as well as auxiliary components such as menus and dialogs. The application programmer provides an abstract specification of the pane hierarchy, which CLIM uses in conjunction with user preferences and other factors to select a specific “look and feel” for the application. In many environments, a CLIM application can use the facilities of the host window system toolkit via a set of adaptive panes, allowing a portable CLIM application to take on the look and feel of a native application user interface.

Panes are rectangular objects that are implemented as special sheet classes. An application will typically create a tree of panes that divide up the application frame’s screen space. Panes can be structurally classified according to their location in pane hierarchies. Panes that can have child panes are called composite panes; those that cannot are called leaf panes. Composite panes are used to provide a mechanism for spatially organizing (“laying out”) other panes. Some leaf panes implement gadgets that have some appearance and react to user input by invoking application code. Another kind of leaf pane, known as an extended stream pane, provides an area of the application’s screen real estate for the presentation of text and graphics.

Abstract panes are panes that are defined only in terms of their programmer interface, or behavior. The protocol for an abstract pane (that is, the specified set of initialization options, accessors, and callbacks) is designed to specify the pane in terms of its overall purpose, rather then in terms of its specific appearance or particular interactive details. This abstract definition allows multiple implementations of the abstract pane to define their own specific look and feel individually. CLIM can then select the appropriate pane implementation based on factors outside of the application domain, such as user preferences or the look and feel of the host operating environment. A subset of the abstract panes, the adaptive panes, have been defined to integrate well across all CLIM operating platforms.

CLIM provides a general mechanism for automatically selecting the particular implementation of an abstract pane selected by an application based on the current frame manager. The application programmer can override the selection mechanism by using the name of a specific pane implementation in place of the abstract pane name when specifying the application frame’s layout. By convention, the name of the basic, portable implementation of an abstract pane class can be determined by adding the suffix -pane to the name of the abstract class.
10.1.1 Basic Pane Construction

Applications typically define the hierarchy of panes used in their frames with the :pane or :panes options of define-application-frame. These options generate the body of methods on functions that are invoked when the frame is being adopted into a particular frame manager, so the frame manager can select the specific implementations of the abstract panes.

There are two basic interfaces for constructing a pane: make-pane of an abstract pane class name, or make-instance of a “concrete” pane class. The former approach is generally preferable, since it results in more portable code. However, in some cases the programmer may wish to instantiate panes of a specific class (such as an hbox-pane or a vbox-pane). In this case, using make-instance directly circumvents the abstract pane selection mechanism. However, the make-instance approach requires the application programmer to know the name of the specific pane implementation class that is desired, and so is inherently less portable. By convention, all of the concrete pane class names, including those of the implementations of abstract pane protocol specifications, end in -pane.

Using make-pane instead of make-instance invokes the “look and feel” realization process to select and construct a pane. Normally this process is implemented by the frame manager, but it is possible for other “realizers” to implement this process. make-pane is typically invoked using an abstract pane class name, which by convention is a symbol in the CLIM package that doesn’t include the -pane suffix. (This naming convention distinguishes the names of the abstract pane protocols from the names of classes that implement them.) Programmers, however, are allowed to pass any pane class name to make-pane, in which case the frame manager will generally instantiate that specific class.

pane [Protocol Class]
Summary: The protocol class that corresponds to a pane, a subclass of sheet. A pane is a special kind of sheet that implements the pane protocols, including the layout protocols. If you want to create a new class that behaves like a pane, it should be a subclass of pane. Subclasses of pane must obey the pane protocol.

All of the subclasses of pane are mutable.

panep [Function]
Arguments: object
Summary: Returns t if object is a pane; otherwise, it returns nil.

basic-pane [Class]

Panes and Gadgets
Summary: The basic class on which all CLIM panes are built, a subclass of pane. This class is an abstract class, intended only to be subclassed, not instantiated.

**make-pane**

Arguments: `abstract-class-name &rest initargs`

Summary: Selects a class that implements the behavior of the abstract pane `abstract-class-name` and constructs a pane of that class. **make-pane** must be used either within the dynamic scope of a call to **with-look-and-feel-realization**, or within the :pane or :panes options of a define-application-frame (which implicitly invokes **with-look-and-feel-realization**).

**make-pane-1**

Arguments: `realizer frame abstract-class-name &rest initargs`

Summary: The generic function that is invoked by a call to **make-pane**. The object that realizes the pane, `realizer`, is established by **with-look-and-feel-realization**. Usually `realizer` is a frame manager, but it could be another object that implements the pane realization protocol. `frame` is the frame for which the pane will be created, and `abstract-class-name` is the type of pane to create.

**with-look-and-feel-realization**

Arguments: `(realizer frame) &body forms`

Summary: Establishes a dynamic context that installs `realizer` as the object responsible for realizing panes. All calls to **make-pane** within the context of **with-look-and-feel-realization** result in **make-pane-1** being invoked on `realizer`. This macro can be nested dynamically; inner uses shadow outer uses. `body` may have zero or more declarations as its first forms.

`realizer` is usually a frame manager, but in some cases `realizer` may be some other object. For example, within the implementation of a pane that uses its own subpanes to achieve its functionality, this form might be used with `realizer` being the pane itself.

### 10.1.2 Pane Initialization Options

The following options must be accepted by all pane classes.
:foreground [Option]

:background [Option]
Summary: These options specify the default foreground and background inks for a pane. Client code should be cautious about passing values for these two options, since the desktop’s look and feel or the user’s preferences should usually determine these values.

:text-style [Option]
Summary: This option specifies the default text style that should be used for any sort of pane that supports text output. Panes that do not support text output ignore this option. Client code should be cautious about passing values for this option, since the desktop’s look and feel or the user’s preferences should usually determine this value.

:name [Option]
Summary: This option specifies the name of the pane. It defaults to nil.

10.1.3 Pane Properties

pane-frame [Generic Function]
Arguments: pane
Summary: Returns the frame that “owns” the pane. pane-frame can be invoked on any pane in a frame’s pane hierarchy, but it can only be invoked on “active” panes, that is, those panes that are currently adopted into the frame’s pane hierarchy.

pane-name [Generic Function]
Arguments: pane
Summary: Returns the name of the pane.

pane-foreground [Generic Function]
Arguments: pane
Summary: Returns the foreground ink of the pane.

pane-background [Generic Function]
Arguments: pane

Panes and Gadgets
Summary: Returns the background ink of the pane.

## 10.2 Layout Panes

This section describes the various layout panes provided by CLIM and the protocol that these panes obey.

The layout panes described in this section are all composite panes that are responsible for positioning their children according to various layout rules. Layout panes can be selected in the same way as other panes by using `make-pane` or `make-instance`. For convenience and readability of pane layouts, many of these panes also provide a macro that expands into a `make-pane` form, passing a list of the panes created in the body of the macro as the :contents argument. For example, you can express a layout of a vertical column of two label panes either as:

```lisp
(make-instance 'vbox-pane :contents (list
  (make-instance 'label-pane :text "One")
  (make-instance 'label-pane :text "Two")))
```

or as:

```lisp
(vertically ()
  (make-instance 'label-pane :text "One")
  (make-instance 'label-pane :text "Two"))
```

### 10.2.1 Layout Pane Options

**:contents**

Summary: All layout pane classes accept the :contents options, which specifies the child panes to be laid out.
:width  [Option]
:max-width  [Option]
:min-width  [Option]
:height  [Option]
:max-height  [Option]
:min-height  [Option]

Summary: These options control the space requirement parameters for laying out the pane. The :width and :height options specify the preferred horizontal and vertical sizes. The :max-width and :max-height options specify the maximum amount of space that may be consumed by the pane, and give CLIM’s pane layout engine permission to grow the pane beyond the preferred size. The :min-width and :min-height options specify the minimum amount of space that may be consumed by the pane, and give CLIM’s pane layout engine permission to shrink the pane below the preferred size.

If either the :max-width or the :min-width option is not supplied, it defaults to the value of the :width option. If either the :max-height or the :min-height option is not supplied, it defaults to the value of the :height option.

:max-width, :min-width, :max-height, and :min-height can also be specified as a relative size by supplying a list of the form (number :relative). In this case, the number indicates the number of device units that the pane is willing to stretch or shrink.

The values of these options are specified in the same way as the :x-spacing and :y-spacing options to formatting-table. (Note that :character and :line may only be used on those panes that display text, such as a clim-stream-pane or a label-pane.)

+fill+  [Constant]

Summary: Use this constant as a value to any of the relative size options. It indicates that pane’s willingness to adjust an arbitrary amount in the specified direction.
The \texttt{align-x} option is one of \texttt{right}, \texttt{center}, or \texttt{left}. The \texttt{align-y} option is one of \texttt{top}, \texttt{center}, or \texttt{bottom}. These specify how child panes are aligned within the parent pane. These have the same semantics as for \texttt{formatting-cell}.

\textbf{10.2.2 Layout Pane Classes}

\texttt{hbox-pane} \hfill \texttt{[Composite Pane]}

\texttt{horizontally} \hfill \texttt{[Macro]}
\texttt{\texttt{Arguments}:\&rest options \&key spacing \&allow-other-keys) \&body contents}

Summary: The \texttt{hbox-pane} class lays out all its child panes horizontally from left to right. The \texttt{horizontally} macro is a convenient interface for creating an \texttt{hbox-pane}.

\textit{contents} is one or more forms that are the child panes. Each form in \textit{contents} is of the form:

\begin{itemize}
  \item A pane—the pane is inserted at this point and its space requirements are used to compute the size.
  \item A number—the specified number of device units should be allocated at this point.
  \item The symbol \texttt{+fill+}—an arbitrary amount of space can be absorbed at this point in the layout.
\end{itemize}
A list whose first element is a number and whose second element evaluates to a pane—if the number is less than 1, then it means that percentage of excess space or deficit should be allocated to the pane. If the number is greater than or equal to 1, then that many device units are allocated to the pane. For example:

```
(horizontally ()
  (1/3 (make-pane 'label-button-pane))
  (2/3 (make-pane 'label-button-pane)))
```

would create a horizontal stack of two button panes. The first button takes one-third of the space, and the second takes two-thirds of the space.

**vbox-pane**

**vertically**

Arguments: (&rest options &key spacing &allow-other-keys) &body contents

Summary: The **vbox-pane** class lays out all of its child panes vertically, from top to bottom. The **vertically** macro serves as a convenient interface for creating an **vbox-pane**.

**table-pane**

**tabling**

Arguments: (&rest options) &body contents

Summary: This pane lays out its child panes in a two-dimensional table arrangement. Each column of the table is specified by an extra level of list in **contents**. For example:

```
(tabling ()
  ((make-pane 'label :text "Red")
   (make-pane 'label :text "Green")
   (make-pane 'label :text "Blue"))
  ((make-pane 'label :text "Intensity")
   (make-pane 'label :text "Hue")
   (make-pane 'label :text "Saturation")))
```

**spacing-pane**

**spacing**

Arguments: (&rest options &key thickness &allow-other-keys) &body contents
Summary: This pane reserves some margin space around a single child pane. The space requirement keys that are passed in indicate the requirements for the surrounding space, not including the requirements of the child.

outlined-pane

outlining

Arguments: ( &rest options &key thickness &allow-other-keys )
Summary: This layout pane puts an outline of thickness thickness around its contents.

Use the :background option to control the ink used to draw the background.

bboard-pane

Summary: A pane that allows its children to be any size and lays them out wherever they want to be (for example, a desktop manager).

scroller-pane

scrolling

Arguments: ( &rest options )
Summary: Creates a composite pane that allows the single child specified by contents to be scrolled. options may include a :scroll-bar option. The value of this option may be t (the default), which indicates that both horizontal and vertical scroll bars should be created; :vertical, which indicates that only a vertical scroll bar should be created; or :horizontal, which indicates that only a horizontal scroll bar should be created.

The pane created by the scrolling includes a scroller-pane that has as children the scroll bars and a viewport. The viewport of a pane is the area of the window’s drawing plane that is currently visible to the user. The viewport has as its child the specified contents.

hrack-pane

vrack-pane

Summary: Similar to the hbox-pane and vbox-pane classes, except that these ensure that all children are the same size in the minor dimension. In other words, these panes are used to create stacks of same-sized items, such as menu items.
An **hrack-pane** is created when the `:equalize-height` option to **horizontally** is `t`. A **vrack-pane** is created when the `:equalize-width` option to **vertically** is `t`.

Note: **hrack-pane** and **vrack-pane** are available only in Liquid CLIM.

**restraining-pane**

[Composite Pane]

**restraining**

[Macro]
Arguments: `(&rest options) &body contents`
Summary: Wraps the contents with a pane that prevents changes to the space requirements for `contents` from causing re-layout of panes outside the restraining context. This prevents the size constraints of the child from propagating up beyond this point.

Note: **restraining-pane** and **restraining** are available only in Liquid CLIM.

### 10.2.3 Composite Pane Generic Functions

**pane-viewport**

[Generic Function]
Arguments: `pane`
Summary: Returns the pane’s viewport, if one exists.

**pane-viewport-region**

[Generic Function]
Arguments: `pane`
Summary: If a viewport for the pane exists, the viewport’s region is returned.

**pane-scroller**

[Generic Function]
Arguments: `pane`
Summary: Checks to see whether a pane has an associated scroller pane, and returns it if it does.

**scroll-extent**

[Generic Function]
Arguments: `pane x y`
Summary: If the `pane` argument has an associated viewport, it resets the viewport to display the portion of the underlying stream starting at `(x, y)`.
10.2.4 The Layout Protocol

The layout protocol is triggered by layout-frame, which is called when a frame is adopted by a frame manager.

CLIM uses a two-pass algorithm to lay out a pane hierarchy. In the first pass (called space composition), the top-level pane is asked how much space it requires. This may in turn lead to the same question being asked recursively of all the panes in the hierarchy, with the answers being composed to produce the top-level pane’s answer. Each pane answers the query by returning a space requirement (or space-requirement) object, which specifies the pane’s desired width and height, as well as its willingness to shrink or grow along its width and height.

In the second pass (called space allocation), the frame manager attempts to obtain the required amount of space from the host window system. The top-level pane is allocated the space that is actually available. Each pane, in turn, allocates space recursively to each of its descendants in the hierarchy according to the pane’s rules of composition.

For many types of panes, the application programmer can indicate the space requirements of the pane at creation time by using the space requirement options, as well as by calling the change-space-requirements function. Panes are used to display application-specific information, so the application can determine how much space should normally be given to them.

Other pane types automatically calculate their space needs based on the information they have to present. For example, the space requirement for a label pane is a function of the text to be displayed.

A composite pane calculates its space requirement based on the requirements of its children and its own particular rule for arranging them. For example, a pane that arranges its children in a vertical stack would return as its desired height the sum of the heights of its children. Note, however, that a composite pane is not required by the layout protocol to respect the space requests of its children; in fact, composite panes aren’t even required to ask their children.

Space requirements are expressed for each of the two dimensions as a preferred size, a minimum size below which the pane cannot be shrunk, and a maximum size above which the pane cannot be grown. (The minimum and maximum sizes can also be specified as relative amounts.) All sizes are specified as a real number indicating the number of device units (such as pixels).
space-requirement [Protocol Class]
Summary: The protocol class of all space requirement objects. There are one or more subclasses of space-requirement with implementation-dependent names that implement space requirements. The exact names of these classes is explicitly unspecified. If you want to create a new class that behaves like a space requirement, it should be a subclass of space-requirement. Subclasses of space-requirement must obey the space requirement protocol.

All the instantiable space requirement classes provided by CLIM are immutable.

make-space-requirement [Function]
Arguments: &key (width 0) (max-width 0) (min-width 0) (height 0) (max-height 0) (min-height 0)
Summary: Constructs a space requirement object with the given characteristics :width, :height, and so on.

space-requirement-width [Function]
Arguments: space-req

(setf space-requirement-width) [Function]
Arguments: size space-req

space-requirement-max-width [Function]
Arguments: space-req

(setf space-requirement-max-width) [Function]
Arguments: size space-req

space-requirement-min-width [Function]
Arguments: space-req

(setf space-requirement-min-width) [Function]
Arguments: size space-req

space-requirement-height [Function]
Arguments: space-req

Panes and Gadgets
(setf space-requirement-height)  [Function]
Arguments:size space-req

space-requirement-max-height  [Function]
Arguments:space-req

(setf space-requirement-max-height)  [Function]
Arguments:size space-req

space-requirement-min-height  [Function]
Arguments:space-req

(setf space-requirement-min-height)  [Function]
Arguments:size space-req
Summary: These read or modify the components of the space requirement space-req.

space-requirement-components  [Generic Function]
Arguments:space-req
Summary: Returns the components of the space requirement space-req as six values: the width, minimum width, maximum width, height, minimum height, and maximum height.

space-requirement-combine  [Function]
Arguments:function sr1 sr2
Summary: Returns a new space requirement, each component of which is the result of applying the function function to each of the components of the two space requirements sr1 and sr2.

function is a function of two arguments, both of which are real numbers. It has dynamic extent.

space-requirement+  [Function]
Arguments:sr1 sr2
Summary: Returns a new space requirement whose components are the sum of each of the components of the two space requirements sr1 and sr2.

space-requirement+*  [Function]
Arguments:space-req &key width max-width min-width height max-height min-height

<Bold>232CLIM User Guide
Summary: Returns a new space requirement whose components are the sum of each of the components of `space-req` added to the appropriate keyword argument (for example, the width component of `space-req` is added to `width`). This is intended to be a more efficient, spread version of `space-requirement+`.

`change-space-requirements` [Generic Function]
Arguments: `pane &key resize-frame &rest space-req-keys`
Summary: This function can be invoked to indicate that the space requirements for `pane` have changed. Any of the options that applied to the pane at creation time can be passed into this function as well.

`resize-frame` determines whether the frame should be resized to accommodate the new space requirement of the hierarchy. If `resize-frame` is `t`, then `layout-frame` will be invoked on the frame. If `resize-frame` is `nil`, then the frame may or may not get resized depending on the pane hierarchy and the `:resize-frame` option that was supplied to `define-application-frame`.

`note-space-requirements-changed` [Generic Function]
Arguments: `sheet pane`
Summary: This function is invoked whenever `pane`’s space requirements have changed. `sheet` must be the parent of `pane`. Invoking this function essentially means that `compose-space` will be reinvoked on `pane`, and it will reply with a space requirement that is not equal to the reply that was given on the last call to `compose-space`.

This function is automatically invoked by `change-space-requirements` in the cases that `layout-frame` isn’t invoked. In the case that `layout-frame` is invoked, it isn’t necessary to call `note-space-requirements-changed`, since a complete re-layout of the frame will be executed.

`changing-space-requirements` [Macro]
Arguments: `(&key resize-frame layout) &body body`
Summary: This macro supports batching the invocation of the layout protocol by calls to `change-space-requirements`. Within the body, all calls to `change-space-requirements` change the internal structures of the pane and are recorded. When the body is exited, the layout protocol is invoked appropriately. `body` may have zero or more declarations as its first forms.
compose-space  [Generic Function]
Arguments: pane
Summary: During the space composition pass, a composite pane will typically ask each of its children how much space it requires by calling \texttt{compose-space}. They answer by returning \texttt{space-requirement} objects. The composite will then form its own space requirement by composing the space requirements of its children according to its own rules for laying out its children.

allocate-space  [Generic Function]
Arguments: pane width height
Summary: During the space allocation pass, a composite pane will arrange its children within the available space and allocate space to them according to their space requirements and its own composition rules by calling \texttt{allocate-space} on each of the child panes. \texttt{width} and \texttt{height} are the width and height of \texttt{pane} in device units.

10.3 Extended Stream Panes

In addition to the various layout panes and gadgets, an application usually needs some space to display textual and graphic output as well as to receive application-specific input from the user. For example, a paint program needs a “canvas” pane for displaying the picture and handling “mouse strokes.” This can be accomplished in CLIM through the use of extended stream panes.

This section describes the basic CLIM extended stream pane types. Programmers are free to customize pane behavior by defining subclasses of these pane classes. Writing methods to change the repaint or event-handling behavior is a possible starting place.

10.3.1 Extended Stream Pane Options

CLIM extended stream panes accept the :foreground, :background, and :text-style options as well as those options applicable to layout panes. The space requirement options (:width, :height, and so forth) can also take a size specification of :compute, which causes CLIM to run the display function for the pane and make the pane large enough to hold the output of the display function.
In addition to those listed previously, CLIM extended stream frames accept the following options:

**:display-after-commands** [Option]

**Summary:** This specifies how the display function will be run. If t, the “print” part of the read-eval-print loop runs the display function; this is the default for most pane types. If nil, you are responsible for implementing the display after commands.

Do not use **:display-after-commands** with **accept-values** panes, as the redisplay for those panes is managed at a slightly lower level for efficiency. Avoid code such as the following:

```
(in-package :clim-user)

(define-application-frame test-frame () ()
  (:command-table (test-frame :inherit-from
    (clim:accept-values-pane)))
  (:command-definer t)
  (:panes
    (test-input-pane :accept-values :display-function
      '(clim:accept-values-pane-displayer
        :displayer test-input)
      ;; THIS WILL NOT WORK
      :display-after-commands t)
    (dummy :application)
    (menu :command-menu
      :display-function '(display-command-menu :n-rows 1))
    (mouse :pointer-documentation))
  (:layouts (:default
    (vertically ()
      menu test-input-pane DUMMY mouse))))

(defmethod test-input ((frame test-frame) stream)
  (accept 'integer :stream stream :prompt "prompt" :default 1)
  (terpri stream)
  (accept 'integer :stream stream :prompt "foo" :default 1)
  (terpri stream))

(defun test-it (&key (port (find-port))))
(run-frame-top-level
  (make-application-frame 'test-frame
    :frame-manager
    (find-frame-manager :port port))))
```
**:display-function** [Option]
Summary: This specifies a function to be called in order to display the contents of a CLIM stream pane. CLIM’s default top-level function, `default-frame-top-level`, will invoke the pane’s display function at the appropriate time (see the :display-time option). The value of this option is either the name of a function to invoke, or a cons whose car is the name of a function and whose cdr is additional arguments to the function. The function will be invoked on the frame, the pane, and the additional function arguments, if any. The default for this option is nil.

**:display-time** [Option]
Summary: This tells CLIM when the pane’s display function should be run. If it is :command-loop, CLIM erases the pane’s contents and runs the display function after each time a frame command is executed. If it is t, the pane is displayed once but not again until `pane-needs-redisplay` is called on the pane. If it is nil, CLIM waits until it is explicitly requested, either via `pane-needs-redisplay` or `redisplay-frame-pane`. The default value varies according to the pane type.

**:display-string** [Option]
Summary: For title-panes only, you can use this option instead of :display-function to specify a constant string to be displayed in the title-pane.

**:incremental-redisplay** [Option]
Summary: When t, the redisplay function will initially be executed inside of an invocation to updating-output and the resulting output record will be saved. Subsequent calls to `redisplay-frame-pane` will simply use `redisplay` to redisplay the pane. The default for this option is nil.

**:text-margin** [Option]
Summary: This specifies the default text margin, that is, how much space is left around the inside edge of the pane. The default for :text-margin is the width of the window.

**:vertical-spacing** [Option]
Summary: This specifies the default vertical spacing for the pane, that is, how much space there is between each text line. The default for :vertical-spacing is 2.
:end-of-line-action [Option]
Summary: This specifies the end-of-line action to be used. The default is :wrap. (The other possible value is :allow.)

:end-of-page-action [Option]
Summary: This specifies the end-of-page action to be used. The default is :scroll. (The other possible value is :allow.)

:output-record [Option]
Summary: This names the output record class to be used for the output history of the pane. The default is standard-tree-output-history.

:draw [Option]
:record [Option]
Summary: These options specify whether the pane should initially allow drawing and output recording, respectively. The default for both options is t.

10.3.2 Extended Stream Pane Classes

clim-stream-pane [Leaf Pane]
Summary: This class implements a pane that supports the CLIM graphics, extended input and output, and output recording protocols. Any extended stream panes used will most commonly be subclasses of this class.

The five following panes classes are subclasses of clim-stream-pane. Fundamentally, these panes have the same capabilities; however, by convention, the different pane classes have distinct roles. For instance, interactor panes are used for standard input, whereas application panes, by default, specify the destination for standard output.

interactor-pane [Leaf Pane]
Summary: The pane class that implements “interactor” panes. The default method for frame-standard-input will return the first pane of this type.

The default for :display-time is nil and for :scroll-bars is :vertical.
application-pane
Summary: The pane class that implements “application” panes. The default method for frame-standard-output will return the first pane of this type.

The default for :display-time is :command-loop and for :scroll-bars is t.

command-menu-pane
Summary: The pane class that implements command menu panes that are not menu bars. The default display function for panes of this type is display-command-menu.

For command-menu-pane, the default for :display-time is :command-loop, the default for :incremental-redisplay is t, and the default for :scroll-bars is t.

title-pane
Summary: The pane class that implements a title pane. The default display function for panes of this type is display-title. If the title to be displayed will not change, it can be supplied using the option :display-string described in Section 10.3.1, “Extended Stream Pane Options”. If neither :display-function or :display-string is supplied, the title will be taken from frame-pretty-name (see Section 9.9.1, “Finding Frame Managers”).

The default for :display-time is t and for :scroll-bars is nil.

pointer-documentation-pane
Summary: The pane class that implements the pointer documentation pane.

The default for :display-time is nil and for :scroll-bars is nil.

10.3.3 Making CLIM Extended Stream Panes

Most CLIM extended stream panes will contain more information than can be displayed in the allocated screen space, so scroll bars are nearly always desirable. CLIM therefore provides a convenient form for creating composite panes that include a CLIM stream pane, scroll bars, labels, and so forth. For window stream pane functions, see Section 13.7, “CLIM Window Stream Pane Functions.”

make-clim-stream-pane
Arguments: &rest options &key type label scroll-bars &allow-other-keys
Summary: Creates a pane of type type, which defaults to clim-stream-pane. If label is supplied, it is a string used to label the pane. scroll-bars may be t to indicate that both vertical and horizontal scroll bars should be included, :vertical (the default) to indicate that vertical scroll bars should be included, or :horizontal to indicate that horizontal scroll bars should be included.

The other options may include all the valid CLIM extended stream pane options.

make-clim-interactor-pane  [Function]
Arguments: &rest options
Summary: Like make-clim-stream-pane, but the type is forced to be interactor-pane.

make-clim-application-pane  [Function]
Arguments: &rest options
Summary: Like make-clim-stream-pane, but the type is forced to be application-pane.

10.4  Defining A New Pane Type: Leaf Panes

To define a gadget pane implementation, first define the appearance and layout behavior of the gadget, next define the callbacks, and last define the specific user interactions that trigger the callbacks.

For example, to define an odd new kind of button that displays itself as a circle and is activated whenever the mouse is moved over it, proceed as follows:

;;;; A new kind of button
(defclass sample-button-pane (gadget-pane) ())

;;;; An arbitrary size parameter
(defparameter *sample-button-radius* 10)

;;;; Define the sheet’s repaint method to draw the button.
(defmethod handle-repaint ((button sample-button-pane) region &key medium &allow-other-keys)
  (let ((radius *sample-button-radius*)
        (half (round *sample-button-radius* 2)))
    ;; Larger circle with small one in the center.
    (draw-circle* medium radius radius radius :filled nil)
    (draw-circle* medium radius radius half :filled t)))

Panes and Gadgets
Define the pane’s compose-space method to always request the fixed size of the pane.

```lisp
(defun compose-space ((pane sample-button-pane))
  (make-space-requirement :width (* 2 *sample-button-radius*)
                           :height (* 2 *sample-button-radius*))
```

The previous code is enough to allow you to instantiate the button pane in an application frame. It will fit in with the space composition protocol of, for example, an `hbox-pane`. It will display itself as two nested circles.

The next step is to define the callbacks supported by this gadget, and the user interaction that triggers them. The resulting pane is a `leaf pane`.

```lisp
(defun value-change-callback ((button sample-button-pane) client id value)
  (declare (ignore client id value)))
```

```lisp
(defun enter-region ((pane sample-button-pane) &key &allow-other-keys)
  (value-change-callback pane (gadget-client pane) (gadget-id pane) nil))
```

10.5 Gadgets

**Gadgets** are panes that implement such common toolkit components as push buttons or scroll bars. Each gadget class has a set of associated generic functions that serve the same role that callbacks serve in traditional toolkits. (A callback is a function that informs an application that one of its gadgets has been used.) For example, a push button has an “activate” callback function that is invoked when its button is “pressed;” a scroll bar has a “value changed” callback that is invoked after its indicator has been moved.

The gadget definitions specified by CLIM are abstract; that is, the gadget definition does not specify the exact user interface of the gadget, but only specifies the semantics that the gadget should provide. For instance, it is not defined whether the user clicks on a push button with the mouse, or moves the mouse over the button and then presses some key on the keyboard to invoke the “activate” callback. Each toolkit implementation will specify the
look and feel of their gadgets. Typically, the look and feel will be derived directly from the underlying toolkit.

Each of CLIM’s abstract gadgets has at least one standard implementation that is written using the facilities provided solely by CLIM itself. The gadgets’ appearances are achieved via calls to the CLIM graphics functions, and their interactive behavior is defined in terms of the CLIM input event processing mechanism. Since these gadget implementations are written entirely in terms of CLIM, they are portable across all supported CLIM host window systems. Furthermore, since the specific look and feel of each such gadget is “fixed” in CLIM Lisp code, the gadget implementation will look and behave the same in all environments.

10.5.1 Abstract Gadgets

The push button and slider gadgets alluded to previously are abstract gadgets. The callback interface to all of the various implementations of the gadget is defined by the abstract class. In the :panes clause of define-application-frame, the abbreviation for a gadget is the name of the abstract gadget class.

At pane creation time (that is, make-pane), the frame manager resolves the abstract class into a specific implementation class; the implementation classes specify the detailed look and feel of the gadget. Each frame manager will keep a mapping from abstract gadgets to an implementation class; if the frame manager does not implement its own gadget for the abstract gadget classes in the following sections, it will use the portable class provided by CLIM. Since every implementation of an abstract gadget class is a subclass of the abstract class, they all share the same programmer interface.

10.5.1.1 Using Gadgets

Every gadget has a client that is specified when the gadget is created. The client is notified via the callback mechanism when any important user interaction takes place. Typically, a gadget’s client will be an application frame or a composite pane. Each callback generic function is invoked on the gadget, its client, the gadget id, and other arguments that vary depending on the callback.

For example, the argument list for activate-callback looks like (gadget client gadget-id). Assuming the programmer has defined an application frame called button-test that has a CLIM stream pane in the slot output-pane, she could write the following method:
(defmethod activate-callback
  ((button push-button) (client button-test) gadget-id)
  (with-slots (output-pane) client
    (format output-pane
      "The button ~S was pressed, client ~S, id ~S."
      button client gadget-id)))

One problem with this example is that it differentiates on the class of the gadget, not on the particular gadget instance. That is, the same method will run for every push button that has the button-test frame as its client.

One way to distinguish between the various gadgets is via the gadget id, which is also specified when the gadget is created. The value of the gadget id is passed as the third argument to each callback generic function. In this case, if we have two buttons, we might install start and stop as the respective gadget ids and then use eql specializers on the gadget ids. We could then refine the previous method as:

(defmethod activate-callback
  ((button push-button) (client button-test)
   (gadget-id (eql 'start)))
  (start-test client))

(defmethod activate-callback
  ((button push-button) (client button-test)
   (gadget-id (eql 'stop)))
  (stop-test client))

;; Create the start and stop push buttons
(make-pane 'push-button
  :label "Start"
  :client frame :id 'start)
(make-pane 'push-button
  :label "Stop"
  :client frame :id 'stop)

Another way to distinguish between gadgets is to specify explicitly what function should be called when the callback is invoked. This is done by supplying an appropriate initarg when the gadget is created. The previous example could then be written as follows:

;; No callback methods needed; just create the push buttons.
(make-pane 'push-button
  :label "Start"
  :client frame :id 'start
  :activate-callback
  #'(lambda (gadget)
      (start-test (gadget-client gadget)))))
(make-pane 'push-button
  :label "Stop"
  :client frame :id 'stop
  :activate-callback
  #'(lambda (gadget)
    (stop-test (gadget-client gadget))))

10.5.1.2 Implementing Gadgets

The following shows how a push button gadget might be implemented.

;; A PUSH-BUTTON uses the ACTIVATE-CALLBACK, and has a label.
;; This is the abstract class
(defclass push-button (action-gadget labelled-gadget) ())

;; Here is a concrete implementation of a PUSH-BUTTON.
;; The "null" frame manager create a pane of type PUSH-BUTTON-PANE when
;; asked to create a PUSH-BUTTON.
(defclass push-button-pane
  (push-button leaf-pane space-requirement-mixin)
  (show-as-default :initarg :show-as-default
                   :accessor push-button-show-as-default)
  (armed :initform nil))

;; General highlight-by-inverting method
(defmethod highlight-button ((pane push-button-pane) medium)
  (with-bounding-rectangle* (left top right bottom) (sheet-region pane)
    (draw-rectangle* medium left top right bottom
                       :ink +flipping-ink+ :filled t)))

;; Compute the amount of space required by a PUSH-BUTTON-PANE
(defmethod compose-space ((pane push-button-pane) &key width height)
  (multiple-value-bind (width height)
                          (compute-gadget-label-size pane)
                          (make-space-requirement :width width :height height)))
;; This gets invoked to draw the push button.
(defmethod repaint-sheet ((pane push-button-pane) region)
  (declare (ignore region))
  (with-sheet-medium (medium pane)
    (let ((text (gadget-label pane))
           (text-style (slot-value pane 'text-style))
           (armed (slot-value pane 'armed))
           (region (sheet-region pane))
           (multiple-value-call #'draw-rectangle*
                                medium (bounding-rectangle*
                                         (sheet-region pane))
                                :filled nil)
      (draw-textmedium
       text
       (clim-utils::bounding-rectangle-center region)
       :text-style text-style
       :align-x ':center
       :align-y ':top)
    (when (eql armed ':button-press)
      (highlight-button pane medium))))))

;; When we enter the push button’s region, arm it.
(defmethod handle-event ((pane push-button-pane) (event pointer-enter-event))
  (with-slots (armed) pane
    (unless armed
      (setf armed t)
      (armed-callback
       pane (gadget-client pane) (gadget-id pane))))))

;; When we leave the push button’s region, disarm it.
(defmethod handle-event ((pane push-button-pane) (event pointer-exit-event))
  (with-slots (armed) pane
    (when armed
      (when (eql armed ':button-press)
        (highlight-button pane medium))
      (setf armed nil)
      (disarmed-callback
       pane (gadget-client pane) (gadget-id pane))))))

<Bold>244CLIM User Guide
;;; When the user presses a pointer button, ensure that the button
;;; is armed, and highlight it.
(defun handle-event ((pane push-button-pane)
  (event pointer-button-press-event))
  (with-slots (armed) pane
    (unless armed
      (setf armed ':button-press)
      (armed-callback
        pane (gadget-client pane) (gadget-id pane))
      (with-sheet-medium (medium pane)
        (highlight-button pane medium))))

;;; When the user releases the button and the button is still armed,
;;; call the activate callback.
(defun handle-event ((pane push-button-pane)
  (event pointer-button-release-event))
  (with-slots (armed) pane
    (when (eql armed ':button-press)
      (activate-callback
        pane (gadget-client pane) (gadget-id pane))
      (setf armed t)
      (with-sheet-medium (medium pane)
        (highlight-button pane medium))))

10.5.2 Basic Gadget Classes

The following are the basic gadget classes upon which all abstract gadgets are built.

**gadget**

[Protocol Class]

Summary: The protocol class that corresponds to a gadget, a subclass of **pane**. If you want to create a new class that behaves like a gadget, it should be a subclass of **gadget**. Subclasses of **gadget** must obey the gadget protocol.

All of the subclasses of **gadget** are mutable.

**gadgetp**

[Function]

Arguments: **object**

Summary: Returns **t** if **object** is a gadget; otherwise, it returns **nil**.

**basic-gadget**

[Class]

Summary: The base class on which all CLIM gadget classes are built.
Summary: All subclasses of gadget must handle these four initargs, which are used to specify, respectively, the gadget id, client, armed callback, and disarmed callback of the gadget.

**gadget-id**

Arguments: gadget

Summary: Returns (or sets) the gadget id of the gadget gadget. The id is typically a simple Lisp object that uniquely identifies the gadget.

**gadget-client**

Arguments: gadget

Summary: Returns the client of the gadget gadget. The client is usually an application frame, but it could be another gadget (for example, a push button contained in a radio box).

**gadget-armed-callback**

Arguments: gadget

**gadget-disarmed-callback**

Arguments: gadget

Summary: Returns the functions that will be called when the armed or disarmed callback, respectively, are invoked. These functions will be invoked with a single argument, the gadget.
When these functions return \texttt{nil}, there is no armed (or disarmed) callback for the gadget.

\textbf{armed-callback} \hspace{1cm} \texttt{[Callback]}

\textit{Arguments:} \texttt{gadget client gadget-id}

\textbf{disarmed-callback} \hspace{1cm} \texttt{[Callback]}

\textit{Arguments:} \texttt{gadget client gadget-id}

\textit{Summary:} These callbacks are invoked when the gadget \texttt{gadget} is, respectively, armed or disarmed. The exact definition of arming and disarming varies from gadget to gadget, but typically a gadget becomes armed when the pointer is moved into its region, and disarmed when the pointer moves out of its region.

The default methods (on \texttt{basic-gadget}) call the function stored in \texttt{gadget-armed-callback} or \texttt{gadget-disarmed-callback} with one argument, the gadget.

\textbf{activate-gadget} \hspace{1cm} \texttt{[Generic Function]}

\textit{Arguments:} \texttt{gadget}

\textit{Summary:} Causes the host gadget to become active, that is, available for input.

\textbf{deactivate-gadget} \hspace{1cm} \texttt{[Generic Function]}

\textit{Arguments:} \texttt{gadget}

\textit{Summary:} Causes the host gadget to become inactive, that is, unavailable for input. In some environments this may cause the gadget to become grayed over; in others, no visual effect may be detected.

\textbf{gadget-active-p} \hspace{1cm} \texttt{[Generic Function]}

\textit{Arguments:} \texttt{gadget}

\textit{Summary:} Returns \texttt{t} if \texttt{gadget} is active, that is, has been activated with \texttt{activate-gadget}.

\textbf{note-gadget-activated} \hspace{1cm} \texttt{[Generic Function]}

\textit{Arguments:} \texttt{client gadget}

\textit{Summary:} This function is invoked after a gadget is made active.

\textbf{note-gadget-deactivated} \hspace{1cm} \texttt{[Generic Function]}

\textit{Arguments:} \texttt{client gadget}

\textit{Summary:} This function is invoked after a gadget is made inactive.
value-gadget [Class]
Summary: The class used by gadgets that have a value; a subclass of basic-gadget.

:value [Initarg]
:value-changed-callback [Initarg]
Summary: All subclasses of value-gadget must handle these two initargs, which specify, respectively, the initial value and the value changed callback of the gadget.

gadget-value [Generic Function]
Arguments: value-gadget
Summary: Returns the value of the gadget value-gadget. The interpretation of the value varies from gadget to gadget. For example, a scroll bar’s value might be a number between 0 and 1, while a toggle button’s value is either t or nil. (The documentation of each individual gadget specifies how to interpret the value.)

(setf gadget-value) [Generic Function]
Arguments: value value-gadget &key invoke-callback
Summary: Sets the gadget’s value to the specified value. In addition, if invoke-callback is t (the default is nil), the value-changed callback for the gadget is invoked. The syntax for using (setf gadget-value) is:

(setf (gadget-value gadget :invoke-callback t) new-value)

gadget-value-changed-callback [Generic Function]
Arguments: value-gadget
Summary: Returns the function that will be called when the value changed callback is invoked. This function will be invoked with two arguments, the gadget and the new value.

When this function returns nil, there is no value-changed callback for the gadget.

value-changed-callback [Callback]
Arguments: value-gadget client gadget-id value
Summary: This callback is invoked whenever the value of a gadget is changed.

The default method (on value-gadget) calls the function stored in gadget-value-changed-callback with two arguments, the gadget and the new value.
CLIM implements or inherits a method for `value-changed-callback` for every gadget that is a subclass of `value-gadget`.

### action-gadget

**[Class]**

**Summary:** The class used by gadgets that perform some kind of action, such as a push button; a subclass of `basic-gadget`.

### :activate-callback

**[Initarg]**

**Summary:** All subclasses of `action-gadget` must handle this initarg, which specifies the activate callback of the gadget.

### gadget-activate-callback

**[Generic Function]**

**Arguments:** `action-gadget`

**Summary:** Returns the function that will be called when the gadget is activated. This function will be invoked with one argument, the gadget.

When this function returns `nil`, there is no value-activate callback for the gadget.

### activate-callback

**[Callback]**

**Arguments:** `action-gadget client gadget-id`

**Summary:** This callback is invoked when the gadget is activated.

The default method (on `action-gadget`) calls the function stored in `gadget-activate-callback` with one argument, the gadget.

CLIM implements or inherits a method for `activate-callback` for every gadget that is a subclass of `action-gadget`.

### oriented-gadget-mixin

**[Class]**

**Summary:** The class that is mixed into a gadget that has an orientation associated with it, for example, a slider. This class is not intended to be instantiated.

### :orientation

**[Initarg]**

**Summary:** All subclasses of `oriented-gadget-mixin` must handle this initarg, which is used to specify the orientation of the gadget. The value is either `:horizontal` or `:vertical`.
**gadget-orientation**  
*Generic Function*  
**Arguments:** oriented-gadget  
**Summary:** Returns the orientation of the gadget oriented-gadget. Typically, this will be a keyword such as :horizontal or :vertical.

**labelled-gadget-mixin**  
*Class*  
**Summary:** The class that is mixed into a gadget that has a label, for example, a push button. This class is not intended to be instantiated.

**:label**  
*Initarg*  

**:align-x**  
*Initarg*  

**:align-y**  
*Initarg*  
**Summary:** All subclasses of labelled-gadget-mixin must handle these initargs, which are used to specify the label and its x and y alignment. Labelled gadgets will also have a text style for the label, but this is managed by the usual text-style mechanism for panes.

**gadget-label**  
*Generic Function*  
**Arguments:** labelled-gadget  

**(setf gadget-label)**  
*Generic Function*  
**Arguments:** label labelled-gadget  
**Summary:** Returns (or sets) the label of the gadget labelled-gadget. The label must be a string. Changing the label of a gadget may result in invoking the layout protocol on the gadget and its ancestor sheets.

**gadget-label-align-x**  
*Generic Function*  
**Arguments:** labelled-gadget  

**(setf gadget-label-align-x)**  
*Generic Function*  
**Arguments:** alignment labelled-gadget  

**gadget-label-align-y**  
*Generic Function*  
**Arguments:** labelled-gadget
(setf gadget-label-align-y)  [Generic Function]
Arguments:alignment labelled-gadget
Summary: Returns (or sets) the alignment of the label of the gadget labelled-gadget. Changing the alignment a gadget may result in invoking the layout protocol on the gadget and its ancestor sheets.

gadget-label-text-style  [Generic Function]
Arguments:labelled-gadget

(setf gadget-label-text-style)  [Generic Function]
Arguments:text-style labelled-gadget
Summary: Returns (or sets) the text style of the label of the gadget labelled-gadget. This must be a CLIM text style object. Changing the label text style of a gadget may result in invoking the layout protocol on the gadget and its ancestor sheets.

range-gadget-mixin  [Class]
Summary: The class that is mixed into a gadget that has a range, for example, a slider.

:min-value  [Initarg]

:max-value  [Initarg]
Summary: All subclasses of range-gadget-mixin must handle these two initargs, which are used to specify the minimum and maximum value of the gadget.

gadget-min-value  [Generic Function]
Arguments:range-gadget

(setf gadget-min-value)  [Generic Function]
Arguments:min-value range-gadget
Summary: Returns (or sets) the minimum value of the gadget range-gadget, a real number.

gadget-max-value  [Generic Function]
Arguments:range-gadget

(setf gadget-max-value)  [Generic Function]
Arguments:max-value range-gadget

Panes and Gadgets
Summary: Returns (or sets) the maximum value of the gadget \textit{range-gadget}, a real number.

\texttt{gadget-range} \hspace{1cm} [Generic Function]
\texttt{Arguments:range-gadget}
Summary: Returns the range of \textit{range-gadget}, that is, the difference of the maximum value and the minimum value.

\texttt{gadget-range*} \hspace{1cm} [Generic Function]
\texttt{Arguments:range-gadget}
Summary: Returns the minimum value and the maximum value of \textit{range-gadget} as two values.

10.5.3 Abstract Gadget Classes

CLIM supplies a set of gadgets that have been designed to be compatible with a variety of user interface toolkits, including Xt widget-based toolkits (such as Motif), OpenLook, and the MacToolbox.

Each gadget maps to an implementation-specific object that is managed by the underlying toolkit. For example, when a CLIM program manipulates an object of class \texttt{scroll-bar}, the underlying implementation-specific object might be a Motif \texttt{ScrollBar} widget. As events are processed on the underlying object, the corresponding generic operations are applied to the Lisp gadget.

Note that not all operations will necessarily be generated by particular toolkit implementations. For example, a user interface toolkit that is designed for a 3-button mouse may generate significantly more gadget events than one designed for a 1-button mouse.

10.5.3.1 The Label Gadget

\texttt{label-pane} \hspace{1cm} [Leaf Pane]

\texttt{labelling} \hspace{1cm} [Macro]
\texttt{Arguments:(&rest options &key label label-alignment &allow-other-keys) &body contents}
Summary: Creates a pane that consists of the specified label, which is a string.
Valid options are :align-x (one of :left, :right, or :center) and :text-style.  
label-alignment may be one of :top or :bottom),

contents must be a single (but possibly compound) pane.

10.5.3.2 The List-Pane and Option-Pane Gadgets

A list pane is a list of buttons. An option pane is a single button that, when pressed, pops up a menu of selections.

**list-pane**  
[Class]  
Summary: The class that implements an abstract list pane. It is a subclass of value-gadget.

:mode  
[Initarg]  
Summary: Either :one-of or :some-of. When it is :one-of, the list pane acts like a radio box; that is, only one item can be selected. When it is :some-of (the default), zero or more items can be selected at a time.

:items  
[Initarg]  
:name-key  
[Initarg]  
:value-key  
[Initarg]  
:test  
[Initarg]  
Summary: The :items initarg specifies a sequence of items to use as the items of the list pane. The name of the item is extracted by the function that is the value of the :name-key initarg, which defaults to princ-to-string. The value of the item is extracted by the function that is the value of the :value-key initarg, which defaults to identity. The :test initarg specifies a function of two argument that is used to compare items; it defaults to eql. For example:

```
(make-pane 'list-pane  
 :value '("Lisp" "C++")  
 :mode :some-of  
 :items '("Lisp" "Fortran" "C" "C++" "Cobol" "Ada")  
 :test 'string=)
```
gadget-value [Generic Function]

Arguments: (button list-pane)
Summary: Returns the single selected item when the mode is :one-of, or a sequence of selected items when the mode is :some-of.

generic-list-pane [Class]
Summary: The class that implements a portable list pane; a subclass of list-pane.

option-pane [Class]
Summary: The class that implements an abstract option pane. It is a subclass of value-gadget.

:items [Initarg]
: name-key [Initarg]
: value-key [Initarg]
: test [Initarg]
Summary: The :items initarg specifies a sequence of items to use as the items of the option pane. The name of the item is extracted by the function that is the value of the :name-key initarg, which defaults to princ-to-string. The value of the item is extracted by the function that is the value of the :value-key initarg, which defaults to identity. The :test initarg specifies a function of two argument that is used to compare items; it defaults to eql.

gadget-value [Generic Function]
Arguments: (button option-pane)
Summary: Returns the single selected item.

generic-option-pane [Class]
Summary: The class that implements a portable option pane; a subclass of option-pane.
10.5.3.3 The Menu-Button Gadget

Note: The Menu-Button gadget is available only in Liquid CLIM.

The menu-button gadget provides similar behavior to the toggle-button gadget, except that it is intended for items in menus. The returned value is generally the item chosen from the menu.

arm-callback will be invoked when the menu button becomes armed (such as when the pointer moves into it, or a pointer button is pressed over it). When the menu button is actually activated (by releasing the pointer button over it), value-changed-callback will be invoked. Finally, disarm-callback will be invoked after value-changed-callback, or when the pointer is moved outside of the menu button.

menu-button [Class]
Summary: The class that implements an abstract menu button. It is a subclass of value-gadget and labelled-gadget-mixin.

menu-button-pane [Class]
Summary: The class that implements a portable menu button; a subclass of menu-button.

10.5.3.4 The Push-Button Gadget

The push-button gadget provides press-to-activate switch behavior.

arm-callback will be invoked when the push button becomes armed (such as when the pointer moves into it, or a pointer button is pressed over it). When the button is actually activated (by releasing the pointer button over it), activate-callback will be invoked. Finally, disarm-callback will be invoked after activate-callback, or when the pointer is moved outside of the button.

push-button [Class]
Summary: The class that implements an abstract push button. It is a subclass of active-gadget and labelled-gadget-mixin.

:show-as-default [Initarg]
Summary: This initializes the “show as default” property for the gadget.
**push-button-show-as-default**  
*Arguments:* `push-button`  
*Summary:* Returns the “show as default” property for the push button gadget. When `t`, the push button will be drawn with a heavy border, which indicates that this button is the “default operation.”

**push-button-pane**  
*Class*  
*Summary:* The class that implements a portable push button; a subclass of `push-button`.

### 10.5.3.5 The Radio-Box and Check-Box Gadgets

A radio box is a special kind of gadget that constrains one or more toggle buttons. At any one time, only one of the buttons managed by the radio box may be “on.” A radio box is responsible for laying out its contents (the buttons that it contains). So that the value of the radio box can be properly computed, it is a client of each of its buttons. As the current selection changes, the previously selected button and the newly selected button both have their `value-changed-callback` handlers invoked.

Like a radio box, a check box is a gadget that constrains a number of toggle buttons, but unlike a radio box, a check box may have zero or more of its buttons selected at a time.

**radio-box**  
*Class*  
*Summary:* The class that implements a radio box. It is a subclass of `value-gadget` and `oriented-gadget-mixin`.

:current-selection  
*Initarg*  
*Summary:* This is used to specify which button, if any, should be initially selected.

**radio-box-current-selection**  
*Arguments:* `radio-box`  
*Generic Function*

**(setf radio-box-current-selection)**  
*Arguments:* `button radio-box`  
*Generic Function*

*Summary:* Returns (or sets) the current selection for the radio box. The current selection will be one of the toggle buttons in the box.
**radio-box-selections**  
[Generic Function]  
**Arguments:** radio-box  
**Summary:** Returns a sequence of all the selections in the radio box. The elements of the sequence will be toggle buttons.

**gadget-value**  
[Generic Function]  
**Arguments:** (button radio-box)  
**Summary:** Returns the selected button (i.e., returns the same value as radio-box-current-selection).

**radio-box-pane**  
[Class]  
**Summary:** The class that implements a portable radio box; it is a subclass of radio-box.

**check-box**  
[Class]  
**Summary:** The class that implements a check box. check-box is a subclass of value-gadget and oriented-gadget-mixin.

**:current-selection**  
[Initarg]  
**Summary:** This is used to specify which button, if any, should be initially selected.

**check-box-current-selection**  
[Generic Function]  
**Arguments:** check-box  

**(setf check-box-current-selection)**  
[Generic Function]  
**Arguments:** button check-box  
**Summary:** Returns (or sets) the current selection for the check box. The current selection will be a list of zero or more of the toggle buttons in the box.

**check-box-selections**  
[Generic Function]  
**Arguments:** check-box  
**Summary:** Returns a sequence of all the selections in the check box. The elements of the sequence will be toggle buttons.

**gadget-value**  
[Generic Function]  
**Arguments:** (button check-box)  
**Summary:** Returns the selected buttons as a list (i.e., returns the same value as check-box-current-selection).
check-box-pane

Summary: The class that implements a portable check box; it is a subclass of check-box.

with-radio-box

Arguments: (&rest options &key (type one-of) &allow-other-keys) &body body

Summary: Creates a radio box whose buttons are created by the forms in body. The macro radio-box-current-selection can be wrapped around one of forms in body in order to indicate that that button is the current selection.

A radio box will be created if type is :one-of, a check box if :some-of.

For example, the following creates a radio box with three buttons in it, the second of which is initially selected.

(with-radio-box ()
  (make-pane 'toggle-button :label "Mono")
  (radio-box-current-selection
   (make-pane 'toggle-button :label "Stereo")
   (make-pane 'toggle-button :label "Quad")))

The following simpler form can be used when you do not need to control the appearance of each button closely.

(with-radio-box () "Mono" "Stereo" "Quad")

10.5.3.6 The Scroll-Bar Gadget

The scroll-bar gadget corresponds to a scroll bar.

scroll-bar

Summary: The class that implements a scroll bar. This is a subclass of value-gadget, oriented-gadget-mixin, and range-gadget-mixin.
:drag-callback

:scroll-to-bottom-callback

:scroll-to-top-callback

:scroll-down-line-callback

:scroll-up-line-callback

:scroll-down-page-callback

:scroll-up-page-callback

Summary: Specifies the various callbacks for the scroll bar.

scroll-bar-drag-callback

Arguments: scroll-bar

Summary: Returns the function that will be called when the indicator of the scroll bar is dragged. This function will be invoked with a two arguments, the scroll bar and the new value.

scroll-bar-scroll-to-bottom-callback

Arguments: scroll-bar

scroll-bar-scroll-to-top-callback

Arguments: scroll-bar

scroll-bar-scroll-down-line-callback

Arguments: scroll-bar

scroll-bar-scroll-up-line-callback

Arguments: scroll-bar

scroll-bar-scroll-down-page-callback

Arguments: scroll-bar

Panes and Gadgets
scroll-bar-scroll-up-page-callback  [Generic Function]
Arguments: scroll-bar
Summary: Returns the functions that will be used as callbacks when various parts of the scroll bar are clicked on. These are all functions of one argument, the scroll bar. When any of these functions returns nil, there is no callback of that type for the gadget.

drag-callback  [Callback]
Arguments: scroll-bar client gadget-id value
Summary: This callback is invoked when the value of the scroll bar is changed while the indicator is being dragged. The function stored in scroll-bar-drag-callback is called with two arguments, the scroll bar and the new value.

The value-changed-callback is invoked only after the indicator is released after dragging it.

scroll-to-top-callback  [Callback]
Arguments: scroll-bar client gadget-id

scroll-to-bottom-callback  [Callback]
Arguments: scroll-bar client gadget-id

scroll-up-line-callback  [Callback]
Arguments: scroll-bar client gadget-id

scroll-up-page-callback  [Callback]
Arguments: scroll-bar client gadget-id

scroll-down-line-callback  [Callback]
Arguments: scroll-bar client gadget-id

scroll-down-page-callback  [Callback]
Arguments: scroll-bar client gadget-id
Summary: All the callbacks above are invoked when appropriate parts of the scroll bar are clicked on. Note that each implementation may not have “hot spots” corresponding to each of these callbacks.
gadget-value [Generic Function]
Arguments: (button scroll-bar)
Summary: Returns a real number within the specified range.

scroll-bar-pane [Class]
Summary: The class that implements a portable scroll bar; it is a subclass of scroll-bar.

10.5.3.7 The Slider Gadget

The slider gadget corresponds to a slider.

slider [Class]
Summary: The class that implements a slider. This is a subclass of value-gadget, oriented-gadget-mixin, range-gadget-mixin, and labelled-gadget-mixin.

:drag-callback [Initarg]
:show-value-p [Initarg]
:decimal-places [Initarg]
Summary: Specifies the drag callback for the slider, whether the slider should show its current value, and how many decimal places to the right of the decimal point should be displayed when the slider is showing its current value.

slider-drag-callback [Generic Function]
Arguments: slider
Summary: Returns the function that will be called when the slider’s indicator is dragged. This function will be invoked with two arguments, the slider and the new value. When this function returns nil, there is no drag callback for the gadget.

drag-callback [Callback]
Arguments: slider client gadget-id value
Summary: This callback is invoked when the value of the slider is changed while the indicator is being dragged. The function stored in slider-drag-callback is called with two arguments, the slider and the new value.
The value-changed-callback is invoked only after the indicator is released after dragging it.

**gadget-value**  
**Arguments:** (button slider)  
**Summary:** Returns a real number that is the value of button.

**slider-pane**  
**Summary:** The class that implements a portable slider; a subclass of slider.

**:number-of-tick-marks**  
**[Initarg]**

**:number-of-quanta**  
**[Initarg]**

**Summary:** Specifies the number of tick marks that should be drawn on the scroll bar, and the number of quanta in the scroll bar. If the scroll bar is quantized, it will consist of discrete (rather than continuous) values.

**Note:** :number-of-tick-marks and :number-of-quanta are available only in Liquid CLIM.

**gadget-show-value-p**  
**Arguments:** slider  
**Summary:** Returns t if the slider shows its value; otherwise, it returns nil.

**Note:** gadget-show-value-p is available only in Liquid CLIM.

### 10.5.3.8 The Text-Field and Text-Editor Gadgets

The text-field gadget corresponds to a small field containing text. The text-editor gadget corresponds to a large field containing multiple lines of text.

**text-field**  
**[Class]**

**Summary:** The class that implements a text field. This is a subclass of value-gadget and action-gadget. The value of a text field is the text string.
:editable-p
Summary: Specifies whether or not the text field can be edited.

gadget-value
Arguments: (value-gadget text-field)
Summary: Returns the resulting string.

text-field-pane
Summary: The instantiable class that implements a portable text field; it is a subclass of text-field.

text-editor
Summary: The instantiable class that implements an abstract large text field. This is a subclass of text-field.

The value of a text editor is the text string.

:ncolumns
:nlines
Summary: Specifies the width and height of the text editor in columns and number of lines.

gadget-value
Arguments: (value-gadget text-editor)
Summary: Returns the resulting string.

text-editor-pane
Summary: The instantiable class that implements a portable text editor; it is a subclass of text-editor.

10.5.3.9 The Toggle-Button Gadget

The toggle-button gadget provides “on/off” switch behavior. This gadget typically appears as a recessed or prominent box. If the box is recessed, the gadget’s value is t; if it is prominent, the value is nil.
arm-callback will be invoked when the toggle button becomes armed (such as when the pointer moves into it, or a pointer button is pressed over it). When the toggle button is actually activated (by releasing the pointer button over it), value-changed-callback will be invoked. Finally, disarm-callback will be invoked after value-changed-callback, or when the pointer is moved outside of the toggle button.

**toggle-button**

Summary: The class that implements an abstract toggle button. It is a subclass of value-gadget and labelled-gadget-mixin.

**:indicator-type**

Summary: This initializes the indicator type property for the gadget.

**toggle-button-indicator-type**

Arguments: toggle-button

Summary: Returns the indicator type for the toggle button. This will be either :one-of or :some-of. The indicator type controls the appearance of the toggle button. For example, many toolkits present a one-of-many choice differently from a some-of-many choice.

**gadget-value**

Arguments: (value-gadget toggle-button)

Summary: Returns t if the button is selected; otherwise, it returns nil.

**toggle-button-pane**

Summary: The class that implements a portable toggle button; a subclass of toggle-button.

### 10.5.4 Integrating Gadgets and Output Records

In addition to gadget panes, CLIM allows gadgets to be used inside of CLIM stream panes. For instance, an accepting-values pane whose fields consist of gadgets may appear in an ordinary CLIM stream pane.

Note that many of the functions in the output record protocol must correctly manage the case where output records contain gadgets. For example, (setf output-record-position) may need to notify the host window system that the toolkit object representing the gadget has moved, window-clear needs to deactivate any gadgets, and so forth.
**gadget-output-record**

[Class]

Summary: The instantiable class that represents an output record class that contains a gadget. This is a subclass of `output-record`.

**with-output-as-gadget**

[Macro]

Arguments: `(stream &rest options) &body body`

Summary: Invokes `body` to create a gadget, and then creates a gadget output record that contains the gadget and installs it into the output history of the output recording stream `stream`. The returned value of `body` must be the gadget.

The options in `options` are passed as initargs to the call to `invoke-with-new-output-record` that is used to create the gadget output record.

The `stream` argument is not evaluated, and must be a symbol that is bound to an output recording stream. If `stream` is `t`, `*standard-output*` is used. `body` may have zero or more declarations as its first forms.

For example, the following could be used to create an output record containing a radio box that contains several toggle buttons:

```lisp
(with-output-as-gadget
 (stream)
 (let* ((radio-box
   (make-pane 'radio-box
     :client stream :id 'radio-box)))
   (dolist (item sequence)
     (make-pane 'toggle-button
       :label (princ-to-string (item-name item))
       :value (item-value item)
       :id item :parent radio-box))
   radio-box))
```

An example of a push button that calls back into the presentation type system to execute a command might be as follows:

```lisp
(with-output-as-gadget
 (stream)
 (make-pane 'push-button
   :label "Click here to exit"
   :activate-callback
   #'(lambda (button)
      (frame-exit (pane-frame button))))
```
Chapter 11  Commands
11.1 Introduction to CLIM Commands

In CLIM, users interact with applications through the use of commands. Commands are a way of representing an operation in an application.

Commands are performed by the command loop, which accepts input of presentation type command and then executes the accepted command. Section 11.3, “Command Objects,” discusses how commands are represented.

CLIM also supports actions, which are performed directly by the user interface. Actions are seldom necessary, as it is usually the functionality of commands that is desired. See the macro define-presentation-action for a discussion of when presentation actions are appropriate.

CLIM supports four main styles of interaction:

■ Mouse interaction via command menus
  A command is invoked by clicking on an item in a menu.

■ Mouse interaction via command translators
  A command can be invoked by clicking on any object displayed by the interface. The particular combination of mouse-buttons and modifier keys (e.g., SHIFT, CONTROL) is called a gesture. As part of the presentation system, a command translator turns a gesture on an object into a command.

■ Keyboard interaction using a command-line processor
  The user types a complete textual representation of command names and arguments. The text is parsed by the command-line processor to form a command. A special character (usually NEWLINE) indicates to the command-line processor that the text is ready to be parsed.

■ Keyboard interaction using keystroke accelerators
  A single keystroke invokes the associated command.

The choice of interaction styles is independent of the command loop or the set of commands. The relationship between a user’s interactions and the commands to be executed is governed by command tables. A command table is an object that mediates between a command input context (e.g., the top level of an application frame), a set of commands, and these interaction styles.
For simple CLIM applications, define-application-frame will automatically create a command table and a top-level command input context, and define a command-defining macro for you.

Following a discussion of the simple approach, this chapter discusses command tables and the command processor in detail. This information is provided for the curious and for those who feel they require further control over their application’s interactions. These are some circumstances that might suggest something beyond the simple approach:

- Your application requires more than one command table if, for example, it has multiple modes with different sets of commands available in each mode.
- If you have sets of commands that are common among several modes or even among several applications, you could use several command tables and inheritance to help organize your command sets.
- Your application may be complex enough that you may want to develop more powerful tools for examining and manipulating command tables.

If you do not require this level of detail, only read Section 11.2, “Defining Commands the Easy Way.”

### 11.2 Defining Commands the Easy Way

The easiest way to define commands is to use define-application-frame, which automatically creates a command table for your application. This behavior is controlled by the :command-table option. It also defines the command-defining macro you will use to define the commands for your application. This is controlled by the :command-definer option.

This command-definer macro behaves similarly to define-command, but automatically uses your application’s command table, so you needn’t specify one.

Here is a code fragment illustrating the use of define-application-frame, which defines an application named editor. A command table named editor-command-table is defined to mediate the user’s interactions with the editor application. define-application-frame also defines a macro named define-editor-command, which you will use to define commands for the editor application and install them in the command table editor-command-table.

```lisp
(clim:define-application-frame editor () ()
  (:command-table editor-command-table)
  (:command-definer define-editor-command) ...)
```

Commands
Note that for this particular example, the :command-table and :command-definer options are not specified, since the names that they specify are the ones that would be generated by default. Provide these options only when you want different names than the default ones, you don’t want a command definer, or you want to specify which command tables the application’s command table inherits from. See the macro define-application-frame, in Section 9.2, “Defining CLIM Application Frames,” for a description of these options.

11.2.1 Command Names and Command Line Names

Every command has a command name, which is a symbol. The symbol names the function that implements the command. The body of the command is the function definition of that symbol.

By convention, commands are named with a com- prefix, although CLIM does not enforce this convention.

To avoid collisions among command names, each application should live in its own package; for example, there might be several commands named com-show-chart defined for each of a spreadsheet, a navigation program, and a medical application.

CLIM supports a command line name which is the "command" that the end user sees and uses, as opposed to the construct that is the command’s actual name. For example, the command com-show-chart would have a command-line name of Show Chart. When defining a command using define-command (or the application’s command defining macro), you can have a command line name generated automatically. As you can see from this example, the automatically generated command line name consists of the command’s name with the hyphens replaced by spaces and the words capitalized. Any com- prefix is removed.

11.2.2 The Command-Defining Macro

The define-editor-command macro, automatically generated by the define-application-frame code previously, is used to define a command for the editor application. It is just like define-command, except that you don’t need to specify editor-command-table as the command table in which to define the command. define-editor-command will automatically use editor-command-table.
Through the appropriate use of the options to `define-editor-command` (see `define-command` for details), you can provide the command via any number of the previously mentioned interaction styles. For example, you could install the command in the editor application’s menu, as well as specifying a single keystroke command accelerator character for it.

The following example defines a command whose command name is `com-save-file`. The `com-save-file` command will appear in the application’s command menu as Save File. (The command-menu name is derived from the command name in the same way as the command-line name.) The single keystroke CONTROL-s is also defined to invoke the command.

```
(define-editor-command
  (com-save-file :menu t
                 :keystroke #\c-\s) () ...)
```

Here, a command line name of Save File is associated with the `com-save-file` command. The user can then type Save File to the application’s interaction pane to invoke the command.

```
(define-editor-command
  (com-save-file :name "Save File") () ...)
```

Since the command processor works by establishing an input context of presentation type `command` and executing the resulting input, any displayed presentation can invoke a command, so long as there is a translator defined that translates from the presentation type of the presentation to the presentation type `command`. In this way, you can associate a command with a pointer gesture when it is applied to a displayed presentation. (See Chapter 8, “Presentation Translators in CLIM,” for details.)

```
define-presentation-to-command-translator
  [Macro]
Arguments: name (from-type command-name command-table &key (:gesture :select)
             :tester :documentation :pointer-documentation (:menu t) :priority (:echo t)) arglist &body body
Summary: Defines a presentation translator that translates a displayed presentation into a command.
```

Commands
11.3 Command Objects

A command is an object that represents a single user interaction. Each command has a command name, which is a symbol. A command can also have both positional and keyword arguments.

CLIM represents commands as command objects. The internal representation of a command object is a cons of the command name and a list of the command’s arguments, and is therefore analogous to a Lisp expression. Functions are provided for extracting the command name and the arguments list from a command object:

command-name [Function]
Arguments: command
Summary: Given a command object command, returns the command name.

command-arguments [Function]
Arguments: command
Summary: Given a command object command, returns the command’s arguments.

It is possible to represent a command for which some of the arguments have not yet been specified. The value of the symbol *unsupplied-argument* is used in place of any argument that has not yet been specified.

partial-command-p [Function]
Arguments: command
Summary: Returns t if the command is a partial command, that is, has any occurrences of *unsupplied-argument-marker* in it. Otherwise, this function returns nil.

One can think of define-command as defining templates for command objects. It defines a symbol as a command name and associates with it the presentation types corresponding to each of the command’s arguments.

define-command [Macro]
Arguments:name-and-options arguments &body body

The most basic command-defining form. Usually the programmer will not use define-command directly, but will instead use a define-frame-command form automatically generated by define-application-frame, define-frame-com-
The `define-command` form is used to define a new command in the application frame’s command table. By default, `define-command` does not add the command to any command table.

`define-command` defines two functions. The first function has the same name as the command name and implements the body of the command. It takes as required arguments the arguments to the command as specified by the `define-command` form. The name of the other function defined by Lisp is unspecified. It implements the code used by the command processor for parsing and returning the command’s arguments.

The `name-and-options` is either a command name or a cons of the command name and a list of keyword-value pairs.

- **:command-table** `command-table-name`, where `command-table-name` either names a command table to which the command will be added, or is `nil` (the default), indicating that the command should not be added to any command table. If the command table does not exist, the `command-table-not-found` error will be signaled. This keyword is only accepted by `define-command`, not by `define-frame-command`.

- **:name** `string`, where `string` is a string that will be used as the command-line name for the command for keyboard interactions in the command table specified by the `:command-table` option. The default is `nil`, meaning that the command will not be available via command-line interactions. If `string` is `t`, then the command-line name will be generated automatically, as described in `add-command-to-command-table`.

- **:menu** `menu-spec`, where `menu-spec` describes an item in the menu of the command table specified by the `:command-table` option. The default is `nil`, meaning that the command will not be available via menu interactions. If `menu-spec` is a string, then that string will be used as the name of the command in the menu. If `menu-spec` is `t`, and if a command-line name was supplied, it will be used as the name of the command in the menu; otherwise the menu name will be generated automatically, as described in `add-command-to-command-table`. Otherwise, `menu-spec` must be a cons of the form `(string . menu-options)`, where `string` is the menu name and `menu-options` consists of keyword-value pairs. The valid keywords are `:after`, `:documentation`, and `:text-style`, which are as for `add-menu-item-to-command-table`.

- **:keystroke** `gesture`, where `gesture` is a keyboard gesture name that specifies a keystroke accelerator to use for this command in the command table specified by the `:command-table` option. The default is `nil`, meaning that there is no keystroke accelerator.
The :name, :menu, and :keystroke options are only allowed if the :command-table option is supplied explicitly or implicitly, as in define-frame-command.

arguments is a list consisting of argument descriptions. A single occurrence of the symbol &key may appear in arguments to separate required command arguments from keyword arguments. Each argument description consists of a parameter variable, followed by a presentation type specifier, followed by keyword-value pairs. The keywords can be:

- :default value, where value is the default that should be used for the argument, as for accept.
- :default-type is the same as for accept.
- :display-default is the same as for accept.
- :mentioned-default value, where value is the default that should be used for the argument when a keyword is explicitly supplied via the command-line processor, but no value is supplied for it. :mentioned-default is only allowed on keyword arguments.
- :prompt string, where string is a prompt to print out during command-line parsing, as for accept.
- :documentation string, where string is a documentation string that describes what the argument is.
- :when form, form is evaluated in a scope where the parameter variables for the required parameters are bound, and if the result is nil, the keyword argument is not available. :when is only allowed on keyword arguments, and form cannot use the values of other keyword arguments.
- :gesture gesture, where gesture is either a pointer gesture name or a list of a pointer gesture name followed by keyword-value pairs. When a gesture is supplied, a presentation translator will be defined that translates from this argument’s presentation type to an instance of this command with the selected object as the argument; the other arguments will be filled in with their default values. The keyword-value pairs are used as options for the translator. Valid keywords are :tester, :menu, :priority, :echo, :documentation, and :pointer-documentation. The default for gesture is nil, meaning no translator will be written. :gesture is only allowed when the :command-table option was supplied to the command-defining form.
body implements the body of the command. It has lexical access to all of the
commands arguments. If the body of the command needs access to the application
frame itself, it should use *application-frame*. The returned values of body are
ignored. body may have zero or more declarations as its first forms.

define-command must arrange for the function that implements the body of the
command to get the proper values for unsupplied keyword arguments.

name-and-options and body are not evaluated. In the argument descriptions, the
parameter variable name is not evaluated. The others are evaluated at run-time
when argument parsing reaches them, except that the value for :when is evalu-
ated when parsing reaches the keyword arguments. :gesture is not evaluated.

11.4 CLIM Command Tables

CLIM command tables are represented by instances of the CLOS class command-table. A
command table serves to mediate between a command input context, a set of commands,
and the interactions of the application’s user. Command tables contain the following infor-
mation:

- The name of the command table, which is a symbol
- An ordered list of command tables to inherit from
- The set of commands that are present in this command table
- A table that associates command-line names to command names (used to support com-
mand-line processor interactions)
- A set of presentation translators, defined via define-presentation-translator and de-
define-presentation-to-command-translator
- A table that associates keyboard gesture names to menu items (used to support key-
stroke accelerator interactions). The keystroke accelerator table does not contain any
items inherited from superior command tables.
- A menu that associates menu names with command menu items (used to support inter-
action via command menus). The command menu items can invoke commands or sub-
menus. The menu does not contain any command menu items inherited from superior
command tables.

We say that a command is present in a command table when it has been added to that com-
mand table by being associated with some form of interaction. We say that a command is

Commands
accessible in a command table when it is present in that command table or is present in any of the command tables from which that command table inherits.

**command-table**

[Protocol Class]

Summary: The protocol class that corresponds to command tables. If you want to create a new class that behaves like a command table, it should be a subclass of `command-table`. Subclasses of `command-table` must obey the command table protocol. Members of this class are mutable.

**command-table-p**

[Function]

Arguments: `object`

Summary: Returns `t` if `object` is a command table; otherwise, it returns `nil`.

**standard-command-table**

[Class]

Summary: The instantiable class that implements command tables, a subclass of `command-table`. `make-command-table` returns objects that are members of this class.

**command-table-name**

[Generic Function]

Arguments: `command-table`

Summary: Returns the name of the command table `command-table`.

**command-table-inherit-from**

[Generic Function]

Arguments: `command-table`

Summary: Returns a list of the command tables from which the command table `command-table` inherits. This function returns objects that reveal CLIM’s internal state; do not modify those objects.

**define-command-table**

[Macro]

Arguments: `name` &key `inherit-from` `menu`

Summary: Defines a command table whose name is the symbol `name`. The new command table inherits from all of the command tables specified by `inherit-from`, which is a list of `command table designators` (that is, either a command table or a symbol that names a command table). The inheritance is done by union with shadowing. If no inheritance is specified, the command table will be made to inherit from CLIM’s global command table. (This command table contains such things as the “menu” translator that is associated with the right-hand button on pointers.)
Commands

Menu can be used to specify a menu for the command table. The value of menu is a list of clauses. Each clause is a list with the syntax (string type value &key keystroke documentation text-style), where string, type, value, keystroke, documentation, and text-style are as for add-menu-item-to-command-table.

If the command table named by name already exists, define-command-table will modify the existing command table to have the new value for inherit-from and menu, and leaves the other attributes for the existing command table alone.

None of define-command-table’s arguments are evaluated.

make-command-table
[Function]
Arguments:name &key inherit-from menu (errorp t)
Summary: Creates a command table named name. inherit-from and menu are the same as for define-command-table. make-command-table does not implicitly include CLIM’s global command table in the inheritance list for the new command table. If the command table already exists and errorp is t, the command-table-already-exists error will be signaled. If the command table already exists and errorp is nil, then the old command table will be discarded. The returned value is the command table.

find-command-table
[Function]
Arguments:name &key (errorp t)
Summary: Returns the command table named by name. If name is itself a command table, it is returned. If the command table is not found and errorp is t, the command-table-not-found error will be signaled.

add-command-to-command-table
[Function]
Arguments:command-name command-table &key name menu keystroke (errorp t)
Summary: Adds the command named by command-name to the command table specified by the command table designator command-table.

name is the command-line name for the command, and can be nil, t, or a string. When it is nil, the command will not be available via command-line interactions. When it is a string, that string is the command-line name for the command. When it is t, the command-line name is generated automatically. (The automatically generated name consists of the command’s name with the hyphens replaced by spaces, and the words capitalized; any com- prefix is removed. For example, if the command name is com-show-file, the command-line name will be Show
For the purposes of command-line-name lookup, the character case of *name* is ignored.

*menu* is a menu item for the command, and can be *nil*, *t*, a string, or a cons. When it is *nil*, the command will not be available via menus. When it is a string, the string will be used as the menu name. When *menu* is *t* and *name* is a string, then *name* will be used as the menu name. When *menu* is *t* and *name* is not a string, an automatically generated menu name will be used. When *menu* is a cons of the form (string, menu-options), *string* is the menu name and *menu-options* consists of keyword-value pairs. The valid keywords are :after, :documentation, and :text-style, which are interpreted as for add-menu-item-to-command-table.

The value for *keystroke* is either a keyboard gesture name or *nil*. When it is a gesture name, it is the keystroke accelerator for the command; if it is *nil*, the command will not be available via keystroke accelerators.

If the command is already present in the command table and *errorp* is *t*, the command-already-present error will be signaled. When *errorp* is *nil*, the old command-line name, menu, and keystroke accelerator will first be removed from the command table.

remove-command-from-command-table  
[Function]

Arguments: *command-name command-table &key (errorp t)*
Summary: Removes the command named by *command-name* from the command table specified by the command table designator *command-table*.

If the command is not present in the command table and *errorp* is *t*, the command-not-present error will be signaled.

## 11.5 CLIM Predefined Command Tables

CLIM provides these command tables:

global-command-table  
[Command Table]
Summary: The “global” command table from which all command tables inherit.

user-command-table  
[Command Table]
Summary: A command table reserved for user-defined commands.

accept-values-pane  
[Command Table]
Summary: When you use an `accept-values` pane in a `define-application-frame`, you must inherit from this command table.

It is recommended that an application’s command table inherit from `user-command-table`. `user-command-table` inherits from `global-command-table`. If your application uses an `:accept-values` pane, then its command table must inherit from the `accept-values-pane` command table in order for it to work properly.

11.6 Conditions Relating to CLIM Command Tables

Command table operations can signal these conditions:

- **command-table-already-exists**  
  Summary: This condition is signaled by `make-command-table` when you try to create a command table that already exists.

- **command-table-not-found**  
  Summary: This condition is signaled by functions such as `find-command-table` when the named command table cannot be found.

- **command-already-present**  
  Summary: The error that is signaled when a function tries to add a command that is already present in a command table to that command table.

- **command-not-present**  
  Summary: A condition that is signaled when the command you are looking for is not present in the command table.

- **command-not-accessible**  
  Summary: A condition that is signaled when the command you are looking for is not accessible in the command table.

- **command-table-error**  
  Summary: 

Commands
Summary: The class that is the superclass of the previous four conditions. This class is a subclass of error.

11.7 Styles of Interaction Supported by CLIM

CLIM supports four main styles of interaction:

- Mouse interaction via command menus
- Mouse interaction via translators
- Keyboard interaction using a command-line processor
- Keyboard interaction using keystroke accelerators

See Section 11.2, “Defining Commands the Easy Way,” for a simple description of how to use define-command to associate a command with any of these interaction styles. See Section 11.10, “Advanced Topics,” for an in-depth discussion of CLIM interaction styles.

11.8 Command-Related Presentation Types

CLIM provides several presentation types pertaining to commands:

command [Presentation Type]
Arguments:&key command-table
Summary: The presentation type used to represent a command and its arguments; the command must be accessible in command-table and enabled in *application-frame*. command-table is a command table designator. If command-table is not supplied, it defaults to the command table for the current application frame.

The object returned by the accept presentation method for command must be a command object, that is, a cons of the command name and the list of the command’s arguments.

The accept presentation method for the command type must call the command parser stored in *command-parser* to read the command. The parser will recursively call accept to read a command-name and all of the command’s argu-
ments. The parsers themselves must be implemented by accepting objects whose presentation type is `command`.

If the command parser returns a partial command, the accept presentation method for the `command` type must call the partial command parser stored in `*partial-command-parser*`.

The present presentation method for the `command` type must call the command unparser stored in `*command-unparser*`.

If a presentation history is maintained for the `command` presentation type, it should be maintained separately for each instance of an application frame.

**command-name**

[Presentation Type]

Arguments: &key command-table

Summary: The presentation type used to represent the name of a command that is both accessible in the command table `command-table` and enabled in `*application-frame*`. `command-table` is a command table designator. If `command-table` is not supplied, it defaults to the command table for the current application frame, `(frame-command-table *application-frame*)`

The textual representation of a `command-name` object is the command-line name of the command, while the internal representation is the command name.

**command-or-form**

[Presentation Type]

Arguments: &key command-table

Summary: The presentation type used to represent an object that is either a Lisp form or a command and its arguments. The command must be accessible in `command-table` and enabled in `*application-frame*`. `command-table` is a command table designator. If `command-table` is not supplied, it defaults to the command table for the current application frame, `(frame-command-table *application-frame*)`

The accept presentation method for this type reads a Lisp form, except that if the first character in the user’s input is one of the characters in `*command-dispatchers*`, it will read a command. The two returned values from the accept presentation method will be the command or form object and a presentation type specifier that is either `command` or `form`.

A presentation history is maintained separately for the `command-or-form` presentation type for each instance of an application frame.
*command-dispatchers*  [Variable]

Summary: This is a list of the characters that indicates that CLIM reads a command when it is accepting a command-or-form. The standard set of command argument delimiters includes the colon character, #\:.

### 11.9 The CLIM Command Processor

Once a set of commands has been defined, CLIM provides a variety of means to read a command. These are all mediated by the command processor.

The command loop of a CLIM application is performed by the application’s top-level function (see Chapter 9, “Defining Application Frames”). By default, this is default-frame-top-level. After performing some initializations, default-frame-top-level enters an infinite loop, reading and executing commands. It invokes the generic function read-frame-command to read a command that is then passed to the generic function execute-frame-command for execution. The specialization of these generic functions is the simplest way to modify the command loop for your application. Other techniques would involve replacing default-frame-top-level with your own top-level function.

read-frame-command invokes the command parser by establishing an input context of command. The input editor keeps track of the user’s input, both from the keyboard and the pointer. Each of the command’s arguments is parsed by establishing an input context of the arguments presentation type as described in the command’s definition. Presentation translators provide the means by which the pointer can be used to enter command names and arguments.

read-command  [Function]

Arguments: command-table &key (stream *query-io*) command-parser command-unparser partial-command-parser use-keystrokes

Summary: read-command is the standard interface used to read a command line. stream is an extended input stream, and command-table is a command table designator.

command-parser is a function of two arguments, a command table and a stream. It reads a command from the user and returns a command object. The default value for command-parser is the value of *command-parser*.

command-unparser is a function of three arguments, a command table, a stream, and a command to “unparse.” It prints a textual description of the command and
its supplied arguments onto the stream. The default value for `command-unparser` is the value of `*command-unparser*`.

`partial-command-parser` is a function of four arguments, a command table, a stream, a partial command, and a start position. The partial command is a command object with the value of `*unsupplied-argument-marker*` in place of any argument that needs to be filled in. The function reads the remaining unsupplied arguments in any way it sees fit (for example, via an `accepting-values` dialog), and returns a command object. The start position is the original input-editor scan position of the stream, when the stream is an interactive stream. The default value for `partial-command-parser` is the value of `*partial-command-parser*`.

`command-parser`, `command-unparser`, and `partial-command-parser` have dynamic extent.

When `use-keystrokes` is `t`, the command reader will also process keystroke accelerators.

Input editing, while conceptually an independent facility, fits into the command processor via its use of `accept`. That is, `read-command` calls `accept` to read command objects, and `accept` itself makes use of the input editing facilities.

---

**read-frame-command**

[Generic Function]

**Arguments:** `frame &key stream`

**Summary:** `read-frame-command` reads a command from the user on the stream `stream`, and returns the command object. `frame` is an application frame.

The default method for `read-frame-command` calls `read-command` on `frame`'s current command table.

**execute-frame-command**

[Generic Function]

**Arguments:** `frame command`

**Summary:** `execute-frame-command` executes the command `command` on behalf of the application frame `frame`.

**with-command-table-keystrokes**

[Macro]

**Arguments:** `(keystroke-var command-table) &body body`

**Summary:** Binds `keystroke-var` to a sequence that contains all of the keystroke accelerators in `command-table`’s menu, and then executes `body` in that context. `command-table` is a command table designator. `body` may have zero or more declarations as its first forms.

Commands
read-command-using-keystrokes  [Function]
Arguments: command-table keystrokes &key (stream *query-io*) command-parser command-unparser partial-command-parser
Summary: Reads a command from the user via command lines, the pointer, or a single keystroke, and returns either a command object or a keyboard gesture object. (The latter only occurs when the user types a keystroke that is in keystrokes but does not have a command associated with it in command-table.)

keystrokes is a sequence of keyboard gesture names that are the keystroke accelerators.

command-table, stream, command-parser, command-unparser, and partial-command-parser are as for read-command.

An application can control which commands are enabled and which are disabled on an individual basis by using enable-command and disable-command. The user is not allowed to enter a disabled command via any interaction style.

enable-command  [Function]
Arguments: command-name frame
Summary: Enables the use of the command named by command-name while in the application frame.

disable-command  [Function]
Arguments: command-name frame
Summary: Disables the use of the command named by command-name while in the application frame.

The special variable *command-dispatchers* controls the behavior of the command-or-form presentation type.

*command-dispatchers*  [Variable]
Summary: This is a list of the characters that indicates that CLIM reads a command when it is accepting a command-or-form. The standard set of command argument delimiters includes the colon character, #\:.

284CLIM User Guide
11.10 Advanced Topics

The material in this section is advanced; most CLIM programmers can skip to the next chapter.

11.10.1 CLIM Command Tables

For more information on CLIM command tables, see Section 11.4.

**do-command-table-inheritance**

*Macro*

**Arguments:**

(command-table-var command-table) &body body

**Summary:**

Successively executes body with command-table-var bound first to the command table specified by the command table designator command-table, and then (recursively) to all of the command tables from which command-table inherits.

The command-table-var argument is not evaluated. body may have zero or more declarations as its first forms.

**map-over-command-table-commands**

*Function*

**Arguments:**

function command-table &key (inherited t)

**Summary:**

Applies function to all of the commands accessible in the command table specified by the command table designator command-table. function must be a function that takes a single argument, the command name; it has dynamic extent.

If inherited is **nil**, this applies function only to those commands present in command-table, that is, it does not map over any inherited command tables. If inherited is t, then the inherited command tables are traversed in the same order as for do-command-table-inheritance.

**map-over-command-table-names**

*Function*

**Arguments:**

function command-table &key (inherited t)

**Summary:**

Applies function to all of the command-line name accessible in the command table specified by the command table designator command-table. function must be a function of two arguments, the command-line name and the command name; it has dynamic extent.

If inherited is **nil**, this applies function only to those command-line names present in command-table, that is, it does not map over any inherited command tables. If
inherited is t, then the inherited command tables are traversed in the same order as for do-command-table-inheritance.

command-present-in-command-table-p [Function]
Arguments:command-name command-table
Summary: Returns t if the command named by command-name is present in the command table specified by the command table designator command-table; otherwise, it returns nil.

canmand-accessible-in-command-table-p [Function]
Arguments:command-name command-table
Summary: If the command named by command-name is not accessible in the command table specified by the command table designator command-table, then this function returns nil. Otherwise, it returns the command table in which the command was found.

find-command-from-command-line-name [Function]
Arguments:name command-table &key (errorp t)
Summary: Given a command-line name name and a command table, returns two values, the command name and the command table in which the command was found. If the command is not accessible in command-table and errorp is t, the command-not-accessible error will be signaled. command-table is a command table designator.

find-command-from-command-line-name ignores character case.

command-line-name-for-command [Function]
Arguments:command-name command-table &key (errorp t)
Summary: Returns the command-line name for command-name as it is installed in command-table. command-table is a command table designator.

If the command is not accessible in command-table or has no command-line name, then there are three possible results. If errorp is nil, then the returned value will be nil. If errorp is :create, then a command-line name will be generated, as described in add-command-to-command-table. Otherwise, if errorp is t, then the command-not-accessible error will be signaled. The returned command-line name should not be modified.
**command-table-complete-input**

**Arguments:** `command-table string action &key frame`

**Summary:** A function that can be used as in conjunction with `complete-input` in order to complete over all of the command lines names accessible in the command table `command-table`. `string` is the input string to complete over, and `action` is as for `complete-from-possibilities`.

`frame` is either an application frame, or `nil`. If `frame` is supplied, no disabled commands should be offered as valid completions.

### 11.10.2 CLIM Command Menu Interaction Style

Each command table may describe a menu consisting of an ordered sequence of command menu items. The menu specifies a mapping from a menu name (the name displayed in the menu) to either a command object or a submenu. The menu of an application’s top-level command table may be presented in a window-system specific way, for example, as a menu bar or in a `:menu` application frame pane.

Command menu items are stored as a list of the form `(type value options)`, where `type` and `value` are as for `add-menu-item-to-command-table`, and `options` is a list of keyword-value pairs. The allowable keywords are `:documentation`, which is used to supply optional pointer documentation for the command menu item, and `:text-style`, which is used to indicate what text style should be used for this command menu item when it is displayed in a command menu.

The following functions can be used to display a command menu in one of the panes of an application frame or to choose a command from a menu. `add-menu-item-to-command-table`, `remove-menu-item-from-command-table`, and `find-menu-item` ignore the character case of the command menu item’s name when searching through the command table’s menu.

**display-command-table-menu**

**Arguments:** `command-table stream &key max-width max-height n-rows n-columns x-spacing y-spacing initial-spacing (cell-align-x :left) (cell-align-y :top) (move-cursor t)`

**Summary:** Displays `command-table`’s menu on `stream`. It may use `formatting-item-list` or display the command table’s menu in a platform-dependent manner, such as using the menu bar on a Macintosh. `command-table` is a command table designator.
max-width, max-height, n-rows, n-columns, x-spacing, y-spacing, initial-spacing, cell-align-x, cell-align-y, and move-cursor are as for formatting-item-list.

display-command-menu

[Generic Function]

:initial-spacing :n-rows :n-columns (:cell-align-x :left) (:cell-align-y :top)

Summary: Displays the menu described by the command table associated with the application frame frame on stream. This is generally used as the display function for extended stream panes of type :command-menu.

menu-choose-command-from-command-table

[Function]

Arguments: command-table &key associated-window default-style label cache unique-id id-test cache-value cache-test

Summary: Invokes a window-system-specific routine that displays a menu of commands from command-table’s menu, and allows the user to choose one of the commands. command-table is a command table designator. The returned value is a command object. This may invoke itself recursively when there are submenus.

associated-window, default-style, label, cache, unique-id, id-test, cache-value, and cache-test are as for menu-choose.

A number of lower level functions for manipulating command menus are also provided:

add-menu-item-to-command-table

[Function]

Arguments: command-table string type value &key documentation (after :end) keystroke text-style (errorp t)

Summary: Adds a command menu item to command-table’s menu. string is the name of the command menu item; its character case is ignored. type is either :command, :function, :menu, or :divider. command-table is a command table designator.

When type is :command, value must be a command (a cons of a command name followed by a list of the command’s arguments), or a command name. (When value is a command name, it behaves as though a command with no arguments was supplied.) In the case where all of the command’s required arguments are supplied, clicking on an item in the menu invokes the command immediately. Otherwise, the user will be prompted for the remaining required arguments.

When type is :function, value must be a function having indefinite extent that, when called, returns a command. It is called with two arguments, the gesture the user used to select the item (either a keyboard or button press event) and a “numeric argument.”
When type is :menu, this item indicates that a submenu will be invoked, and so value must be another command table or the name of another command table.

When type is :divider, some sort of a dividing line is displayed in the menu at that point. If string is supplied, it will be drawn as the divider instead of a line. If the look and feel provided by the underlying window system has no corresponding concept, :divider items may be ignored.

documentation is a documentation string, which can be used as mouse documentation for the command menu item.

text-style is either a text style spec or nil. It is used to indicate that the command menu item should be drawn with the supplied text style in command menus.

after must be either :start (meaning to add the new item to the beginning of the menu), :end or nil (meaning to add the new item to the end of the menu), or a string naming an existing entry (meaning to add the new item after that entry). If after is :sort, then the item is inserted in such as way as to maintain the menu in alphabetical order.

If keystroke is supplied, the item will be added to the command table’s keystroke accelerator table. The value of keystroke must be a keyboard gesture name. This is exactly equivalent to calling add-keystroke-to-command-table with the arguments command-table, keystroke, type and value. When keystroke is supplied and type is :command or :function, typing a key on the keyboard that matches to the keystroke accelerator gesture will invoke the command specified by value. When type is :menu, the command will continue to be read from the submenu indicated by value in a window-system-specific manner.

If the item named by string is already present in the command table’s menu and errorp is t, then the command-already-present error will be signaled. When the item is already present in the command table’s menu and errorp is nil, the old item will first be removed from the menu. Note that the character case of string is ignored when searching the command table’s menu.

remove-menu-item-from-command-table [Function]
Arguments:command-table string &key (errorp t)
Summary: Removes the item named by string from command-table’s menu. command-table is a command table designator.

If the item is not present in the command table’s menu and errorp is t, then the command-not-present error will be signaled. Note that the character case of string is ignored when searching the command table’s menu.
map-over-command-table-menu-items

Arguments: function command-table
Summary: Applies function to all of the items in command-table’s menu. function must be a function of three arguments, the menu name, the keystroke accelerator gesture (which will be nil if there is none), and the command menu item; it has dynamic extent. The command menu items are mapped over in the order specified by add-menu-item-to-command-table. command-table is a command table designator.

find-menu-item

Arguments: menu-name command-table &key (errorp t)
Summary: Given a menu name and a command table, returns two values, the command menu item and the command table in which it was found. (Since menus are not inherited, the second returned value will always be command-table.) command-table is a command table designator. This function returns objects that reveal CLIM’s internal state; do not modify those objects.

If there is no command menu item corresponding to menu-name present in command-table and errorp is t, then the command-not-accessible error will be signaled. Note that the character case of string is ignored when searching the command table’s menu.

command-menu-item-type

Arguments: menu-item
Summary: Returns the type of the command menu item menu-item, for example, :menu or :command. If menu-item is not a command menu item, the result is unspecified.

command-menu-item-value

Arguments: menu-item
Summary: Returns the value of the command menu item menu-item. For example, if the type of menu-item is :command, this will return a command or a command name. If menu-item is not a command menu item, the result is unspecified.

command-menu-item-options

Arguments: menu-item
Summary: Returns a list of the options for the command menu item menu-item. If menu-item is not a command menu item, the result is unspecified.
11.10.3 Mouse Interaction Via Presentation Translators

A command table maintains a database of presentation translators. A presentation translator translates from its from-presentation-type to its to-presentation-type when its associated gesture (e.g., clicking a mouse button) is input. A presentation translator is triggered when its to-presentation-type matches the input context and its from-presentation-type matches the presentation type of the displayed presentation (the appearance of one of your application’s objects on the display) on which the gesture is performed.

define-presentation-to-command-translator can be used to associate a presentation and a gesture with a command to be performed on the object which the presentation represents.

Translators can also be used to translate from an object of one type to an object of another type based on context. For example, consider a computer-aided design system for electrical circuits. You might have a translator that translates from a resistor object to the numeric value of its resistance. When asked to enter a resistance (as an argument to a command or for some other query), the user could click on the presentation of a resistor.

Here are some utilities for maintaining presentation translators in command tables. See Section 6.1, “Conceptual Overview of CLIM Presentation Types,” for a discussion of the facilities supporting the mouse translator interaction style.

add-presentation-translator-to-command-table

Arguments: command-table translator-name &key (errorp t)
Summary: Adds the translator named by translator-name to command-table. The translator must have been previously defined with define-presentation-translator or define-presentation-to-command-translator. command-table is a command table designator.

If translator-name is already present in command-table and errorp is t, then the command-already-present error will be signaled. When the translator is already present and errorp is nil, the old translator will first be removed.

remove-presentation-translator-from-command-table

Arguments: command-table translator-name &key (errorp t)
Summary: Removes the translator named by translator-name from command-table. command-table is a command table designator.

If the translator is not present in the command table and errorp is t, then the command-not-present error will be signaled.
map-over-command-table-translators

Arguments: function command-table &key (inherited t)
Summary: Applies function to all of the translators accessible in command-table. function must be a function of one argument, the translator; it has dynamic extent. command-table is a command table designator.

If inherited is nil, this applies function only to those translators present in command-table, that is, it does not map over any inherited command tables. If inherited is t, then the inherited command tables are traversed in the same order as for do-command-table-inheritance.

find-presentation-translator

Arguments: translator-name command-table &key (errorp t)
Summary: Given a translator name and a command table, returns two values, the presentation translator and the command table in which it was found. If the translator is not present in command-table and errorp is t, then the command-not-accessible error will be signaled. command-table is a command table designator.

11.10.4 CLIM Command Line Interaction Style

One interaction style supported by CLIM is the command line style of interaction provided on most conventional operating systems. A command prompt is displayed in the application’s :interactor pane. The user enters a command by typing its command line name followed by its arguments. What the user types (or enters via the pointer) is echoed to the interactor window. When the user has finished typing the command, it is executed.

In CLIM, this interaction style is augmented by the input editing facility, which allows the user to correct typing mistakes, and by the prompting and help facilities, which provide a description of the command and the expected arguments (see Chapter 16, “Input Editing and Completion Facilities”). Command entry is also facilitated by the presentation substate, which allows the input of objects matching the input context, both for command names and command arguments.

See Section 11.4, “CLIM Command Tables,” and Subsection 11.10.1, “CLIM Command Tables,” for complete descriptions of these functions.

find-command-from-command-line-name

Arguments: name command-table &key (errorp t)
**Summary:** Given a command-line name `name` and a `command-table`, this function returns two values, the command name and the command table in which the command was found.

**command-line-name-for-command**  
*Function*

**Arguments:** `command-name` `command-table` &key (errorp t)

**Summary:** Returns the command-line name for `command-name` as it is installed in `command-table`.

**map-over-command-table-names**  
*Function*

**Arguments:** `function` `command-table` &key (inherited t)

**Summary:** Applies `function` to all the command-line names accessible in `command-table`.

### 11.10.5 CLIM Keystroke Interaction Style

Each command table may have a mapping from keystroke accelerator gesture names to command menu items. When a user presses a key that corresponds to the gesture for keystroke accelerator, the corresponding command menu item will be invoked. Command menu items are shared among the command table’s menu and the accelerator table. This lets the menu display the keystroke associated with a particular item, if there is one.

Note that, despite the fact the keystroke accelerators are specified using keyboard gesture names rather than characters, the conventions for typed characters vary widely from one platform to another. Therefore the programmer must be careful in choosing keystroke accelerators. Some sort of per-platform conditionalization is to be expected.

Keystroke accelerators will typically be associated with commands through the use of the `:keystroke` option to `define-command` (or the application’s command defining macro).

**add-keystroke-to-command-table**  
*Function*

**Arguments:** `command-table gesture type value` &key documentation (errorp t)

**Summary:** Adds a command menu item to `command-table`’s keystroke accelerator table. `gesture` is a keyboard gesture name to be used as the accelerator. `type` and `value` are as for `add-menu-item-to-command-table`, except that `type` must be either `:command`, `:function` or `:menu`. `command-table` is a command table designator. `documentation` is a documentation string, which can be used as documentation for the keystroke accelerator.
If the command menu item associated with \textit{gesture} is already present in the command table’s accelerator table and \textit{errorp} is \texttt{t}, then the \texttt{command-already-present} error will be signaled. When the item is already present in the command table’s accelerator table and \textit{errorp} is \texttt{nil}, the old item will first be removed.

\textbf{remove-keystroke-from-command-table} \hfill \textbf{[Function]}

\textbf{Arguments}: \texttt{command-table gesture \&key (errorp \texttt{t})}

\textbf{Summary}: Removes the command menu item named by keyboard gesture name \textit{gesture} from \textit{command-table}’s accelerator table. \textit{command-table} is a command table designator.

The \texttt{command-not-present} error will be signaled if the command menu item associated with \textit{gesture} is not in the command table’s menu and \textit{errorp} is \texttt{t}.

\textbf{map-over-command-table-keystrokes} \hfill \textbf{[Function]}

\textbf{Arguments}: \texttt{function command-table}

\textbf{Summary}: Applies \textit{function} to all the keystroke accelerators in \textit{command-table}’s accelerator table. \textit{function} must be a function of three arguments, the menu name (which will be \texttt{nil} if there is none), the keystroke accelerator, and the command menu item; it has dynamic extent. \textit{command-table} is a command table designator.

\textit{map-over-command-table-keystrokes} is not recursive. If you want it to descend into submenus, check that the type of the command menu item is \texttt{eql} to :menu before using \textit{map-over-command-table-keystrokes} recursively.

\textbf{find-keystroke-item} \hfill \textbf{[Function]}

\textbf{Arguments}: \texttt{gesture command-table \&key (errorp \texttt{t})}

\textbf{Summary}: Given a keyboard gesture \textit{gesture} and a command table, returns two values, the command menu item associated with the gesture and the command table in which it was found. (Since keystroke accelerators are not inherited, the second returned value will always be \textit{command-table}.)

This function returns objects that reveal CLIM’s internal state; do not modify those objects.

Note that \textit{gesture} may be either a keyboard gesture name of a gesture object. When it is a gesture name, \texttt{eql} will be used to compare the supplied gesture to the gesture names stored in the command table’s menu. When it is a gesture object, \texttt{event-matches-gesture-name-p} will be used to do the comparison.
If the keystroke accelerator is not present in command-table and errorp is t, then the command-not-present error will be signaled. command-table is a command table designator.

**lookup-keystroke-item**

[Function]

Arguments: gesture command-table

Summary: Given a keyboard gesture gesture and a command table, returns two values, the command menu item associated with the gesture and the command table in which it was found. gesture may be either a keyboard gesture name or a gesture object, and is handled in the same way as in find-keystroke-item. This function returns objects that reveal CLIM’s internal state; do not modify those objects.

Unlike find-keystroke-item, this follows the submenu chains that can be created with add-menu-item-to-command-table. If the keystroke accelerator cannot be found in the command table or any of the command tables from which it inherits, lookup-keystroke-item will return nil. command-table is a command table designator.

**lookup-keystroke-command-item**

[Function]

Arguments: gesture command-table &key numeric-arg

Summary: Given a keyboard gesture gesture and a command table, returns the command associated with the keystroke, or gesture if no command is found. Note that gesture may be either a keyboard gesture name or a gesture object, and is handled in the same way as in find-keystroke-item. This function returns objects that reveal CLIM’s internal state; do not modify those objects.

This is like find-keystroke-item, except that only keystrokes that map to an enabled application command will be matched. command-table is a command table designator.

numeric-arg (which defaults to 1) is substituted into the resulting command for any occurrence of *numeric-argument-marker* in the command. This is intended to allow programmers to define keystroke accelerators that take simple numeric arguments, which will be passed on by the input editor.

**substitute-numeric-argument-marker**

[Function]

Arguments: command numeric-arg

Summary: Given a command object command, this substitutes the value of numeric-arg for all occurrences of the value of *numeric-argument-marker* in the command, and returns a command object with those substitutions.
For a description of the CLIM command processor, see Section 11.9.

11.10.6 The CLIM Command Processor

command-line-command-parser

Arguments: command-table stream

Summary: The default command-line parser. It reads a command name and the command’s arguments as a command line from stream (with completion as much as is possible), and returns a command object. command-table is a command table designator that specifies the command table to use; the commands are read via the textual command-line name.

command-line-command-unparser

Arguments: command-table stream command

Summary: The default command-line unparser. It prints the command command as a command name and its arguments as a command line on stream. command-table is a command table designator that specifies the command table to use; the commands are displayed using the textual command-line name.

command-line-read-remaining-arguments-for-partial-command

Arguments: command-table stream partial-command start-position

Summary: The default partial command-line parser. If the remaining arguments are at the end of the command line, it reads them as a command line; otherwise, it constructs a dialog using accepting-values and reads the remaining arguments from the dialog. command-table is a command table designator.

menu-command-parser

Arguments: command-table stream

Summary: The default menu-driven command parser. It uses only pointer clicks to construct a command. It relies on presentations of all arguments being visible. command-table and stream are as for command-line-parser.

There is no menu-driven command unparsers, since it makes no sense to unpars a completely menu-driven command.

menu-read-remaining-arguments-for-partial-command

Arguments: command-table stream partial-command start-position

296CLIM User Guide
Summary: The default menu-driven partial command parser. It uses only pointer clicks to fill in the command. Again, it relies on presentations of all arguments being visible. *command-table* is a command table designator.

*command-parser* [Variable]
Summary: Contains the currently active command parsing function. The default value is the function *command-line-command-parser*, which is the default command-line parser.

*command-unparser* [Variable]
Summary: Contains the currently active command unparsing function. The default value is the function *command-line-command-unparser*, which is the default command-line unparser.

*partial-command-parser* [Variable]
Summary: Contains the currently active partial command parsing function. The default value is the function *command-line-read-remaining-arguments-for-partial-command*.

*unsupplied-argument-marker* [Variable]
Summary: The value of *unsupplied-argument-marker* is an object that can be uniquely identified as standing for an unsupplied argument in a command object.

*n numeric-argument-marker* [Variable]
Summary: The value of *numeric-argument-marker* is an object that can be uniquely identified as standing for a numeric argument in a command object.

*command-name-delimiters* [Variable]
Summary: This is a list of the characters that separate the command name from the command arguments in a command line. The standard set of command name delimiters includes #\Space.

*command-argument-delimiters* [Variable]
Summary: This is a list of the characters that separate the command arguments from each other in a command line. The standard set of command argument delimiters includes #\Space.
Chapter 12   Menus and Dialogs
12.1 Conceptual Overview of Menus and Dialogs

CLIM provides three powerful menu routines for allowing user to interact with an application through various kinds of menus and dialogs:

- **menu-choose** is a straightforward menu generator that provides a quick way to construct menus. You can call it with a list of menu items. (For a complete definition of menu items, see the function `menu-choose`.)

- **menu-choose-from-drawer** is a lower-level routine that allows the user much more control in specifying the appearance and layout of a menu. You can call it with a window and a drawing function. Use this function for more advanced, customized menus.

- **accepting-values** enables you to build a dialog. Unlike menus, you can specify several items that can be individually selected or modified within the dialog before dismissing it. To abort the dialog, press `control-z`. To exit the dialog, unless you are editing the field, press `control-{}`. These key bindings can be changed by using `add-key-stroke-to-command-table` and `remove-keystroke-from-command-table`.

12.2 CLIM Menu Operators

**menu-choose**

[Generic Function]

Arguments: `items &key associated-window printer presentation-type default-item text-style label cache unique-id id-test cache-value cache-test max-width max-height n-rows n-columns x-spacing y-spacing row-wise cell-align-x cell-align-y pointer-documentation scroll-bars`

Summary: Displays a menu whose choices are given by the elements of the sequence `items`. It returns three values: the value of the chosen item, the item itself, and the pointer button event corresponding to the gesture that the user used to select it. If the user aborts out of the menu, a single value is returned, `nil`.

`menu-choose` calls `frame-manager-menu-choose` on the frame manager being used by `associated-window` (or the frame manager of the current application frame). All the arguments to `menu-choose` will be passed on to `frame-manager-menu-choose`.

`items` is a sequence of menu items. Each menu item has a visual representation derived from a display object, an internal representation that is a value object, and a set of menu item options. The form of a menu item is one of the following:
• An atom—the item is both the display object and the value object.

• A cons—the car is the display object and the cdr is the value object. The value object must be an atom. If you need to return a list as the value, use the :value option in the list menu item format.

• A list—the car is the display object and the cdr is a list of alternating option keywords and values. The value object is specified with the keyword :value and defaults to the display object if :value is not present.

The menu item options are:

• :value—This specifies the value object.

• :text-style—This specifies the text style used to prin the display object when neither presentation-type nor printer is supplied.

• :items—This specifies a sequence of menu items for a submenu used if this item is selected.

• :documentation—This associates some documentation with the menu item. When pointer-documentation is not nil, this will be used as pointer documentation for the item.

• :active—When t (the default), this item is active. When nil, the item is inactive, and cannot be selected. CLIM will generally provide some visual indication that an item is inactive, such as by “graying over” the item.

• :type—This specifies the type of the item. :item (the default) indicates that the item is a normal menu item. :label indicates that the item is simply an inactive label; labels will not be “grayed over.” :divider indicates that the item serves as a separator between groups of other items; separator items will usually be drawn as a horizontal line.

The visual representation of an item depends on the printer and presentation-type keyword arguments. If presentation-type is supplied, the visual representation is produced by present of the menu item with that presentation type. Otherwise, if printer is supplied, the visual representation is produced by the printer function, which receives two arguments, the item and a stream to do output on. The printer function should output some text or graphics at the stream’s cursor position, but need not call present. If neither presentation-type nor printer is supplied, the visual representation is produced by prin of the display object. Note that if presentation-type or printer is supplied, the visual representation is produced from the entire menu item, not just from the display object.
associated-window is the CLIM window with which the menu is associated. This defaults to the top-level window of the current application frame.

default-item is the menu item that is indicated as the default either by some form of highlighting or by warping the mouse to appear over it.

default-style is a text style that defines how the menu items are presented.

label is a string to which the menu title will be set.

printer is a function of two arguments used to print the menu items in the menu. The two arguments are the menu item and the stream to output it on. It has dynamic extent.

presentation-type specifies the presentation type of the menu items.

cache is a boolean that indicates whether CLIM should cache this menu for later use. (Caching menus might speed up later uses of the same menu.) If cache is t, then unique-id and id-test serve to identify this menu uniquely. When cache is t, unique-id defaults to items, but programmers will generally wish to specify a more efficient tag. id-test is a function of two arguments used to compare unique-ids, which defaults to equal. cache-value is the value that is used to indicate that a cached menu is still valid. It defaults to items, but programmers may wish to supply a more efficient cache value than that. cache-test is a function of two arguments that is used to compare cache values, which defaults to equal. Both cache-value and unique-id have dynamic extent.

max-width and max-height specify the maximum width and height of the menu, in device units. They can be overridden by n-rows and n-columns.

n-rows and n-columns specify the number of rows and columns in the menu.

x-spacing specifies the amount of space to be inserted between columns of the table; the default is the width of a space character. It is specified the same way as the :x-spacing option to formatting-table.

y-spacing specifies the amount of blank space inserted between rows of the table; the default is the vertical spacing for the stream. The possible values for this option are the same as for the :y-spacing option to formatting-table.

If row-wise is t (the default) and the item list requires multiple columns, each successive element in the item list is laid out from left to right. If row-wise is nil and the item list requires multiple columns, each successive element in the item list is laid out below its predecessor, as in a telephone book.
cell-align-x specifies the horizontal placement of the cell’s contents. Like the :align-x option to \texttt{formatting-cell}, it is one of :left (the default), :right, or :center. See Subsection 17.1.2, “CLIM Operators for Formatting Tables.”

\textit{cell-align-y} specifies the vertical placement of the contents of the cell. It can be one of :top, :bottom, or :center. The default is :top. The semantics are the same as for the :align-y option to \texttt{formatting-cell}.

\textit{pointer-documentation} is either \texttt{nil} (the default), meaning that no pointer documentation should be computed, or a stream on which pointer documentation should be displayed.

\textbf{frame-manager-menu-choose} \hspace{1cm} \textbf{[Generic Function]}

\textbf{Arguments:} \texttt{frame-manager items} \&key \texttt{associated-window printer presentation-type default-item text-style label cache unique-id id-test cache-value cache-test max-width max-height n-rows n-columns x-spacing y-spacing row-wise cell-align-x cell-align-y pointer-documentation scroll-bars toolkit-p}

\textbf{Summary:} Displays a menu whose choices are given by the elements of the sequence \texttt{items}. It returns three values: the value of the chosen item, the item itself, and the pointer button event corresponding to the gesture that the user used to select it. If the user aborts out of the menu, a single value is returned, \texttt{nil}.

For the values of the arguments, see \texttt{menu-choose}.

\textbf{frame-manager-menu-view} \hspace{1cm} \textbf{[Generic Function]}

\textbf{Arguments:} \texttt{frame-manager view}

\textbf{Summary:} Returns the view used by default for \texttt{menu-choose} when mediated by \texttt{frame-manager}.

\textbf{(setf frame-manager-menu-view)} \hspace{1cm} \textbf{[Generic Function]}

\textbf{Arguments:} \texttt{frame-manager view}

\textbf{Summary:} Sets the view used by default for \texttt{menu-choose} when mediated by \texttt{frame-manager}. Useful values for the view are \texttt{+gadget-menu-view+} and \texttt{+textual-menu-view+}.

\textbf{menu-choose-from-drawer} \hspace{1cm} \textbf{[Generic Function]}

\textbf{Arguments:} \texttt{menu type drawer} \&key \texttt{x-position y-position cache unique-id id-test cache-value cache-test default-presentation pointer-documentation leave-menu-visible}

\textbf{Summary:} This is a lower-level routine for displaying menus. It allows the programmer much more flexibility in the menu layout. Unlike \texttt{menu-choose}, which automat-
ically creates and lays out the menu, **menu-choose-from-drawer** takes a programmer-provided window and drawing function. The drawing function is responsible for drawing the contents of the menu; generally it will be a lexical closure over the menu items.

**menu-choose-from-drawer** draws the menu items into that window using the drawing function. The drawing function gets called with two arguments, *stream* and *type*. It can use *type* for its own purposes, for example, as the type argument in a call to `present`.

**menu-choose-from-drawer** returns two values: the object the user clicked on, and the pointer button event. If the user aborts out of the menu, `nil` is returned.

*menu* is a CLIM window to use for the menu. This argument may be specialized to provide a different look-and-feel for different host window systems.

*type* is a presentation type specifier for each of the mouse-sensitive items in the menu. This is the input context that will be established once the menu is displayed. For programmers who don’t need to define their own types, a useful presentation type is **menu-item**.

**drawer** is a function that takes two arguments, *stream* and *type*, and draws the contents of the menu. It has dynamic extent.

*x-position* and *y-position* are the requested x and y positions of the menu. They may be `nil`, meaning that the position is unspecified.

If `leave-menu-visible` is `t`, the window will not disappear once the selection has been made. The default is `nil`, meaning that the window will disappear once the selection has been made.

**default-presentation** is used to identify the presentation that the mouse is pointing to when the menu comes up.

*cache*, *unique-id*, *id-test*, *cache-value*, and *cache-test* are as for `menu-choose`.

---

**draw-standard-menu**

Arguments: `stream` presentation-type items default-item &key item-printer max-width max-height n-rows n-columns x-spacing y-spacing cell-align-x cell-align-y

Summary: **draw-standard-menu** is the function used by CLIM to draw the contents of a menu, unless the current frame manager determines that host window toolkit should be used to draw the menu instead. *stream* is the stream onto which to draw the menu, *presentation-type* is the presentation type to use for the menu items.
(usually **menu-item**), and **item-printer** is a function used to draw each item. **item-printer** defaults to **print-menu-item**.

**items**, **default-item**, **max-width**, **max-height**, **n-columns**, **x-spacing**, **y-spacing**, **cell-align-x**, and **cell-align-y** are as for **menu-choose**

**print-menu-item** [Function]
Arguments: *menu-item* &optional *stream*
Summary: Given a menu item *menu-item*, displays it on the stream *stream*. This is the function that **menu-choose** uses to display menu items if no printer is supplied.

**menu-item-value** [Function]
Arguments: *menu-item*
Summary: Returns the value of *menu-item*, where the format of a menu item is described under **menu-choose**.

**menu-item-display** [Function]
Arguments: *menu-item*
Summary: Returns the display object of the menu item *menu-item*, where the format of a menu item is described under **menu-choose**.

**menu-item-options** [Function]
Arguments: *menu-item*
Summary: Returns the options of the menu item *menu-item*, where the format of a menu item is described under **menu-choose**.

**with-menu** [Macro]
Arguments: *(menu &optional associated-window &key (deexpose t)) &body body*
Summary: Binds **menu** to a “temporary” window, exposes the window on the same screen as the **associated-window** and runs the body. After the body has been run, the window disappears only if the boolean **deexpose** is t (the default).

The values returned by **with-menu** are the values returned by **body**. **body** may have zero or more declarations as its first forms.

**menu** must be a variable name. **associated-window** is as for **menu-choose**.

None of the arguments is evaluated.
12.3 CLIM Dialog Operators

accepting-values  [Macro]

Arguments: (&optional stream &key own-window exit-boxes
  initially-select-query-identifier resynchronize-every-pass label scroll-bars
  x-position y-position frame-class) &body body

Summary: Builds a dialog for user interaction based on calls to accept within body. The
user can select the values and change them, or use defaults if they are supplied.
The dialog will also contain some sort of “end” and “abort” choices. If “end” is
selected, then accepting-values returns whatever values the body returns. If
“abort” is selected, accepting-values will invoke the abort restart.

stream is an interactive stream that accepting-values will use to build up the dia-
log. The stream argument is not evaluated, and must be a symbol that is bound
to a stream. If stream is t (the default), *query-io* is used.

body is the body of the dialog, which contains calls to accept that will be inter-
cepted by accepting-values and used to build up the dialog. body may have zero
or more declarations as its first forms.

An accepting-values dialog is a looping structure. First, body is evaluated in
order to collect the output. During the evaluation, all calls to accept call the
accept-present-default presentation methods instead of calling the accept pre-
sentation methods. The output is displayed with incremental redisplay. accepting-values
awaits a user gesture, such as clicking on one of the fields of the
dialog. When that happens, accepting-values reads a new value for that field
using accept and replaces the old value with the new value. The loop is started
again, until the user either exits or aborts from the dialog.

Because of this looping structure, accepting-values uses the query identifier to
uniquely identify each call to accept in the body of the dialog. The query identi-
ifier is computed on each loop through the dialog, and should therefore be free of
side-effects. Query identifiers are compared using equal. Inside of accept-
ing-values, the :query-identifier argument should be supplied to each call to
accept. If it is not explicitly supplied, the prompt for that call to accept is used
as the query identifier. Thus, if :query-identifier is not supplied, programmers
must ensure that all of the prompts are different. If there is more than one call to
accept with the same query identifier, the behavior of accepting-values is
unspecified.
While inside \texttt{accepting-values}, calls to \texttt{accept} return a third value, the boolean \texttt{changed-p} that indicates whether the object is the result of new input by the user, or is just the previously supplied default. The third value will be \texttt{t} in the former case, \texttt{nil} in the latter.

When \texttt{own-window} is non-\texttt{nil}, the dialog will appear in its own “popped-up” window. In this case the initial value of \texttt{stream} is a window with which the dialog is associated. (This is similar to the \texttt{associated-window} argument to \texttt{menu-choose}.) Within the body, the value of \texttt{stream} will be the “popped-up” window. \texttt{own-window} is either \texttt{t} or a list of alternating keyword options and values. The accepted options are \texttt{:right-margin} and \texttt{:bottom-margin}; their values control the amount of extra space to the right of and below the dialog (useful if the user’s responses to the dialog take up more space than the initially displayed defaults). The allowed values for \texttt{:right-margin} are the same as for the \texttt{:x-spacing} option to \texttt{formatting-table}; the allowed values for \texttt{:bottom-margin} are the same as for the \texttt{:y-spacing} option.

\texttt{exit-boxes} specifies what the exit boxes should look like. The default behavior is though the following were supplied:

```
'(:exit "Control-\] uses these values")
(:abort "Control-z aborts")
```

\texttt{initially-select-query-identifier} specifies that a particular field in the dialog should be pre-selected when the user interaction begins. The field to be selected is tagged by the \texttt{:query-identifier} option to \texttt{accept}; use this tag as the value for the \texttt{:initially-select-query-identifier} keyword, as in this example:

```
(defun avv ()
  (let (a b c)
    (accepting-values
      (*query-io* :initially-select-query-identifier 'the-tag)
      (setq a (accept 'pathname :prompt "A pathname"))
      (terpri *query-io*)
      (setq b (accept 'integer :prompt "A number"
                       :query-identifier 'the-tag))
      (terpri *query-io*)
      (setq c (accept 'string :prompt "A string")))
    (values a b c)))
```

When the initial display is output, the input editor cursor appears after the prompt of the tagged field, just as if the user had selected that field by clicking on it. The default value, if any, for the selected field is not displayed.

\texttt{resynchronize-every-pass} is a boolean option specifying whether earlier queries depend on later values; the default is \texttt{nil}. When it is \texttt{t}, the contents of the dialog

\textbf{Menus and Dialogs}
are redisplayed an additional time after each user interaction. This has the effect of ensuring that, when the value of some field of a dialog depends on the value of another field, all of the displayed fields will be up to date.

You can use this option to alter the dialog dynamically. The following example initially displays an integer field that disappears if a value other than 1 is entered; a two-field display appears in its place.

```lisp
(defun alter-multiple-accept ()
  (let ((flag 2))
    (accepting-values
      (*query-io* :resynchronize-every-pass t)
      (setq flag (accept 'integer :default flag :prompt "Number"))
      (when (= flag 1)
        (terpri *query-io*)
        (accept 'string :prompt "String"
          (terpri *query-io*)
          (accept 'pathname :prompt "Pathname"))))))
```

`label` is as for `menu-choose`. `scroll-bars` controls what and whether scroll-bars appear on the dialog. The value is one of: `:vertical`, `:horizontal`, `:both`, and `nil` (the default). `x-position` and `y-position` are as for `menu-choose-from-drawer`.

**accept-values**

[Application Frame]

Summary: `accepting-values` is a CLIM application frame that uses `accept-values` as the name of the frame class.

**accept-values-pane-displayer**

[Function]

Arguments: frame pane &key displayer resynchronize-every-pass

Summary: When you use an `:accept-values` pane, the display function must use `accept-values-pane-displayer`. `displayer` is a function that is the body of an `accepting-values` dialog. It takes two arguments, the frame and a stream. The display function does not need to call `accepting-values` itself, since that is done by `accept-values-pane-displayer`. `resynchronize-every-pass` is as for `accepting-values`.

**display-exit-boxes**

[Generic Function]

Arguments: frame stream view

Summary: Displays the exits boxes for the `accepting-values` frame `frame` on the stream `stream`, in the view `view`. The exit boxes specification is not passed in directly, but is a slot in the frame. The default method (on `accept-values`) simply writes
Menus and Dialogs

a line of text associating the Exit and Abort strings with presentations that either exit or abort from the dialog.

The frame, stream, and view arguments may be specialized to provide a different look-and-feel for different host window systems.

**accept-values-resynchronize**

[Generic Function]

Arguments: `stream`

Summary: Causes `accepting-values` to resynchronizes the dialog once on the accepting values stream `stream` before it restarts the dialog loop.

**accept-values-command-button**

[Macro]

Arguments: `(&optional stream &key documentation query-identifier cache-value cache-test resynchronize) prompt &body body`

Summary: The prompt `prompt` creates the button area by writing to the appropriate `accepting-values` stream `stream`. `prompt` should not produce a string itself. When a pointer button is clicked in this area at runtime, `body` will be evaluated.

`accept-values-command-button` expands into a call to `invoke-accept-values-command-button`, supplying a function that executes `body` as the `continuation` argument to `invoke-accept-values-command-button`.

The `stream` argument is not evaluated, and must be a symbol that is bound to a stream. If `stream` is `t` (the default), `*query-io*` is used. `body` may have zero or more declarations as its first forms.

**invoke-accept-values-command-button**

[Generic Function]

Arguments: `stream continuation view prompt &key documentation query-identifier cache-value cache-test resynchronize`

Summary: Displays the prompt `prompt` on the stream `stream` and creates the button areas. When a pointer button is clicked in this area at runtime, the continuation will be called. `continuation` is a function that takes no arguments. `view` is a view.

`prompt` may be either a string (which will be displayed via `write-string`), or a form that will be evaluated to draw the button.

`documentation` is an object that will be used to produce pointer documentation for the button. It defaults to `prompt`. If it is a string, the string itself will be used as the pointer documentation. Otherwise it must be a function of one argument, the stream to which the documentation should be written.

Menus and Dialogs
When \textit{resynchronize} is \texttt{t}, the dialog will be redisplayed an additional time whenever the command button is clicked on. See the \textit{resynchronize-every-pass} argument to \texttt{accepting-values}.

\textit{cache-value} and \textit{cache-test} are as for \texttt{updating-output}. That is, \textit{cache-value} should evaluate to the same value if and only if the output produced by \textit{prompt} does not ever change. \textit{cache-test} is a function of two arguments that is used to compare cache values. \textit{cache-value} defaults to \texttt{t} and \textit{cache-test} defaults to \texttt{eql}.

This function may only be used inside the dynamic context of an \texttt{accepting-values}.

\textbf{frame-manager-dialog-view}  
\texttt{Arguments: frame-manager}  
\texttt{Summary: Returns the view used by default for \texttt{accepting-values} when mediated by frame-manager.}

\textbf{(setf frame-manager-dialog-view)}  
\texttt{Arguments: frame-manager view}  
\texttt{Summary: Sets the view used by default for \texttt{accepting-values} when mediated by frame-manager. Useful values for \texttt{view} are \texttt{+gadget-dialog-view+} and \texttt{+textual-dialog-view+}.}

\section*{12.4 Examples of Menus and Dialogs in CLIM}

\subsection*{12.4.1 Using \texttt{accepting-values}}

This example sets up a dialog in the CLIM window \texttt{stream} that displays the current month, date, hour, and minute (as obtained by a call to \texttt{get-universal-time}) and allows the user to modify those values. The user can select values to change by using the mouse to select values, typing in new values, and pressing \texttt{RETURN}. When done, the user selects \texttt{<END>} to accept the new values, or \texttt{<ABORT>} to terminate without changes.
(defun reset-clock (stream)
  (multiple-value-bind (second minute hour day month)
      (decode-universal-time (get-universal-time))
    (declare (ignore second))
    (format stream "Enter the time~%")
    (restart-case
      (progn
        (clim:accepting-values (stream)
          (setq month
              (clim:accept 'integer :stream stream
                :default month :prompt "Month")))
        (terpri stream)
        (setq day
            (clim:accept 'integer :stream stream
                :default day :prompt "Day"))
        (terpri stream)
        (setq hour
            (clim:accept 'integer :stream stream
                :default hour :prompt "Hour"))
        (terpri stream)
        (setq minute
            (clim:accept 'integer :stream stream
                :default minute :prompt "Minute")))
        ;; This could be code to reset the time, but instead
        ;; we're just printing it out
        (format t "%-New values: Month: -D, Day: -D, Time: -D:-2,'0D."
          minute day hour minute))
      (abort () (format t "&Time not set")))))))

Note that in CLIM, calls to accept do not automatically insert newlines. If you want to put each query on its own line of the dialog, use terpri between the calls to accept.

### 12.4.2 Using accept-values-command-button

Here is the reset-clock example with the addition of a command button that will set the number of seconds to zero.

(defun reset-clock (stream)
  (multiple-value-bind (second minute hour day month)
      (decode-universal-time (get-universal-time))
    (declare (ignore second))
    (format stream "Enter the time~%")
    (restart-case
      (progn
        (clim:accepting-values (stream)
          (setq month
              (clim:accept 'integer :stream stream
                :default month :prompt "Month")))
        (terpri stream)
        (setq day
            (clim:accept 'integer :stream stream
                :default day :prompt "Day"))
        (terpri stream)
        (setq hour
            (clim:accept 'integer :stream stream
                :default hour :prompt "Hour"))
        (terpri stream)
        (setq minute
            (clim:accept 'integer :stream stream
                :default minute :prompt "Minute")))
        ;; This could be code to reset the time, but instead
        ;; we're just printing it out
        (format t "%-New values: Month: -D, Day: -D, Time: -D:-2,'0D."
          minute day hour minute))
      (abort () (format t "&Time not set")))))

Menus and Dialogs
(progn
  (clim:accepting-values (stream)
    (setq month
      (clim:accept 'integer :stream stream
        :default month :prompt "Month"))
    (terpri stream)
    (setq day
      (clim:accept 'integer :stream stream
        :default day :prompt "Day"))
    (terpri stream)
    (setq hour
      (clim:accept 'integer :stream stream
        :default hour :prompt "Hour"))
    (terpri stream)
    (setq minute
      (clim:accept 'integer :stream stream
        :default minute :prompt "Minute")))
  (terpri stream)
  ;; Print the current time to the terminal.
  (accept-values-command-button (stream) "Print-Clock"
    (format t
      "~%Current values: Month: ~D, Day: ~D, Time: ~D:~2,'0D."
      month day hour minute))
  (abort () (format t "&Time not set")))

12.4.3 Using :resynchronize-every-pass in accepting-values

It often happens that the programmer wants to present a dialog where the individual fields of the dialog depend on one another. For example, consider a spreadsheet with seven columns representing the days of a week. Each column is headed with that day’s date. If the user inputs the date of any single day, the other dates can be computed from that single piece of input.

If you build CLIM dialogs using accepting-values, you can achieve this effect by using the :resynchronize-every-pass argument to accepting-values in conjunction with the :default argument to accept. There are three points to remember:

- The entire body of the accepting-values runs each time the user modifies any field. The body can be made to run an extra time by specifying :resynchronize-every-pass t. Code in the body may be used to enforce constraints among values.
If the :default argument to accept is used, then every time that call to accept is run, it will pick up the new value of the default.

Inside accepting-values, accept returns a third value, a boolean that indicates whether the returned value is the result of new input by the user or is just the previously supplied default.

In this example we show a dialog that accepts two real numbers, delimiting an interval on the real line. The two values are labelled Min and Max, but we wish to allow the user to supply a Min that is greater than the Max, and automatically exchange the values rather than signalling an error.

```
(defun accepting-interval (&key (min -1.0) (max 1.0) (stream *query-io*))
  (clim:accepting-values (stream :resynchronize-every-pass t)
   (fresh-line stream)
   (setq min
     (clim:accept
      'clim:real :default min
      :prompt "Min" :stream stream))
   (fresh-line stream)
   (setq max
     (clim:accept
      'clim:real :default max
      :prompt "Max" :stream stream))
   (when (< max min)
     (rotatef min max)))
  (values min max))
```

(You may want to try this example after dropping the :resynchronize-every-pass and see the behavior. Without :resynchronize-every-pass, the constraint is still enforced, but the display lags behind the values and doesn’t reflect the updated values immediately.)

### 12.4.4 Using the third value from accept in accepting-values

As a second example, consider a dialog that accepts four real numbers that delimit a rectangular region in the plane, but we wish to enforce a constraint that the region be a square. We allow the user to input any of Xmin, Xmax, Ymin, or Ymax, but enforce the constraint that:
$X_{\text{max}} - X_{\text{min}} = Y_{\text{max}} - Y_{\text{min}}$

We want to avoid changing the value that a user inputs, so we decide (in cases where the constraint has to be enforced) to change the $X$ value if the user inputs a $Y$ value, and to change the $Y$ value if the user inputs an $X$ value. When changing values, we preserve the center of the interval. We use the third returned value from \texttt{accept} to control the constraint enforcement.

\begin{verbatim}
(defun accepting-square
  (&key (xmin -1.0) (xmax 1.0) (ymin -1.0) (ymax 1.0) (stream *query-io*))
)
\end{verbatim}
(let (xmin-changed xmax-changed ymin-changed ymax-changed ptype)
  (clim:accepting-values
   (stream :resynchronize-every-pass t)
   (fresh-line stream)
   (multiple-value-setq
    (xmin ptype xmin-changed)
    (clim:accept 'clim:real :default xmin
      :prompt "Xmin" :stream stream))
   (fresh-line stream)
   (multiple-value-setq
    (xmax ptype xmax-changed)
    (clim:accept 'clim:real :default xmax
      :prompt "Xmax" :stream stream))
   (fresh-line stream)
   (multiple-value-setq
    (ymin ptype ymin-changed)
    (clim:accept 'clim:real :default ymin
      :prompt "Ymin" :stream stream))
   (fresh-line stream)
   (multiple-value-setq
    (ymax ptype ymax-changed)
    (clim:accept 'clim:real :default ymax
      :prompt "Ymax" :stream stream))
   (cond ((or xmin-changed xmax-changed)
     (let ((y-center (/ (+ ymax ymin) 2.0))
       (x-half-width (/ (- xmax xmin) 2.0)))
       (setq ymin (- y-center x-half-width)
         ymax (+ y-center x-half-width)))
     (setq xmin-changed nil
       ymax-changed nil))
   ((or ymin-changed ymax-changed)
     (let ((x-center (/ (+ xmax xmin) 2.0))
       (y-half-width (/ (- ymax ymin) 2.0)))
       (setq xmin (- x-center y-half-width)
         xmax (+ x-center y-half-width)))
     (setq ymin-changed nil
       xmax-changed nil)))))
  (values xmin xmax ymin ymax))

12.4.5 Using menu-choose

The simplest use of menu-choose is when each item is not a list. In that case, the entire item will be printed and is also the value to be returned.
If you want to return a value that is different from what was printed, the simplest method is as follows. Each item is a list; the first element is what will be printed, the remainder of the list is treated as a *plist*—the :value property will be returned. (Note nil is returned if you click on Seventeen since it has no :value.)

(clim:menu-choose
  '(
    "One" :value 1 :documentation "the loneliest number")
  "Two" :value 2 :documentation "for tea"
  "Seventeen"
  :documentation "teen magazine"))

The list of items you pass to menu-choose can serve other purposes in your application, so you might not want to put the printed appearance in the first element. You can supply a :printer function that will be called on the item to produce its printed appearance.

(clim:menu-choose
  '(1 2 17)
  :printer #'(lambda (item stream)
                (format stream "~R" item)))

The items in the menu needn’t be printed textually:

(clim:menu-choose
  '(circle square triangle)
  :printer
  #'(lambda (item stream)
      (case item
        (circle (clim:draw-circle* stream 0 0 10))
        (square (clim:draw-polygon* stream '(-8 -8 -8 8 8 8 -8)))
        (triangle (clim:draw-polygon* stream '(10 8 0 -10 -10 8)))))

The :item-list option of the list form of menu item can be used to describe a set of hierarchical menus.
12.4.6 Using menu-choose-from-drawer

This example displays in the window *page-stream* the choices One through Ten in bold-face type. When the user selects one, the string is returned along with the gesture that selected it.

```
(clim:menu-choose
 '((("Class: Osteichthyes" :documentation "Bony fishes"
   :style (nil :italic nil))
  ("Class: Chondrichthyes"
   :documentation "Cartilaginous fishes"
   :style (nil :italic nil)
   :item-list ("Order: Squaliformes" :documentation "Sharks"
   ("Order: Rajiformes" :documentation "Rays")))
  ("Class: Mammalia" :documentation "Mammals" :style (nil :italic nil)
   :item-list
   ("Order Rodentia" :item-list ("Family Sciuridae"
     "Family Muridae"
     "Family Cricetidae"
     ("..." :value nil)))
  ("Order Carnivora" :item-list ("Family: Felidae"
    "Family: Canidae"
    "Family: Ursidae"
    ("..." :value nil))
   ("..." :value nil))
  ("..." :value nil))

This example shows how you can use menu-choose-from-drawer with with-menu to create a temporary menu:
```

```
(clim:menu-choose-from-drawer
 *page-stream* 'string
 #'(lambda (stream type)
   (clim:with-text-face (:bold stream)
    (dotimes (count 10)
     (clim:present (string-capitalize (format nil "~R" (1+ count)))
      type :stream stream)
     (terpri stream))))
```

This example shows how you can use menu-choose-from-drawer with with-menu to create a temporary menu:
(defun choose-compass-direction (parent-window)
  (labels
    ((draw-compass-point
        (stream ptype symbol x y)
        (clim:with-output-as-presentation
          (:stream stream :object symbol :type ptype)
          (clim:draw-string* stream
            (symbol-name symbol) x y
            :align-x :center
            :align-y :center
            :text-style
            "'(:sans-serif :roman :large))
          )
        ))
    (draw-compass
     (stream ptype)
     (clim:draw-line* stream 0 25 0 -25 :line-thickness 2)
     (clim:draw-line* stream 25 0 -25 0 :line-thickness 2)
     (loop for point in '((n 0 -30) (s 0 30) (e 30 0) (w -30 0))
        do (apply #'draw-compass-point
          stream ptype point)))))
  (clim:with-menu (menu parent-window)
    (clim:menu-choose-from-drawer menu 'clim:menu-item
      #'draw-compass)))
Chapter 13  Extended Stream Output Facilities
13.1 Basic Output Streams

CLIM performs all of its input and output operations on objects called \textit{streams}. Stream functionality is partitioned into two layers: the basic stream protocol and the extended stream protocol. The stream-oriented output layer is implemented on top of the sheet output architecture. The basic CLIM output stream protocol is based on the character output stream protocol proposal submitted to the ANSI Common Lisp committee by David Gray. This proposal was not approved by the committee, but CLIM provides an implementation of the basic output stream facilities. This protocol is documented in Appendix D, “Common Lisp Streams.”

\texttt{standard-output-stream} \hspace{1cm} \textbf{[Class]}

Summary: This class provides an implementation of the CLIM basic output stream protocol, based on the CLIM output kernel. Members of this class are mutable.

\texttt{stream-write-char} \hspace{1cm} \textbf{[Generic Function]}

Arguments: \textit{stream character}

Summary: Writes the character \textit{character} to the output stream \textit{stream}, and returns \textit{character} as its value.

\texttt{stream-line-column} \hspace{1cm} \textbf{[Generic Function]}

Arguments: \textit{stream}

Summary: This function returns the column number where the next character will be written on the output stream \textit{stream}. The first column on a line is numbered 0.

\texttt{stream-start-line-p} \hspace{1cm} \textbf{[Generic Function]}

Arguments: \textit{stream}

Summary: Returns \texttt{t} if the output stream \textit{stream} is positioned at the beginning of a line (that is, column 0); otherwise, it returns \texttt{nil}.

\texttt{stream-write-string} \hspace{1cm} \textbf{[Generic Function]}

Arguments: \textit{stream string} \texttt{&optional (start 0) end}

Summary: Writes the string \textit{string} to the output stream \textit{stream}. If \textit{start} and \textit{end} are supplied, they are integers that specify what part of \textit{string} to output. \textit{string} is returned as the value.
stream-terpri
Arguments: stream
Summary: Writes an end-of-line character on the output stream stream, and returns nil.

stream-fresh-line
Arguments: stream
Summary: Writes an end-of-line character on the output stream stream only if the stream is not at the beginning of the line.

stream-finish-output
Arguments: stream
Summary: Ensures that all the output sent to the output stream stream has reached its destination, and only then does it return nil.

stream-force-output
Arguments: stream
Summary: Like stream-finish-output, except that it may immediately return nil without waiting for the output to complete.

stream-clear-output
Arguments: stream
Summary: Aborts any outstanding output operation in progress on the output stream stream, and returns nil.

stream-advance-to-column
Arguments: stream column
Summary: Writes enough blank space on the output stream stream so that the next character will be written at the position specified by column, which is an integer.

13.2 Extended Output Streams

In addition to the basic output stream protocol, CLIM defines an extended output stream protocol. This protocol extends the stream model to maintain the state of a text cursor, margins, text styles, inter-line spacing, and so forth.
The extended output stream protocol is discussed in the following two sections, “The Text Cursor” and “Text.”

**extended-output-stream** [Protocol Class]
Summary: The protocol class for CLIM extended output streams. This is a subclass of `output-stream`. If you want to create a new class that behaves like an extended output stream, it should be a subclass of `extended-output-stream`. Subclasses of `extended-output-stream` must obey the extended output stream protocol.

**extended-output-stream-p** [Function]
Arguments: `object`
Summary: Returns `t` if `object` is a CLIM extended output stream; otherwise, it returns `nil`.

**:foreground** [Initarg]
**:background** [Initarg]
**:default-text-style** [Initarg]
**:vertical-spacing** [Initarg]
**:text-margin** [Initarg]
**:end-of-line-action** [Initarg]
**:end-of-page-action** [Initarg]
**:default-view** [Initarg]
Summary: All subclasses of `extended-output-stream` must handle these initargs, which are used to specify, respectively, the medium foreground and background inks, default text style, vertical spacing, default text margin, end of line and end of page actions, and the default view for the stream.

**standard-extended-output-stream** [Class]
Summary: This class provides an implementation of the CLIM extended output stream protocol, based on the CLIM output kernel.

Members of this class are mutable.
13.3 The Text Cursor

In the days when display devices displayed only two-dimensional arrays of fixed-width characters, the text cursor was a simple thing. A discrete position was selected in integer character units, and a character could go there and nowhere else. Even for variable-width fonts, it was enough to address a character by the pixel position of one of its corners. However, variable-height fonts with variable baselines on pixel-addressable displays upset this simple model. The “logical” vertical reference point is the baseline, as it is in typesetting. In typesetting, however, an entire line of text is created with baselines aligned and padded to the maximum ascent and descent, and then the entire line is put below the previous line.

It is clearly desirable to have the characters on a line aligned with their baselines, but when the line on the display is formed piece by piece, it is impossible to pick in advance the proper baseline. The solution CLIM adopts is to choose a provisional baseline.

We assume that text has at least six properties. With a reference point of (0, 0) at the upper left of the text, it has a bounding box consisting of ascent, descent, left kerning, right extension, and a displacement to the next reference point in both $x$ and $y$. CLIM determines the position of the reference point and draws the text relative to that, and then the cursor position is adjusted by the displacement. In this way, text has width and height, but the $x$ and $y$ displacements need not equal the width and height.

CLIM adopts the following approach to the actual rendering of a glyph. Textual output using the stream functions (not the graphics functions) maintains text on a “line.” Note that a line is not an output record, but is rather a collection of “text so far,” a top (positioned at the bottom of the previous line plus the stream’s vertical spacing), a baseline, a bottom, and a “cursor position.” The cursor position is defined to be at the top of the line, not at the baseline. The reason for this is that the baseline can move, but the top is relative to the previous line, which has been completed and therefore doesn’t move. If text is drawn on the current line whose ascent is greater than the current ascent of the line, then the line is moved down to make room. This can be done easily using the output records for the existing text on the line. When there is enough room, the reference point for the text is the $x$ position of the cursor at the baseline, and the cursor position is adjusted by the displacement.

Figure 21 shows this in action before and after each of three characters are drawn. In all three cases, the small circle is the “cursor position.” At first, there is nothing on the line.
Figure 21. Determining the Position of the Text Cursor
The first character establishes the initial baseline and is then drawn. The upper left corner of the character is where the cursor was (as in the traditional model), but this will not remain the case. Drawing the second character, which is larger than the first, requires moving the first character down in order to get the baselines to align; during this time, the top of the line remains the same. Again, the upper left of the second character is where the cursor was, but that is no longer the case for the first character (which has moved down). The third character is smaller than the second, so no moving of characters needs to be done. However, the character is drawn to align the baselines, which in this case means the upper left is not where the cursor was. Nor is the cursor at the upper right of the character as it was for the previous two characters. It is, however, at the upper right of the collective line.

### 13.3.1 The Text Cursor Protocol

Many streams that maintain a text cursor also display some visible indication of it. The object that represents this display is (somewhat confusingly) also called a cursor.

#### cursor

[Protocol Class]

Summary: The protocol class that corresponds to cursors. If you want to create a new class that behaves like cursor, it should be a subclass of cursor. Subclasses of cursor must obey the cursor protocol. Members of this class are mutable.

#### cursorp

[Function]

Arguments: object

Summary: Returns t if object is a cursor; otherwise, it returns nil.

#### :sheet

[Initarg]

Summary: The :sheet initarg is used to specify the sheet with which the cursor is associated.

#### standard-text-cursor

[Class]

Summary: The instantiable class that implements a text cursor. Typically, ports will further specialize this class.

#### cursor-sheet

[Generic Function]

Arguments: cursor

Summary: Returns the sheet with which the cursor cursor is associated.
**cursor-position**  
**[Generic Function]**  
**Arguments:** cursor  
**Summary:** Returns the x and y position of the cursor cursor as two values.

**(setq* cursor-position)**  
**[Generic Function]**  
**Arguments:** x y cursor  
**Summary:** Sets the x and y position of the cursor cursor to the specified position. For the details of **setq**, see Appendix C.4, “Multiple-Value Setf.”

**cursor-visibility**  
**[Generic Function]**  
**Arguments:** cursor  

**(setq cursor-visibility)**  
**[Generic Function]**  
**Arguments:** visibility cursor  
**Summary:** Returns (or sets) the visibility of the cursor cursor. The visibility is one of :on (the cursor will be visible at its current position), :off (the cursor is active, but not visible at its current position), or nil (the cursor is to be deactivated).

**display-cursor**  
**[Generic Function]**  
**Arguments:** cursor state  
**Summary:** This draws or erases the cursor cursor. If state is :draw, the cursor will be drawn. If the state is :erase, the cursor will be erased.

### 13.3.2 The Stream Text Cursor Protocol

The following generic functions comprise the stream text cursor protocol. Any extended output stream class must implement methods for these generic functions.

**stream-text-cursor**  
**[Generic Function]**  
**Arguments:** stream  

**(setq stream-text-cursor)**  
**[Generic Function]**  
**Arguments:** cursor stream  
**Summary:** Returns (or sets) the text cursor object for the stream stream.
stream-cursor-position

Arguments: stream
Summary: Returns the current text cursor position for the extended output stream stream as two integer values, the x and y positions.

(setf* stream-cursor-position)

Arguments: x y stream
Summary: Sets the text cursor position of the extended output stream stream to x and y. x and y are in device units, and must be integers. For the details of setf*, see Appendix C.4, “Multiple-Value Setf.”

stream-increment-cursor-position

Arguments: stream dx dy
Summary: Moves the text cursor position of the extended output stream stream relatively, adding dx to the x coordinate and dy to the y coordinate. Either of dx or dy may be nil, meaning the the x or y cursor position will be unaffected. Otherwise, dx and dy must be integers.

13.4 Text

This section addresses text as it relates to output streams.

13.4.1 The Text Protocol

The following generic functions comprise the text protocol. Any extended output stream class must implement methods for these generic functions.

stream-character-width

Arguments: stream character &key text-style
Summary: Returns a rational number corresponding to the amount of horizontal motion of the cursor position that would occur if the character character were output to the extended output stream stream in the text style text-style (which defaults to the current text style for the stream). This ignores the stream’s text margin.
stream-string-width [Generic Function]
Arguments: stream string &key start end text-style
Summary: Computes how the cursor position would move horizontally if the string string were output to the extended output stream stream in the text style text-style (which defaults to the current text style for the stream) starting at the left margin. This ignores the stream’s text margin.

The first returned value is the x coordinate that the cursor position would move to. The second returned value is the maximum x coordinate the cursor would visit during the output. (This is the same as the first value unless the string contains a #\Newline character.)

start and end are integers that default to 0 and the string length, respectively.

stream-text-margin [Generic Function]
Arguments: stream

(setf stream-text-margin) [Generic Function]
Arguments: margin stream
Summary: Returns the x coordinate at which text wraps around on the extended output stream stream (see stream-end-of-line-action). The default setting is the width of the viewport, which is the right-hand edge of the viewport when it is horizontally scrolled to the “initial position.”

You can use setf with stream-text-margin to establish a new text margin. If margin is nil, then the width of the viewport will be used. If the width of the viewport is later changed, the text margin will change, too.

stream-line-height [Generic Function]
Arguments: stream &key text-style
Summary: Returns what the line height of a line on the extended output stream stream containing text in the text style text-style would be, as a rational number. text-style defaults to the current text style for the stream.

stream-vertical-spacing [Generic Function]
Arguments: stream
Summary: Returns the current inter-line spacing (as a rational number) for the extended output stream stream.
stream-baseline

Arguments: stream
Summary: Returns the current text baseline (as a rational number) for the extended output stream stream.

13.4.2 Mixing Text and Graphics

The following macro provides a convenient way to mix text and graphics on the same output stream.

with-room-for-graphics

Arguments: (optional stream &key (move-cursor t) height record-type) &body body
Summary: Binds the dynamic environment to establish a local Cartesian coordinate system for doing graphics output onto the extended output stream designated by stream. The origin (0, 0) of the local coordinate system is placed at the current cursor position, and is in the lower left corner of the area created. If the boolean move-cursor is t (the default), then after the graphic output is completed, the cursor is positioned past (immediately below) this origin. The bottom of the vertical block allocated is at this location (that is, just below point (0, 0), not necessarily at the bottom of the output done).

The stream argument is not evaluated, and must be a symbol that is bound to a stream. If stream is t (the default), *standard-output* is used. body may have zero or more declarations as its first forms.

If height is supplied, it must be a rational number that specifies the amount of vertical space to allocate for the output, in device units. If it is not supplied, the height is computed from the output.

record-type specifies the class of output record to create to hold the graphical output. The default is standard-sequence-output-record.

13.4.3 Wrapping Text Lines

stream-end-of-line-action

Arguments: stream

Extended Stream Output Facilities
The end-of-line action controls what happens if the text cursor position moves horizontally out of the viewport or if text output reaches the text margin. (By default the text margin is the width of the viewport, so these often coincide.)

`stream-end-of-line-action` returns the end-of-line action for the extended output stream `stream`. It can be changed by using `setf` on `stream-end-of-line-action`.

The end-of-line action is one of:

- `:wrap`—when doing text output, wrap the text around (that is, break the text line and start another line). When setting the cursor position, scroll the window horizontally to keep the cursor position inside the viewport. This is the default.
- `:scroll`—scroll the window horizontally to keep the cursor position inside the viewport, then keep doing the output.
- `:allow`—ignore the text margin and do the output on the drawing plane beyond the visible part of the viewport.

The `stream` argument is not evaluated, and must be a symbol that is bound to a stream. If `stream` is `t`, `*standard-output*` is used. `body` may have zero or more declarations as its first forms.

The end-of-page action controls what happens if the text cursor position moves vertically out of the viewport.

`stream-end-of-page-action` returns the end-of-page action for the extended output stream `stream`. Change it by using `setf` on `stream-end-of-page-action`.

The end-of-page action is one of:
Extended Stream Output Facilities

- **:wrap**—when doing text output, wrap the text around (that is, go back to the top of the viewport).
- **:scroll**—scroll the window vertically to keep the cursor position inside the viewport, then keep doing output. This is the default.
- **:allow**—ignore the viewport and do the output on the drawing plane beyond the visible part of the viewport.

**with-end-of-page-action**

*Macro*

Arguments: `(stream action) &body body`

Summary: Temporarily changes stream’s end-of-page action for the duration of execution of body. *action* must be one of the actions described in `stream-end-of-page-action`.

The `stream` argument is not evaluated, and must be a symbol that is bound to a stream. If `stream` is `t`, `*standard-output*` is used. *body* may have zero or more declarations as its first forms.

### 13.5 Attracting the User’s Attention

**beep**

*Generic Function*

Arguments: `&optional sheet`

Summary: Attracts the user’s attention, usually with an audible sound.

### 13.6 Buffering Output

Some mediums that support the output protocol may buffer output. When buffering is enabled on a medium, the time at which output is actually done on the medium is unpredictable. `force-output` or `finish-output` can be used to ensure that all pending output gets completed. If the medium is a bidirectional stream, a `force-output` is performed whenever any sort of input is requested on the stream.

`with-buffered-output` provides a way to control when buffering is enabled on a medium. By default, CLIM’s interactive streams are buffered if the underlying window system supports buffering.
medium-buffering-output-p  [Generic Function]
Arguments: medium
Summary: Returns t if the medium medium is currently buffering output; otherwise, it returns nil.

(setf medium-buffering-output-p)  [Generic Function]
Arguments: buffer-p medium
Summary: Sets medium-buffering-output-p of the medium medium to buffer-p.

with-output-buffered  [Macro]
Arguments: (medium &optional (buffer-p t)) &body body
Summary: If buffer-p is t (the default), this causes the medium designated by medium to start buffering output, and evaluates body in that context. If buffer-p is nil, force-output will be called before body is evaluated. When body is exited (or aborted from), force-output will be called if output buffering will be disabled after with-output-buffered is exited.

The medium argument is not evaluated, and must be a symbol that is bound to a medium. If medium is t, *standard-output* is used. body may have zero or more declarations as its first forms.

13.7 CLIM Window Stream Pane Functions

The following functions can be called on any pane that is a subclass of clim-stream-pane. (Such a pane is often simply referred to as a window.) These are provided as a convenience for programmers and for compatibility with CLIM 1.1.

window-clear  [Generic Function]
Arguments: window
Summary: Clears the entire drawing plane by filling it with the background design of the CLIM stream pane window. If window has an output history, it is cleared as well. The text cursor position of window, if there is one, is reset to the upper left corner.

window-refresh  [Generic Function]
Arguments: window
Summary: Clears the visible part of the drawing plane of the CLIM stream pane \textit{window}, and then if the window stream is an output recording stream, the output records in the visible part of the window are replayed.

\textbf{window-viewport} \hspace{1cm} [Generic Function]
\textbf{Arguments:} \textit{window}
\textbf{Summary:} Returns the viewport region of the CLIM stream pane \textit{window}. The returned region will usually be a \texttt{standard-bounding-rectangle}.

\textbf{window-erase-viewport} \hspace{1cm} [Generic Function]
\textbf{Arguments:} \textit{window}
\textbf{Summary:} Clears the visible part of the drawing plane of the CLIM stream pane \textit{window} by filling it with the background design.

\textbf{window-viewport-position} \hspace{1cm} [Generic Function]
\textbf{Arguments:} \textit{window}
\textbf{Summary:} Returns two values, the \texttt{x} and \texttt{y} position of the upper left corner of the CLIM stream pane \textit{window}'s viewport.

\textbf{(setf window-viewport-position)} \hspace{1cm} [Generic Function]
\textbf{Arguments:} \texttt{x} \texttt{y} \textit{window}
\textbf{Summary:} Sets the position of the upper left corner of the CLIM stream pane \textit{window}'s viewport to \texttt{x} and \texttt{y}.
334CLIM User Guide
Chapter 14  Output Recording and Redisplay
14.1  Conceptual Overview of Output Recording

Output recording is an important part of CLIM. It provides the basis for scrolling windows, for formatted output of tables and graphs, for the ability of presentations to retain their semantics, and for incremental redisplay.

The output recording mechanism is enabled by default. Unless you turn it off, all output that occurs on a window is captured and saved by the output recording mechanism. The output is captured in output records. An output record is an object that contains either other output records or an output record element.

Since output records can contain other output records, we can view the organization of output records as a tree structure. The top-level output record, which contains all the output done on that window, is called the history of the window.

![Figure 22. The Tree Structure of an Output Record](image)

Each rectangle in Figure 22 is an output record. The top-level record is an output record called a history. Each output record is a leaf of the tree and is called a displayed output record element. The intermediate output records are both output records and output record elements of their immediate superior.

CLIM automatically segments the output into output records. The result of each atomic drawing operation is put into a new output record. Each presentation is put into a new output record. Strings are treated differently; CLIM concatenates strings into one output record until a newline is encountered, which begins a new output record.
One use of an output record is to *replay* it; that is, to produce the output again. Scrolling is implemented by replaying the appropriate output records. When using the techniques of incremental redisplay, your code determines which portions of the display have changed, whereupon the appropriate output records are updated to the new state and the output records are replayed.

CLIM’s table and graph formatters use output records. For example, your code uses `formatting-table` to format output into rows and cells; this output is sent to a particular stream. Invisibly to you, CLIM temporarily binds this stream to an intermediate stream and runs a constraint engine over the code to determine the layout of the table. The result is a set of output records which contain the table, its rows, and its cells. Finally, CLIM replays these output records to your original stream.

*Presentations* are a special case of output records that remember the object and the type of object associated with the output.

The concept of the tree structure organization of output records is further illustrated by the organization of the output records of a formatted table. The table itself is stored in an output record; each row has its own output record and each cell has its own output record.

![Figure 23. The Output Records of a Formatted Table](image)

### 14.2 CLIM Operators for Output Recording

The purpose of output recording is to capture the output done by an application onto a stream. The objects used to capture output are called output records and output record elements. The following classes and predicates correspond to the objects used in output recording.
output-record [Protocol Class]
Summary: The protocol class used to indicate that an object is an output record. A subclass of bounding-rectangle, output records obey the bounding rectangle protocol. If you want to create a new class that behaves like an output record, it should be a subclass of output-record. Subclasses of output-record must obey the output-record protocol.

All output records are mutable.

output-record-p [Function]
Arguments: object
Summary: Returns t if object is an output record; otherwise, it returns nil.

displayed-output-record [Protocol Class]
Summary: The protocol class that is used to indicate that an object is a displayed output record, that is, an object that represents a visible piece of output on some output stream. This is a subclass of bounding-rectangle. If you want to create a new class that behaves like a displayed output record, it should be a subclass of displayed-output-record. Subclasses of displayed-output-record must obey the displayed output record protocol.

All displayed output records are mutable.

displayed-output-record-p [Function]
Arguments: object
Summary: Returns t if object is a displayed output record; otherwise, it returns nil.

:x-position [Initarg]
:y-position [Initarg]
:parent [Initarg]
Summary: All subclasses of either output-record or displayed-output-record must handle these three initargs, which are used to specify, respectively, the x and y position of the output record, and the parent of the output record.

:size [Initarg]
Summary: All subclasses of output-record must handle the size initarg, which specifies how much room should be left for child output records (if, for example, the chi-
dren are stored in a vector). \texttt{name} may be ignored, provided that the resulting output record is able to store the specified number of child output records.

### 14.2.1 The Basic Output Record Protocol

All subclasses of \texttt{output-record} and \texttt{displayed-output-record} must inherit or implement methods for the following generic functions. For details of \texttt{setf*}, see Appendix C.4, “Multiple-Value Setf.”

#### output-record-position

Arguments: \texttt{record}

Summary: Returns the \texttt{x} and \texttt{y} position of the output record \texttt{record} as two rational numbers. The position of an output record is the position of the upper-left corner of its bounding rectangle. The position is relative to the stream, where \((0, 0)\) is (initially) the upper-left corner of the stream.

#### (setf* output-record-position)

Arguments: \texttt{x y record}

Summary: Changes the \texttt{x} and \texttt{y} position of the output record \texttt{record} to be \texttt{x} and \texttt{y} (which are rational numbers), and updates the bounding rectangle to reflect the new position (and saved cursor positions, if the output record stores it). If \texttt{record} has any children, all of the children (and their descendants as well) will be moved by the same amount as \texttt{record} was moved. The bounding rectangles of all of \texttt{record}'s ancestors will also be updated to be large enough to contain \texttt{record}. This does not replay the output record, but the next time the output record is replayed it will appear at the new position.

#### output-record-start-cursor-position

Arguments: \texttt{record}

Summary: Returns the \texttt{x} and \texttt{y} starting cursor position of the output record \texttt{record} as two integer values. The positions are relative to the stream, where \((0, 0)\) is (initially) the upper-left corner of the stream.

Text output records and updating output records maintain the cursor position. Graphical output records and other output records that do not require or affect the cursor position will return \texttt{nil} as both of the values.
(setf* output-record-start-cursor-position)  [Generic Function]
Arguments:  x  y  record
Summary:  Changes the x and y starting cursor position of the output record record to be x
and y (which are integers). This does not affect the bounding rectangle of record, nor does it replay
the output record. For those output records that do not require or affect the cursor position, the
method for this function does nothing.

output-record-end-cursor-position  [Generic Function]
Arguments:  record
Summary:  Returns the x and y ending cursor position of the output record record as two
integer values. The positions are relative to the stream, where (0, 0) is initially the upper-left
corner. Graphical output records do not track the cursor position, so only text output record and some
others will return meaningful values for this. Text output records and updating output records maintain
the cursor position. Graphical output records and other output records that do not require or affect
the cursor position will return nil as both of the values.

(setf* output-record-end-cursor-position)  [Generic Function]
Arguments:  x  y  record
Summary:  Changes the x and y ending cursor position of the output record record to be x
and y (which are integers). This does not affect the bounding rectangle of record, nor does it replay
the output record. For those output records that do not require or affect the cursor position, the
method for this function does nothing.

output-record-parent  [Generic Function]
Arguments:  record
Summary:  Returns the output record that is the parent of the output record record, or nil if
the record has no parent.

replay  [Function]
Arguments:  record  stream  &optional  region
Summary:  This function binds stream-recording-p of stream to nil, and then calls
replay-output-record on the arguments record, stream, and region. If
stream-drawing-p of stream is nil, replay does nothing. replay is typically
called during scrolling, by repaint handlers, and so on.

region defaults to nil.
**replay-output-record**  
**Arguments:** record stream &optional region x-offset y-offset  
**Summary:** Displays the output captured by the output record record on the output recording stream stream, exactly as it was originally captured (subject to subsequent modifications). The current user transformation, line style, text style, ink, and clipping region of stream are all ignored during the replay operation. Instead, these are gotten from the output record.

If record is not a displayed output record, then replaying it involves replaying all of its children. If record is a displayed output record, then replaying it involves redoing the graphics operation captured in the record.

region is a region that limits what records are displayed. Only those records that overlap region are replayed. The default for region is +everywhere+.

stream must be the same stream on which the output records were originally recorded.

**erase-output-record**  
**Arguments:** record stream  
**Summary:** Erases the output record record from the output recording stream stream, removes record from stream’s output history, and ensures that all other output records that were covered by record are visible. In effect, this draws background ink over the record, and then redraws all the records that overlap record.

**output-record-refined-sensitivity-test**  
**Arguments:** record x y  
**Summary:** This is used to definitively answer hit detection queries, that is, determining that the point (x, y) is contained within the output record record. Once the position (x, y) has been determined to lie within output-record-hit-detection-rectangle*, output-record-refined-sensitivity-test is invoked. Output record subclasses can provide a method that defines a hit more precisely; for example, output records for elliptical rings will implement a method that detects whether the pointing device is on the elliptical ring.

**highlight-output-record**  
**Arguments:** record stream state  
**Summary:** This method is called in order to draw a highlighting box around the output record record on the output recording stream stream. state will be either :highlight (meaning to draw the highlighting) or :unhighlight (meaning to erase the
highlighting). The default method (on CLIM’s standard output record class) simply draws a rectangle that corresponds to the bounding rectangle of record.

### 14.2.2 The Output Record “Database” Protocol

All classes that are subclasses of `output-record` must implement methods for the following generic functions.

- **output-record-children**
  - **Arguments:** `record`
  - **Summary:** Returns a fresh list of all the children of the output record `record`.

- **add-output-record**
  - **Arguments:** `child record`
  - **Summary:** Adds the new output record `child` to the output record `record`. The bounding rectangle for `record` and all its ancestors is updated accordingly.
    
    The methods for the `add-output-record` will typically specialize only the `record` argument.

- **delete-output-record**
  - **Arguments:** `child record &optional (errorp t)`
  - **Summary:** Removes the output record `child` from the output record `record`. The bounding rectangle for `record` (and all its ancestors) is updated to account for the child having been removed.
    
    If `errorp` is `t` (the default) and `child` is not contained within `record`, then an error is signaled.

    The methods for the `delete-output-record` will typically specialize only the `record` argument.

- **clear-output-record**
  - **Arguments:** `record`
  - **Summary:** Removes all of the children from the output record `record`, and resets the bounding rectangle of `record` to its initial state.
output-record-count

Arguments: record
Summary: Returns the number of children contained within the output record record.

map-over-output-records-containing-position

Arguments: function record x y &optional x-offset y-offset
Summary: Maps over all of the children of the output record record that contain the point at (x, y), calling function on each one. function is a function of one argument, the record containing the point; it has dynamic extent.

If there are multiple records that contain the point and that overlap each other, map-over-output-records-containing-position hits the most recently inserted record first and the least recently inserted record last.

map-over-output-records-overlapping-region

Arguments: function record region &optional x-offset y-offset
Summary: Maps over all of the children of the output record record that overlap the region region, calling function on each one. function is a function of one argument, the record overlapping the region; it has dynamic extent.

If there are multiple records that overlap the region and that overlap each other, map-over-output-records-overlapping-region hits the least recently inserted record first and the most recently inserted record last.

14.2.3 Types of Output Records

This section discusses several types of output records, including two standard classes of output records and the displayed output record protocol.

14.2.3.1 Standard Output Record Classes

standard-sequence-output-record

Summary: The standard class provided by CLIM to store a relatively short sequence of output records; a subclass of output-record. The retrieval complexity of this class is O(n). Most of the formatted output facilities (such as formatting-table) create output records that are a subclass of standard-sequence-output-record.
standard-tree-output-record

Summary: The standard class provided by CLIM to store longer sequences of output records. Typically, the child records of a tree output record will be sorted in some way, such as an ordering on the $x$ and $y$ coordinates of the children. The retrieval complexity of this class is $O(\log n)$.

14.2.3.2 Graphics Displayed Output Records

Graphics displayed output records are used to record the output produced by the graphics functions, such as `draw-line*`. Each graphics displayed output record describes the output produced by a call to one of the graphics functions.

CLIM graphics displayed output records capture the following information, so that the original output can be redrawn exactly at replay time:

- The description of the graphical object itself, for example, the end points of a line or the center point and radius of a circle
- The programmer-supplied ink at the time the drawing function was called (indirect inks are not resolved, so you can later change the default foreground and background ink of the medium and have that change affect the already-created output records during replay)
- For paths, the programmer-supplied line-style at the time the drawing function was called
- The programmer-supplied clipping region at the time the drawing function was called
- The user transformation

graphics-displayed-output-record

Summary: The protocol class that corresponds to output records for the graphics functions, such as `draw-line*`. This is a subclass of `displayed-output-record`. If you want to create a new class that behaves like a graphics displayed output record, it should be a subclass of `graphics-displayed-output-record`. Subclasses of `graphics-displayed-output-record` must obey the graphics displayed output record protocol.

graphics-displayed-output-record-p

Arguments: object
14.2.3.3 Text Displayed Output Records

Text displayed output records are used to record the textual output produced by such functions as `stream-write-char` and `stream-write-string`. Each text displayed output record corresponds to no more than one line of textual output (that is, line breaks caused by `terpri` and `fresh-line` create a new text output record, as do certain other stream operations).

Text displayed output records store the following information:
- The displayed text strings
- The starting and ending cursor positions
- The text style in which the text string was written
- The programmer-supplied ink at the time the drawing function was called (indirect inks are not resolved, so that you can later change the default foreground and background ink of the medium and have that change affect the already-created output records during replay)
- The programmer-supplied clipping region at the time the drawing function was called

The following three generic functions comprise the text displayed output record protocol.

`text-displayed-output-record-p` [Function]
Arguments: `object`
Summary: Returns `t` if `object` is a text displayed output record; otherwise, it returns `nil`.

The following three generic functions comprise the text displayed output record protocol.

`add-character-output-to-text-record` [Generic Function]
Arguments: `text-record character text-style width height baseline`
Summary: Adds the character \textit{character} to the text displayed output record \textit{text-record} in the text style \textit{text-style}. \textit{width} and \textit{height} are the width and height of the character in device units, and are used to compute the bounding rectangle for the text record. \textit{baseline} is the new baseline for characters in the output record.

\textbf{add-string-output-to-text-record} \hspace{1cm} [Generic Function]
\textbf{Arguments:} \textit{text-record} \textit{string} \textit{start} \textit{end} \textit{text-style} \textit{width} \textit{height} \textit{baseline}
\textbf{Summary:} Adds the string \textit{string} to the text displayed output record \textit{text-record} in the text style \textit{text-style}. \textit{start} and \textit{end} are integers that specify the substring within \textit{string} to add to the text output record. \textit{width} and \textit{height} are the width and height of the character in device units, and are used to compute the bounding rectangle for the text record. \textit{baseline} is the new baseline for characters in the output record.

\textbf{text-displayed-output-record-string} \hspace{1cm} [Generic Function]
\textbf{Arguments:} \textit{text-record}
\textbf{Summary:} Returns the string contained by the text displayed output record \textit{text-record}. This function returns objects that reveal CLIM’s internal state; do not modify those objects.

14.2.3.4 Top-Level Output Records

Top-level output records are similar to ordinary output records, except that they maintain additional state, such as the information required to display scroll bars.

\textbf{stream-output-history-mixin} \hspace{1cm} [Class]
\textbf{Summary:} This class is mixed into some other output record class to produce a new class that is suitable for use as a top-level output history.

When the bounding rectangle of a member of this class is updated, any window decorations (such as scroll bars) associated with the stream with which the output record \textit{history} is associated are updated, too.

\textbf{standard-tree-output-history} \hspace{1cm} [Class]
\textbf{Summary:} The standard class provided by CLIM to use as the top-level output history. This will typically be a subclass of both \textbf{standard-tree-output-record} and \textbf{stream-output-history-mixin}.
14.2.4 Output Recording Streams

CLIM defines an extension to the stream protocol that supports output recording. The stream has an associated output history record and provides controls to enable and disable output recording.

output-recording-stream

[Protocol Class]
Summary: The protocol class that indicates that a stream is an output recording stream. If you want to create a new class that behaves like an output recording stream, it should be a subclass of output-recording-stream. Subclasses of output-recording-stream must obey the output recording stream protocol.

output-recording-stream-p

[Function]
Arguments: object
Summary: Returns t if object is an output recording stream; otherwise, it returns nil.

standard-output-recording-stream

[Class]
Summary: The class used by CLIM to implement output record streams. This is a subclass of output-recording-stream. Members of this class are mutable.

14.2.4.1 The Output Recording Stream Protocol

The following generic functions comprise the output recording stream protocol. All subclasses of output-recording-stream implement methods for these generic functions.

stream-recording-p

[Generic Function]
Arguments: stream
Summary: Returns t when the output recording stream stream is recording all output performed to it; otherwise, it returns nil.

(setf stream-recording-p)

[Generic Function]
Arguments: recording-p stream
Summary: Changes the state of stream-recording-p to be recording-p, which must be either t or nil.
stream-drawing-p
Arguments: stream
Summary: Returns t when the output recording stream stream will actually draw on the viewport when output is being performed to it; otherwise, it returns nil.

(setf stream-drawing-p)
Arguments: drawing-p stream
Summary: Changes the state of stream-recording-p to be drawing-p, which must be either t or nil.

stream-output-history
Arguments: stream
Summary: Returns the history (or top-level output record) for the output recording stream stream.

stream-current-output-record
Arguments: stream
Summary: The current “open” output record for the output recording stream stream, to which stream-add-output-record will add a new child record. Initially, this is the same as stream-output-history. As nested output records are created, this acts as a “stack.”

(setf stream-current-output-record)
Arguments: record stream
Summary: Sets the current “open” output record for the output recording stream stream to the output record record.

stream-add-output-record
Arguments: stream record
Summary: Adds the output record record to the current output record on the output recording stream stream. (The current output record is the output record returned by stream-current-output-record.)

stream-replay
Arguments: stream &optional region
Summary: Directs the output recording stream stream to invoke replay on its output history. Only those records that overlap the region region (which defaults to the viewport of the stream) are replayed.
14.2.4.2 Graphics Output Recording

We use draw-line* as an example here, but calling any of the drawing functions specified in Section 2.3, “CLIM Drawing Functions,” and Section 2.4, “Graphics Protocols,” results in the following series of function calls on an output recording stream:

- A program calls draw-line* on arguments sheet, x1, y1, x2, y2, and perhaps some drawing options.
- draw-line* merges the supplied drawing options into the sheet’s medium, and then calls medium-draw-line* on the sheet.
- The :around method for medium-draw-line* on the output recording stream is called. This creates an output record with all of the information necessary to replay the output record, if stream-recording-p is t. If stream-drawing-p is t, this then does a call-next-method.
- The primary method for medium-draw-line* performs the necessary user transformations by applying the medium transformation to x1, y1, x2, y2, and the clipping region. Then it draws on the underlying window.

replay-output-record for a graphics displayed output record simply calls the medium drawing function (such as medium-draw-line*) directly on the sheet (not on the medium) with stream-recording-p set to nil and stream-drawing-p set to t.

14.2.4.3 Text Output Recording

This is the place where write-string and similar functions are captured in order to create an output record. The generic functions include protocol like stream-write-string that are specialized by output recording streams to do the output recording.

stream-text-output-record [Generic Function]
Arguments: stream text-style
Summary: Returns a text output record for the output recording stream stream suitable for holding characters in the text style text-style. If there is a currently “open” text output record that can hold characters in the specified text style, it is returned. Otherwise a new text output record is created that can hold characters in that text style, and its starting cursor position is set to the cursor position of stream.
stream-close-text-output-record

Arguments: stream
Summary: Closes the output recording stream stream’s currently “open” text output record by recording the stream’s current cursor position as the ending cursor position of the record and adding the text output record to stream’s current output record by calling stream-add-output-record.

If there is no “open” text output record, stream-close-text-output-record does nothing. Calling stream-finish-output, stream-force-output, calling redisplay, setting the text cursor position (via stream-set-cursor-position, terpri, or fresh-line), creating a new output record (for example, via with-new-output-record), or changing the state of stream-recording-p closes the current text output record.

stream-add-character-output

Arguments: stream character text-style width height baseline
Summary: Adds the character character to the output recording stream stream’s text output record in the text style text-style. width and height are the width and height of the character in device units. baseline is the new baseline for the stream. stream-add-character-output calls add-character-output-to-text-record.

stream-write-char on an output recording stream will call stream-add-character-output when stream-recording-p is t.

stream-add-string-output

Arguments: stream string start end text-style width height baseline
Summary: Adds the string string to the output recording stream stream’s text output record in the text style text-style. start and end are integers that specify the substring within string to add to the text output record. width and height are the width and height of the string in device units. baseline is the new baseline for the stream. stream-add-string-output calls add-string-output-to-text-record.

stream-write-string on an output recording stream will call stream-add-string-output when stream-recording-p is t.

14.2.4.4 Output Recording Utilities

CLIM provides several helper macros to control the output recording facility.
with-output-recording-options
[Macro]
Arguments: (stream &key record draw) &body body
Summary: Enables or disables output recording and/or drawing on the output recording stream designated by stream, within the extent of body.

The stream argument is not evaluated, and must be a symbol that is bound to an output recording stream. If stream is t, *standard-output* is used. body may have zero or more declarations as its first forms.

with-output-recording-options expands into a call to invoke-with-output-recording-options, supplying a function that executes body as the continuation argument to invoke-with-output-recording-options.

invoke-with-output-recording-options
[Generic Function]
Arguments: stream continuation record draw
Summary: Enables or disables output recording and/or drawing on the output recording stream stream, and calls the function continuation with the new output recording options in effect. continuation is a function of one argument, the stream; it has dynamic extent.

If draw is nil, output to the stream is not drawn on the viewport, but recording proceeds according to record; if draw is t, the output is drawn. If record is nil, output recording is disabled, but output otherwise proceeds according to draw; if draw is t, output recording is enabled.

All output recording streams must implement a method for invoke-with-output-recording-options.

with-new-output-record
[Macro]
Arguments: (stream &optional record-type record &rest init-args) &body body
Summary: Creates a new output record of type record-type (which defaults to standard-sequence-output-record), captures the output of body into the new output record, and inserts the new record into the current “open” output record associated with the output recording stream stream. While body is being evaluated, the current output record for stream will be bound to the new output record.

If record is supplied, it is the name of a variable that will be lexically bound to the new output record inside of body. init-args are CLOS initialization arguments that are passed to make-instance when the new output record is created.

with-new-output-record returns the output record it creates.
The `stream` argument is not evaluated, and must be a symbol that is bound to an output recording stream. If `stream` is `t`, `*standard-output*` is used. `body` may have zero or more declarations as its first forms.

`with-new-output-record` expands into a call to `invoke-with-new-output-record`, supplying a function that executes `body` as the `continuation` argument to `invoke-with-new-output-record`.

### with-output-to-output-record

[Macro]

**Arguments:** `(stream & optional record-type record &rest init-args) &body body`

**Summary:** Like `with-new-output-record`, except that the new output record is not inserted into the output record hierarchy, and the text cursor position of `stream` is initially bound to `(0, 0)`.

`record-type` is the type of output record to create, which defaults to `standard-sequence-output-record`. `init-args` are CLOS initialization arguments that are passed to `make-instance` when the new output record is created. `record`, if supplied, is a variable that will be bound to the new output record while `body` is evaluated.

`with-output-to-output-record` returns the output record it creates.

The `stream` argument is not evaluated, and must be a symbol that is bound to an output recording stream. If `stream` is `t`, `*standard-output*` is used. Unlike facilities such as `with-output-to-string`, `stream` must be an actual stream, but no output will be done to it. `body` may have zero or more declarations as its first forms.

`with-output-to-output-record` expands into a call to `invoke-with-output-to-output-record`, supplying a function that executes `body` as the `continuation` argument to `invoke-with-output-to-output-record`.

### invoke-with-new-output-record

[Generic Function]

**Arguments:** `stream continuation record-type &rest init-args &key`

**Summary:** Creates a new output record of type `record-type`. The function `continuation` is then called, and any output it does to the output recording stream `stream` is captured in the new output record. The new record is then inserted into the current “open” output record associated with `stream` (or the top-level output record if there is no currently “open” one). While `continuation` is being executed, the current output record for `stream` will be bound to the new output record.
continuation is a function of two arguments, the stream and the output record; it has dynamic extent. init-args are CLOS initialization arguments that are passed to make-instance when the new output record is created.

invoke-with-new-output-record returns the output record it creates.

All output recording streams must implement a method for invoke-with-new-output-record.

invoke-with-output-to-output-record

Arguments: stream continuation record-type &rest init-args &key

Summary: Like invoke-with-new-output-record except that the new output record is not inserted into the output record hierarchy, and the text cursor position of stream is initially bound to (0, 0). That is, when invoke-with-output-to-output-record is used, no drawing on the stream occurs and nothing is put into the stream’s normal output history. The function continuation is called, and any output it does to stream is captured in the output record.

continuation is a function of two arguments, the stream and the output record; it has dynamic extent. record-type is the type of output record to create. init-args are CLOS initialization arguments that are passed to make-instance when the new output record is created.

invoke-with-output-to-output-record returns the output record it creates.

All output recording streams must implement a method for invoke-with-output-to-output-record.

make-design-from-output-record

Arguments: record

Summary: Makes a design that replays the output record record when drawn via draw-design. If record is changed after the design is made, the consequences are unspecified. Applying a transformation to the design and calling draw-design on the new design is equivalent to establishing the same transformation before creating the output record.

The current version of CLIM supports this only for those output records that correspond to the geometric object classes (for example, the output records created by draw-line* and draw-ellipse*).
14.3 Conceptual Overview of Incremental Redisplay

Some kinds of applications can benefit greatly from the ability to redisplay information on a window only when that information has changed. This feature, called incremental redisplay, can significantly improve the speed at which your application updates information on the screen. Incremental redisplay is very useful for programs that display a window of changing information where some portions of the window are static and some are continually changing.

Incremental redisplay is a facility to allow you to change the output in an output history (and hence on the screen or other output device). It allows you to redisplay pieces of the existing output differently, under your control. “Incremental” here means that CLIM redisplay only the part of the output history visible in the viewport that has changed and thus needs to be redisplayed.

There are two different ways to do incremental redisplay:

- Call **redisplay** on an output record.
  
  This essentially tells the system to start that output record over from scratch. It compares the results with the existing output and tries to do minimal redisplay. The **updating-output** form allows you to assist the system by informing it that entire branches of the output history are known not to have changed. **updating-output** also allows you to communicate the fact that a piece of the output record hierarchy has moved.

- Update the output history manually, and then notify the output record that its child has changed.

  This causes CLIM to propagate the changes up the output record tree and allows parent output records to readjust themselves to account for the changes.

Each way is appropriate under different circumstances. **redisplay** is often easier to code and is more useful in cases where there might be large changes between two passes, or where you have little idea as to what the changes might be. Notifying the output record of changes can be more efficient for small changes at the bottom of the output-record hierarchy, or in cases where you are well informed as to the specific changes necessary and can describe these changes to the system.
14.4 CLIM Operators for Incremental Redisplay

The following functions are used to create an output record that should be incrementally redisplayed, and then to redisplay that record.

**updating-output**  
[Macro]  
Arguments: (stream &rest args &key unique-id (id-test #'eql) cache-value (cache-test #'eql) copy-cache-value fixed-position all-new parent-cache record-type) &body body  
Summary: Introduces a caching point for incremental redisplay.  

The stream argument is not evaluated, and must be a symbol that is bound to an output recording stream. If stream is t, *standard-output* is used. body may have zero or more declarations as its first forms.

*record-type* specifies the class of output record to create. The default is *standard-updating-output-record*. This argument should only be supplied by a programmer if there is a new class of output record that supports the updating output record protocol.

*updating-output* expands into a call to *invoke-updating-output*, supplying a function that executes body as the continuation argument to *invoke-updating-output*.

**invoke-updating-output**  
[Generic Function]  
Arguments: stream continuation record-type unique-id id-test cache-value cache-test copy-cache-value &key all-new parent-cache  
Summary: Introduces a caching point for incremental redisplay. Calls the function continuation, which generates the output records to be redisplayed. continuation is a function of one argument, the stream; it has dynamic extent.

If this is used outside the dynamic scope of an incremental redisplay, it has no particular effect. However, when incremental redisplay is occurring, the supplied cache-value is compared with the value stored in the cache identified by *unique-id*. If the values differ or the code in body has not been run before, the code in body runs, and cache-value is saved for next time. If the cache values are the same, the code in body is not run, because the current output is still valid.

*unique-id* uniquely identifies the output done by body. If *unique-id* is not supplied, CLIM will generate one that is guaranteed to be unique. *unique-id* may be any object as long as it is unique with respect to the *id-test* predicate among all
such unique ids in the current incremental redisplay. id-test is a function of two arguments that is used for comparing unique ids; it has indefinite extent.

cache-value is a value that remains constant if and only if the output produced by body does not need to be recomputed. If the cache value is not supplied, CLIM will not use a cache for this piece of output. cache-test is a function of two arguments that is used for comparing cache values; it has indefinite extent. If copy-cache-value is t, then the supplied cache value will be copied using copy-seq before it is stored in the output record. The default for copy-cache-value is nil.

If fixed-position is t, then the location of this output is fixed relative to its parent output record. When CLIM redisplay an output record that has a fixed position, then if the contents have not changed, the position of the output record will not change. If the contents have changed, CLIM assumes that the code will take care to preserve its position. The default for fixed-position is nil.

If all-new is t, that indicates that all of the output done by body is new, and will never match output previously recorded. In this case, CLIM will discard the old output and do the redisplay from scratch. The default for all-new is nil.

The output record tree created by updating-output defines a caching structure where mappings from a unique-id to an output record are maintained. If the programmer specifies an output record P via the parent-cache argument, then CLIM will try to find a corresponding output record with the matching unique-id in the cache belonging to P. If neither parent-cache is not provided, then CLIM looks for the unique-id in the output record created by immediate dynamically enclosing call to updating-output. If that fails, CLIM use the unique-id to find an output record that is a child of the output history of stream. Once CLIM has found an output record that matches the unique-id, it uses the cache value and cache test to determine whether the output record has changed. If the output record has not changed, it may have moved, in which case CLIM will simply move the display of the output record on the display device.

redisplay
[Function]
Arguments: record stream &key (check-overlapping t)
Summary: This function calls redisplay-output-record on the arguments record and stream. When check-overlapping is t, redisplay checks overlapped output records more carefully in order to display them correctly. The default is nil.
redisplay-output-record

Arguments: record stream &optional (check-overlapping t) x y parent-x parent-y

Summary: (redisplay-output-record record stream) causes the output of record to be recomputed. CLIM redisplays the changes “incrementally”; that is, it only displays those parts that have been changed. record must already be part of the output history of the output recording stream stream, although it can be anywhere inside the hierarchy.

When check-overlapping is t, redisplay checks overlapped output records more carefully in order to display them correctly. The default is nil. This means that CLIM can assume that no sibling output records overlap each other at any level. Supplying a false value for this argument can improve performance of redisplay.

The other optional arguments can be used to specify where on the stream the output record should be redisplayed. x and y represent where the cursor should be, relative to (output-record-parent record), before record is redisplayed. parent-x and parent-y can be supplied to say: do the output as if the parent started at positions parent-x and parent-y (which are in absolute coordinates). The default values for x and y are (output-record-start-position record). The default values for parent-x and parent-y are:

(convert-from-relative-to-absolute-coordinates stream (output-record-parent record))

record will usually be an output record created by updating-output. If it is not, then redisplay-output-record will be equivalent to replay-output-record.

14.5 Using updating-output

One technique of incremental redisplay is to use updating-output to inform CLIM what output has changed, and to use redisplay to recompute and redisplay that output.

The outermost call to updating-output identifies a program fragment that produces incrementally redisplayable output. A nested call to updating-output (that is, a call to updating-output that occurs during the execution of the body of the outermost updating-output and that specifies the same stream) identifies an individually redisplayable piece of output, the program fragment that produces that output, and the circumstances under which that output needs to be redrawn.
The outermost call to **updating-output** executes its body, producing the initial version of the output, and returns an **updating-output-record** that captures the body in a closure. Each nested call to **updating-output** caches its :unique-id and :cache-value arguments and the portion of the output produced by its body.

**redisplay** takes an **updating-output-record** and executes the captured body of **updating-output** over again. When a nested call to **updating-output** is executed during redisplay, **updating-output** decides whether the cached output can be reused or the output needs to be redrawn. This is controlled by the :cache-value argument to **updating-output**. If its value matches its previous value, the body would produce output identical to the previous output, which would thus be unnecessary. In this case, the cached output is reused and **updating-output** does not execute its body. If the :cache-value does not match, the output needs to be redrawn, so **updating-output** executes its body and the new output drawn on the stream replaces the previous output. The :cache-value argument is only meaningful for nested calls to **updating-output**.

If the :incremental-redisplay pane option is used, CLIM supplies the outermost call to **updating-output**, saves the **updating-output-record**, and calls **redisplay**. The function specified by the :display-function pane option performs only the nested calls to **updating-output**.

If you use incremental redisplay without using the :incremental-redisplay pane option, you must perform the outermost call to **updating-output**, save the **updating-output-record**, and call **redisplay** yourself.

In order to compare the cache to the output record, two pieces of information are necessary:

1. An association between the output being done by the program and a particular cache. This is supplied in the :unique-id option to **updating-output**.
2. A means of determining whether this particular cache is valid. This is the :cache-value option to **updating-output**.

Normally you would supply both options. The :unique-id would be some data structure associated with the corresponding part of output. The cache value would be something in that data structure that changes whenever the output changes.

It is valid to give the :unique-id and not the :cache-value. This is done to identify a superior in the hierarchy. By this means, the inferiors essentially get a more complex :unique-id when they are matched for output. (It is rather like using a telephone area code.) The cache without a cache value is never valid. Its inferiors always have to be checked.
It is also valid to give the `:cache-value` and not the `:unique-id`. In this case, unique ids are just assigned sequentially. So, if output associated with the same thing is done in the same order each time, it isn’t necessary to invent new unique ids for each piece. This is especially true in the case of inferiors of a cache with a unique id and no cache value of its own. In this case, the superior marks the particular data structure, whose components can change individually, and the inferiors are always in the same order and properly identified by their superior and the order in which they are output.

A `:unique-id` need not be unique across the entire redisplay, only among the inferiors of a given output cache, that is, among all possible (current and additional) uses you make of `updating-output` that are dynamically (not lexically) within another.

To make your incremental redisplay maximally efficient, you should attempt to give as many caches with `:cache-value` as possible. For instance, if you have a deeply nested tree, it is better to be able to know when whole branches have not changed than to have to recurse to every single leaf and check it. So, if you are maintaining a modification tick in the leaves, it is better to maintain one in their superiors as well and to propagate the modification up when things change. While the simpler approach works, it requires CLIM to do more work than is necessary.

### 14.6 Example of Incremental Redisplay in CLIM

The following function illustrates the standard use of incremental redisplay:

```lisp
(defun test (stream)
  (let* ((list (list 1 2 3 4 5))
    (record
     (clim:updating-output
      (stream)
      (do* ((elements list (cdr elements))
            (count 0 (1+ count))
            (element (first elements) (first elements)))
           ((null elements))
        (clim:updating-output (stream :unique-id count
                                      :cache-value element)
                               (format stream "Element ~D~%" element))))
    (force-output stream)
    (sleep 10)
    (setf (nth 2 list) 17)
    (clim:redisplay record stream)))
```

Output Recording and Redisplay
When *test* is run on a window, the initial display looks like:

Element 1
Element 2
Element 3
Element 4
Element 5

After the sleep has terminated, the display looks like:

Element 1
Element 2
Element 17
Element 4
Element 5

Incremental redisplay takes care of ensuring that only the third line gets erased and redisplayed. In the case where items have moved around, Incremental Redisplay ensures that the minimum amount of work is done in updating the display, thereby minimizing “flashiness” while providing a powerful user interface. For example, try substituting the following for the form after the sleep:

```lisp
(setf list (sort list #'(lambda (&rest args) (zerop (random 2))))))
```
Chapter 15  Extended Stream Input Facilities
15.1 Basic Input Streams

CLIM provides a stream-oriented input layer that is implemented on top of the sheet input architecture. The basic CLIM input stream protocol is based on the character input stream protocol proposal submitted to the ANSI Common Lisp committee by David Gray. This proposal was not approved by the committee, but CLIM provides an implementation of the basic input stream facilities.

standard-input-stream  
[Class]
Summary: This class provides an implementation of the CLIM’s basic input stream protocol based on CLIM’s input kernel. It defines a handle-event method for keystroke events and queues the resulting characters in a per-stream input buffer. Members of this class are mutable.

stream-read-char  
[Generic Function]
Arguments: stream
Summary: Returns the next character available in the input stream stream, or :eof if the stream is at end-of-file. If no character is available, this function will wait until one becomes available.

stream-unread-char  
[Generic Function]
Arguments: stream character
Summary: Places the character character back into the input stream stream’s input buffer. The next call to read-char on stream will return the unread character. The character supplied must be the most recent character read from the stream.

stream-read-char-no-hang  
[Generic Function]
Arguments: stream
Summary: Like stream-read-char, except that if no character is available, the function returns nil.

stream-peek-char  
[Generic Function]
Arguments: stream
Summary: Returns the next character available in the input stream stream. The character is not removed from the input buffer, so the same character will be returned by a subsequent call to stream-read-char.
Extended Stream Input Facilities

**stream-listen**

[Generic Function]

Arguments: *stream*

Summary: Returns `t` if there is input available on the input stream *stream*, `nil` if not.

**stream-read-line**

[Generic Function]

Arguments: *stream*

Summary: Reads and returns a string containing a line of text from the input stream *stream*, delimited by the `#\Newline` character.

**stream-clear-input**

[Generic Function]

Arguments: *stream*

Summary: Clears any buffered input associated with the input stream *stream* and returns `nil`.

### 15.2 Extended Input Streams

In addition to the basic input stream protocol, CLIM defines an extended input stream protocol. This protocol extends the stream model to allow manipulation of non-character user gestures, such as pointer button presses. The extended input protocol provides the programmer with more control over input processing, including the options of specifying input wait timeouts and auxiliary input test functions.

**extended-input-stream**

[Protocol Class]

Summary: The protocol class for CLIM extended input streams. This is a subclass of `input-stream`. If you want to create a new class that behaves like an extended input stream, it should be a subclass of `extended-input-stream`. Subclasses of `extended-input-stream` must obey the extended input stream protocol.

**extended-input-stream-p**

[Function]

Arguments: *object*

Summary: Returns `t` if *object* is a CLIM extended input stream; otherwise, it returns `nil`.
:input-buffer [Initarg]
:pointer [Initarg]
:text-cursor [Initarg]

Summary: All subclasses of extended-input-stream must handle these initargs, which are used to specify, respectively, the input buffer, pointer, and text cursor for the extended input stream.

standard-extended-input-stream [Class]

Summary: This class provides an implementation of the CLIM extended input stream protocol based on CLIM’s input kernel. The extended input stream maintains the state of the display’s pointing devices (such as a mouse) in pointer objects associated with the stream. It defines a handle-event methods for keystroke and pointer motion and button press events and updates the pointer object state and queues the resulting events in a per-stream input buffer.

Members of this class are mutable.

15.2.1 The Extended Input Stream Protocol

The following generic functions comprise the extended input stream protocol. All extended input streams must implement methods for these generic functions.

stream-input-buffer [Generic Function]
Arguments: stream

(setf stream-input-buffer) [Generic Function]
Arguments: buffer stream
Summary: These functions provide access to the stream’s input buffer. Normally programs do not need to manipulate the input buffer directly. It is sometimes useful to cause several streams to share the same input buffer so that input that comes in on one of them is available to an input call on any of the streams. The input buffer must be a vector with a fill pointer capable of holding general input gesture objects (such as characters and event objects).

stream-pointers [Generic Function]
Arguments: stream
Summary: Returns the list of pointer objects corresponding to the pointing devices of the port associated with stream. This function returns objects that reveal CLIM’s internal state; do not modify those objects.

stream-primary-pointer
Arguments: stream

(setf stream-primary-pointer)
Arguments: pointer stream
Summary: Returns (or sets) the pointer object corresponding to the primary pointing device of the console.

Note: CLIM currently supports only a single pointer for any port. Therefore, the length of the list returned by stream-pointers will always be one, and stream-primary-pointer will always return an object that is the only element of that list.

stream-pointer-position
Arguments: stream &key pointer
Summary: Returns the current position of the pointing device pointer for the extended input stream stream as two values, the x and y positions in the stream’s drawing surface coordinate system. If pointer is not supplied, it defaults to the stream-primary-pointer of the stream.

(setf* stream-pointer-position)
Arguments: x y stream &key pointer
Summary: Sets the position of the pointing device for the extended input stream stream to x and y, which are integers. pointer is as for stream-pointer-position. For the details of setf*, see Appendix C.4, “Multiple-Value Setf.”

stream-set-input-focus
Arguments: stream
Summary: Sets the “input focus” to the extended input stream stream and returns the old input focus as its value.

stream-restore-input-focus
Arguments: stream old-focus
Summary: Restores the “input focus” of the extended input stream stream to old-focus.
**with-input-focus**

[Macro]

*Arguments:* `(stream) &body body*

*Summary:* Temporarily gives the keyboard input focus to the extended input stream `stream`. By default, an application frame gives the input focus to the window associated with `frame-query-io`.

The `stream` argument is not evaluated, and must be a symbol that is bound to a stream. If `stream` is `t`, `*standard-input*` is used. `body` may have zero or more declarations as its first forms.

---

**:*input-wait-test* [Variable]**

---

**:*input-wait-handler* [Variable]**

---

**:*pointer-button-press-handler* [Variable]**

*Summary:* These three variables are used to hold the default values for the current input wait test, wait handler, and pointer button press handler. These variables are globally bound to `nil`.

---

**read-gesture**

[Function]


*Summary:* Calls `stream-read-gesture` on the extended input stream `stream` and all of the other keyword arguments. Returns the next gesture available in the extended input stream `stream`; the gesture will be a character, an event (such as a pointer button event), or `(values nil :timeout)` if no input is available. The input is not echoed.

These arguments are the same as for `stream-read-gesture`.

---

**stream-read-gesture**

[Generic Function]


*Summary:* Returns the next gesture available in the extended input stream `stream`; the gesture will be either a character or an event (such as a pointer button event). The input is not echoed.
If the user types an abort gesture (that is, a gesture that matches any of the gesture names in *abort-gestures*), then the abort-gesture condition will be signaled.

If the user types an accelerator gesture (that is, a gesture that matches any of the gesture names in *accelerator-gestures*), then the accelerator-gesture condition will be signaled.

stream-read-gesture works by invoking stream-input-wait on stream, input-wait-test, and timeout, and then processing the input, if there is any.

timeout is either nil or an integer that specifies the number of seconds that stream-read-gesture will wait for input to become available. If no input is available, stream-read-gesture will return two values, nil and :timeout.

If peek-p is t, the returned gesture will be left in the stream’s input buffer.

input-wait-test is a function of one argument, the stream. The function should return t when there is input to process, otherwise it should return nil. This argument will be passed on to stream-input-wait. stream-read-gesture will bind *input-wait-test* to input-wait-test.

input-wait-handler is a function of one argument, the stream. It is called when stream-input-wait returns nil (that is, no input is available). This option can be used in conjunction with input-wait-test to handle conditions other than keyboard gestures, or to provide some sort of interactive behavior (such as highlighting applicable presentations). stream-read-gesture will bind *input-wait-handler* to input-wait-handler.

pointer-button-press-handler is a function of two arguments, the stream and a pointer button press event. It is called when the user clicks a pointer button. stream-read-gesture will bind *pointer-button-press-handler* to pointer-button-press-handler.

input-wait-test, input-wait-handler, and pointer-button-press-handler have dynamic extent.

stream-input-wait

Arguments: stream &key timeout input-wait-test
Summary: Waits for input to become available on the extended input stream stream. timeout and input-wait-test are as for stream-read-gesture.

unread-gesture

Arguments: gesture &key (stream *standard-input*)
Summary: Calls `stream-unread-gesture` on `gesture` and `stream`. These arguments are the same as for `stream-unread-gesture`.

`stream-unread-gesture` [Generic Function]
Arguments: `stream` `gesture`
Summary: Places `gesture` back into the extended input stream `stream`'s input buffer. The next call to `stream-read-gesture` request will return the unread gesture. `gesture` must be the most recent gesture read from the stream via `read-gesture`.

15.2.2 Extended Input Stream Conditions

*abort-gestures* [Variable]
Summary: A list of all of the gesture names that correspond to abort gestures. The global set of standard abort gestures is unspecified; it includes the `:abort` gesture name. The actual keystroke sequence is `Control-z`.

abort-gesture [Condition]
Summary: This condition is signaled by `read-gesture` whenever an abort gesture (one of the gestures in `*abort-gestures*`) is read from the user.

abort-gesture-event [Generic Function]
Arguments: `condition`
Summary: Returns the event that cause the abort gesture condition to be signaled. `condition` is an object of type `abort-gesture`.

*accelerator-gestures* [Variable]
Summary: A list of all of the gesture names that correspond to keystroke accelerators. The global value for this is `nil`.

accelerator-gesture [Condition]
Summary: This condition is signaled by `read-gesture` whenever an keystroke accelerator gesture (one of the gestures in `*accelerator-gestures*`) is read from the user.

accelerator-gesture-event [Generic Function]
Arguments: `condition`
Summary: Returns the event that causes the accelerator gesture condition to be signaled. `condition` is an object of type `accelerator-gesture`.

`accelerator-gesture-numeric-argument` [Generic Function]
Arguments: `condition`
Summary: Returns the accumulated numeric argument (maintained by the input editor) at the time the accelerator gesture condition was signaled. `condition` is an object of type `accelerator-gesture`.

### 15.3 Gestures and Gesture Names

A *gesture* is some sort of input action by the user, such as typing a character or clicking a pointer button. A *keyboard gesture* refers to those gestures that are input by typing something on the keyboard. A *pointer gesture* refers to those gestures that are input by doing something with the pointer, such as clicking a button.

A *gesture name* is a symbol that gives a name to a set of similar gestures. Gesture names are used in order to provide a level of abstraction above raw device events; greater portability can be achieved by avoiding referring directly to platform-dependent constructs, such as character objects that refer to a particular key on the keyboard. For example, the `complete` gesture is used to name the gesture that causes the `complete-input` complete the current input string; on Genera, this may correspond to the COMPLETE key on the keyboard (which generates a `#\Complete` character), but on a Unix workstation, it may correspond to TAB or some other key. Another example is `select`, which is commonly used to indicate a left button click on the pointer.

Note that gesture names participate in a one-to-many mapping, that is, a single gesture name can name a group of physical gestures. For example, an `edit` might include both a pointer button click and a key press.

CLIM uses *event* objects to represent user gestures. Some of the more common events are those of the class `pointer-button-event`. Event objects store the sheet associated with the event, a timestamp, and the modifier key state (a quantity that indicates which modifier keys were held down on the keyboard at the time the event occurred). Pointer button event objects also store the pointer object, the button that was clicked on the pointer, the window the pointer was over, and the x and y position within that window. Keyboard gestures store the key name.
In some contexts, the object used to represent a user gesture is referred to as a gesture object. An gesture object might be exactly the same as an event object, or might contain less information. For example, for a keyboard gesture that corresponds to a standard printing character, it may be enough to represent the gesture object as a character.

**define-gesture-name**

*Macro*

**Arguments:** `name type gesture-spec &key (unique t)`

**Summary:** Defines a new gesture named by the symbol `name`. It expands into a call to `add-gesture-name`

`type` is the type of gesture being created, and is either `:keyboard` or `:pointer-button`. `gesture-spec` specifies the physical gesture that corresponds to the named gesture; its syntax depends on the value of `type`.

- **When** `type` is `:keyboard`, `gesture-spec` is a list of the form `(key-name . modifier-key-names)`. `key-name` is the name of a non-modifier key on the keyboard. `modifier-key-names` is a (possibly empty) list of modifier key names (`:shift`, `:control`, `:meta`, `:super`, and `:hyper`).

  For the standard Common Lisp characters (the 95 ASCII printing characters including `# \ Space`), `key-name` is the character object itself. For the other “semi-standard” characters, `key-name` is a keyword symbol naming the character (`:newline`, `:linefeed`, `:return`, `:tab`, `:backspace`, `:page`, and `:rubout`).

  The names of the modifier keys have been chosen to be uniform across all platforms, even though not all platforms will have keys on the keyboard with these names. The per-port part of CLIM simply chooses a sensible mapping from the modifier key names to the names of the keys on the keyboard. For example, CLIM on the Macintosh maps `:meta` to the COMMAND SHIFT key, and `:super` to the OPTION SHIFT key.

- **When** `type` is `:pointer-button`, `gesture-spec` is a list of the form `(button-name . modifier-key-names)`. `button` is the name of a pointer button (`:left`, `:middle`, or `:right`), and `modifier-key-names` is as for when `type` is `:keyboard`.

If `unique` is `t` (the default), all old gestures named by `name` are removed.

None of the arguments to `define-gesture-name` are evaluated.

**add-gesture-name**

*Function*

**Arguments:** `name type gesture-spec &key unique`

**Summary:** Adds a gesture named by the symbol `name` to the set of gesture names. `type` and `gesture-spec` are as for `define-gesture-name`. 
If `unique` is `t`, all old gestures named by `name` are removed. `unique` defaults to `nil`.

As an example, the `:edit` gesture name could be defined as follows using `define-gesture-name`:

```lisp
(define-gesture-name :edit :pointer-button (:left :meta))
(define-gesture-name :edit :keyboard (#\E :control))
```

**delete-gesture-name**  
[Function]  
*Arguments:* `name`  
*Summary:* Removes the gesture named by the symbol `name`.

CLIM provides a standard set of gesture names that correspond to a common set of gestures. Here are the required, standard keyboard gesture names:

- **:abort**—corresponds to gestures that cause the currently running application to be aborted back to top-level. In LispWorks CLIM, this may match the event corresponding to typing `CONTROL-Z`.
- **:clear-input**—corresponds to gestures that cause the current input buffer to be cleared. In LispWorks CLIM, this may match the event corresponding to typing `CONTROL-BACKSPACE`.
- **:complete**—corresponds to the gestures that tell the completion facility to complete the current input. On most systems, this will typically match the `\#\Tab` or `\#\Escape` character.
- **:help**—corresponds to the gestures that tell `accept` and the completion facility to display a help message. On most systems, this will typically match the event corresponding to typing `CONTROL-?`.
- **:possibilities**—corresponds to the gestures that tell the completion facility to display the current set of possible completions. On most systems, this will typically match the event corresponding to typing `CONTROL-?`.

Here are the required, standard pointer gesture names:

- **:select**—corresponds to the gesture that is used to “select” the object being pointed to with the pointer. Typically, this will correspond to the left button on the pointer.
- **:describe**—corresponds to the gesture that is used to “describe” or display some sort of documentation on the object being pointed to with the pointer. Typically, this will correspond to the middle button on the pointer.
- **:menu**—corresponds to the gesture that is used to display a menu of all possible operations on the object being pointed to with the pointer. Typically, this will correspond to the right button on the pointer.

- **:edit**—corresponds to the gesture that is used to “edit” the object being pointed to with the pointer. Typically, this will correspond to the left button on the pointer with some modifier key held down (such as the META key).

- **:delete**—corresponds to the gesture that is used to “delete” the object being pointed to with the pointer. Typically, this will correspond to the middle button on the pointer with some modifier key held down (such as the SHIFT key).

### 15.4 The Pointer Protocol

**pointer**

[Protocol Class]

Summary: The protocol class that corresponds to a pointing device. If you want to create a new class that behaves like pointer, it should be a subclass of **pointer**. Subclasses of **pointer** must obey the pointer protocol. Members of this class are mutable.

**pointerp**

[Function]

Arguments: **object**

Summary: Returns **t** if **object** is a pointer; otherwise, it returns **nil**.

**:port**

[Initarg]

Summary: Specifies the port with which the pointer is associated.

**standard-pointer**

[Class]

Summary: The instantiable class that implements a pointer.

**pointer-port**

[Generic Function]

Arguments: **pointer**

Summary: Returns the port with which the pointer **pointer** is associated.

**pointer-sheet**

[Generic Function]

Arguments: **pointer**
### (setf pointer-sheet) [Generic Function]

**Arguments:** sheet pointer  
**Summary:** Returns (or sets) the sheet over which the pointer pointer is located.

### pointer-button-state [Generic Function]

**Arguments:** pointer  
**Summary:** Returns the state of the buttons of the pointer pointer. This is represented as the logior of the values obtained from pointer-event-button.

### pointer-position [Generic Function]

**Arguments:** pointer  
**Summary:** Returns the x and y position of the pointer pointer as two values.

### (setf* pointer-position) [Generic Function]

**Arguments:** x y pointer  
**Summary:** Sets the x and y position of the pointer pointer to the specified position. For the details of setf*, see Appendix C.4, “Multiple-Value Setf.”

### pointer-cursor [Generic Function]

**Arguments:** pointer  

### (setf pointer-cursor) [Generic Function]

**Arguments:** cursor pointer  
**Summary:** A pointer object usually has a visible cursor associated with it. These functions return (or set) the cursor associated with the pointer pointer.

---

### 15.5 Pointer Tracking

#### tracking-pointer [Macro]

**Arguments:** (:optional stream &key pointer multiple-window transformp context-type highlight) &body body  
**Summary:** The tracking-pointer macro provides a general means for running code while following the position of a pointing device and monitoring for other input events. The programmer supplies code (the clauses in body) to be run upon the occurrence of any of the following types of events:
• Motion of the pointer
• Motion of the pointer over a presentation
• Clicking or releasing a pointer button
• Clicking or releasing a pointer button while the pointer is over a presentation
• Keyboard event (typing a character)

The stream argument is not evaluated, and must be a symbol that is bound to an input sheet or stream. If stream is t, *standard-output* is used. body may have zero or more declarations as its first forms.

The pointer argument specifies a pointer to track. It defaults to the primary pointer for the sheet, (port-pointer stream).

When multiple-window is t, the pointer will be tracked across multiple windows; when nil, it will be tracked only in the window corresponding to stream.

When the boolean transformp is t, then the coordinates supplied to the :pointer-motion clause will be in the “user” coordinate system rather than in stream coordinates; that is, the medium’s transformation will be applied to the coordinates.

context-type is used to specify the presentation type of presentations that will be “visible” to the tracking code for purposes of highlighting and for the :presentation, :presentation-button-press, and :presentation-button-release clauses. Supplying context-type is only useful when sheet is an output recording stream. context-type defaults to t, meaning that all presentations are visible.

When highlight is t, tracking-pointer will highlight applicable presentations as the pointer is positioned over them. highlight defaults to t when any of the :presentation, :presentation-button-press, or :presentation-button-release clauses is supplied; otherwise, it defaults to nil.

The body of tracking-pointer consists of a list of clauses. Each clause is of the form (clause-keyword arglist . clause-body) and defines a local function to be run upon occurrence of each type of event. The possible values for clause-keyword and the associated arglist are:

• :pointer-motion (&key window x y) Defines a clause to run whenever the pointer moves. In the clause, window is bound to the window in which the motion occurred, and x and y to the coordinates of the pointer. (See the keyword argument :transformp for a description of the coordinate system in which x and y are expressed.)
• :presentation ($key\ presentation\ window\ x\ y$) Defines a clause to run whenever the pointer moves over a presentation of the desired type. (See the keyword argument :context-type for a description of how to specify the desired type.) In the clause, presentation is bound to the presentation, window to the window in which the motion occurred, and $x$ and $y$ to the coordinates of the pointer. (See the keyword argument :transformp for a description of the coordinate system in which $x$ and $y$ are expressed.)

When both :presentation and :pointer-motion clauses are provided, the two are mutually exclusive. The :presentation clause will run only if the pointer is over an applicable presentation; otherwise the :pointer-motion clause will run.

• :pointer-button-press ($key\ event\ x\ y$) Defines a clause to run whenever a pointer button is pressed. In the clause, event is bound to the pointer button press event. (The window and the coordinates of the pointer are part of event.) $x$ and $y$ are the transformed $x$ and $y$ positions of the pointer. These will be different from pointer-event-$x$ and pointer-event-$y$ if the user transformation is not the identity transformation.

• :presentation-button-press ($key\ presentation\ event\ x\ y$) Defines a clause to run whenever the pointer button is pressed while the pointer is over a presentation of the desired type. (See the keyword argument :context-type for a description of how to specify the desired type.) In the clause, presentation is bound to the presentation, and event to the pointer button press event. (The window and the stream coordinates of the pointer are part of event.) $x$ and $y$ are as for the :pointer-button-press clause.

When both :presentation-button-press and :pointer-button-press clauses are provided, the two clauses are mutually exclusive. The :presentation-button-press clause will run only if the pointer is over an applicable presentation; otherwise, the :pointer-button-press clause will run.

• :pointer-button-release ($key\ event\ x\ y$) Defines a clause to run whenever a pointer button is released. In the clause, event is bound to the pointer button release event. (The window and the coordinates of the pointer are part of event.) $x$ and $y$ are the transformed $x$ and $y$ positions of the pointer. These will be different from pointer-event-$x$ and pointer-event-$y$ if the user transformation is not the identity transformation.

• :presentation-button-release ($key\ presentation\ event\ x\ y$) Defines a clause to run whenever a pointer button is released while the pointer is over a presentation of the desired type. (See the keyword argument :context-type for a de-
scription of how to specify the desired type.) In the clause, presentation is bound to the presentation, and event to the pointer button release event. (The window and the stream coordinates of the pointer are part of event.) x and y are as for the :pointer-button-release clause.

When both :presentation-button-release and :pointer-button-release clauses are provided, the two clauses are mutually exclusive. The :presentation-button-release clause will run only if the pointer is over an applicable presentation; otherwise, the :pointer-button-release clause will run.

- :keyboard (&key gesture) Defines a clause to run whenever a character is typed on the keyboard. In the clause, gesture is bound to the keyboard gesture corresponding to the character typed.

Here is an example of tracking-pointer:

```
(in-package `clim-user)

(define-application-frame test ()
  ()
  (:panes
   (main :application)))

(define-test-command (rubberband :menu t) ()
  (let ((x1 0);; x1, y1 represents the fix point
         (y1 0)
         (x2 0);; x2,y2 represents the point that is changing
         (y2 0)
         (mouse-button-press nil);; set to T when mouse button has
         ;; press to select pivot
         (stream (get-frame-pane *application-frame* 'main)))
```

<Bold>376CLIM User Guide
(tracking-pointer
(stream)
(:pointer-button-press
(event x y )
(setf x1 x
y1 y
x2 x
y2 y)
(draw-rectangle* stream x1 y1 x2 y2
   :ink +flipping-ink+ :filled nil)
(setf mouse-button-press t))
(:pointer-motion
(window x y)
(when mouse-button-press
;;;;erase
(draw-rectangle* stream x1 y1 x2 y2
   :ink +flipping-ink+ :filled nil)
;;;; draw
(draw-rectangle* stream x1 y1 x y
   :ink +flipping-ink+ :filled nil)
(setf x2 x y2 y))
(:pointer-button-release (event x y )
(when mouse-button-press
   (return (list x1 y1 x2 y2)))))

(define-test-command (com-exit :menu "EXEUNT" :keystroke #-) ()
(frame-exit *application-frame*))

drag-output-record [Generic Function]
Arguments: stream output-record &key repaint multiple-window erase feedback
   finish-on-release
Summary: Enters an interaction mode in which the user moves the pointer and output-record “follows” the pointer by being dragged on the output recording stream stream. By default, the dragging is accomplished by erasing the output record from its previous position and redrawing at the new position. output-record remains in the output history of stream at its final position.

The returned values are the final x and y positions of the pointer, and the delta-x and delta-y position of the mouse with respect to the origin of the object at the time it was originally selected by the pointer.

The boolean repaint controls the appearance of the windows as the pointer is dragged. If repaint is t (the default), displayed contents of windows are not disturbed as the output record is dragged over them (that is, those regions of the
screen are repainted). If it is `nil`, then no repainting is done as the output record is dragged.

`erase` identifies a function that will be called to erase the output record as it is dragged. It must be a function of two arguments, the output record to erase and the stream; it has dynamic extent. The default is `erase-output-record`.

`feedback` allows the programmer to identify a “feedback” function of seven arguments: the output record, the stream, the initial x and y position of the pointer, the current x and y position of the pointer, and a drawing argument (either `:erase` or `:draw`). It has dynamic extent. The default is `nil`, meaning that the feedback behavior will be for the output record to track the pointer. (The `feedback` argument is used when the programmer desires more complex feedback behavior, such as drawing a “rubber band” line as the user moves the mouse.) Note that if `feedback` is supplied, `erase` is ignored.

If the boolean `finish-on-release` is `nil` (the default), `drag-output-record` is exited when the user presses a pointer button. When it is `t`, `drag-output-record` is exited when the user releases the pointer button currently being held down.

**dragging-output**  
**[Macro]**

**Arguments:** 
(`&optional stream &key repaint multiple-window finish-on-release) &body body`

**Summary:**  
This macro is used by functions that want to move output records in an interactive fashion in a CLIM window. The body of the macro invocation contains code to draw a CLIM graphic. The resulting graphic tracks mouse motion in the window until the mouse button is pressed (or released, depending on the options).

`body` is evaluated inside of `with-output-to-output-record` to produce an output record for the stream `stream`, and then invokes `drag-output-record` on the record in order to drag the output. The output record is not inserted into `stream`’s output history.

The returned values are the final x and y positions of the pointer, and the delta-x and delta-y position of the mouse with respect to the origin of the object at the time it was originally selected by the pointer.

The `stream` argument is not evaluated, and must be a symbol that is bound to an output recording stream `stream`. If `stream` is `t` (the default), `*standard-output*` is used. `body` may have zero or more declarations as its first forms.

`repaint` and `finish-on-release` are as for `drag-output-record`. 

"378CLIM User Guide"
**pointer-place-rubber-band-line**

*Function*

Arguments: &key start-x start-y stream pointer multiple-window finish-on-release

Summary: This function is used to place a rubber-band line. The input is the end points of a rubber-band line on the stream `stream` (which defaults to `*standard-input*`) via the pointer `pointer`.

If `start-x` and `start-y` are provided, the start point of the line is at `(start-x,start-y)`. Otherwise, the start point of the line is selected by pressing a button on the pointer.

The `pointer` argument specifies a pointer from which to take input. It defaults to `(port-pointer stream)`.

When the boolean `multiple-window` argument is `t`, input can be taken from a window other than the default window. However, input cannot be taken from more than one window at the same time. For instance, you cannot press the pointer button in one window to begin the line and release it in another window to indicate the end point of the line; the press and release must happen in the same window.

When the boolean `finish-on-release` is `t`, `pointer-place-rubber-band-line` is exited when the user releases the pointer button currently being held down. When it is `nil`, `pointer-place-rubber-band-line` is exited when the user presses a pointer button.

`pointer-place-rubber-band-line` returns five values: the start X and Y of the line, the end X and Y of the line, and the window on which the line was drawn. The final value is useful only when `multiple-window` is `t`.

**pointer-input-rectangle**

*Function*

Arguments: &key left top right bottom stream pointer multiple-window finish-on-release

Summary: This function is used to input a rectangle via the pointer `pointer`. The input is the corners of a rectangle on the stream `stream`, which defaults to `*standard-input*`.

If `left` and `top` are provided, the upper left corner of the rectangle will be placed at `(left,top)`. If `right` and `bottom` are provided, the lower right corner of the rectangle will be placed at `(right,bottom)`. Otherwise, the upper left corner of the rectangle is selected by pressing a button on the pointer.

`pointer, multiple-window, and finish-on-release` are as for `pointer-place-rubber-band-line`.

`pointer-input-rectangle` returns five values: the left, top, right, and bottom corners of the rectangle, and the window on which the rectangle was drawn. The final value is useful only when `multiple-window` is true.

Extended Stream Input Facilities
**pointer-input-rectangle**  
[Function]

Arguments: &rest options &key rectangle stream pointer multiple-window  
             finish-on-release &allow-other-keys

*pointer-input-rectangle* is exactly like *pointer-input-rectangle* except that it takes as input and returns a rectangle object.
Chapter 16   Input Editing and Completion Facilities
16.1 Input Editing

An input editing stream “encapsulates” an interactive stream. That is, most operations are handled by the encapsulated interactive stream, but some operations are handled directly by the input editing stream itself. (See Appendix D, “Common Lisp Streams,” for a discussion of encapsulating streams.)

An input editing stream has the following components:

- The encapsulated interactive stream
- A buffer with a fill pointer, which we shall refer to as $FP$. The buffer contains all of the user’s input, and $FP$ is the length of that input.
- An insertion pointer, which we shall refer to as $IP$. The insertion pointer is the point in the buffer at which the “editing cursor” is.
- A scan pointer, which we shall refer to as $SP$. The scan pointer is the point in the buffer from which CLIM will get the next input gesture object (in the sense of read-gesture).
- A “rescan queued” flag, indicating that the programmer (or CLIM) requested that a “rescan” operation should take place before the next gesture is read from the user.
- A “rescan in progress” flag, indicating that CLIM is rescanning the user’s input, rather than reading freshly supplied gestures from the user.

The input editor reads either “real” gestures from the user (such as characters from the keyboard or pointer button events) or input editing commands, which can modify the state of the input buffer. When they do so, the input buffer must be “rescanned”; that is, the scan pointer $SP$ must be reset to its original state, and the contents of the input editor buffer must be reparsed before any other gestures from the user are read. While this rescanning operation is taking place, the “rescan in progress” flag is set to t. The relationship $SP \leq IP \leq FP$ always holds.

The overall control structure of the input editor is:

```
(catch 'rescan ;thrown to when a rescan is invoked
 (reset-scan-pointer stream) ;sets STREAM-RESCANNING-P to T
 (loop
   (funcall continuation stream))))
```

where `stream` is the input editing stream and `continuation` is the code supplied by the programmer, which typically contains calls to such functions as `accept` and `read-token` (which
will eventually call \texttt{stream-read-gesture}). When a rescan operation is invoked, it throws to the \texttt{rescan} tag in the previous example. The loop is terminated when an activation gesture is seen, and at that point the values produced by \textit{continuation} are returned as values from the input editor.

The important point is that functions such as \texttt{accept}, \texttt{read-gesture}, and \texttt{unread-gesture} read (or restore) the next gesture object from the buffer at the position pointed to by the scan pointer \texttt{SP}. However, insertion and input editing commands take place at the position pointed to by \texttt{IP}. The purpose of the rescanning operation is to ensure that all the input gestures issued by the user (typed characters, pointer button presses, and so forth) have been read by CLIM. During input editing, the input editor maintains some sort of visible cursor to remind the user of the position of \texttt{IP}.

The overall structure of \texttt{stream-read-gesture} on an input editing stream is:

\begin{verbatim}
(progn
  (rescan-if-necessary stream)
  (loop
    ;; If SP is less than FP
    ;;     Then get the next gesture from the input editor buffer at SP
    ;;     and increment SP
    ;; Else read the next gesture from the encapsulated stream
    ;;     and insert it into the buffer at IP
    ;; Set the "rescan in progress" flag to false
    ;; Call STREAM-PROCESS-GESTURE on the gesture
    ;;     If it was a "real" gesture
    ;;     Then exit with the gesture as the result
    ;; Else it was an input editing command (which has already been
    ;;     processed), so continue looping
  )
)
\end{verbatim}

A new gesture object is inserted into the input editor buffer at the insertion pointer \texttt{IP}. If \texttt{IP} = \texttt{FP}, this is accomplished by a \texttt{vector-push-extend}-like operation on the input buffer and \texttt{FP}, and then incrementing \texttt{IP}. If \texttt{IP} < \texttt{FP}, CLIM must first “make room” for the new gesture in the input buffer, then insert the gesture at \texttt{IP}, and finally increment both \texttt{IP} and \texttt{FP}.

When the user requests an input editor motion command, only the insertion pointer \texttt{IP} is affected. Motion commands do not need to request a rescan operation.

When the user requests an input editor deletion command, the sequence of gesture objects at \texttt{IP} is removed, and \texttt{IP} and \texttt{FP} must be modified to reflect the new state of the input buffer. Deletion commands (and other commands that modify the input buffer) must call \texttt{immediate-rescan} when they are done modifying the buffer.
CLIM is free to put special objects in the input editor buffer, such as “noise strings” and “accept results.” A “noise string” is used to represent some sort of in-line prompt and is never seen as input; the prompt-for-accept method may insert a noise string into the input buffer. An “accept result” is an object in the input buffer that is used to represent some object that was inserted into the input buffer (typically via a pointer gesture) that has no readable representation (in the Lisp sense); presentation-replace-input may create accept results. Noise strings are skipped over by input editing commands, and accept results are treated as a single gesture.


16.1.1 Operators for Input Editing

interactive-stream-p  [Generic Function]
Arguments: object
Summary: Returns t if object is an interactive stream, that is, a bidirectional stream intended for user interactions. Otherwise it returns nil. This is exactly the same function as in X3J13 Common Lisp, except that in CLIM it is a generic function.

The input editor is only fully implemented for interactive streams.

input-editing-stream  [Protocol Class]
Summary: The protocol class that corresponds to an input editing stream. If you want to create a new class that behaves like an input editing stream, it should be a subclass of input-editing-stream. Subclasses of input-editing-stream must obey the input editing stream protocol.

input-editing-stream-p  [Function]
Arguments: object
Summary: Returns t if object is an input editing stream (that is, a stream of the sort created by a call to with-input-editing), otherwise returns nil.

standard-input-editing-stream  [Class]
Summary: The class that implements CLIM’s standard input editor. This is the class of stream created by calling with-input-editing.

Members of this class are mutable.
with-input-editing
[Macro]
Arguments: (@optional stream &key input-sensitizer initial-contents) &body body
Summary: Establishes a context in which the user can edit the input typed in on the interactive stream stream. body is then executed in this context, and the values returned by body are returned as the values of with-input-editing. body may have zero or more declarations as its first forms.

The stream argument is not evaluated, and must be a symbol that is bound to an input stream. If stream is t (the default), *query-io* is used. If stream is a stream that is not an interactive stream, then with-input-editing is equivalent to prog. input-sensitizer, if supplied, is a function of two arguments, a stream and a continuation function; the function has dynamic extent. The continuation, supplied by CLIM, is responsible for displaying output corresponding to the user’s input on the stream. The input-sensitizer function will typically call with-output-as-presentation in order to make the output produced by the continuation sensitive.

If initial-contents is supplied, it must be either a string or a list of two elements, an object and a presentation type. If it is a string, it will be inserted into the input buffer using replace-input. If it is a list, the printed representation of the object will be inserted into the input buffer using presentation-replace-input.

with-input-editor-typeout
[Macro]
Arguments: (@optional stream) &body body
Summary: Establishes a context inside of with-input-editing in which output can be done by body to the input editing stream stream. with-input-editor-typeout should call fresh-line before and after evaluating the body. body may have zero or more declarations as its first forms.

The stream argument is not evaluated, and must be a symbol that is bound to a stream. If stream is t (the default), *query-io* is used. If stream is a stream that is not an input editing stream, then with-input-editor-typeout is equivalent to calling fresh-line, evaluating the body, and then calling fresh-line again.

16.1.2 Input Editor Commands

Keyboard input to accept can be edited until an activation character is typed to terminate it. If the input cannot be parsed after an activation character is entered, it must be edited and re-activated. The input editor has several keystroke commands, as listed in Table 4, “Input
Editor Keystroke Commands”. Prefix numeric arguments to input editor commands can be entered using digits and the minus sign (-) with CONTROL and META (as in Emacs).

The function **add-input-editor-command** can be used to bind one or more keys to an input editor command. Any character can be an input editor command, but by convention only non-graphic characters should be used.

<table>
<thead>
<tr>
<th>Command</th>
<th>Character</th>
<th>Command</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward character</td>
<td>C-f</td>
<td>Delete previous character</td>
<td>Rubout</td>
</tr>
<tr>
<td>Forward word</td>
<td>M-f</td>
<td>Delete previous word</td>
<td>M-Rubout</td>
</tr>
<tr>
<td>Backward character</td>
<td>C-b</td>
<td>Kill to end of line</td>
<td>C-k</td>
</tr>
<tr>
<td>Backward word</td>
<td>M-b</td>
<td>Clear input buffer</td>
<td>Liquid: C-M-delete</td>
</tr>
<tr>
<td>Beginning of line</td>
<td>C-a</td>
<td>Insert new line</td>
<td>C-o</td>
</tr>
<tr>
<td>End of line</td>
<td>C-e</td>
<td>Transpose adjacent characters</td>
<td>C-t</td>
</tr>
<tr>
<td>Next line</td>
<td>C-n</td>
<td>Transpose adjacent words</td>
<td>M-t</td>
</tr>
<tr>
<td>Previous line</td>
<td>C-p</td>
<td>Yank from ring</td>
<td>C-y</td>
</tr>
<tr>
<td>Beginning of buffer</td>
<td>M-&lt;</td>
<td>Yank from presentation history</td>
<td>C-M-y</td>
</tr>
<tr>
<td>End of buffer</td>
<td>M-&gt;</td>
<td>Yank next item</td>
<td>M-y</td>
</tr>
<tr>
<td>Delete next character</td>
<td>C-d</td>
<td>Scroll output history forward</td>
<td>C-v</td>
</tr>
<tr>
<td>Delete next word</td>
<td>M-d</td>
<td>Scroll output history backward</td>
<td>M-v</td>
</tr>
</tbody>
</table>

**Table 4. Input Editor Keystroke Commands**

The input also supports “numeric arguments” (such as C-0, C-1, M-0, etc.) that modify the behavior of the input editing commands. For instance, the motion and deletion commands will be repeated as many times as specified by the numeric argument. Furthermore, the accumulated numeric argument will be passed to the command processor in such a way...
that substitute-numerical-marker can be used to insert the numeric argument into a command that was read via a keystroke accelerator.
16.2 Activation and Delimiter Gestures

Activation gestures terminate an input “sentence,” such as a command or anything else being read by accept. When an activation gesture is entered by the user, CLIM will cease reading input and “execute” the input that has been entered.

Delimiter gestures terminate an input “word,” such as a recursive call to accept.

*activation-gestures* [Variable]
Summary: The set of currently active activation gestures. The global value of this is nil. The exact format of *activation-gestures* is unspecified. *activation-gestures* and the elements in it may have dynamic extent.

*standard-activation-gestures* [Variable]
Summary: The default set of activation gestures. The exact set of standard activation is unspecified; it includes the gesture corresponding to the #\Newline character.

with-activation-gestures [Macro]
Arguments: (gestures &key override) &body body
Summary: Specifies a list of gestures that terminate input during the execution of body. body may have zero or more declarations as its first forms. gestures must be either a single gesture name or a form that evaluates to a list of gesture names.

If the boolean override is t, then gestures will override the current activation gestures. If it is nil (the default), then gestures will be added to the existing set of activation gestures. with-activation-gestures must bind *activation-gestures* to the new set of activation gestures.

See also the :activation-gestures and :additional-activation-gestures options to accept.

activation-gesture-p [Function]
Arguments: gesture
Summary: Returns t if the gesture object gesture is an activation gesture; otherwise, it returns nil.
*delimiter-gestures*  
[Variable]
Summary: The set of currently active delimiter gestures. The global value of this is nil. The exact format of *delimiter-gestures* is unspecified. *delimiter-gestures* and the elements in it may have dynamic extent.

with-delimiter-gestures  
[Macro]
Arguments: (gestures &key override) &body body
Summary: Specifies a list of gestures that terminate an individual token, but not the entire input, during the execution of body. body may have zero or more declarations as its first forms. gestures must be either a single gesture name or a form that evaluates to a list of gesture names.

If the boolean override is t, then gestures will override the current delimiter gestures. If it is nil (the default), then gestures will be added to the existing set of delimiter gestures. with-delimiter-gestures must bind *delimiter-gestures* to the new set of delimiter gestures.

See also the :delimiter-gestures and :additional-delimiter-gestures options to accept.

delimiter-gesture-p  
[Function]
Arguments: gesture
Summary: Returns t if the gesture object gesture is a delimiter gesture; otherwise, it returns nil.
16.3  Signalling Errors Inside accept Methods

Sometimes an accept method may wish to signal an error while it is parsing the user’s input, or a nested call to accept may signal such an error itself. The following functions and conditions may be used:

**parse-error**  
[Condition]  
Summary: The error that is signaled by parse-error. This is a subclass of error.

**parse-error**  
[Function]  
Arguments: format-string &rest format-arguments  
Summary: Reports an error while parsing an input token. Does not return. format-string and format-arguments are as for the Common Lisp function format.

**simple-parse-error**  
[Condition]  
Summary: The error that is signaled by simple-parse-error. This is a subclass of parse-error.

**simple-parse-error**  
[Function]  
Arguments: format-string &rest format-arguments  
Summary: Signals a simple-parse-error when CLIM does not know how to parse some sort of user input while inside accept. Does not return. format-string and format-arguments are as for the Common Lisp function format.

**input-not-of-required-type**  
[Condition]  
Arguments: object type  
Summary: This condition is signalled by input-not-of-required-type. This is a subclass of parse-error.

**input-not-of-required-type**  
[Function]  
Arguments: object type  
Summary: Reports that input does not satisfy the specified type by signalling an input-not-of-required-type error. object is a parsed object or an unparsed token (a string). type is a presentation type specifier. Does not return.
16.4 Reading and Writing Tokens

Sometimes after an accept method has read some input from the user, it may be necessary to insert a modified version of that input back into the input buffer. The following two functions can be used to modify the input buffer:

**replace-input**

**Arguments:** stream new-input &key start end buffer-start rescan

**Summary:** Replaces the part of the input editing stream stream’s input buffer that extends from buffer-start to its scan pointer with the string new-input. buffer-start defaults to the current input position of stream. start and end can be supplied to specify a subsequence of new-input; start defaults to 0 and end defaults to the length of new-input.

replace-input queues a rescan by calling queue-rescan if the new input does not match the old output, or if rescan is t.

The returned value is the position in the input buffer.

**presentation-replace-input**

**Arguments:** stream object type view &key buffer-start rescan query-identifier for-context-type

**Summary:** Like replace-input, except that the new input to insert into the input buffer is obtained by presenting the object object with the presentation type type and view view. buffer-start and rescan are as for replace-input, query-identifier is as for accept, and for-context-type is as for present.

If the object does not have a readable representation (in the Lisp sense), presentation-replace-input may create an “accept result” to represent the object and insert it into the input buffer. For the purposes of input editing, “accept results” must be treated as a single input gesture.

The following two functions are used to read or write a token (that is, a string):

**read-token**

**Arguments:** stream &key input-wait-handler pointer-button-press-handler click-only

**Summary:** Reads characters from the interactive stream stream until it encounters a delimiter, activation, or pointer gesture. Returns the accumulated string that was delimited by the delimiter or activation gesture, leaving the delimiter unread.
If the first character of typed input is a quotation mark ('"'), then \texttt{read-token}
will ignore delimiter gestures until another quotation mark is seen. When the
closing quotation mark is seen, \texttt{read-token} will proceed as discussed previously.

If the boolean \textit{click-only} is \texttt{t}, then no keyboard input is allowed. In that case,
\texttt{read-token} will simply ignore any typed characters.

\textit{input-wait-handler} and \textit{pointer-button-press-handler} are as for
\texttt{stream-read-gesture}. Refer to Section 15.2.1, “The Extended Input Stream Pro-
tocol,” for details.

\textbf{write-token} \hfill [Function]

\textbf{Arguments: } \texttt{token stream \&key acceptably}

\textbf{Summary:} \texttt{write-token} is the opposite of \texttt{read-token}; given the string \texttt{token}, it writes it to
the interactive stream \texttt{stream}. If \texttt{acceptably} is \texttt{t} and there are any characters in
the token that are delimiter gestures (see \texttt{with-delimiter-gestures}), then
\texttt{write-token} will surround the token with quotation marks ('"').

Typically, present methods will use \texttt{write-token} instead of \texttt{write-string}. 

\textbf{392CLIM User Guide}
16.5 Completion

CLIM provides a completion facility that completes a string provided by a user against some set of possible completions (which are themselves strings). Each completion is associated with some Lisp object. CLIM provides "chunkwise" completion; that is, if the user input consists of several tokens separated by "partial delimiters," CLIM completes each token separately against the set of possibilities.

*completion-gestures*  [Variable]
Summary: A list of the gesture names that cause complete-input to complete the user’s input as fully as possible. The exact global contents of this list is unspecified; it includes the :complete gesture name. *completion-gestures* is bound to \\Control-Tab.

*help-gestures*  [Variable]
Summary: A list of the gesture names that cause accept and complete-input to display a (possibly input context-sensitive) help message, and for some presentation types a list of possibilities as well. The exact global contents of this list is unspecified; it includes the :help gesture name. *help-gestures* is bound to \\Control-l in LispWorks CLIM and \\Meta-? in Liquid CLIM.

*possibilities-gestures*  [Variable]
Summary: A list of the gesture names that cause complete-input to display a (possibly input context-sensitive) help message and a list of possibilities. The exact global contents of this list is unspecified; it includes the :possibilities gesture name. *possibilities-gestures* is bound to \\Control-?.

complete-input  [Function]
Arguments: stream function &key partial-completers allow-any-input possibility-printer (help-displays-possibilities t)
Summary: Reads input from the user from the input editing stream stream, completing over a set of possibilities. complete-input only works on input editing streams. function is a function of two arguments. It is called to generate the completion possibilities that match the user’s input; it has dynamic extent. Usually, programmers will pass a function which calls either complete-from-possibilities or complete-from-generator as the value of function. Its first argument is a string

Input Editing and Completion Facilities
containing the user’s input “so far.” Its second argument is the completion mode, one of the following:

- **:complete-limited**—the function completes the input up to the next partial delimiter. This is the mode used when the user types a partial completer.

- **:complete-maximal**—the function completes the input as much as possible. This is the mode used when the user issues a gesture that matches any of the gesture names in `*completion-gestures*`.

- **:complete**—the function completes the input as much as possible, except that if the user’s input exactly matches one of the possibilities, even if it is a left substring of another possibility, the shorter possibility is returned as the result. This is the mode used when the user issues a delimiter or activation gesture that is not a partial completer.

- **:possibilities**—the function returns an alist of the possible completions as its fifth value. This is the mode used when the user a gesture that matches any of the gesture names in `*possibilities-gestures*` or `*help-gestures*` (if `help-displays-possibilities` is `t`).

`complete-input` returns five values:

- **string**—the completed input string
- **success**—`t` if completion was successful, otherwise `nil`
- **object**—the object corresponding to the completion, otherwise `nil`
- **nmatches**—the number of possible completions of the input
- **possibilities**—a newly-created alist of completions (lists of a string and an object), returned only when the completion mode is :possibilities.

`complete-input` returns three values: `object`, `success`, and `string`. In addition, the printed representation of the completed input will be inserted into the input buffer of `stream` in place of the user-supplied string by calling `replace-input`.

`partial-completers` is a list of characters that delimit portions of a name that can be completed separately. The default is an empty list.

If the boolean `allow-any-input` is `t`, then `complete-input` returns as soon as the user issues an activation gesture, even if the input is not any of the possibilities. If the input is not one of the possibilities, the three values returned by `complete-input` will be `nil`, `t`, and the string. The default for `allow-any-input` is `nil`.

<Bold>394CLIM User Guide
If `possibility-printer` is supplied, it must be a function of three arguments, a possibility, a presentation type, and a stream; it has dynamic extent and displays the possibility on the stream. The possibility will be a list of two elements, the first being a string and the second being the object corresponding to the string.

If `help-display-possibilities` is `t` (the default), then when the user issues a help gesture (a gesture that matches one of the gesture names in `*help-gestures*`), CLIM will display all the matching possibilities. If it is `nil`, then CLIM will not display the possibilities unless the user issues a possibility gesture (a gesture that matches one of the gesture names in `*possibilities-gestures*`).

Here is an example:

```lisp
(defvar *my-possibilities* '(
  ("Raspberry" :rasp)
  ("Strawberry" :straw)
  ("Blueberry" :blue)))

(flet ((possibilities-generator (string-so-far mode)
    (complete-from-possibilities string-so-far
      *my-possibilities* nil :action mode)))
  (complete-input stream #'possibilities-generator))
```

**complete-from-generator**

[Function]

Arguments: `string generator delimiters` &key (action :complete) predicate

Summary: Given an input string `string` and a list of delimiter characters `delimiters` that act as partial completion characters, `complete-from-generator` completes against possibilities that are generated by the function `generator`. `generator` is a function of two arguments, the string `string` and another function that it calls in order to process the possibility; it has dynamic extent.

`action` will be one of :complete, :complete-maximal, :complete-limited, or :possibilities. These are described under the function `complete-input`.

`predicate` is a function of one argument, an object. If the predicate returns `t`, the possibility corresponding to the object is processed. It has dynamic extent.

`complete-from-generator` returns five values, the completed input string, the success value (if the completion was successful, otherwise `nil`), the object matching the completion (or `nil` if unsuccessful), the number of matches, and a list of possible completions if `action` was :possibilities.

A caller of this function will typically be passed as the second argument to `complete-input`. 
**complete-from-possibilities**  
**[Function]**

**Arguments:** string completions delimiters &key (action :complete) predicate name-key value-key

**Summary:** Given an input string `string` and a list of delimiter characters `delimiters` that act as partial completion characters, `complete-from-possibilities` completes against the possibilities in the sequence `completions`. The completion string is extracted from the possibilities by applying `name-key`, which is a function of one argument. The object is extracted by applying `value-key`, which is a function of one argument. `name-key` defaults to `first`, and `value-key` defaults to `second`.

`action` will be one of :complete, :complete-maximal, :complete-limited, or :possibilities. These are described under the function `complete-input`.

`predicate` must be a function of one argument, an object. If the predicate returns `t`, the possibility corresponding to the object is processed, otherwise it is not.

`predicate`, `name-key`, and `value-key` have dynamic extent.

`complete-from-possibilities` returns five values, the completed input string, the success value (`t` if the completion was successful, `nil` otherwise), the object matching the completion (or `nil` if unsuccessful), the number of matches, and a list of possible completions if `action` was :possibilities.

A caller of this function will typically be passed as the second argument to `complete-input`.

**completing-from-suggestions**  
**[Macro]**

**Arguments:** (stream &key partial-completers allow-any-input possibility-printer) &body body

**Summary:** Reads input from input editing stream `stream`, completing over a set of possibilities generated by calls to `suggest` in `body`. Returns `object`, `success`, and `string`.

The `stream` argument is not evaluated, and must be a symbol that is bound to a stream. If `stream` is `t` (the default), `*query-io*` is used.

See `complete-input` for `partial-completers`, `allow-any-input`, and `possibility-printer`.

For example:
(completing-from-suggestions (stream)
  (map nil
    #'(lambda (x)
      (suggest
        (car x) (cdr x)))
    '(("One" . 1)
      ("Two" . 2)
      ("Three" . 3))))

suggest [Function]
Arguments: completion object
Summary: Specifies one possibility for completing-from-suggestions. completion is a string, the printed representation. object is the internal representation.

This function has lexical scope and is defined only within the body of completing-from-suggestions.

accept generates help messages based on the name of the presentation type, but sometimes this is not enough. Use with-accept-help to create more complex help messages.

with-accept-help [Macro]
Arguments: options &body body
Summary: Binds the dynamic environment to control the documentation produced by help and possibilities gestures during user input in calls to accept with the dynamic scope of body. body may have zero or more declarations as its first forms.

options is a list of option specifications. Each specification is itself a list of the form (help-option help-string). help-option is either a symbol that is a help-type or a list of the form (help-type mode-flag). help-type must be one of:

• :top-level-help—specifies that help-string be used instead of the default help documentation provided by accept.

• :subhelp—specifies that help-string be used in addition to the default help documentation provided by accept.

mode-flag must be one of:

• :append—specifies that the current help string be appended to any previous help strings of the same help type. This is the default mode.

• :override—specifies that the current help string is the help for this help type; no lower-level calls to with-accept-help can override this. (:override works from the outside in.)
The `:establish-unless-overridden` specifies that the current help string be the help for this help type unless a higher-level call to `with-accept-help` has already established a help string for this help type in the `:override` mode. This is what `accept` uses to establish the default help.

`help-string` is a string or a function that returns a string. If it is a function, it receives three arguments, the stream, an action (either `:help` or `:possibilities`) and the help string generated so far.

None of the arguments are evaluated.
16.6 Using with-accept-help: some examples

(clim:with-accept-help
 ((:subhelp "This is a test."))
 (clim:accept 'pathname))

[ACCEPT does this] ==> You are being asked to enter a pathname.
[done via :SUBHELP] This is a test.

(clim:with-accept-help ((:top-level-help "This is a test."))
 (clim:accept 'pathname))

[done via :TOP-LEVEL-HELP] ==> This is a test.

(clim:with-accept-help (((:subhelp :override) "This is a test."))
 (clim:accept 'pathname))

[ACCEPT does this] ==> You are being asked to enter a pathname.
[done via :SUBHELP] This is a test.

(clim:define-presentation-type test ())

(clim:define-presentation-method clim:accept
 (type test) stream view &key)
 (values (clim:with-accept-help
 ((:subhelp "A test is made up of three things:")
 (clim:completing-from-suggestions (...) ...)))))

(clim:accept 'test)
==> You are being asked to enter a test.
   A test is made up of three things:
16.7 Advanced Topics

The material in this section is advanced; most CLIM programmers can skip to the next chapter. This section discusses the Input Editing Stream Protocol.

Input editing streams obey both the extended input and extended output stream protocols, and must support the generic functions that comprise those protocols. For the most part, this simply entails “trampolining” those operations to the encapsulated interactive stream. However, such generic functions as stream-read-gesture and stream-unread-gesture will need methods that observe the use of the input editor’s scan pointer.

Input editing streams implement methods for prompt-for-accept (in order to provide in-line prompting that interacts correctly with input editing) and stream-accept (in order to cause accept to obey the scan pointer).

The following generic functions comprise the remainder of the input editing protocol, and must be implemented for all classes that inherit from input-editing-stream.

**stream-input-buffer**

[Generic Function]

Arguments: (stream input-editing-stream)

Summary: Returns the input buffer (that is, the string being edited) associated with the input editing stream *stream*. This must be an unspecialized vector with a fill pointer. The fill pointer of the vector points past the last gesture object in the buffer. This buffer is affected during input editing. The effects of modifying the input buffer other than by the specified API (such as replace-input) are unspecified.

**stream-insertion-pointer**

[Generic Function]

Arguments: *stream*

Summary: Returns an integer corresponding to the current input position in the input editing stream *stream*’s buffer, that is, the point in the buffer at which the next user input gesture will be inserted. The insertion pointer will always be less than or equal to (fill-pointer (stream-input-buffer *stream*)). The insertion pointer can also be thought of as an editing cursor.

**setf stream-insertion-pointer**

[Generic Function]

Arguments: *pointer* *stream*
Summary: Changes the input position of the input editing stream `stream` to `pointer`, an integer less than or equal to `(fill-pointer (stream-input-buffer stream))`.

**stream-scan-pointer**

**[Generic Function]**

**Arguments:** `stream`

**Summary:** Returns an integer corresponding to the current scan pointer in the input editing stream `stream`’s buffer, that is, the point in the buffer at which calls to `accept` have stopped parsing input. The scan pointer will always be less than or equal to `(stream-insertion-pointer stream)`.

**(setf stream-scan-pointer)**

**[Generic Function]**

**Arguments:** `pointer stream`

**Summary:** Changes the scan pointer of the input editing stream `stream` to `pointer`, an integer less than or equal to `(stream-insertion-pointer stream)`.

**stream-rescanning-p**

**[Generic Function]**

**Arguments:** `stream`

**Summary:** Returns the state of the input editing stream `stream`’s “rescan in progress” flag, which is `t` if `stream` is performing a rescan operation, but otherwise `nil`. All extended input streams must implement a method for this, but non-input editing streams will always returns `nil`.

**reset-scan-pointer**

**[Generic Function]**

**Arguments:** `stream` &optional `(scan-pointer 0)`

**Summary:** Sets the input editing stream `stream`’s scan pointer to `scan-pointer`, and sets the state of `stream-rescanning-p` to `t`.

**immediate-rescan**

**[Generic Function]**

**Arguments:** `stream`

**Summary:** Invokes a rescan operation immediately by “throwing” out to the most recent invocation of `with-input-editing`.

**queue-rescan**

**[Generic Function]**

**Arguments:** `stream`

**Summary:** Sets the “rescan queued” flag to `t`, meaning that the input editing stream `stream` should be rescanned after the next non-input editing gesture is read.
rescan-if-necessary  [Generic Function]
Arguments: stream
Summary: Invokes a rescan operation on the input editing stream stream if queue-rescan was called on the same stream and no intervening rescan operation has taken place. Resets the state of the “rescan queued” flag to nil.

erase-input-buffer  [Generic Function]
Arguments: stream &optional (start-position 0)
Summary: Erases the part of the display that corresponds to the input editor’s buffer, starting at the position start-position.

redraw-input-buffer  [Generic Function]
Arguments: stream &optional (start-position 0)
Summary: Displays the input editor’s buffer starting at the position start-position on the interactive stream that is encapsulated by the input editing stream stream.

stream-process-gesture  [Generic Function]
Arguments: stream gesture type
Summary: If gesture is an input editing command, stream-process-gesture performs the input editing operation on the input editing stream stream and returns nil. Otherwise, it returns the two values gesture and type.

stream-read-gesture  [Generic Function]
Arguments: (stream standard-input-editing-stream) &allow-other-keys
Summary: Reads and returns a gesture from the user on the input editing stream stream. The stream-read-gesture method calls stream-process-gesture, which will either return a “real” gesture (such as a typed character, a pointer gesture, or a timeout) or nil (indicating that some sort of input editing operation was performed). stream-read-gesture only returns when a real gesture has been read; if an input editing operation was performed, stream-read-gesture will loop until a “real” gesture is typed by the user.

stream-unread-gesture  [Generic Function]
Arguments: (stream standard-input-editing-stream) gesture
Summary: Inserts the gesture gesture back into the input editor’s buffer, maintaining the scan pointer.

<Bold>402CLIM User Guide
Chapter 17  Formatted Output
17.1 Formatting Tables in CLIM

17.1.1 Conceptual Overview of Formatting Tables

CLIM makes it easy to construct tabular output. The usual way of making tables is by indicating what you want to put in the table and letting CLIM choose the placement of the row and column cells. CLIM also allows you to specify constraints on the placement of the table elements with some flexibility.

In the CLIM model of formatting tables, each cell of the table is handled separately:

- The code for a cell draws to a stream that has a “private” (local to that cell) drawing plane. The code puts ink on the drawing plane, in the form of text, graphics, or both.
- After output for a cell has finished, the bounding rectangle of all output on the “private” drawing plane is found. The region within that bounding rectangle forms the contents of a cell.
- Additional rectangular regions, containing only background ink, are attached to the edges of the cell contents. These regions ensure that the cells satisfy the tabular constraints that within a row all cells have the same height, and within a column all cells have the same width. CLIM may also introduce additional background for other purposes.
- The cells are assembled into rows and columns.

You are responsible only for specifying the contents of the cell. CLIM’s table formatter will figure out how to lay out the table so that all the cells fit together properly. It derives the width of each column from the the widest cell within the column, and the height of each row from the the tallest cell within the row.

All the cells in a row have the same height. All the cells in a column have the same width. The contents of the cells can be of irregular shapes and sizes. You can impose both vertical and horizontal constraints on the objects within the cell, aligning them vertically at the top, bottom, or center of the cell, and horizontally at the left, right, or center of the cell.

Some tables are “multiple column” tables, in which two or more rows of the table are placed side by side (usually with intervening spacing) rather than all rows being aligned vertically. Multiple column tables are generally used to produce a table that is more aesthetically pleasing, or to make more efficient use of space on the output device. When a table is a multiple column table, one additional step takes place in the formatting of the table: the
rows of the table are rearranged into multiple columns in which some rows are placed side by side.

The programmer can give CLIM the following advice about assembling the table:

- How to place the contents of the cell within the cell (such as centered vertically, flush-left, and so forth). The possibilities for this advice are described later.
- Optionally, how much additional space to insert between columns and between rows of the table.
- Optionally, whether to make all columns the same size.

You can specify other constraints that affect the appearance of the table, such as the width or length of the table.

Note that table formatting is inherently two-dimensional from the point of view of the application. Item list formatting is inherently one-dimensional output that is presented two-dimensionally. The canonical example is a menu, where the programmer specifies a list of items to be presented. If the list is small enough, a single column or row of menu entries suffices. In this case, formatting is done when viewport requirements make it desirable.

These constraints affect the appearance of item lists:

- The number of rows (that is, allowing CLIM to choose the number of columns)
- The number of columns (that is, allowing CLIM to choose the number of rows)
- The maximum height (or width) of the column (that is, letting CLIM determine the number of rows and columns that satisfy that constraint)

See Section 17.5, “Advanced Topics,” for the table and item list formatting protocols.

### 17.1.2 CLIM Operators for Formatting Tables

This subsection covers the general-purpose table formatting operators.

**formatting-table**

[Macro]

Arguments: (&optional stream &key x-spacing y-spacing multiple-columns multiple-columns-x-spacing equalize-column-widths (move-cursor t) record-type) &body body

Formatted Output
**Summary:** Binds the local environment in such a way the output of `body` will be done in a tabular format. This must be used in conjunction with `formatting-row` or `formatting-column`, and `formatting-cell`. The table is placed so that its upper left corner is at the current text cursor position of `stream`. If the boolean `move-cursor` is `t` (the default), then the text cursor will be moved so that it immediately follows the last cell of the table.

The returned value is the output record corresponding to the table.

`stream` is an output recording stream to which output will be done. The `stream` argument is not evaluated, and must be a symbol that is bound to a stream. If `stream` is `t` (the default), `*standard-output*` is used. `body` may have zero or more declarations as its first forms.

`x-spacing` specifies the number of units of spacing to be inserted between columns of the table; the default is the width of a space character in the current text style. `y-spacing` specifies the number of units of spacing to be inserted between rows in the table; the default is the default vertical spacing of the stream. Possible values for these two options are:

- An integer—a size in the current units to be used for spacing
- A string or character—the spacing is the width or height of the string or character in the current text style
- A function—the spacing is the amount of horizontal or vertical space the function would consume when called on the stream
- A list—the list is of the form `(number unit)`, where `unit` is one of `:point`, `:pixel`, `:mm`, `:character`, or `:line`. When `unit` is `:character`, the width of an “M” in the current text style is used as the width of one character.

`multiple-columns` is either `nil`, `t`, or an integer. If it is `t` or an integer, the table rows will be broken up into a multiple columns. If it is `t`, CLIM will determine the optimal number of columns. If it is an integer, it will be interpreted as the desired number of columns. `multiple-columns-x-spacing` has the same format as `x-spacing`. It controls the spacing between the multiple columns. It defaults to the value of the `x-spacing` option.

When the boolean `equalize-column-widths` is `t`, all the columns will have the same width (the width of the widest cell in any column in the entire table).

`record-type` specifies the class of output record to create. The default is `standard-table-output-record`. This argument should only be supplied by a pro-
grammer if there is a new class of output record that supports the table formatting protocol.

formatting-row

Arguments: (&optional stream &key record-type) &body body

Summary: Binds the local environment in such a way the output of body will be grouped into a table row. All of the output performed by body becomes the contents of one row. This must be used inside of formatting-table, and in conjunction with formatting-cell.

stream is an output recording stream to which output will be done. The stream argument is not evaluated, and must be a symbol that is bound to a stream. If stream is t (the default), *standard-output* is used. body may have zero or more declarations as its first forms.

Once a table has had a row added to it via formatting-row, no columns may be added to it.

record-type specifies the class of output record to create. The default is standard-row-output-record. This argument should only be supplied by a programmer if there is a new class of output record that supports the row formatting protocol.

formatting-column

Arguments: (&optional stream &key record-type) &body body

Summary: Binds the local environment in such a way the output of body will be grouped into a table column. All of the output performed by body becomes the contents of one column. This must be used inside of formatting-table, and in conjunction with formatting-cell.

stream is an output recording stream to which output will be done. The stream argument is not evaluated, and must be a symbol that is bound to a stream. If stream is t (the default), *standard-output* is used. body may have zero or more declarations as its first forms.

Once a table has had a column added to it via formatting-column, no rows may be added to it.

record-type specifies the class of output record to create. The default is standard-column-output-record. This argument should only be supplied if there is a new class of output record that supports the column formatting protocol.
formatting-cell

[Macro]
Arguments: (&optional stream &key (align-x #:left) (align-y #:baseline) min-width min-height record-type) &body body
Summary: Controls the output of a single cell inside a table row or column, or of a single item inside formatting-item-list. All of the output performed by body becomes the contents of the cell.

stream is an output recording stream to which output will be done. The stream argument is not evaluated, and must be a symbol that is bound to a stream. If stream is t (the default), *standard-output* is used. body may have zero or more declarations as its first forms.

align-x specifies how the output in a cell will be aligned relative to other cells in the same table column. The default, #:left, causes the cells to be flush-left in the column. The other possible values are #:right (meaning flush-right in the column) and #:center (meaning centered in the column). Each cell within a column may have a different alignment; thus it is possible, for example, to have centered legends over flush-right numeric data.

align-y specifies how the output in a cell will be aligned vertically. The default, #:baseline, causes textual cells to be aligned along their baselines and graphical cells to be aligned at the bottom. The other possible values are #:bottom (align at the bottom of the output), #:top (align at the top of the output), and #:center (center the output in the cell).

min-width and min-height are used to specify minimum width or height of the cell. The default, nil, causes the cell to be only as wide or high as is necessary to contain the cell’s contents. Otherwise, min-width and min-height are specified in the same way as the :x-spacing and :y-spacing arguments to formatting-table.

record-type specifies the class of output record to create. The default is standard-cell-output-record. This argument should only be supplied by a programmer if there is a new class of output record that supports the cell formatting protocol.

formatting-item-list

[Macro]
Arguments: (&optional stream &key x-spacing y-spacing n-columns n-rows stream-width stream-height max-width max-height initial-spacing (row-wise t) (move-cursor t) record-type) &body body
Summary: Binds the local environment in such a way that the output of body will be done in an item list (that is, menu) format. This must be used in conjunction with formatting-cell, which delimits each item. The item list is placed so that its upper left corner is at the current text cursor position of stream. If the boolean
move-cursor is t (the default), then the text cursor will be moved so that it imme-
diately follows the last cell of the item list.

“Item list output” means that each row of the item list consists of a single cell. The
first row is on top, and each succeeding row has its top aligned with the bot-
tom of the previous row (plus the specified y-spacing). Multiple rows and col-
umns are constructed after laying the item list out in a single column. Item list
output takes place in a normalized +y-downward coordinate system.

The returned value is the output record corresponding to the table.

stream is an output recording stream to which output will be done. The stream
argument is not evaluated, and must be a symbol that is bound to a stream. If
stream is t (the default), *standard-output* is used. body may have zero or
more declarations as its first forms.

x-spacing specifies the number of units of spacing to be inserted between col-
umns of the item list; the default is the width of a \Space character in the cur-
rent text style. y-spacing specifies the number of units of spacing to be inserted
between rows in the item list; the default is default vertical spacing of the stream. The format of these arguments is as for formatting-table.

When the boolean equalize-column-widths is t, all the columns will have the
same width (the width of the widest cell in any column in the entire item list).

n-columns and n-rows specify the number of columns or rows in the item list. The
default for both is nil, which causes CLIM to pick an aesthetically pleasing lay-
out, possibly constrained by the other options. If both n-columns and n-rows are
supplied and the item list contains more elements than will fit according to the
specification, CLIM will format the item list as if n-rows were supplied as nil.

max-width and max-height constrain the layout of the item list. max-width can be
overridden by n-rows. max-height can be overridden by n-columns.

formatting-item-list normally spaces items across the entire width of the stream. When initial-spacing is t, it inserts some whitespace (about half as much space as is between each item) before the first item on each line. When it is nil (the default), the initial whitespace is not inserted. If row-wise is t (the default) and the item list requires multiple columns, each successive element in the item list is laid out from left to right. If row-wise is nil and the item list requires multiple columns, each successive element in the item list is laid out below its predeces-
or, as in a telephone book.
record-type specifies the class of output record to create. The default is standard-item-list-output-record. Supply this argument only if there is a new class of output record that supports the item list formatting protocol.

format-items
Arguments: items &key stream printer presentation-type x-spacing y-spacing n-columns n-rows max-width max-height cell-align-x cell-align-y initial-spacing (move-cursor t) record-type
Summary: This is a function interface to the item list formatter. The elements of the sequence items are formatted as separate cells within the item list.

stream is an output recording stream to which output will be done. It defaults to *standard-output*.

printer (default is prin1) is a function that takes two arguments, an item and a stream, and outputs the item on the stream. printer has dynamic extent.

presentation-type is a presentation-type. When printer is not supplied, the items will be printed as if printer were:

```
#'(lambda (item stream)
  (present item presentation-type :stream stream))
```

When printer is supplied, each item will be enclosed in a presentation whose type is presentation-type.

x-spacing, y-spacing, n-columns, n-rows, max-width, max-height, initial-spacing, and move-cursor are as for formatting-item-list.

cell-align-x and cell-align-y are used to supply :align-x and :align-y to an implicitly used formatting-cell.

record-type is as for formatting-item-list.

format-textual-list
Arguments: sequence printer &key (:stream *standard-output*) (:separator "", ",") :conjunction
Summary: Outputs a sequence of items as a textual list.

Note that format-items is similar to formatting-item-list. Both operators do the same thing, except they accept their input differently:

- formatting-item-list accepts its input as a body that calls formatting-cell for each item.
- format-items accepts its input as a list of items with a specification of how to print them.

Note that menus use the one-dimensional table formatting model.

17.1.3 Examples of Formatting Tables

17.1.3.1 Formatting a Table From a List

The example1 function formats a simple table whose contents come from a list.

```lisp
(defun example1 (&optional (items *alphabet*) &key (stream *standard-output*) (n-columns 6) (y-spacing (stream)) (x-spacing (stream)))
  (clim:formatting-table
   (stream :y-spacing y-spacing :x-spacing x-spacing)
   (do ()
     ((null items))
   (clim:formatting-row (stream)
     (do ((i 0 (1+ i)))
       ((or (null items) (= i n-columns)))
     (clim:formatting-cell (stream)
       (format stream "~A" (pop items))))))))
```

Evaluating (example1 *alphabet* :stream *my-window*) shows this table:

```
A B C D E F
G H I J K L
M N O P Q R
S T U V W X
Y Z
```

Figure 24. Example1 With No :y-spacing

Figure 24 shows the result of evaluating the example1 function call without providing the :y-spacing and :x-spacing keywords. The defaults for these keywords makes tables whose elements are characters look reasonable.
You can easily vary the number of columns, and the spacing between rows or between columns. In the following example, we provide keyword arguments that change the appearance of the table.

Evaluating this form

```lisp
(example1 *alphabet* :stream *my-window*
   :n-columns 10 :x-spacing 10
   :y-spacing 10)
```

shows this table:

```
  A  B  C  D  E  F  G  H  I  J
 K L M N O P Q R S T
 U V W X Y Z
```

Figure 25. Example1 With :y-spacing

(Note that this example can be done with `formatting-item-list` as shown in example4.)

### 17.1.3.2 Formatting a Table Representing a Calendar Month

The `calendar-month` function shows how you can format a table that represents a calendar month. The first row in the table acts as column headings representing the days of the week. The following rows are numbers representing the days of the month.

This example shows how you can align the contents of a cell. The column headings (Sun, Mon, Tue, etc.) are centered within the cells. However, the dates themselves (1, 2, 3, ... 31) are aligned to the right edge of the cells. The resulting calendar looks good because the dates are aligned in the natural way.

```
(in-package :clim-user)
(defun day-of-the-week-string (day-of-week)
  (cdr (assoc day-of-week *day-of-the-week-string*))
)
(defvar *days-in-month* '((1 . 31)(2 . 28) ( 3 . 31)( 4 . 30)
                           (5 . 31)(6 . 30) ( 7 . 31)( 8 . 31)
                           (9 . 30)(10 . 31)(11 . 30)(12 . 31))
  "alist whose first element is numeric value returned by decode-universal-time and second is the number of days in that month")
```

412CLIM User Guide
;; In a leap year, the month-length function increments the number of
;; days in February as required
(defun leap-year-p (year)
  (cond ((and (integerp (/ year 100))
              (integerp (/ year 400)))
         t
         ((and (not (integerp (/ year 100)))
              (integerp (/ year 4)))
         t
         (t nil)))
(defun month-length (month year)
  (let ((days (cdr (assoc month *days-in-month*))))
    (when (and (eql month 2)
               (leap-year-p year)
               (incf days))
    days))
(defun calendar-month (month year &key (stream *standard-output*))
  (let ((days-in-month (month-length month year)))
    (multiple-value-bind (sec min hour date month year start-day)
      (decode-universal-time (encode-universal-time
                               0 0 0 1 month year))
      (setq start-day (mod (+ start-day 1) 7))
      (clim:formatting-table (stream)
        (clim:formatting-row (stream)
          (dotimes (d 7)
            (clim:formatting-cell (stream :align-x :center)
               (write-string (day-of-the-week-string
                              (mod (- d 1) 7)) stream)))))
    (do ((date 1)
         (first-week t nil)
         (> date days-in-month))
      (clim:formatting-row (stream)
        (dotimes (d 7)
          (clim:formatting-cell (stream :align-x :right)
            (when (and (<= date days-in-month)
                        (or (not first-week) (>= d start-day)))
              (format stream "~D" date)
              (incf date))))))))))
(define-application-frame calendar ()
  (:panes
   (main :application
         :width :compute :height :compute
         :display-function 'display-main)))
(define-calendar-command (com-exit-calendar :menu "Exit") ()
  (frame-exit *application-frame*))
(defmethod display-main ((frame calendar) stream &key)
  (multiple-value-bind (sec min hour date month year start-day)
      (decode-universal-time (get-universal-time))
    (calendar-month month year :stream stream)))

(defun run ()
  (find-application-frame 'calendar))

Evaluating (calendar-month 5 90 :stream *my-stream*) shows this table:

<table>
<thead>
<tr>
<th>Sun</th>
<th>Mon</th>
<th>Tue</th>
<th>Wed</th>
<th>Thu</th>
<th>Fri</th>
<th>Sat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 26. A Table Representing a Calendar Month

17.1.3.3 Formatting a Table With Regular Graphic Elements

The example2 function shows how you can draw graphics within the cells of a table. Each cell contains a rectangle of the same dimensions.

(defun example2 (&key (stream *standard-output*)
  y-spacing
  x-spacing)
  (clim:formatting-table
    (stream :y-spacing y-spacing :x-spacing x-spacing)
    (dotimes (i 3)
      (clim:formatting-row
       (stream)
       (dotimes (j 3)
         (clim:formatting-cell
          (stream)
          (clim:draw-rectangle* stream 10 10 50 50)))))))

Evaluating (example2 :stream *my-stream* :y-spacing 5) shows this table:
17.1.3.4 Formatting a Table With Irregular Graphics in the Cells

The example3 function shows how you can format a table in which each cell contains graphics of different sizes.

**(defun example3** (&optional (items *alphabet*) &key (stream *standard-output*) (n-columns 6) y-spacing x-spacing)
  (clim:formatting-table
   (stream :y-spacing y-spacing :x-spacing x-spacing)
   (do ()
     ((null items))
     (clim:formatting-row (stream)
      (do ((i 0 (1+ i)))
        ((or (null items) (= i n-columns)))
        (clim:formatting-cell (stream)
         (clim:draw-polygon* stream
          (list 0 0 (* 10 (1+ (random 3))) 5 5 (* 10 (1+ (random 3))))
          :filled nil)
         (pop items))))))

Evaluating `(example3 *alphabet* :stream *my-stream*)` shows this table:
17.1.3.5 Formatting a Table of a Sequence of Items

The example4 function shows how you can use formatting-item-list to format a table of a sequence of items when the exact arrangement of the items and the table is not important. Note that you use formatting-cell inside the body of formatting-item-list to output each item. You do not use formatting-column or formatting-row, because CLIM figures out the number of columns and rows automatically (or obeys a constraint given in a keyword argument).

(defun example4 (&optional (items *alphabet*) &key (stream *standard-output*) n-columns n-rows y-spacing x-spacing :max-width max-height)
  (clim:formatting-item-list (stream :y-spacing y-spacing :x-spacing x-spacing :n-columns n-columns :n-rows n-rows :max-width max-width :max-height max-height)
    (do ()
      ((null items))
        (clim:formatting-cell (stream)
          (format stream "~A" (pop items)))))
)

Evaluating (example4 :stream *my-window*) shows this table:

<416CLIM User Guide>
You can easily add a constraint specifying the number of columns.

Evaluating \( \text{example4 :stream *my-stream* :n-columns 8} \) gives this:

```
A B C D E F G H
I J K L M N O P
Q R S T U V W X
Y Z
```

Figure 29. Example4 Table Reformatted
17.2 Formatting Graphs in CLIM

17.2.1 Conceptual Overview of Formatting Graphs

When you need to format a graph, you specify the nodes to be in the graph and the scheme for organizing them. The CLIM graph formatter does the layout automatically, obeying any constraints that you supply.

You can format any graph in CLIM. The CLIM graph formatter is most successful with directed acyclic graphs (DAG). “Directed” means that the arcs on the graph have a direction. “Acyclic” means that there are no loops in the graph.

Here is an example of such a graph:

![Directed Acyclic Graph]

To specify the elements and the organization of the graph, you provide CLIM with the following information:

- The root node
- A “node printer,” that is, a function used to display each node. The function is passed the object associated with a node and the stream on which to do output.
- An “inferior producer,” a function that takes one node and returns its inferior nodes (the nodes to which it points)

Based on that information, CLIM lays out the graph for you. You can specify a number of options that control the appearance of the graph. For example, you can specify whether you want the graph to grow vertically (downward) or horizontally (to the right). Note that
CLIM’s algorithm does the best layout it can, but complicated graphs can be difficult to lay out in a readable way.

See Section 17.5, “Advanced Topics,” for the graph formatting protocol.

### 17.2.2 CLIM Operators for Graph Formatting

**format-graph-from-roots**  
*Function*

**Arguments:**  
`root-objects` `object-printer` `inferior-producer` `key` `stream` `orientation`  
`cutoff-depth` `merge-duplicates` `duplicate-key` `duplicate-test`  
`generation-separation` `within-generation-separation` `center-nodes`  
`arc-drawer` `arc-drawing-options` `graph-type` `(move-cursor t)`

**Summary:** Draws a graph whose roots are specified by the sequence `root-objects`. The nodes of the graph are displayed by calling the function `object-printer`, which takes two arguments, the node to display and a stream. `inferior-producer` is a function of one argument that is called on each node to produce a sequence of inferiors (or `nil` if there are none). Both `object-printer` and `inferior-producer` have dynamic extent.

The output from graph formatting takes place in a normalized +y-downward coordinate system. The graph is placed so that the upper left corner of its bounding rectangle is at the current text cursor position of `stream`. If the boolean `move-cursor` is `t` (the default), then the text cursor will be moved so that it immediately follows the lower right corner of the graph.

The returned value is the output record corresponding to the graph.

`stream` is an output recording stream to which output will be done. It defaults to `*standard-output*`.

`orientation` specifies the direction from root to leaves in the graph. `orientation` may be either `:horizontal` (the default) or `:vertical`. In LispWorks, it may also be `:down` or `:up; :right` is a synonym for `:horizontal` and `:down` is a synonym for `:vertical`.

`cutoff-depth` specifies the maximum depth of the graph. It defaults to `nil`, meaning that there is no cutoff depth. Otherwise it must be an integer, meaning that no nodes deeper than `cutoff-depth` will be formatted or displayed.

If the boolean `merge-duplicates` is `t`, then duplicate objects in the graph will share the same node in the display of the graph. That is, when `merge-duplicates` is `t`, the resulting graph will be a tree. If `merge-duplicates` is `nil` (the default), then
duplicate objects will be displayed in separate nodes. **duplicate-key** is a function of one argument that is used to extract the node object component used for duplicate comparison; the default is **identity**. **duplicate-test** is a function of two arguments that is used to compare two objects to see if they are duplicates; the default is **eql**. **duplicate-key** and **duplicate-test** have dynamic extent.

**generation-separation** is the amount of space to leave between successive generations of the graph; the default is 20. **within-generation-separation** is the amount of space to leave between nodes in the same generation of the graph; the default is 10. **generation-separation** and **within-generation-separation** are specified in the same way as the y-spacing argument to **formatting-table**.

When **center-nodes** is **t**, each node of the graph is centered with respect to the widest node in the same generation. The default is **nil**.

**arc-drawer** is a function of seven positional and some unspecified keyword arguments that is responsible for drawing the arcs from one node to another; it has dynamic extent. The positional arguments are the stream, the “from” node, the “to” node, the “from” x and y position, and the “to” x and y position. The keyword arguments gotten from **arc-drawing-options** are typically line drawing options, such as for **draw-line**. If **arc-drawer** is unsupplied, the default behavior is to draw a thin line from the “from” node to the “to” node using **draw-line**.

**graph-type** is a keyword that specifies the type of graph to draw. CLIM supports graphs of type **:tree**, **:directed-graph** (and its synonym **:digraph**), and **:directed-acyclic-graph** (and its synonym **:dag**). **graph-type** defaults to **:tree** when **merge-duplicates** is **t**; otherwise, it defaults to **:digraph**.

The following is an example demonstrating the use of **format-graph-from-roots** to draw an arrow. Note that **draw-arrow** is available internally.

```lisp
(define-application-frame graph-it ()
  ((root-node :initform (find-class 'clim:design)
                :initarg :root-node
                :accessor root-node)
   (app-stream :initform nil :accessor app-stream))
  (:panes (display :application
                    :display-function 'draw-display
                    :display-after-commands :no-clear))
  (:layouts
   (:defaults
    (:horizontally () display)))))
```

<Bold>420CLIM User Guide
(defmethod draw-display ((frame graph-it) stream)
  (format-graph-from-roots (root-node *application-frame*)
    #'draw-node
    #'clos:Class-direct-subclasses
    :stream stream
    :arc-drawer
    #'(lambda (stream from-object
      to-object x1 y1
      x2 y2
      &rest
      drawing-options)
      (declare (dynamic-extent
      drawing-options))
      (declare (ignore from-object
      to-object))
      (apply #'draw-arrow* stream
      x1 y1 x2 y2 drawing-options))
    :merge-duplicates t)
  (setf (app-stream frame) stream))

(define-presentation-type node ())
(defun draw-node (object stream)
  (with-output-as-presentation (stream object 'node)
    (surrounding-output-with-border
      (stream :shape :rectangle)
      (format stream "~A"
        (class-name object))))

(define-graph-it-command (exit :menu "Exit") ()
  (frame-exit *application-frame*))

(defun graph-it (&optional (root-node (find-class 'basic-sheet))
  (port (find-port)))
  (if (atom root-node) (setf root-node (list root-node)))
  (let ((graph-it (make-application-frame 'graph-it
    :frame-manager
    (find-frame-manager
      :port port)
    :width 800
    :height 600
    :root-node root-node)))
    (run-frame-top-level graph-it)))
17.2.3 Examples of CLIM Graph Formatting

(defstruct node (name "") (children nil))

(defvar g1 (let* ((2a (make-node :name "2A"))
                   (2b (make-node :name "2B"))
                   (2c (make-node :name "2C")))
               (1a (make-node :name "1A" :children (list 2a 2b)))
               (1b (make-node :name "1B" :children (list 2b 2c))))
               (make-node :name "0" :children (list 1a 1b))))

(defun test-graph (root-node &rest keys)
  (apply #'clim:format-graph-from-root root-node
         #'(lambda (node s)
             (write-string (node-name node) s))
         #'node-children keys))

Evaluating (test-graph g1 :stream *my-window*) results in the following graph:

![A Horizontal Graph]

In Figure 32, the graph has a horizontal orientation and grows toward the right by default. We can supply the :orientation keyword to control this. Evaluating (test-graph g1 :stream *my-window* :orientation :vertical) results in the following graph:
The following example uses **format-graph-from-roots** to create a graph with multiple parents, that is, a graph in which node D is a child of both nodes B and C. Note that it interprets its first argument as a list of top-level graph nodes, so we have wrapped the root node inside a list.

```lisp
(defun test-graph (win)
  (window-clear win)
  (format-graph-from-roots '((a (b (d)) (c (d))))
    #'(lambda (x s) (princ (car x) s))
    #'cdr
    :stream win
    :orientation :vertical
    :merge-duplicates t
    :duplicate-key #'car)
  (force-output win))
```

Figure 33. A Vertical Graph
17.3 Formatting Text in CLIM

CLIM provides the following three forms for creating textual lists, indenting output, and breaking up lengthy output into multiple lines.

format-textual-list [Function]
Arguments: sequence printer &key stream separator conjunction
Summary: Outputs the sequence of items in sequence as a “textual list.” For example, the list (1 2 3 4) might be printed as

1, 2, 3, and 4

printer is a function of two arguments: an element of the sequence and a stream; it has dynamic extent. It is called to output each element of the sequence.

stream specifies the output stream. The default is *standard-output*.

The separator and conjunction arguments control the appearance of each element of the sequence and the separators used between each pair of elements. separator is a string that is output after every element but the last one; the default for separator is “,” (a comma followed by a space). conjunction is a string that is output before the last element. The default is nil, meaning that there is no conjunction. Typical values for conjunction are the strings “and” and “or”.

indenting-output [Macro]
Arguments: (stream indentation &key (move-cursor t)) &body body
Summary: Binds stream to a stream that inserts whitespace at the beginning of each line of output produced by body, and then writes the indented output to the stream that is the original value of stream.

The stream argument is not evaluated, and must be a symbol that is bound to an output recording stream. If stream is t, *standard-output* is used. body may have zero or more declarations as its first forms.

indentation specifies how much whitespace should be inserted at the beginning of each line. It is specified in the same way as the :x-spacing option to formatting-table.

If the boolean move-cursor is t (the default), CLIM moves the cursor to the end of the table.
Programmers using `indenting-output` should begin the body with a call to `fresh-line` (or some equivalent) to position the stream to the indentation initially. There is a restriction on interactions between `indenting-output` and `filling-output`: a call to `indenting-output` should appear outside of a call to `filling-output`.

**filling-output**

Arguments: `(stream &key fill-width break-characters after-line-break after-line-break-initially) &body body`

Summary: Binds `stream` to a stream that inserts line breaks into the textual output written to it (by such functions as `write-char` and `write-string`) so that the output is usually no wider then `fill-width`. The filled output is then written on the original stream.

The `stream` argument is not evaluated, and must be a symbol that is bound to a stream. If `stream` is `t`, `*standard-output*` is used. `body` may have zero or more declarations as its first forms.

`fill-width` specifies the width of filled lines, and defaults to 80 characters. It is specified the same way as the `x-spacing` option for `formatting-table`. See Subsection 17.1.2, "CLIM Operators for Formatting Tables."

“Words” are separated by the characters specified in the list `break-characters`. When a line is broken to prevent wrapping past the end of a line, the line break is made at one of these separators. That is, `filling-output` does not split “words” across lines, so it might produce output wider than `fill-width`.

`after-line-break` specifies a string to be sent to `stream` after line breaks; the string appears at the beginning of each new line. The string must not be wider than `fill-width`.

If the boolean `after-line-break-initially` is `t`, then the `after-line-break` text is to be written to `stream` before executing `body`, that is, at the beginning of the first line. The default is `nil`. 
17.4 Bordered Output in CLIM

CLIM provides a mechanism for surrounding arbitrary output with some kind of a border. The programmer annotates some output-generating code with an advisory macro that describes the type of border to be drawn. The following code produces the output shown in Figure 34.

For example, the following produces three pieces of output, which are surrounded by a rectangle, highlighted with a dropshadow, and underlined, respectively.

```
(defun border-test (stream)
  (fresh-line stream)
  (surrounding-output-with-border (stream :shape :rectangle)
    (format stream "This is some output with a rectangular border"))
  (terpri stream) (terpri stream)
  (surrounding-output-with-border (stream :shape :drop-shadow)
    (format stream "This has a drop-shadow under it"))
  (terpri stream) (terpri stream)
  (surrounding-output-with-border (stream :shape :underline)
    (format stream "And this output is underlined")))
```

This is some output with a rectangular border.

This has a drop-shadow under it.

And this output is underlined.

Figure 34. Examples of Bordered Output

**surrounding-output-with-border**

*Macro*

**Arguments:** &optional stream &key shape (move-cursor t) &body body

**Summary:** Binds the local environment in such a way the output of body will be surrounded by a border of the specified shape. Supported shapes are :rectangle (the default), :oval, :drop-shadow, and :underline. :rectangle draws a rectangle around the bounding rectangle of the output. :oval draws an oval around the bounding rect-
angle of the output. :drop-shadow draws a “drop shadow” around the lower right edge of the bounding rectangle of the output. :underline draws a thin line along the baseline of all of the text in the output, but does not draw anything underneath non-textual output.

If the boolean move-cursor is t (the default), then the text cursor will be moved so that it immediately follows the lower right corner of the bordered output.

stream is an output recording stream to which output will be done. The stream argument is not evaluated, and must be a symbol that is bound to a stream. If stream is t (the default), *standard-output* is used. body may have zero or more declarations as its first forms.

define-border-type
[Macro]
Arguments: shape arglist &body body
Summary: Defines a new kind of border named shape. arglist must be a subset of the “canonical” arglist (using string-equal to do the comparison) (&key stream record left top right bottom). body is the code that actually draws the border. It has lexical access to stream, record, left, top, right, and bottom, which are respectively, the stream being drawn on, the output record being surrounded, and the coordinates of the left, top, right, and bottom edges of the bounding rectangle of the record. body may have zero or more declarations as its first forms.
17.5 Advanced Topics

The material in this subsection is advanced; most CLIM programmers can skip to the next section. This section discusses Table, Item List, and Graph Formatting Protocols.

All of table, item list, and graph formatting is implemented on top of the basic output recording protocol, using \texttt{with-new-output-record} to specify the appropriate type of output record. The following examples show specifically how tables and graphs are implemented.

Example 1: Tables \texttt{formatting-table} first collects all the output that belongs in the table into a collection of row, column, and cell output records, all of which are children of a single table output record. During this phase, \texttt{stream-drawing-p} is bound to \texttt{nil} and \texttt{stream-recording-p} is bound to \texttt{t}. When all the output has been generated, the table layout constraint solver (\texttt{adjust-table-cells} or \texttt{adjust-item-list-cells}) is called to compute the table layout, taking into account such factors as the widest cell in a given column. If the table is to be split into multiple columns, \texttt{adjust-multiple-columns} is now called. Finally, the table output record is positioned on the stream at the current text cursor position and then displayed by calling \texttt{replay} on the table (or item list) output record.

Example 2: Graphs \texttt{format-graph-from-roots} first collects all the graph node output records that belong in the graph by calling \texttt{generate-graph-nodes}. All these output records are children of a single graph output record. During this phase, \texttt{stream-drawing-p} is bound to \texttt{nil} and \texttt{stream-recording-p} is bound to \texttt{t}. When all the output has been generated, the graph layout code (\texttt{layout-graph-nodes} and \texttt{layout-out-graph-edges}) is called to compute the graph layout. Finally, the graph output record is positioned on the stream at the current text cursor position and then displayed by calling \texttt{replay} on the graph output record.
17.5.1 The Table Formatting Protocol

Any output record class that implements the following generic functions is said to support the table formatting protocol.

In the following subsections, the term “non-table output records” will be used to mean any output record that is not a table, row, column, cell, or item list output record. When CLIM “skips over intervening non-table output records,” this means that it will bypass all the output records between two such table output records (such as a table and a row, or a row and a cell) that are not records of those classes (most notably, presentation output records).

CLIM detects invalid nesting of table output records, such as a row within a row, a cell within a cell, or a row within a cell. Note that this does not prohibit the nesting of calls to formatting-table, it simply requires that programmers include the inner table within one of the cells of the outer table.

**table-output-record** [Protocol Class]
Summary: The protocol class that represents tabular output records; a subclass of output-record. If you want to create a new class that behaves like a table output record, it should be a subclass of table-output-record. Subclasses of table-output-record must obey the table output record protocol.

**table-output-record-p** [Function]
Arguments: object
Summary: Returns t if object is a table output record; otherwise, it returns nil.

:x-spacing [Initarg]
:y-spacing [Initarg]
:multiple-columns-x-spacing [Initarg]
:equalize-column-widths [Initarg]
Summary: All subclasses of table-output-record must handle these initargs, which are used to specify, respectively, the x and y spacing, the multiple column x spacing, and equal-width columns attributes of the table.

**standard-table-output-record** [Class]
Summary: The instantiable class of output record that represents tabular output. Its children will be a sequence of either rows or columns, with presentation output records possibly intervening. This is a subclass of table-output-record.
map-over-table-elements

Arguments: function table-record type

Summary: Applies function to all the rows or columns of table-record that are of type type. type is either :row, :column, or :row-or-column. function is a function of one argument, an output record; it has dynamic extent. map-over-table-elements ensures that rows, columns, and cells are properly nested. It skips over intervening non-table output record structure, such as presentations.

adjust-table-cells

Arguments: table-record stream

Summary: This function is called after the tabular output has been collected, but before it has been replayed. The method on standard-table-output-record implements the usual table layout constraint solver by moving the rows or columns of the table output record table-record and the cells within the rows or columns. stream is the stream on which the table is displayed.

adjust-multiple-columns

Arguments: table-record stream

Summary: This is called after adjust-table-cells to account for the case where the programmer wants to break the entire table up into multiple columns. Each of those columns will have some of the rows of the “original” table, and those rows may each have several columns. For example:

Original table:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>alpha</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>beta</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>gamma</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>delta</td>
</tr>
</tbody>
</table>

Multiple column version:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>alpha</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>beta</td>
<td>d</td>
<td>4</td>
</tr>
</tbody>
</table>

table-record and stream are as for adjust-table-cells.
17.5.1.1 The Row and Column Formatting Protocol

Any output record class that implements the following generic functions is said to support the row (or column) formatting protocol.

**row-output-record** [Protocol Class]
Summary: The protocol class that represents one row in a table; a subclass of output-record. If you want to create a new class that behaves like a row output record, it should be a subclass of row-output-record. Subclasses of row-output-record must obey the row output record protocol.

**row-output-record-p** [Function]
Arguments: object
Summary: Returns t if object is a row output record; otherwise, it returns nil.

**standard-row-output-record** [Class]
Summary: The instantiable class of output record that represents a row of output within a table. Its children will be a sequence of cells, and its parent (skipping intervening non-tabular records such as presentations) will be a table output record. This is a subclass of row-output-record.

**map-over-row-cells** [Generic Function]
Arguments: function row-record
Summary: Applies function to all the cells in the row row-record, skipping intervening non-table output record structure. function is a function of one argument, an output record corresponding to a table cell within the row; it has dynamic extent.

**column-output-record** [Protocol Class]
Summary: The protocol class that represents one column in a table; a subclass of output-record. If you want to create a new class that behaves like a column output record, it should be a subclass of column-output-record. Subclasses of column-output-record must obey the column output record protocol.

**column-output-record-p** [Function]
Arguments: object
Summary: Returns t if object is a column output record; otherwise, it returns nil.
standard-column-output-record  [Class]
Summary: The instantiable class of output record that represents a column of output within a table. Its children will be a sequence of cells, and its parent (skipping intervening non-tabular records such as presentations) will be a table output record; presentation output records may intervene. This is a subclass of column-output-record.

map-over-column-cells  [Generic Function]
Arguments: function column-record
Summary: Applies function to all the cells in the column column-record, skipping intervening non-table output record structure. function is a function of one argument, an output record corresponding to a table cell within the column; it has dynamic extent.

17.5.1.2 The Cell Formatting Protocol

Any output record class that implements the following generic functions is said to support the cell formatting protocol.

cell-output-record  [Protocol Class]
Summary: The protocol class that represents one cell in a table or an item list; a subclass of output-record. If you want to create a new class that behaves like a cell output record, it should be a subclass of cell-output-record. Subclasses of cell-output-record must obey the cell output record protocol.

cell-output-record-p  [Function]
Arguments: object
Summary: Returns t if object is a cell output record; otherwise, it returns nil.

:align-x  [Initarg]
:align-y  [Initarg]
:min-width  [Initarg]
:min-height  [Initarg]
Summary: All subclasses of cell-output-record must handle these initargs, which are used to specify, respectively, the x and y alignment, and the minimum width and height attributes of the cell.
standard-cell-output-record  [Class]
Summary: The instantiable class of output record that represents a single piece of output within a table row or column, or an item list. Its children will either be presentations or output records that represent displayed output. This is a subclass of cell-output-record.

cell-align-x  [Generic Function]
Arguments: cell

cell-align-y  [Generic Function]
Arguments: cell

cell-min-width  [Generic Function]
Arguments: cell

cell-min-height  [Generic Function]
Arguments: cell
Summary: These functions return, respectively, the x and y alignment and minimum width and height of the cell output record cell.

17.5.2 The Item List Formatting Protocol

item-list-output-record  [Protocol Class]
Summary: The protocol class that represents an item list; a subclass of output-record. If you want to create a new class that behaves like an item list output record, it should be a subclass of item-list-output-record. Subclasses of item-list-output-record must obey the item list output record protocol.

item-list-output-record-p  [Function]
Arguments: object
Summary: Returns t if object is an item list output record; otherwise, it returns nil.

:x-spacing  [Initarg]
:y-spacing  [Initarg]
:initial-spacing  [Initarg]
All subclasses of item-list-output-record must handle these initargs, which specify, respectively, the x and y spacing, the initial spacing, the desired number of rows and columns, and maximum width and height attributes of the item list.

standard-item-list-output-record
Summary: The output record that represents item list output. Its children will be a sequence of cells, with presentations possibly intervening. This is a subclass of item-list-output-record.

map-over-item-list-cells
Arguments: function item-list-record
Summary: Applies function to all of the cells in item-list-record. map-over-item-list-cells skips over intervening non-table output record structure, such as presentations. function is a function of one argument, an output record corresponding to a cell in the item list; it has dynamic extent.

adjust-item-list-cells
Arguments: item-list-record stream
Summary: This function is called after the item list output has been collected, but before the record has been replayed. The method on standard-item-list-output-record implements the usual item list layout constraint solver. item-list-record is the item list output record, and stream is the stream on which the item list is displayed.

17.5.3 The Graph Formatting Protocol

graph-output-record
Summary: The protocol class that represents a graph; a subclass of output-record. If you want to create a new class that behaves like a graph output record, it should be a subclass of graph-output-record. Subclasses of graph-output-record must obey the graph output record protocol.
graph-output-record-p [Function]
Arguments: object
Summary: Returns t if object is a graph output record, otherwise returns nil.

:orientation [Initarg]
:center-nodes [Initarg]
:cutoff-depth [Initarg]
:merge-duplicates [Initarg]
:generation-separation [Initarg]
:within-generation-separation [Initarg]
:hash-table [Initarg]
Summary: All the graph output records must handle these seven initargs, which are used to specify, respectively, the orientation, node centering, cutoff depth, merge duplicates, generation and within-generation spacing, and the node hash table of a graph output record.

define-graph-type [Macro]
Arguments: graph-type class
Summary: Defines a new graph type graph-type that is implemented by the class class (a subclass of graph-output-record). Neither of the arguments is evaluated.

graph-root-nodes [Generic Function]
Arguments: graph-record
Summary: Returns a sequence of the graph node output records corresponding to the root objects for the graph output record graph-record.

(setf graph-root-nodes) [Generic Function]
Arguments: roots graph-record
Summary: Sets the root nodes of graph-record to roots.

generate-graph-nodes [Generic Function]
Arguments: graph-record stream root-objects object-printer inferior-producer &key duplicate-key duplicate-test
Summary: This function is responsible for generating all the graph node output records of the graph. graph-record is the graph output record, and stream is the output stream. The graph node output records are generating by calling the object
printer on the root objects, then (recursively) calling the inferior producer on the root objects and calling the object printer on all inferiors. After all the graph node output records have been generated, the value of `graph-root-nodes` of `graph-record` must be set to be a sequence of those graph node output records that correspond to the root objects.

`root-objects`, `object-printer`, `inferior-producer`, `duplicate-key`, and `duplicate-test` are as for `format-graph-from-roots`.

`layout-graph-nodes`  
[Generic Function]  
Arguments: `graph-record stream`  
Summary: This function is responsible for laying out the nodes in the graph contained in the output record `graph-record`. It is called after the graph output has been collected, but before the graph record has been displayed. The method on `standard-graph-output-record` implements the usual graph layout constraint solver. `stream` is the stream on which the graph is displayed.

`layout-graph-edges`  
[Generic Function]  
Arguments: `graph-record stream arc-drawer arc-drawing-options`  
Summary: This function is responsible for laying out the edges in the graph. It is called after the graph nodes have been laid out, but before the graph record has been displayed. The method on `standard-graph-output-record` simply causes thin lines to be drawn from each node to all of its children. `graph-record` and `stream` are as for `layout-graph-nodes`.

`graph-node-output-record`  
[Protocol Class]  
Summary: The protocol class that represents a node in graph; a subclass of `output-record`. If you want to create a new class that behaves like a graph node output record, it should be a subclass of `graph-node-output-record`. Subclasses of `graph-node-output-record` must obey the graph node output record protocol.

`graph-node-output-record-p`  
[Function]  
Arguments: `object`  
Summary: Returns `t` if `object` is a graph node output record; otherwise, it returns `nil`.

`standard-graph-node-output-record`  
[Class]  
Summary: The instantiable class of output record that represents a graph node. Its parent will be a graph output record. This is a subclass of `graph-node-output-record`.
graph-node-parents  [Generic Function]
Arguments: graph-node-record
Summary: Returns a sequence of the graph node output records whose objects are “parents” of the object corresponding to the graph node output record graph-node-record. This differs from output-record-parent, as graph-node-parents can return output records that are not the parent records of graph-node-record.

(setf graph-node-parents)  [Generic Function]
Arguments: parents graph-node-record
Summary: Sets the parents of graph-node-record to be parents. parents must be a list of graph node records.

graph-node-children  [Generic Function]
Arguments: graph-node-record
Summary: Returns a sequence of the graph node output records whose objects are “children” of the object corresponding to the graph node output record graph-node-record. This differs from output-record-children, as graph-node-children can return output records that are not child records of graph-node-record.

(setf graph-node-children)  [Generic Function]
Arguments: children graph-node-record
Summary: Sets the children of graph-node-record to be children. children must be a list of graph node records.

graph-node-object  [Generic Function]
Arguments: graph-node-record
Summary: Returns the object that corresponds to the output record graph-node-record. This function only works correctly while inside the call to format-graph-from-roots. Unspecified results are returned outside format-graph-from-roots, as CLIM does not capture application objects that might have dynamic extent.
Chapter 18   Sheets
18.1 Overview of Window Facilities

A central notion in organizing user interfaces is allocating screen regions to particular tasks and recursively subdividing these regions into subregions. The windowing layer of CLIM defines an extensible framework for constructing, using, and managing such hierarchies of interactive regions. This framework allows uniform treatment of the following things:

- Window objects like those in X or NeWS
- Lightweight gadgets typical of toolkit layers, such as Motif or OpenLook
- Structured graphics such as output records and an application’s presentation objects
- Objects that act as Lisp handles for windows or gadgets implemented in a different language (such as OpenLook gadgets implemented in C)

From the perspective of most CLIM users, CLIM’s windowing layer plays the role of a window system. However, CLIM usually uses the services of a window system platform to provide efficient windowing, input, and output facilities. We will refer to such window system platforms as host window systems or as display servers.

The fundamental window abstraction defined by CLIM is called a sheet. A sheet can participate in a relationship called a windowing relationship. This relationship is one in which one sheet called the parent provides space to a number of other sheets called children. Support for establishing and maintaining this kind of relationship is the essence of what window systems provide. At any point in time, CLIM allows a sheet to be a child in one relationship and a parent in another relationship.

Programmers can manipulate unrooted hierarchies of sheets (those without a connection to any particular display server). However, a sheet hierarchy must be attached to a display server to make it visible. Ports and grafts provide the functionality for managing this capability. A port is an abstract connection to a display service that is responsible for managing host display server resources and for processing input events received from the host display server. A graft is a special kind of sheet that represents a host window, typically a root window (that is, a screen-level window). A sheet is attached to a display by making it a child of a graft, which represents an appropriate host window. The sheet will then appear to be a child of that host window. In other words, a sheet is put onto a particular screen by making it a child of an appropriate graft and enabling it. Ports and grafts are described in detail in Chapter 19, “Ports, Grafts, and Mirrored Sheets.”
As has been discussed previously, CLIM users will typically be dealing with *panes*, rather than with sheets, ports, grafts, or mediums, as a call to `make-application-frame` automatically results in a port specification, a graft instantiation, and the allocation of a medium, to which output directed to the pane will be forwarded.

### 18.1.1 Properties of Sheets

Sheets have the following properties:

- A coordinate system—Provides the ability to refer to locations in a sheet’s abstract plane.
- A region—Defines an area within a sheet’s coordinate system that indicates the area of interest within the plane, that is, a clipping region for output and input. This typically corresponds to the visible region of the sheet on the display.
- A parent—A sheet that is the parent in a windowing relationship in which this sheet is a child.
- Children—An ordered set of sheets that are each children in a windowing relationship in which this sheet is a parent. The ordering of the set corresponds to the stacking order of the sheets. Not all sheets have children.
- A transformation—Determines how points in this sheet’s coordinate system are mapped into points in its parents’ coordinate system.
- An enabled flag—Indicates whether the sheet is currently actively participating in the windowing relationship with its parent and siblings.
- An event handler—A procedure invoked when the display server wishes to inform CLIM of external events.
- Output state—A set of values used when CLIM causes graphical or textual output to appear on the display. This state is often represented by a medium.

### 18.1.2 Sheet Protocols

A sheet is a participant in a number of protocols. Every sheet must provide methods for the generic functions that make up these protocols. These protocols are:

- The windowing protocol—Describes the relationships between the sheet and its parent and children (and, by extension, all of its ancestors and descendants).
The input protocol—Provides the event handler for a sheet. Events may be handled synchronously, asynchronously, or not at all.

The output protocol—Provides graphical and textual output, and manages descriptive output state such as color, transformation, and clipping.

The repaint protocol—Invoked by the event handler and by user programs to ensure that the output appearing on the display device appears as the program expects it to appear.

The notification protocol—Invoked by the event handler and user programs to ensure that CLIM’s representation of window system information is equivalent to the display server’s.

These protocols may be handled directly by a sheet, queued for later processing by some other agent, or passed on to a delegate sheet for further processing.
18.2 Basic Sheet Classes

There are no standard sheet classes in CLIM, and no pre-packaged way to create sheets in general. If a programmer needs to create an instance of some class of sheet, make-instance must be used. In most cases, application programmers will not deal with sheets directly, but instead will use a subclass of sheets known as panes. Panes can be created by calling the make-pane function. For a more detailed discussion on panes, see Chapter 10, “Panes and Gadgets.”

Sheet

[Protocol Class]
Summary: The protocol class that corresponds to a sheet, a subclass of bounding-rectangle. If you want to create a new class that behaves like a sheet, it should be a subclass of sheet. Subclasses of sheet must obey the sheet protocol.

All of the subclasses of sheet are mutable.

Sheetp

[Function]
Arguments: object
Summary: Returns t if object is a sheet; otherwise, it returns nil.

Basic-sheet

[Class]
Summary: The basic class on which all CLIM sheets are built, a subclass of sheet. This class is an abstract class intended only to be subclassed, not instantiated.
18.3 Relationships Between Sheets

Sheets are arranged in a tree-structured, acyclic, top-down hierarchy. Thus, in general, a sheet has one or no parents and zero or more children. A sheet may have zero or more siblings (that is, other sheets that share the same parent). In order to describe the relationships between sheets, we define the following terms.

- **Adopted**—A sheet is said to be *adopted* if it has a parent. A sheet becomes the parent of another sheet by adopting that sheet.
- **Disowned**—A sheet is said to be *disowned* if it does not have a parent. A sheet ceases to be a child of another sheet by being disowned.
- **Grafted**—A sheet is said to be *grafted* when it is part of a sheet hierarchy whose highest ancestor is a graft. In this case, the sheet may be visible on a particular window server.
- **Degrafted**—A sheet is said to be *degrafted* when it is part of a sheet hierarchy that cannot be visible on a server, that is, the highest ancestor is not a graft.
- **Enabled**—A sheet is said to be *enabled* when it is actively participating in the windowing relationship with its parent. If a sheet is enabled and grafted, and all its ancestors are enabled (they are grafted by definition), then the sheet will be visible if it occupies a portion of the graft region that isn’t clipped by its ancestors or ancestor’s siblings.
- **Disabled**—The opposite of enabled is *disabled*.

18.3.1 Sheet Relationship Functions

The generic functions in this section comprise the sheet protocol. All sheet objects must implement or inherit methods for each of these generic functions.

**sheet-parent**

*Arguments*: `sheet`

*Summary*: Returns the parent of the sheet `sheet`, or `nil` if the sheet has no parent.

**sheet-children**

*Arguments*: `sheet`

*Summary*: Returns a list of sheets that are the children of the sheet `sheet`. Some sheet classes support only a single child; in this case, the result of `sheet-children` will be a list...
of one element. This function returns objects that reveal CLIM’s internal state; do not modify those objects.

**sheet-adopt-child**  
**[Generic Function]**  
**Arguments:** `sheet child`  
**Summary:** Adds the child sheet `child` to the set of children of the sheet `sheet`, and makes the `sheet` the child's parent. If `child` already has a parent, the **sheet-already-has-parent** error will be signaled.

Some sheet classes support only a single child. For such sheets, attempting to adopt more than a single child will cause the **sheet-supports-only-one-child** error to be signaled.

**sheet-disown-child**  
**[Generic Function]**  
**Arguments:** `sheet child &key (errorp t)`  
**Summary:** Removes the child sheet `child` from the set of children of the sheet `sheet`, and makes the parent of the child be `nil`. If `child` is not actually a child of `sheet` and `errorp` is `t`, then the **sheet-is-not-child** error will be signaled.

**sheet-siblings**  
**[Generic Function]**  
**Arguments:** `sheet`  
**Summary:** Returns a list of all of the siblings of the sheet `sheet`. The siblings are all of the children of `sheet`'s parent excluding `sheet` itself. This function returns fresh objects that may be modified.

**sheet-enabled-children**  
**[Generic Function]**  
**Arguments:** `sheet`  
**Summary:** Returns a list of those children of the sheet `sheet` that are enabled. This function returns fresh objects that may be modified.

**sheet-ancestor-p**  
**[Generic Function]**  
**Arguments:** `sheet putative-ancestor`  
**Summary:** Returns `t` if the the sheet `putative-ancestor` is in fact an ancestor of the sheet `sheet`; otherwise, it returns `nil`.

**raise-sheet**  
**[Generic Function]**  
**Arguments:** `sheet`
**bury-sheet**

**Arguments:** sheet

**Summary:** These functions reorder the children of a sheet by raising the sheet *sheet* to the top or burying it at the bottom. Raising a sheet puts it at the beginning of the ordering; burying it puts it at the end. If sheets overlap, the one that appears “on top” on the display device is earlier in the ordering than the one underneath.

This may change which parts of which sheets are visible on the display device.

**reorder-sheets**

**Arguments:** sheet new-ordering

**Summary:** Reorders the children of the sheet *sheet* to have the new ordering specified by new-ordering. new-ordering is an ordered list of the child sheets; elements at the front of new-ordering are “on top” of elements at the rear.

If new-ordering does not contain all of the children of sheet, the *sheet-ordering-underspecified* error will be signaled. If new-ordering contains a sheet that is not a child of *sheet*, the *sheet-is-not-child* error will be signaled.

**sheet-enabled-p**

**Arguments:** sheet

**Summary:** Returns t if the the sheet *sheet* is enabled by its parent; otherwise, it returns nil. Note that all of a sheet’s ancestors must be enabled before the sheet is viewable.

**sheet-viewable-p**

**Arguments:** sheet

**Summary:** Returns t if the sheet *sheet* and all its ancestors are enabled, and if one of its ancestors is a graft. See Chapter 19, “Ports, Grafts, and Mirrored Sheets,” for further information on grafts.

**sheet-occluding-sheets**

**Arguments:** sheet child
Summary: Returns a list of the sheet child’s siblings that occlude part or all of the region of the child. In general, these are the siblings that are enabled and appear earlier in the sheet sheet’s children. If sheet does not permit overlapping among its children, sheet-occluding-sheets will return nil.

This function returns fresh objects that may be modified.

18.3.2 Sheet Genealogy Classes

Different “mix-in” classes are provided that implement the relationship protocol.

**sheet-parent-mixin** [Class]
Summary: This class is mixed into sheet classes that have a parent.

**sheet-leaf-mixin** [Class]
Summary: This class is mixed into sheet classes that will never have children.

**sheet-single-child-mixin** [Class]
Summary: This class is mixed into sheet classes that have at most a single child.

**sheet-multiple-child-mixin** [Class]
Summary: This class is mixed into sheet classes that may have zero or more children.
18.4  Sheet Geometry

Every sheet has a region and a coordinate system. The region refers to its position and extent on the display device. It is represented by a region object, frequently a rectangle. A sheet’s coordinate system is represented by a coordinate transformation that converts coordinates in its coordinate system to coordinates in its parent’s coordinate system.

18.4.1  Sheet Geometry Functions

sheet-transformation  [Generic Function]
Arguments: sheet

(setf sheet-transformation)  [Generic Function]
Arguments: transformation sheet
Summary: Returns a transformation that converts coordinates in the sheet sheet’s coordinate system into coordinates in its parent’s coordinate system. Using setf on this accessor will modify the sheet’s coordinate system, including moving its region in its parent’s coordinate system. When the the transformation is changed, note-sheet-region-changed is called to notify the sheet of the change.

sheet-region  [Generic Function]
Arguments: sheet

(setf sheet-region)  [Generic Function]
Arguments: region sheet
Summary: Returns a region object that represents the set of points to which the sheet sheet refers. The region is in the sheet’s coordinate system. Using setf on this accessor modifies the sheet’s region. When the region is changed, note-sheet-region-region is called to notify the sheet of the change.

move-sheet  [Generic Function]
Arguments: sheet x y
Summary: Moves the sheet sheet to the new position (x, y). x and y are expressed in the coordinate system of sheet’s parent.
**resize-sheet**

**[Generic Function]**

**Arguments:** `sheet width height`

**Summary:** Resizes the sheet `sheet` to have a new width `width` and height `height`. `width` and `height` are real numbers.

**move-and-resize-sheet**

**[Generic Function]**

**Arguments:** `sheet x y width height`

**Summary:** Moves the sheet `sheet` to the new position (`x`, `y`) and changes its size to the new width `width` and height `height`. `x` and `y` are expressed in the coordinate system of `sheet`'s parent. `width` and `height` are real numbers.

**map-sheet-position-to-parent**

**[Generic Function]**

**Arguments:** `sheet x y`

**Summary:** Applies the sheet `sheet`'s transformation to the point `(x, y)`, returning the coordinates of that point in `sheet`'s parent’s coordinate system.

**map-sheet-position-to-child**

**[Generic Function]**

**Arguments:** `sheet x y`

**Summary:** Inverts `sheet`'s transformation of the point `(x, y)` in `sheet`’s parent’s coordinate system. It returns the coordinates of the point in `sheet`’s coordinate system.

**map-sheet-rectangle*-to-parent**

**[Generic Function]**

**Arguments:** `sheet x1 y1 x2 y2`

**Summary:** Applies `sheet`’s transformation to the bounding rectangle specified by the corner points `(x1, y1)` and `(x2, y2)`, returning the bounding rectangle of the transformed region as four values, `min-x`, `min-y`, `max-x`, and `max-y`. The arguments `x1`, `y1`, `x2`, and `y2` are canonicalized in the same way as for `make-bounding-rectangle`.

**map-sheet-rectangle*-to-child**

**[Generic Function]**

**Arguments:** `sheet x1 y1 x2 y2`

**Summary:** Applies the inverse of the sheet `sheet`’s transformation to the bounding rectangle delimited by the corner points `(x1, y1)` and `(x2, y2)` (represented in `sheet`’s parent’s coordinate system), returning the bounding rectangle of the transformed region as four values, `min-x`, `min-y`, `max-x`, and `max-y`. The arguments `x1`, `y1`, `x2`, and `y2` are canonicalized in the same way as for `make-bounding-rectangle`.

Sheets
child-containing-position [Generic Function]
Arguments: sheet x y
Summary: Returns the topmost enabled direct child of the sheet sheet whose region contains the position (x, y). The position is expressed in sheet’s coordinate system.

children-overlapping-region [Generic Function]
Arguments: sheet region

children-overlapping-rectangle* [Generic Function]
Arguments: sheet x1 y1 x2 y2
Summary: Returns the list of enabled direct children of the sheet sheet whose region overlaps the region region. children-overlapping-rectangle* is a special case of children-overlapping-region in which the region is a bounding rectangle whose corner points are (x1, y1) and (x2, y2). The region is expressed in sheet’s coordinate system. This function returns fresh objects that may be modified.

sheet-delta-transformation [Generic Function]
Arguments: sheet ancestor
Summary: Returns a transformation that is the composition of all the sheet transformations between the sheets sheet and ancestor. If ancestor is nil, this returns the transformation to the root of the sheet hierarchy. If ancestor is not an ancestor of sheet, the sheet-is-not-ancestor error will be signaled.

The computation of the delta transformation is likely to be cached.

sheet-allocated-region [Generic Function]
Arguments: sheet child
Summary: Returns the visible region of the sheet child in the sheet sheet’s coordinate system. If child is occluded by any of its siblings, those siblings’ regions are subtracted (using region-difference) from child’s actual region.

18.4.2 Sheet Geometry Classes

Each of the following implements the sheet geometry protocol in a different manner, according to the sheet’s requirements.

sheet-identity-transformation-mixin [Class]
Summary: This class is mixed into sheet classes whose coordinate systems are identical to that of their parent.

**sheet-translation-mixin** [Class]

Summary: This class is mixed into sheet classes whose coordinate systems are related to that of their parent by a simple translation.

**sheet-y-inverting-transformation-mixin** [Class]

Summary: This class is mixed into sheet classes whose coordinate systems are related to that of their parent by inverting the y coordinate system, and optionally translating by some amount in x and y.

**sheet-transformation-mixin** [Class]

Summary: This class is mixed into sheet classes whose coordinate systems are related to that of their parent by an arbitrary affine transformation.
18.5 Sheet Protocols: Input

CLIM’s windowing substrate provides an input architecture and standard functionality for notifying clients of input that is distributed to their sheets. Input includes such events as the pointer entering and exiting sheets, pointer motion (whose granularity is defined by performance limitations), and pointer button and keyboard events. At this level, input is represented as event objects.

Sheets either participate fully in the input protocol or are mute for input. If any functions in the input protocol are called on a sheet that is mute for input, the `sheet-is-mute-for-input` error will be signaled.

In addition to handling input event, a sheet is also responsible for providing other input services, such as controlling the pointer’s appearance, and polling for current pointer and keyboard state.

Input is processed on a per-port basis.

The event-processing mechanism has three main tasks when it receives an event. First, it must determine to which `client` the event is addressed; this process is called `distributing`. Typically, the client is a sheet, but there are other special-purpose clients to which events can also be dispatched. Next, it formats the event into a standard format, and finally it dispatches the event to the client. A client may then either handle the event synchronously, or it may queue it for later handling by another process.

Input events can be broadly categorized into `pointer events` and `keyboard events`. By default, pointer events are dispatched to the lowest sheet in the hierarchy whose region contains the location of the pointer. Keyboard events are dispatched to the port’s keyboard input focus; the accessor `port-keyboard-input-focus` contains the event client that receives the port’s keyboard events.

18.5.1 Input Protocol Functions

In the functions listed here, the `client` argument is typically a sheet, but it may be another object that supports event distribution, dispatching, and handling.
port-keyboard-input-focus
[Generic Function]
Arguments: port

(setf port-keyboard-input-focus)
[Generic Function]
Arguments: focus port
Summary: Returns the client to which keyboard events are to be dispatched.

distribute-event
[Generic Function]
Arguments: port event
Summary: The event is distributed to the port’s proper client. In general, this will be the keyboard input focus for keyboard events, and the lowest sheet under the pointer for pointer events.

dispatch-event
[Generic Function]
Arguments: client event
Summary: This function is called to inform a client about an event of interest. It is invoked synchronously by whatever process called process-next-event, so many methods for this function will simply queue the event for later handling. Certain classes of clients and events may cause this function to call either queue-event or handle-event immediately, or else to ignore the event entirely.

queue-event
[Generic Function]
Arguments: client event
Summary: Places the event event into the queue of events for the client client.

handle-event
[Generic Function]
Arguments: client event
Summary: Implements the client’s policy with respect to the event. For example, if the programmer wishes to highlight a sheet in response to an event that informs it that the pointer has entered its territory, there would be a method to carry out the policy that specializes the appropriate sheet and event classes.

In addition to queue-event, the queued input protocol handles the following generic functions:

event-read
[Generic Function]
Arguments: client
Summary: Takes the next event out of the queue of events for this client.
**event-read-no-hang**  
Arguments: *client*  
Summary: Takes the next event out of the queue of events for this client. It returns `nil` if there are no events in the queue.

**event-peek**  
Arguments: *client* &optional *event-type*  
Summary: Returns the next event in the queue without removing it from the queue. If *event-type* is supplied, events that are not of that type are first removed and discarded.

**event-unread**  
Arguments: *client* *event*  
Summary: Places the *event* at the head of the *client*'s event queue, to be the event read next.

**event-listen**  
Arguments: *client*  
Summary: Returns `t` if there are any events queued for *client*; otherwise, it returns `nil`.

### 18.5.2 Input Protocol Classes

Most classes of sheets will have one of the following input protocol classes mixed in. Of course, a sheet can always have a specialized method for a specific class of event that will override the default. For example, a sheet may need to have only pointer click events dispatched to itself, and may delegate all other events to some other input client. Such a sheet should have `delegate-sheet-input-mixin` as a superclass, and have a more specific method for `dispatch-event` on its class and `pointer-button-click-event`.

**standard-sheet-input-mixin**  
[Class]  
Summary: This class of sheet provides a method for `dispatch-event` that calls `queue-event` on each device event. Configuration events invoke `handle-event` immediately.

**immediate-sheet-input-mixin**  
[Class]  
Summary: This class of sheet provides a method for `dispatch-event` that calls `handle-event` immediately for all events.
mute-sheet-input-mixin [Class]
Summary: This is mixed into any sheet class that does not handle any input events.

delegate-sheet-input-mixin [Class]
Summary: This class of sheet provides a method for dispatch-event that calls dispatch-event on a designated substitute and the event. The initialization argument :delegate or the accessor delegate-sheet-delegate may be used to set the recipient of dispatched events. A value of nil will cause input events to be discarded.

delegate-sheet-delegate [Generic Function]
Arguments: sheet

(setf delegate-sheet-delegate) [Generic Function]
Arguments: delegate sheet
Summary: This may be set to another recipient of events dispatched to a sheet of class delegate-sheet-input-mixin. If the delegate is nil, events are discarded.
18.6 Standard Device Events

An event is a CLIM object that represents some sort of user gesture (such as moving the pointer or pressing a key on the keyboard) or that corresponds to some sort of notification from the display server. Event objects store such things as the sheet associated with the event, the x and y position of the pointer within that sheet, the key name or character corresponding to a key on the keyboard, and so forth.

Figure 35 shows all the event classes. All classes are indented to the right of their superclasses.

```
event
device-event
  keyboard-event
    key-press-event
    key-release-event
  pointer-event
    pointer-button-event
      pointer-button-press-event
      pointer-button-release-event
      pointer-button-hold-event
    pointer-motion-event
    pointer-enter-event
    pointer-exit-event
  window-event
    window-configuration-event
    window-repaint-event
  timer-event
```

**Figure 35. CLIM Event Classes**

**event** [Protocol Class]

Summary: The protocol class that corresponds to any sort of event. If you want to create a new class that behaves like an event, it should be a subclass of **event**. Subclasses of **event** must obey the event protocol.
All of the event classes are immutable.

**eventp**  
[Function]  
Arguments: *object*  
Summary: Returns `t` if *object* is an event; otherwise, it returns `nil`.

**:timestamp**  
[Initarg]  
Summary: All subclasses of `event` must take a `:timestamp` initarg, which is used to specify the timestamp for the event.

**event-timestamp**  
[Generic Function]  
Arguments: *event*  
Summary: Returns an integer that is a monotonically increasing timestamp for the event `event`. The timestamp must have at least as many bits of precision as a fixnum.

**event-type**  
[Generic Function]  
Arguments: *event*  
Summary: For the event `event`, returns a keyword with the same name as the class name, except stripped of the “-event” ending. For example, the keyword `:key-press` is returned by `event-type` for an event whose class is `key-press-event`.

All event classes must implement methods for `event-type` and `event-timestamp`.

**device-event**  
[Class]  
**:sheet**  
[Initarg]  
**:modifier-state**  
[Initarg]  
Summary: The class that corresponds to any sort of device event. This is a subclass of `event`.  

All subclasses of `device-event` must take the `:sheet` and `:modifier-state` initargs, which are used to specify the sheet and modifier state components for the event.

**event-sheet**  
[Generic Function]  
Arguments: `device-event`  
Summary: Returns the sheet associated with the event `device-event`.
event-modifier-state  [Generic Function]
Arguments: device-event
Summary: Returns a value that encodes the state of all the modifier keys on the keyboard. This will be a mask consisting of the logical-or of +shift-key+, +control-key+, +meta-key+, +super-key+, and +hyper-key+.

All device event classes must implement methods for event-sheet and event-modifier-state.

keyboard-event  [Class]
:key-name  [Initarg]
Summary: The class corresponding to any keyboard event; a subclass of device-event.

All subclasses of keyboard-event must take the :key-name initarg, which is used to specify the key name component for the event.

keyboard-event-key-name  [Generic Function]
Arguments: keyboard-event
Summary: Returns the name of the key pressed or released in a keyboard event. This will be a symbol whose value is port-specific. Key names corresponding to standard characters such as the alphanumerics will be symbols in the keyword package.

keyboard-event-character  [Generic Function]
Arguments: keyboard-event
Summary: Returns the character associated with the event keyboard-event, if there is any.

All keyboard event classes must implement methods for keyboard-event-key-name and keyboard-event-character.

key-press-event  [Class]

key-release-event  [Class]
Summary: The classes corresponding to key press or release events. They are subclasses of keyboard-event.

pointer-event  [Class]
:pointer  [Initarg]
:button  [Initarg]
Sheets

: x [ Initarg ]
:y [ Initarg ]

Summary: The class corresponding to any pointer event. This is a subclass of device-event.

All subclasses of pointer-event must take the :pointer, :button, :x, and :y in-
targs, which are used to specify the pointer object, pointer button, and native x and y position of the pointer at the time of the event. The sheet’s x and y positions are derived from the supplied native x and y positions and the sheet itself.

pointer-event-x
Arguments: pointer-event

pointer-event-y
Arguments: pointer-event

Summary: Returns the x and y position of the pointer at the time the event occurred, in the coordinate system of the sheet that received the event. All pointer events must implement a method for these generic functions.

pointer-event-native-x
Arguments: pointer-event

pointer-event-native-y
Arguments: pointer-event

Summary: Returns the x and y position of the pointer at the time the event occurred, in the pointer’s native coordinate system. All pointer events must implement a method for these generic functions.

pointer-event-pointer
Arguments: pointer-event

Summary: Returns the pointer object to which this event refers.

pointer-event-button
Arguments: pointer-event

Summary: Returns an integer, the number of the pointer button that was pressed. Programs should compare this against the constants +pointer-left-button+, +pointer-middle-button+, and +pointer-right-button+ to see what value was returned.
All pointer event classes must implement methods for `pointer-event-x`, `pointer-event-y`, `pointer-event-native-x`, `pointer-event-native-y`, `pointer-event-pointer`, and `pointer-event-button`.

`pointer-event-shift-mask`  
[Generic Function]
Arguments: `pointer-button-event`
Summary: Returns the state of the keyboard’s shift keys when `pointer-button-event` occurred.

`pointer-button-event`  
[Class]
Summary: The class corresponding to any sort of pointer button event. It is a subclass of `pointer-event`.

`pointer-button-press-event`  
[Class]

`pointer-button-release-event`  
[Class]

`pointer-button-hold-event`  
[Class]
Summary: The classes that correspond to a pointer button press, button release, and click-and-hold events. These are subclasses of `pointer-button-event`.

`pointer-button-click-event`  
[Class]

`pointer-button-double-click-event`  
[Class]

`pointer-button-click-and-hold-event`  
[Class]
Summary: The classes that correspond to a pointer button press followed by (respectively) a button release, another button press, or pointer motion. These are subclasses of `pointer-button-event`. Ports are not required to generate these events.

`pointer-motion-event`  
[Class]
Summary: The class that corresponds to any sort of pointer motion event. This is a subclass of `pointer-event`.

`pointer-enter-event`  
[Class]
**pointer-exit-event** [Class]
Summary: The classes that correspond to a pointer enter or exit event. This is a subclass of [pointer-motion-event](#).

**window-event** [Class]
:region [Initarg]
Summary: The class that corresponds to any sort of windowing event. This is a subclass of [device-event](#).

All subclasses of window-event must take a :region initarg, which is used to specify the damage region associated with the event.

**window-event-region** [Generic Function]
Arguments: window-event
Summary: Returns the region of the sheet that is affected by a window event.

**window-event-native-region** [Generic Function]
Arguments: window-event
Summary: Returns the region of the sheet in native coordinates.

**window-event-mirrored-sheet** [Generic Function]
Arguments: window-event
Summary: Returns the mirrored sheet that is attached to the mirror on which the event occurred.

All window event classes must implement methods for window-event-region, window-event-native-region, and window-event-mirrored-sheet.

**window-configuration-event** [Class]
Summary: The class that corresponds to a window changing its size or position. This is a subclass of window-event.

**window-repaint-event** [Class]
Summary: The class that corresponds to a request to repaint the window. This is a subclass of window-event.

**timer-event** [Class]
Summary: The class that corresponds to a timeout event. This is a subclass of event.
+pointer-left-button+ [Constant]

+pointer-middle-button+ [Constant]

+pointer-right-button+ [Constant]
Summary: Constants that correspond to the left, middle, and right button on a pointing device. pointer-event-button will returns one of these three values.

+shift-key+ [Constant]

+control-key+ [Constant]

+meta-key+ [Constant]

+super-key+ [Constant]

+hyper-key+ [Constant]
Summary: Constants that correspond to the SHIFT, CONTROL, META, SUPER, and HYPER modifier keys being held down on the keyboard. These constants must be powers of 2 so that they can be combined with logical-or and tested with logtest. event-modifier-state will return some combination of these values.

CLIM does not provide default key mappings for META, HYPER, or SUPER modifier keys, as they are keyboard/X-server specific.

key-modifier-state-match-p [Macro]
Arguments: button modifier-state &body clauses
Summary: This macro generates code that will check whether the modifier state modifier-state and the pointer button button match all of the clauses. clauses are implicitly grouped by and. Matching a button or a modifier means that the modifier state indicates that the button or modifier is pressed.

A clause may be one of:

- A pointer button (one of :left, :middle, or :right)
- A modifier key (one of :shift, :control, :meta, :super, or :hyper)
- (and [clause]+)
• (or [clause]+)
• (not clause)
18.7 Sheet Protocols: Output

The output protocol is concerned with the appearance of displayed output on the window associated with a sheet. The sheet output protocol is responsible for providing a means of doing output to a sheet, and for delivering repaint requests to the sheet’s client.

Sheets either participate fully in the output protocol or are mute for output. If any functions in the output protocol are called on a sheet that is mute for output, the sheet-is-mute-for-output error will be signaled.

18.7.1 Mediums and Output Properties

Each sheet retains some output state that logically describes how output is to be rendered on its window. Such information as the foreground and background ink, line thickness, and transformation to be used during drawing are provided by this state. This state may be stored in a medium associated with the sheet itself, may be derived from a parent, or may have some global default, depending on the sheet itself.

If a sheet is mute for output, it is an error to set any of these values.

medium [Protocol Class]
Summary: The protocol class that corresponds to the output state for some kind of sheet. There is no single advertised standard medium class. If you want to create a new class that behaves like a medium, it should be a subclass of medium. Subclasses of medium must obey the medium protocol.

mediump [Function]
Arguments: object
Summary: Returns t if object is a medium; otherwise, it returns nil.

basic-medium [Class]
Summary: The basic class on which all CLIM mediums are built, a subclass of medium. This class is an abstract class intended only to be subclassed, not instantiated.

The following generic functions comprise the basic medium protocol. All mediums must implement methods for these generic functions. Often, a sheet class that supports the output
protocol will implement a “trampoline” method that passes the operation on to sheet-medium of the sheet.

**medium-foreground**

Arguments: *medium*

**(setf medium-foreground)**

Arguments: *ink medium*

Summary: Returns (or sets) the current foreground ink for the medium *medium*. For details, see Subsection 3.1, “CLIM Mediums.”

**medium-background**

Arguments: *medium*

**(setf medium-background)**

Arguments: *ink medium*

Summary: Returns (or sets) the current background ink for the medium *medium*. This is described in detail in Subsection 3.1, “CLIM Mediums.”

**medium-ink**

Arguments: *medium*

**(setf medium-ink)**

Arguments: *ink medium*

Summary: Returns (or sets) the current drawing ink for the medium *medium*. This is described in detail in Subsection 3.1, “CLIM Mediums.”

**medium-transformation**

Arguments: *medium*

**(setf medium-transformation)**

Arguments: *transformation medium*

Summary: Returns (or sets) the user transformation that converts the coordinates presented to the drawing functions by the programmer to the medium *medium*'s coordinate system. By default, it is the identity transformation. This is described in detail in Subsection 3.1, “CLIM Mediums.”
**medium-clipping-region**

Arguments: `medium`

Returns (or sets) the clipping region that encloses all output performed on the medium `medium`. It is returned and set in user coordinates. That is, to convert the user clipping region to medium coordinates, it must be transformed by the value of `medium-transformation`. For example, the values returned by:

```lisp
(let (cr1 cr2)
  ;; Ensure that the sheet’s clipping region
  ;; and transformation will be reset:
  (with-drawing-options
    (sheet :transformation +identity-transformation+
            :clipping-region +everywhere+)
    (setf (medium-clipping-region sheet)
          (make-rectangle* 0 0 10 10))
    (setf (medium-transformation sheet)
          (clim:make-scaling-transformation 2 2))
    (setf cr1 (medium-clipping-region sheet))
    (setf (medium-clipping-region sheet)
          (make-rectangle* 0 0 10 10))
    (setf (medium-transformation sheet) +identity-transformation+)
    (setf cr2 (medium-clipping-region sheet)))
  (values cr1 cr2))
```

are two rectangles. The first one has edges of (0, 0, 5, 5), while the second one has edges of (0, 0, 20, 20).

By default, the user clipping region is the value of `+everywhere+`.

**medium-line-style**

Arguments: `medium`

Returns (or sets) the current line style for the medium `medium`. This is described in detail in Subsection 3.1, “CLIM Mediums.”

**medium-text-style**

Arguments: `medium`

Returns (or sets) the current text style for the medium `medium`.
(setf medium-text-style)  [Generic Function]
Arguments: text-style medium  
Summary: Returns (or sets) the current text style for the medium medium of any textual output that may be displayed on the window. This is described in detail in Subsection 3.1, “CLIM Mediums.”

medium-default-text-style  [Generic Function]
Arguments: medium  

(setf medium-default-text-style)  [Generic Function]
Arguments: text-style medium  
Summary: Returns (or sets) the default text style for output on the medium medium. This is described in detail in Section 3.2, “Using CLIM Drawing Options.”

medium-merged-text-style  [Generic Function]
Arguments: medium  
Summary: Returns the actual text style used in rendering text on the medium medium. It returns the result of:

```
(merge-text-styles (medium-text-style medium)
                     (medium-default-text-style medium))
```

Those components of the current text style that are not nil will replace the defaults from medium’s default text style. Unlike the preceding text style function, medium-merged-text-style is read-only.

18.7.2 Output Protocol Functions

The output protocol functions on mediums (and sheets that support the standard output protocol) include those functions described in Section 2.4, “Graphics Protocols.”

18.7.3 Output Protocol Classes

The following classes implement the standard output protocols.

standard-sheet-output-mixin  [Class]

Sheets
Summary: This class is mixed into any sheet that provides the standard output protocol, such as repainting and graphics.

mute-sheet-output-mixin  [Class]
Summary: This class is mixed into any sheet that provides none of the output protocol.

permanent-medium-sheet-output-mixin  [Class]
Summary: This class is mixed into any sheet that always has a medium associated with it.

temporary-medium-sheet-output-mixin  [Class]
Summary: This class is mixed into any sheet that may have a medium associated with it, but does not necessarily have a medium at any given instant.

18.7.4 Associating a Medium With a Sheet

Before a sheet may be used for output, it must be associated with a medium. Some sheets are permanently associated with mediums for output efficiency; for example, CLIM window stream sheets have mediums that are permanently allocated to windows.

However, many kinds of sheets only perform output infrequently, and therefore do not need to be associated with a medium except when output is actually required. Sheets without a permanently associated medium can be much more lightweight than they otherwise would be. For example, in a program that creates a sheet for the purpose of displaying a border for another sheet, the border sheet receives output only when the window’s shape is changed.

To associate a sheet with a medium, use the macro with-sheet-medium.

Usually CLIM application programmers will not deal with mediums directly. In most cases, panes will automatically be associated with a medium upon creation. The specific medium object is chosen based on the port being used. An exception is when a “special” medium is created and used with sheets that normally default to a different medium.

with-sheet-medium  [Macro]
Arguments: (medium sheet) &body body
Summary: Within the body, the variable medium is bound to the sheet’s medium. If the sheet does not have a medium permanently allocated, one will be allocated, associated with the sheet for the duration of the body, and deallocated when the body has
been exited. The values of the last form of the body are returned as the values of
`with-sheet-medium`.

The `medium` argument is not evaluated, and must be a symbol that is bound to a
medium. `body` may have zero or more declarations as its first forms.

`with-sheet-medium-bound` [Macro]

Arguments: `(sheet medium) &body body`

Summary: `with-sheet-medium-bound` is used to associate the specific medium `medium`
with the sheet `sheet` for the duration of the body `body`. Typically, a single medium
will be allocated and passed to several different sheets that can use the same
medium.

If the sheet already has a medium allocated to it, the new medium will not be
given to the sheet. If the value of `medium` is `nil`, `with-sheet-medium-bound` is
exactly equivalent to `with-sheet-medium`. The values of the last form of the
body are returned as the values of `with-sheet-medium-bound`.

`body` may have zero or more declarations as its first forms.

`sheet-medium` [Generic Function]

Arguments: `sheet`

Summary: Returns the medium associated with the sheet `sheet`. If `sheet` does not have a
medium allocated to it, `sheet-medium` returns `nil`.
18.8 Repaint Protocol

The repaint protocol is the mechanism whereby a program keeps the display up-to-date, reflecting the results of both synchronous and asynchronous events. The repaint mechanism may be invoked by user programs each time through their top-level command loop. It may also be invoked directly or indirectly as a result of events received from the display server host. For example, if a window is on display with another window overlapping it and the second window is buried, a “damage notification” event may be sent by the server. CLIM would then cause a repaint to be executed for the newly-exposed region.

18.8.1 Repaint Protocol Functions

queue-repaint  [Generic Function]
Arguments: sheet region
Summary: Requests that a repaint event for the region region be placed in the input queue of the sheet sheet. A program that reads events out of the queue will be expected to call handle-event for the repaint region; the method for that generic function on repaint events will generally call repaint-sheet.

handle-repaint  [Generic Function]
Arguments: sheet region
Summary: Implements repainting for a given sheet class. It may only be called on a sheet that has an associated medium. sheet and region are as for dispatch-repaint.

repaint-sheet  [Generic Function]
Arguments: sheet medium region
Summary: Recursively causes repainting of the sheet sheet and any of its children that overlap the region region. medium is the medium to use for the repainting; if it is nil, handle-repaint will allocate a medium and associate it with the sheet. handle-repaint will call repaint-sheet on sheet, and then call handle-repaint on all of the children of sheet.
18.8.2 Repaint Protocol Classes

**standard-repainting-mixin**

Summary: Defines a `dispatch-repaint` method that calls `queue-repaint`.

**immediate-repainting-mixin**

Summary: Defines a `dispatch-repaint` method that calls `handle-repaint`.

**mute-repainting-mixin**

Summary: Defines a `dispatch-repaint` method that calls `queue-repaint`, and a method on `repaint-sheet` that does nothing. This means that its children will be recursively repainted when the repaint event is handled.
18.9 Sheet Notification Protocol

The notification protocol allows sheet clients to be notified when a sheet hierarchy is changed. Sheet clients can observe modification events by providing \texttt{after} methods for functions defined by this protocol.

18.9.1 Relationship to Window System Change Notifications

\texttt{note-sheet-grafted} [Generic Function]
Arguments: \texttt{sheet}

\texttt{note-sheet-degrafted} [Generic Function]
Arguments: \texttt{sheet}

\texttt{note-sheet-adopted} [Generic Function]
Arguments: \texttt{sheet}

\texttt{note-sheet-disowned} [Generic Function]
Arguments: \texttt{sheet}

\texttt{note-sheet-enabled} [Generic Function]
Arguments: \texttt{sheet}

\texttt{note-sheet-disabled} [Generic Function]
Arguments: \texttt{sheet}
Summary: These notification functions are invoked when a state change has been made to the sheet \texttt{sheet}.

18.9.2 Sheet Geometry Notifications

\texttt{note-sheet-region-changed} [Generic Function]
Arguments: \texttt{sheet}
note-sheet-transformation-changed

Arguments: sheet

Summary: These notification functions are invoked when the region or transformation of the sheet sheet has been changed. When the regions and transformations of a sheet are changed directly, the client is required to call note-sheet-region-changed or note-sheet-transformation-changed.
<Bold>474CLIM User Guide</Bold>
19.1 Introduction

A sheet hierarchy must be attached to a display server so as to permit input and output. This is managed by the use of objects known as *ports* and *grafts*.
19.2 Ports

A port is a logical connection to a display server. It is responsible for managing display output and server resources and for handling incoming input events. Typically, the programmer will create a single port that will manage all of the windows on the display.

A port is described by a **server path**. A server path is a list whose first element is a keyword that selects the kind of port. The remainder of the server path is a list of alternating keywords and values whose interpretation is specific to the port type.

**port**

[Protocol Class]

Summary: The protocol class that corresponds to a port. If you want to create a new class that behaves like a port, it should be a subclass of **port**. Subclasses of **port** must obey the medium protocol.

**portp**

[Function]

Arguments: object

Summary: Returns t if object is a port; otherwise, it returns nil.

**basic-port**

[Class]

Summary: The basic class on which all CLIM ports are built, a subclass of **port**. This class is an abstract class intended only to be subclassed, not instantiated.

**find-port**

[Function]

Arguments: &key (server-path *default-server-path*)

Summary: Finds a port that provides a connection to the window server addressed by server-path. If no such connection exists, a new connection will be constructed and returned. **find-port** is called automatically by **make-application-frame**.

The following server paths are currently supported on the appropriate platforms:

**:motif**

[Server Path]

**:win32**

[Server Path]

Arguments: &key host display-number screen-id
Summary: Given this server path, **find-port** finds a port for the X server on the given *host*, using the *display-id* and *screen-id*.

On a Unix host, if these values are not supplied, the defaults are derived from the **DISPLAY** environment variable.

**default-server-path**  
*Variable*

Summary: This special variable is used by **find-port** and its callers to default the choice of a display service to locate. Binding this variable in a dynamic context will affect the defaulting of this argument to these functions. This variable will be defaulted according to the environment. In the Unix environment, for example, CLIM tries to set this variable based on the value of the **DISPLAY** environment variable.

**port**  
*Generic Function*

Arguments: *object*

Summary: Returns the port associated with *object*. **port** is defined for all sheet classes (including grafts and streams that support the CLIM graphics protocol), mediums, and application frames. For degrafted sheets or other objects that aren’t currently associated with particular ports, **port** will return **nil**.

**with-port-locked**  
*Macro*

Arguments: *port* &body **body**

Summary:Executes **body** after grabbing a lock associated with the port **port**, which may be a port or any object on which the function **port** works. If *object* currently has no port, **body** will be executed without locking.

**port-server-path**  
*Generic Function*

Arguments: *port*

Summary: Returns the server path associated with the port **port**.

**port-properties**  
*Generic Function*

Arguments: *port* indicator

**setf port-properties**  
*Generic Function*

Arguments: *property* *port* indicator

Summary: These functions provide a port-based property list. They are primarily intended to support users of CLIM who may need to associate certain information with
ports. For example, the implementor of a special graphics package may need to maintain resource tables for each port on which it is used.

**map-over-ports** [Function]

**Arguments:** function

**Summary:** Invokes function on each existing port. function is a function of one argument, the port; it has dynamic extent.

**restart-port** [Generic Function]

**Arguments:** port

**Summary:** In a multi-process Lisp, restart-port restarts the global input processing loop associated with the port port. All pending input events are discarded. Server resources may or may not be released and reallocated during or after this action.

**destroy-port** [Generic Function]

**Arguments:** port

**Summary:** Destroys the connection with the window server represented by the port port. All sheet hierarchies that are associated with port are forcibly degrafted by disowning the children of grafts on port using sheet-disown-child. All server resources utilized by such hierarchies or by any graphics objects on port are released as part of the connection shutdown.
19.3 Grafts

A graft is a special sheet that is directly connected to a display server. Typically, a graft is the CLIM sheet that represents the root window of the display. There may be several grafts that are all attached to the same root window but that have differing coordinate systems.

To display a sheet on a display, it must have a graft as an ancestor. In addition, the sheet and all of its ancestors must be enabled, including the graft. In general, a sheet becomes grafted when it (or one of its ancestors) is adopted by a graft.

sheet-grafted-p [Generic Function]
Arguments: sheet
Summary: Returns t if any of the sheet’s ancestors is a graft; otherwise, it returns nil.

find-graft [Function]
Arguments: &key (port (find-port)) (server-path *default-server-path*) (orientation :default) (units :device)
Summary: Finds a graft that represents the display device on the port port that also matches the other supplied parameters. If no such graft exists, a new graft is constructed and returned. find-graft is called automatically by make-application-frame.

If server-path is supplied, find-graft finds a graft whose port provides a connection to the window server addressed by server-path.

It is an error to provide both port and server-path in a call to find-graft.

orientation specifies the orientation of the graft’s coordinate system. It is one of:

- :default—a coordinate system with its origin is in the upper left hand corner of the display device with y increasing from top to bottom and x increasing from left to right.
- :graphics—a coordinate system with its origin in the lower left hand corner of the display device with y increasing from bottom to top and x increasing from left to right.

units specifies the units of the coordinate system and defaults to :device, which means the device units of the host window system (such as pixels). Other supported values include :inches, :millimeters, and :screen-sized, which means that one unit in each direction is the width and height of the display device.
**graft**  
* [Generic Function]  
Arguments: *object*  
Summary: Returns the graft currently associated with *object*. *graft* is defined for all sheet classes (including streams that support the CLIM graphics protocol), mediums, and application frames. For degrafted sheets or other objects that aren’t currently associated with a particular graft, *graft* will return *nil*.

**map-over-grafts**  
* [Function]  
Arguments: *function* *port*  
Summary: Invokes *function* on each existing graft associated with the port *port*. *function* is a function of one argument, the graft; it has dynamic extent.

**with-graft-locked**  
* [Macro]  
Arguments: *graft* &body *body*  
Summary: Executes *body* after grabbing a lock associated with the graft *graft*, which may be a graft or any object on which the function *graft* works. If *object* currently has no graft, *body* will be executed without locking.  

*body* may have zero or more declarations as its first forms.

**graft-orientation**  
* [Generic Function]  
Arguments: *graft*  
Summary: Returns the orientation of the graft *graft*’s coordinate system. The returned value is either :default or :graphics; see the orientation argument to *find-graft*.

**graft-units**  
* [Generic Function]  
Arguments: *graft*  
Summary: Returns the units of *graft*’s coordinate system, which will be one of :device, :inches, :millimeters, or :screen-sized; see the units argument to *find-graft*.

**graft-width**  
* [Generic Function]  
Arguments: *graft* &key (units :device)  
Summary: Returns the width and height of *graft* (and by extension the associated host window) in the units indicated. *units* may be any of :device, :inches, :millimeters.
or :screen-sized: see the units argument to find-graft. Note that if a unit of
:screen-sized is specified, both of these functions will return a value of 1.

**graft-pixels-per-millimeter**

[Function]

Arguments: graft

**graft-pixels-per-inch**

[Function]

Arguments: graft

Summary: Returns the number of pixels per millimeter or inch of graft. These are only for
convenience; you can write similar functions with graft-width or graft-height.
19.4 Mirrors and Mirrored Sheets

A **mirrored sheet** is a special class of sheet that is attached directly to a window on a display server. Grafts, for example, are always mirrored sheets. However, any sheet anywhere in a sheet hierarchy may be a mirrored sheet. A mirrored sheet will usually contain a reference to a window system object, called a mirror. For example, a mirrored sheet attached to a Motif server might have an X window system object stored in one of its slots. Allowing mirrored sheets at any point in the hierarchy enables the adaptive toolkit facilities.

Since not all sheets in the hierarchy have mirrors, there is no direct correspondence between the sheet hierarchy and the mirror hierarchy. However, on those display servers that support hierarchical windows, the hierarchies must be parallel. If a mirrored sheet is an ancestor of another mirrored sheet, their corresponding mirrors must have a similar ancestor/descendant relationship.

CLIM interacts with mirrors when it must display output or process events. On output, the mirrored sheet closest in ancestry to the sheet on which we wish to draw provides the mirror on which to draw. The mirror’s drawing clipping region is set up to be the intersection of the user’s clipping region and the sheet’s region (both transformed to the appropriate coordinate system) for the duration of the output. On input, events are delivered from mirrors to the sheet hierarchy. The CLIM port must determine which sheet shall receive events based on information such as the location of the pointer.

In both of these cases, we must have a coordinate transformation that converts coordinates in the mirror (so-called “native” coordinates) into coordinates in the sheet and vice-versa.

19.4.1 Mirror Functions

A mirror is the Lisp object that is the handle to the actual toolkit window or gadget.

**sheet-direct-mirror**

[Generic Function]

**Arguments:** `sheet`

**Summary:** Returns the mirror of the sheet `sheet`. If the sheet is not mirrored (or does not currently have a mirror), `sheet-mirror` returns `nil`.

**sheet-mirrored-ancestor**

[Generic Function]

**Arguments:** `sheet`
Summary: Returns the nearest mirrored ancestor of the sheet \textit{sheet}.

\textbf{sheet-mirror} [Generic Function]
Arguments: \textit{sheet}
Summary: Returns the mirror of the sheet \textit{sheet}. If the sheet is not itself mirrored, \textbf{sheet-mirror} returns the direct mirror of its nearest mirrored ancestor. \textbf{sheet-mirror} could be implemented as:

\begin{verbatim}
(defun sheet-mirror (sheet)
  (sheet-direct-mirror (sheet-mirrored-ancestor sheet)))
\end{verbatim}

\textbf{realize-mirror} [Generic Function]
Arguments: \textit{port} \textit{mirrored-sheet}
Summary: Creates a mirror for the sheet \textit{mirrored-sheet} on the port \textit{port}, if it does not already have one. The returned value is the sheet’s mirror.

\subsection*{19.4.2 Internal Interfaces for Native Coordinates}

\textbf{sheet-native-transformation} [Generic Function]
Arguments: \textit{sheet}
Summary: Returns the transformation for the sheet \textit{sheet} that converts sheet coordinates into native coordinates. The object returned by this function is volatile, so programmers must not depend on the components of the object remaining constant.

\textbf{sheet-native-region} [Generic Function]
Arguments: \textit{sheet}
Summary: Returns the region for the sheet \textit{sheet} in native coordinates. The object returned by this function is volatile, so programmers must not depend on the components of the object remaining constant.

\textbf{sheet-device-transformation} [Generic Function]
Arguments: \textit{sheet}
Summary: Returns the transformation used by the graphics output routines when drawing on the mirror. This is the composition of the sheet’s native transformation and the user transformation. The object returned by this function is volatile, so programmers must not depend on the components of the object remaining constant.
**sheet-device-region**  
**[Generic Function]**

Arguments: *sheet*

Summary: Returns the actual clipping region to be used when drawing on the mirror. This is the intersection of the user’s clipping region (transformed by the device transformation) with the sheet’s native region. The object returned by this function is volatile, so programmers must not depend on the components of the object remaining constant.

**invalidate-cached-transformations**  
**[Generic Function]**

Arguments: *sheet*

Summary: *sheet-native-transformation* and *sheet-device-transformation* typically cache the transformations for performance reasons. *invalidate-cached-transformations* clears the cached native and device values for the sheet *sheet’s* transformation and clipping region. It is invoked when a sheet’s native transformation changes, which happens when a sheet’s transformation is changed or when *invalidate-cached-transformations* is called on any of its ancestors.

**invalidate-cached-regions**  
**[Generic Function]**

Arguments: *sheet*

Summary: *sheet-native-region* and *sheet-device-region* typically cache the regions for performance reasons. *invalidate-cached-regions* clears the cached native and device values for the sheet *sheet’s* native clipping region. It is invoked when a sheet’s native clipping region changes, which happens when the clipping region changes or when *invalidate-cached-regions* is called on any of its ancestors.
Appendices
### Appendix A  Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract panes</td>
<td>Panes that are defined only in terms of their programmer interface or behavior. The protocol for an abstract pane specifies the pane in terms of its overall purpose, rather than in terms of its specific appearance or particular interactive details, so that multiple implementations of the pane are possible, each defining its own look and feel. CLIM selects the appropriate pane implementation, based on factors outside the control of the application. See adaptive pane.</td>
</tr>
<tr>
<td>adaptive panes</td>
<td>A subset of the abstract panes (q.v.), adaptive panes are defined to integrate well across all CLIM operating platforms.</td>
</tr>
<tr>
<td>adaptive toolkit</td>
<td>A uniform interface to the standard “widget” or “gadget” toolkits available in many environments. The adaptive toolkit enables a single user interface to adopt the individual look and feel of a variety of host systems.</td>
</tr>
<tr>
<td>adopted</td>
<td>A sheet is said to be adopted when it has a parent sheet. A sheet becomes the child of another sheet by adoption.</td>
</tr>
<tr>
<td>affine transformation</td>
<td>See transformation.</td>
</tr>
<tr>
<td>ancestors</td>
<td>The parent of a sheet or an output record, and all of its ancestors, recursively.</td>
</tr>
</tbody>
</table>
applicable  A presentation translator is said to be applicable when the pointer is pointing to a presentation whose presentation type matches the current input context, and the other criteria for translator matching have been met.

application frame  1. A program that interacts directly with a user to perform some specific task.
2. A Lisp object that holds the information associated with such a program, including the panes of the user interface and application state variables.

area  A region that has two dimensions, length and width.

background ink  Ink that has the same design as the background, so that drawing with it results in erasure.

bounded design  A design that is transparent everywhere beyond a certain distance from a certain point. Drawing a bounded design has no effect on the drawing plane outside that distance.

bounded region  A region that contains at least one point and for which there exists a number, $d$, called the region’s diameter, such that if $p1$ and $p2$ are points in the region, the distance between $p1$ and $p2$ is always less than or equal to $d$.

bounding rectangle  1. The smallest rectangle that surrounds a bounded region and contains every point in the region, and that may contain additional points as well. The sides of a bounding rectangle are parallel to the coordinate axes.
2. A Lisp object that represents a bounding rectangle.

cache value  A value used during incremental redisplay to determine whether or not a piece of output has changed.
<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>callback</strong></td>
<td>A function that informs an application that one of its gadgets has been used.</td>
</tr>
<tr>
<td><strong>children</strong></td>
<td>The direct descendants of a sheet or an output record.</td>
</tr>
<tr>
<td><strong>clip, clipping region</strong></td>
<td>A parent window is said to clip its child when only the part of the child window that overlaps the parent is viewable. A clipping region is that part of a window to which the output of a bitmap or a list of rectangles has been restricted.</td>
</tr>
<tr>
<td><strong>color</strong></td>
<td>1. An object representing the intuitive definition of a color, such as black or red.</td>
</tr>
<tr>
<td></td>
<td>2. A Lisp object that represents a color.</td>
</tr>
<tr>
<td><strong>colored design</strong></td>
<td>A design whose points have color.</td>
</tr>
<tr>
<td><strong>colorless design</strong></td>
<td>A design whose points have no color. Drawing a colorless design uses the default color specified by the medium’s foreground design.</td>
</tr>
<tr>
<td><strong>command</strong></td>
<td>1. The way CLIM represents a user interaction.</td>
</tr>
<tr>
<td></td>
<td>2. A Lisp object that represents a command.</td>
</tr>
<tr>
<td><strong>command name</strong></td>
<td>A symbol that designates a particular command.</td>
</tr>
<tr>
<td><strong>command table</strong></td>
<td>1. A way of collecting and organizing a group of related commands and defining the interaction styles that can be used to invoke those commands.</td>
</tr>
<tr>
<td></td>
<td>2. A Lisp object that represents a command table.</td>
</tr>
</tbody>
</table>
command table designator  A Lisp object that is either a command table or a symbol that names a command table.

completion  A facility provided by CLIM for completing user input over a set of possibilities.

composite pane  A pane that can have a child pane (cf. leaf pane).

compositing  The creation of a design whose appearance at each point is a composite of the appearances of two other designs at that point. There are three varieties of compositing: composing over, composing in, and composing out.

composition  The transformation from one coordinate system to another, then from the second to a third, can be represented by a single transformation that is the composition of the two component transformations. Transformations are closed under composition. Composition is not commutative. Any arbitrary transformation can be built up by composing a number of simpler transformations, but that composition is not unique.

context-dependent input  In the presentation-type system, presentation input is context-dependent because only presentations that match the requirements of the application are accepted as input.

DAG  See directed acyclic graph.

degrafted  Not grafted; see grafted.

descendants  All of the children of a sheet or an output record, and all of their descendants, recursively.
| **design** | An object that represents a way of arranging colors and opacities in the drawing plane. A mapping from an \((x, y)\) pair into color and opacity values. |
| **device transformation** | The transformation used by the graphics output routines when drawing on the mirror. It is the composition of the sheet’s native transformation and the user transformation. |
| **directed acyclic graph** | A graph with no closed paths whose arcs have direction. |
| **disowned** | Having no parent. An object ceases being the child of another object by being disowned. See also adopted. |
| **disabled** | Not enabled; See enabled. |
| **dispatching** | The process of sending an input event to the client to which it is addressed. |
| **display server** | A window system; a screen and its input devices, together with the combination of graphics display, hardware, and X server software that drives them. |
| **displayed output record** | 1. An output record that corresponds to a visible piece of output, such as text or graphics.  
2. The leaves of the output record tree. |
| **distributing** | The process of determining to which client an input event (q.v.) is addressed. |
| **drawing plane** | An infinite two-dimensional plane on which graphical output occurs. A drawing plane contains an arrangement |
of colors and opacities that is modified by each graphical output operation.

**enabled**
A sheet is said to be enabled when its parent has provided space for it. If a sheet and its ancestors are enabled and **grafted** (q.v.), then the sheet will be visible if it occupies a portion of the graft region that is not **clipped** (q.v.) by its ancestors or their siblings.

**event**
1. A significant action, such as a user gesture (e.g., moving the pointer, pressing a pointer button, or typing a key-stroke) or a window configuration change (e.g., resizing a window).
2. A Lisp object that represents an event.

**extended input stream**
A kind of sheet that supports CLIM’s extended input stream protocol, e.g., by supporting a pointing device.

**extended output stream**
A kind of sheet that supports CLIM’s extended output stream protocol, e.g., by supporting a variable line-height text rendering.

**false**
1. The boolean value false.
2. The Lisp object **nil**.

**flipping ink**
1. An ink that interchanges occurrences of two designs, such as might be done by \((\text{xor})\) on a monochrome display.
2. A Lisp object that represents a flipping ink.

**foreground**
The design used when drawing with `+foreground-ink+`. 
| **formatted output** | 1. Output that obeys some high-level constraints on its appearance, such as being arranged in a tabular format or justified within some margins.  
2. The CLIM facility that provides a programmer with the tools to produce such output. |
<p>| <strong>frame</strong> | See application frame. |
| <strong>frame manager</strong> | An object that controls the realization of the look and feel of an application frame. |
| <strong>fully specified</strong> | A text style is said to be fully specified when none of its components are nil and when its size is not relative (that is, neither :smaller nor :larger). |
| <strong>gesture</strong> | Some sort of input action by a user, such as typing a character or clicking a pointer button. |
| <strong>gesture name</strong> | A symbol that designates a particular gesture, e.g., :select is commonly used to indicate a left pointer button click. |
| <strong>graft</strong> | A kind of mirrored sheet (q.v.) that represents a host window, typically a root window. The graft is where the CLIM window hierarchy is “spliced” onto that of the host system. The graft maintains screen invariants, such as the number of pixels per inch. |
| <strong>grafted</strong> | A sheet is said to be grafted when it has an ancestor sheet that is a graft. |</p>
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>highlighting</strong></td>
<td>Making some piece of output stand out, for example by changing its color or drawing a colored line around it. CLIM often highlights the presentation under the pointer to indicate that it is sensitive.</td>
</tr>
</tbody>
</table>
| **immutable**                | 1. (of an object) Having components that cannot be modified once the object has been created, such as regions, colors and opacities, text styles, and line styles.  
2. (of a class) An immutable class is a class all of whose objects are immutable. |
| **implementor**              | A programmer who implements CLIM.                                                                                                           |
| **incremental redisplay**    | 1. Redrawing part of some output (typically the portion that has been changed) while leaving other output as is.  
2. The CLIM facility that implements this behavior.                                                                                         |
<p>| <strong>indirect ink</strong>             | An ink such as <code>+foreground+</code> or <code>+background+</code>, whose value is some other ink.                                                               |
| <strong>ink</strong>                      | Any member of the class <code>design</code> supplied as the :ink argument to a CLIM drawing function.                                                   |
| <strong>input context</strong>            | The input requirements of a particular application. Also an object used to implement context-dependent input (q.v.).                       |
| <strong>input editor</strong>             | The CLIM facility allowing a user to modify typed-in input.                                                                                   |
| <strong>input editing stream</strong>     | A CLIM stream that supports input editing.                                                                                                      |</p>
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input stream designator</td>
<td>A Lisp object that is either an input stream or the symbol <code>t</code>, which is taken to mean <code>*query-io*</code>.</td>
</tr>
<tr>
<td>interactive stream</td>
<td>A stream that both accepts input from and supports output to the user.</td>
</tr>
<tr>
<td>layout</td>
<td>1. The arrangement of panes within an application frame. 2. A kind of pane that is responsible for allocating space to its children, taking their preferred sizes into account.</td>
</tr>
<tr>
<td>leaf pane</td>
<td>A pane that cannot have a child pane (cf. composite pane).</td>
</tr>
<tr>
<td>line style</td>
<td>1. Advice to CLIM’s rendering substrate on how to render a path, such as a line or an unfilled ellipse or polygon. 2. A Lisp object that represents a line style.</td>
</tr>
<tr>
<td>medium</td>
<td>1. A destination for output, having a drawing plane, two designs called the medium’s foreground and background, a transformation, a clipping region, a line style, and a text style. 2. A Lisp object that represents a medium.</td>
</tr>
<tr>
<td>mirror</td>
<td>The host window system object associated with a mirrored sheet, such as a window object on an X11 display server.</td>
</tr>
<tr>
<td>mirrored sheet</td>
<td>A special class of sheet attached directly to a window on a display server. A graft (q.v.) is one kind of a mirrored sheet.</td>
</tr>
</tbody>
</table>
| **mutable**         | 1. A mutable object has components that can be modified (via **setf** accessors) once the object has been created, such as streams and output records.  
|                    | 2. A mutable class is a class all of whose objects are mutable. |
| **non-uniform design** | See **uniform design**. |
| **opacity**         | 1. An object that controls how graphical output appears to cover previous output. Opacity ranges from fully opaque through various levels of translucency to completely transparent.  
|                    | 2. A Lisp object that represents an opacity. |
| **output history**  | The highest level output record for an output recording stream. |
| **output record**   | 1. An object that remembers the output performed to a stream or medium; the result of an output recording.  
<p>|                    | 2. A Lisp object that represents an output record. |
| <strong>output recording</strong> | The process of documenting the output performed to a stream. |
| <strong>output recording stream</strong> | A CLIM stream that supports output recording. |
| <strong>output stream designator</strong> | A Lisp object that is either an output stream or the symbol <strong>t</strong>, which is taken to mean <strong>standard-output</strong>. |
| <strong>pane</strong>            | A specialized sheet that understands issues pertaining to space requirements. A pane appears as the child of a |</p>
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
<td>The direct ancestor of a sheet or an output record.</td>
</tr>
<tr>
<td>path</td>
<td>A region that has one dimension, length.</td>
</tr>
<tr>
<td>patterning</td>
<td>The process of creating a bounded rectangular arrangement of designs, such as a checkerboard. A pattern is a design created by this process.</td>
</tr>
<tr>
<td>pixmap</td>
<td>An “off-screen window,” that is, a sheet that can be used for graphical output but that is not visible on any display device.</td>
</tr>
</tbody>
</table>
| point           | 1. A region that has dimensionality 0; i.e., has only a position.  
|                 | 2. A Lisp object that represents a point.                                   |
| pointer         | A physical device used for pointing, such as a mouse, or the cursor that shows the position of the mouse on the screen. |
| pointer documentation | Text that describes something about what the mouse is over; the mechanism for displaying that text, which appears in a pointer-documentation-pane. |
| port            | An abstract connection to a display server that is responsible for managing host display server resources and for processing input events received from the host display server. |
position 1. A location on a plane such as the abstract drawing plane.
2. Two real number values $x$ and $y$ that represent a location.

presentation 1. An association between an object and a presentation type with some output on a output recording stream.
2. A Lisp object that represents a presentation.

presentation tester A predicate that restricts the applicability of a presentation translator.

presentation translator A mapping from an object of one presentation type, input context, and gesture to an object of another presentation type.

presentation type 1. A description of a class of presentations.
2. An extension to CLOS that implements this.

presentation type specifier A Lisp object used to specify a presentation type.

programmer A person who writes application programs using CLIM.

protocol class An “abstract” class having no methods or slots that is used to indicate that a class obeys a certain protocol. For example, all classes that inherit from the bounding-rectangle class obey the bounding rectangle protocol.

rectangle 1. A four-sided polygon whose sides are parallel to the coordinate axes.
2. A Lisp object that represents a rectangle.
| **redisplay** | See incremental redisplay. |
| **reflection** | A transformation in which each point is mapped to a symmetric point with respect to a line; reflections preserve length and magnitude of angles. |
| **region** | 1. A set of mathematical points in a plane; a mapping from an \((x, y)\) pair into either \texttt{t} or \texttt{nil} (meaning member or not a member, respectively, of the region). In CLIM, all regions include their boundaries (i.e., are closed) and have infinite resolution.  
2. A Lisp object that represents a region. |
| **region set** | 1. A “compound” region, that is, a region consisting of several other regions related by one of the operations union, intersection, or difference.  
2. A Lisp object that represents a region set. |
| **rendering** | The process of drawing a shape (such as a line or a circle) on a display device. Rendering is an approximate process, as abstract shapes exist in a continuous coordinate system having infinite precision, whereas display devices must necessarily draw discrete points having some measurable size. |
| **replaying** | The process of redrawing a set of output records. |
| **repainting** | Redrawing a window that has been exposed or uncovered. |
| **rotation** | A transformation that moves all points around the center of rotation. A rotation maintains each point’s distance to the center of rotation and to each other. |
**sensitive**

A presentation is sensitive if some action will take place when the user clicks on it with the pointer. Sensitive presentations are usually highlighted.

**server path**

A server path is a list used to specify a port. The first element is a keyword that selects the kind of port. The remainder of the server path is a list of alternating keywords and values whose interpretation is port-type-specific.

**sheet**

1. A visible interface object that specifies the destination for graphical output. A sheet has properties including a coordinate system, a region, an enabled flag, an event handler, an output state, and optionally a parent, a transformation, and children.
2. A Lisp class, a subclass of **bounding-rectangle**, that represents a sheet.

**sheet region**

The area within a sheet’s coordinate system where actions take place, that is, a clipping region for output and input. This typically corresponds to the visible region of the sheet on the display.

**sheet transformation**

Describes how points in a sheet’s coordinate system are mapped onto points in its parents’ coordinate system.

**solid design**

A design comprised of completely opaque and/or completely transparent points. A solid design can be opaque at some points and transparent at others.

**spread point argument**

Functions that take spread point arguments take a pair of arguments that correspond to the x and y coordinates of the point. Such functions have an asterisk in their name: **draw-line***. Cf. **structured point argument**.
stencil  A kind of pattern that contains only opacities.

stencil opacity  The opacity at one point in a design that would result from drawing the design onto a fictitious medium whose drawing plane is initially completely transparent black (opacity and all color components are zero), and whose foreground and background are both opaque black. The stencil opacity of an opacity is simply its value.

stream  A kind of sheet that implements the stream protocol (such as maintaining a text cursor).

structured point argument  Functions that take structured point arguments take the argument as a single point object. Cf. spread point argument.

text cursor  The visible underscore or block that shows where user input will appear on the command line or in a text editor. Cf. pointer, the cursor that tracks the movement of the mouse.

text style  1. A description of how textual output should appear with respect to its font family, face code, and size. 2. A Lisp object that represents a text style.

tiling  The process of repeating a rectangular portion of a design throughout the drawing plane. A tile is a design so created.

trampoline  A function is said to trampoline when the only thing it does is call the corresponding function in the object’s superclass.
transformation 1. A mapping from one coordinate system onto another that preserves straight lines, such as a translation, scaling, rotation, or reflection.
   2. A Lisp object that represents a transformation.

translation A transformation in which the new coordinate axes are parallel to the original ones. A translation preserves length, angle, and orientation of all geometric entities.

translucent design A design that is not solid, that is, that has at least one point with an opacity somewhere between completely opaque and transparent.

true, t 1. The boolean value true; not false.
   2. Any Lisp object that is not nil.

unbounded design A design that has at least one point of non-zero opacity arbitrarily far from the origin. Drawing an unbounded design affects the entire drawing plane.

unbounded region A region that either contains no points or contains points arbitrarily far apart.

uniform design A design that has the same color and opacity at every point in the drawing plane. Uniform designs are always unbounded, unless they are completely transparent.

unique id During incremental redisplay, the unique id is an object used to identify each piece of output. The output named by the unique id will often have a cache value associated with it.
**user**
A person using an application program written with CLIM.

**user transformation**
A transformation that is apparent to the user (as opposed to an internal transformation, such as that used to deal with disparate display devices). A user transformation may be set by the user and is associated with a medium.

**view**
1. A way of displaying data (e.g., as numbers, bars in a bar graph, etc.).
2. A Lisp object that represents a view.

**viewport**
The portion of the drawing plane of a sheet’s medium that is visible on a display device.

**volatile**
An immutable object is said to be volatile if it has components that cannot be modified by the programmer at the protocol level, but which may be modified internally by CLIM. Volatile objects reflect the internal state of CLIM.

**window**
A pane that is a subclass of **clim-stream-pane**. A window is another name for a stream pane or other pane that supports the stream protocol.
B.1 Setting Up Your Packages to Use CLIM

You can set up your user packages to use CLIM as follows:

```
(in-package :user)

(defpackage "FOO"
  (:use :clim-lisp :clim ))
```

The package :clim-lisp is a version of the :lisp package that shadows some of the Common Lisp symbols. The :clim package is the exported CLIM interface.

B.2 CLIM Packages

LispWorks and Liquid CLIM both make use of the following packages:

- CLIM-USER—This is analogous to the USER package. It uses CLIM and CLIM-LISP.
- COMMON-LISP-USER—The USER package has been renamed.
- COMMON-LISP—The LISP package has been renamed.
- CLIM-INTERNALS—For internal use only.
- CLIM-SILICA—For internal use only.
- CLIM-SYS—Exported, portable Lisp system utilities not officially part of CLIM, such as multitasking, resources, etc.
- CLIM—The official, exported CLIM functionality.
■ CLIM-LISP—CLIM’s carefully constructed LISP package. It imports, shadows, and adds symbols to create a portable namespace for CLIM.

■ CLIM-DEMO—An example of a newly-defined, user-level package that uses CLIM and CLIM-LISP.

■ CLIM-UTILS—Contains unexported Lisp utilities used by the Lisp system.

The official way to make a package for CLIM is as follows:

(defpackage "MY-CLIM-PACKAGE" (:use :CLIM-LISP :CLIM :CLIM-SYS))
Appendix C  The CLIM-SYS Package

The CLIM-SYS package contains useful, “system-like” utilities such as resources and multi-processing primitives. These utilities are neither part of Common Lisp nor conceptually within the province of CLIM itself.

All of the symbols documented in this appendix are accessible as external symbols in the CLIM-SYS package.

C.1  Resources

CLIM provides a facility called resources that allows you to reuse objects. A resource describes how to construct an object, how to initialize and deinitialize it, and how an object should be selected from the resource of objects based on a set of parameters.

`defresource`  
[Macro]  
Arguments: name parameters &key constructor initializer deinitializer matcher initial-copies  
Summary: Defines a resource named name, which must be a symbol. parameters is a lambda-list giving names and default values (for optional and keyword parameters) of parameters to an object of this type.  

constructor is a form that creates an object; it is called when someone tries to allocate an object from the resource and no suitable free objects exist. The constructor form can access the parameters as variables. This argument is required.

initializer is a form used to initialize an object gotten from the resource. It can access the parameters as variables, and also has access to a variable
called name, which is the object to be initialized. The initializer is called both on newly created objects and objects that are being reused.

deinitializer is a form used to deinitialize an object when it is about to be returned to the resource. It can access the parameters as variables, and also has access to a variable called name, the object to be deinitialized. It is called whenever an object is deallocated back to the resource, but is not called by clear-resource. Deinitializers are typically used to clear references to other objects.

matcher is a form that ensures that an object in the resource “matches” the specified parameters, which it can access as variables. The matcher also has access to a variable called name, which is the object in the resource being matched against. If no matcher is supplied, the system remembers the values of the parameters (including optional ones that defaulted) that were used to construct the object, and assumes that it matches those particular values for all time. This comparison is done with equal. The matcher returns t if there is a match, and otherwise nil.

initial-copies specifies the number of objects to be initially put into the resource. It must be an integer or nil (the default), meaning that no initial copies should be made. If initial copies are made and there are parameters, all the parameters must be optional; the initial copies will then have the default values of the parameters.

using-resource [Macro]
Arguments: (variable name &rest parameters) &body body
Summary: The forms in body are evaluated with variable bound to an object allocated from the resource named name, using the parameters given by parameters. The parameters (if any) are evaluated, but name is not.

After body has been evaluated, using-resource returns the object in variable to the resource. If a form in the body sets variable to nil, the object is not returned to the resource. Otherwise, the body should not change the value of variable.

allocate-resource [Function]
Arguments: name &rest parameters
Summary: Allocates an object from the resource name, using the parameters given by para-meters. name must be a symbol naming a resource. It returns the allocated object.
**deallocate-resource**

*Function*

**Arguments:** name object

**Summary:** Returns the object object to the resource name. name must be a symbol naming a resource. object must be an object originally allocated from the same resource.

**clear-resource**

*Function*

**Arguments:** name

**Summary:** Clears the resource named name, that is, removes all of the resourced object from the resource. name must be a symbol that names a resource.

**map-resource**

*Function*

**Arguments:** function name

**Summary:** Calls function once on each object in the resource named name. function is a function of three arguments, the object, a boolean value that is t if the object is in use or nil if it is free, and name. function has dynamic extent.

---

## C.2 Multi-Processing

Most Lisp implementations provide some form of multi-processing. CLIM provides a set of functions that implement a uniform interface to the multi-processing functionality.

**make-process**

*Function*

**Arguments:** function &key name

**Summary:** Creates a process named name. The new process will evaluate the function function. On systems that do not support multi-processing, make-process will signal an error.

**destroy-process**

*Function*

**Arguments:** process

**Summary:** Terminates the process process. process is an object returned by make-process.
current-process [Function]
Summary: Returns the currently running process, which will be the same kind of object as would be returned by make-process.

all-processes [Function]
Summary: Returns a sequence of all of the processes.

process-wait [Function]
Arguments: reason predicate
Summary: Causes the current process to wait until predicate returns t. reason is a string or symbol that gives an explanation for the wait. On systems that do not support multi-processing, process-wait will loop until predicate returns t.

process-wait-with-timeout [Function]
Arguments: reason timeout predicate
Summary: Causes the current process to wait until predicate returns t or the number of seconds specified by timeout has elapsed. reason is a string or symbol that gives an explanation for the wait. On systems that do not support multi-processing, process-wait-with-timeout loops until predicate returns t or the timeout elapses.

process-yield [Function]
Summary: Allows other processes to run. On systems that do not support multi-processing, this does nothing.

process-interrupt [Function]
Arguments: process function
Summary: Interrupts the process process and causes it to evaluate the function function. On systems that do not support multi-processing, this is equivalent to fun-call'ing function.

without-scheduling [Macro]
Arguments: &body body
Summary: Evaluates body in a context that is guaranteed to be free from interruption by other processes. On systems that do not support multi-processing, without-scheduling is equivalent to progn.
C.3  Locks

In the course of multi-processing, it is important to ensure that two processes do not modify the same data simultaneously. This is done by creating a lock, which is an extra memory location in a data structure that can be checked to determine whether that structure is in use. If the value of a lock is nil, no process is using the data structure; otherwise, the value should be a process that is currently using the structure.

The following symbols for creating locks will work with all CLIM ports.

**with-lock-held**  
*Macro*

Arguments: *(place &optional state) &body body*

Summary: Evaluates body with the lock named by place. place is a reference to a lock created by make-lock. state specifies the process to store in the place location; the default value is the value of the variable *current-process*.

On systems that do not support locking, with-lock-held is equivalent to progn.

**make-lock**  
*Function*

Arguments: &optional name

Summary: Creates a lock whose name is name. On systems that do not support locking, this will return a new list of one element, nil.

**with-recursive-lock-held**  
*Macro*

Arguments: *(place &optional state) &body body*

Summary: Evaluates body with the recursive lock named by place. place is a reference to a recursive lock created by make-recursive-lock. A recursive lock differs from an ordinary lock in that a process that already holds the recursive lock can call with-recursive-lock-held on the same lock without blocking.

On systems that do not support locking, with-recursive-lock-held is equivalent to progn.

**make-recursive-lock**  
*Function*

Arguments: &optional name

Summary: Creates a recursive lock whose name is name. On systems that do not support locking, this will return a new list of one element, nil.
C.4 Multiple-Value Setf

CLIM provides a facility, sometimes referred to as setf*, that allows setf to be used on “places” that name multiple values. For example, output-record-position returns the position of an output record as two values that correspond to the x and y coordinates. In order to change the position of an output record, the programmer would like to invoke (setf output-record-position). However, setf only takes a single value with which to modify the specified place. The setf* facility provides a “multiple-value” version of setf that allows an expression that returns multiple values to be used to update the specified place.

`defgeneric*` [Macro]
Arguments: `name lambda-list &body options`
Summary: Defines a setf* generic function named name. The last argument in lambda-list is intended to be class specialized, just as normal setf generic functions are. options are as for defgeneric.

`defmethod*` [Macro]
Arguments: `name (method-qualifier* specialized-lambda-list &body body)`
Summary: Defines a setf* method for the generic function name. The last argument in specialized-lambda-list is intended to be class specialized, just as normal setf methods are. (method-qualifier)* and body are as for defgeneric. For example, output-record-position and its setf* method for a class called sample-output-record might be defined as follows:

```
(defgeneric output-record-position (record)
  (declare (values x y)))
(defgeneric* (setf output-record-position) (x y record))
(defmethod output-record-position ((record sample-output-record))
  (with-slots (x y)
    (values x y)))
(defmethod* (setf output-record-position)
  (nx ny (record sample-output-record))
  (with-slots (x y)
    {setf x nx
     y ny}))
```

The position of such an output record could then be changed as follows:

```
(setf (output-record-position record) (values nx ny))
```
(setf (output-record-position record1)
    (output-record-position record2))
Appendix D  Common Lisp Streams

CLIM performs all of its character-based input and output operations on objects called streams. Streams are divided into two layers, the basic stream protocol, which is character-based and compatible with existing Common Lisp programs, and the extended stream protocol, which introduces extended gestures such as pointer gestures and synchronous window-manager communication.

This appendix describes the basic stream-based input and output protocol used by CLIM. The protocol is taken from the STREAM-DEFINITION-BY-USER proposal to the X3J13 committee, made by David Gray of TI. This proposal was not accepted as part of the ANSI Common Lisp language definition, but CLIM provides an implementation of the basic output stream facilities. For a description of the CLIM specialization of this protocol, see Chapter 15 "Extended Stream Input Facilities".

Note that in CLIM, many of the generic functions described in the following sections are called by Common Lisp stream functions. For example, force-output calls stream-force-output.

D.1  Stream Classes

The following classes are used as superclasses of user-defined stream classes. They are not intended to be directly instantiated; they just provide places to hang default methods.

The predicate functions may return t for other objects that are not members of the fundamental-stream class (or its subclasses) but that claim to serve as streams.
**fundamental-stream**  
Summary: This class is the base class for all CLIM streams. It is a subclass of `stream` and of `standard-object`.

**streamp**  
Arguments: `object`  
Summary: Returns `t` if `object` is a member of the class `fundamental-stream`.

**fundamental-input-stream**  
Summary: A subclass of `fundamental-stream` that implements input streams.

**input-stream-p**  
Arguments: `object`  
Summary: Returns `t` when called on any object that is a member of the class `fundamental-input-stream`.

**fundamental-output-stream**  
Summary: A subclass of `fundamental-stream` that implements output streams.

**output-stream-p**  
Arguments: `object`  
Summary: Returns `t` when called on any object that is a member of the class `fundamental-output-stream`.

Bidirectional streams can be formed by including both `fundamental-input-stream` and `fundamental-output-stream`.

**fundamental-character-stream**  
Summary: A subclass of `fundamental-stream`. It provides a method for `stream-element-type`, which returns `character`.

**fundamental-binary-stream**  
Summary: A subclass of `fundamental-stream`. Any instantiable class that includes this needs to define a method for `stream-element-type`.

**fundamental-character-input-stream**  
Summary: A subclass of `fundamental-stream`.
Summary: A subclass of fundamental-input-stream and fundamental-character-stream, providing default methods for generic functions for character input.

fundamental-character-output-stream [Class]
Summary: A subclass of fundamental-output-stream and fundamental-character-stream, providing default methods for generic functions for character output.

fundamental-binary-input-stream [Class]
Summary: A subclass of fundamental-input-stream and fundamental-binary-stream.

fundamental-binary-output-stream [Class]
Summary: A subclass of fundamental-output-stream and fundamental-binary-stream.

D.2 Basic Stream Functions

These generic functions must be defined for all stream classes.

stream-element-type [Generic Function]
Arguments: stream
Summary: This existing Common Lisp function is made generic, but otherwise behaves the same. Class fundamental-character-stream provides a default method that returns character.

open-stream-p [Generic Function]
Arguments: stream
Summary: This function is made generic. A default method is provided by class fundamental-stream that returns t if close has not been called on the stream.
close
[Generic Function]
Arguments: stream &key abort
Summary: The existing Common Lisp function close is redefined to be a generic function, but otherwise it behaves the same. The default method provided by the class fundamental-stream sets a flag used by open-stream-p. The value returned by close will be as specified by the X3J13 issue closed-stream-operations.

stream-pathname
[Generic Function]
Arguments: stream

stream-truename
[Generic Function]
Arguments: stream
Summary: These are used to implement pathname and truename. There is no default method because these are not valid for all streams.

D.3 Character Input

A character input stream can be created by defining a class that includes fundamental-character-input-stream and defining methods for the following generic functions.

stream-read-char
[Generic Function]
Arguments: stream
Summary: Reads one character from stream, and returns either a character object or the symbol :eof if the stream is at end-of-file. There is no default method for this generic function, so every subclass of fundamental-character-input-stream must define a method.

stream-unread-char
[Generic Function]
Arguments: stream character
Summary: Undoes the last call to stream-read-char, as in unread-char, and returns nil. There is no default method for this, so every subclass of fundamental-character-input-stream must define a method.
**stream-read-char-no-hang**

[Generic Function]

Arguments: *stream*

Summary: Returns either a character, or **nil** if no input is currently available, or **:eof** if end-of-file is reached. This is used to implement **read-char-no-hang**. The default method provided by **fundamental-character-input-stream** simply calls **stream-read-char**; this is sufficient for file streams, but interactive streams should define their own method.

**stream-peek-char**

[Generic Function]

Arguments: *stream*

Summary: Returns either a character or **:eof** without removing the character from the stream’s input buffer. This is used to implement **peek-char**; this corresponds to peek-type of **nil**. The default method calls **stream-read-char** and **stream-unread-char**.

**stream-listen**

[Generic Function]

Arguments: *stream*

Summary: Returns **t** if there is any input pending on *stream*; otherwise, it returns **nil**. This is used by **listen**. The default method uses **stream-read-char-no-hang** and **stream-unread-char**. Most streams should define their own method, as it will usually be trivial and will generally be more efficient than the default method.

**stream-read-line**

[Generic Function]

Arguments: *stream*

Summary: Returns a string as the first value, and **t** as the second value if the string was terminated by end-of-file instead of the end of a line. This is used by **read-line**. The default method uses repeated calls to **stream-read-char**.

**stream-clear-input**

[Generic Function]

Arguments: *stream*

Summary: Clears any buffered input associated with *stream*, and returns **nil**. This is used to implement **clear-input**. The default method does nothing.
D.4 Character Output

A character output stream can be created by defining a class that includes `fundamental-character-output-stream` and defining methods for the following generic functions.

**stream-write-char**

Arguments: stream character

Summary: Writes character to stream, and returns character as its value. Every subclass of `fundamental-character-output-stream` must have a method defined for this function.

**stream-line-column**

Arguments: stream

Summary: This function returns the column number where the next character will be written on stream, or `nil` if that is not meaningful. The first column on a line is numbered 0. This function is used in the implementation of `pprint` and the `format ~T` directive. Every character output stream class must define a method for this, although it is permissible for it to always return `nil`.

**stream-start-line-p**

Arguments: stream

Summary: Returns `t` if stream is positioned at the beginning of a line; otherwise, it returns `nil`. It is permissible to always return `nil`. This is used in the implementation of `fresh-line`.

Note that while a value of 0 from `stream-line-column` also indicates the beginning of a line, there are cases where `stream-start-line-p` can be meaningfully implemented when `stream-line-column` cannot. For example, for a window using variable-width characters, the column number isn’t very meaningful, but the beginning of the line does have a clear meaning. The default method for `stream-start-line-p` on class `fundamental-character-output-stream` uses `stream-line-column`, so if that is defined to return `nil`, a method should be provided for either `stream-start-line-p` or `stream-fresh-line`.

stream-write-string  [Generic Function]
Arguments: stream string &optional (start 0) end
Summary: Writes the string string to stream. If start and end are supplied, they specify what part of string to output. string is returned as the value. This is used by write-string. The default method provided by fundamental-character-output-stream uses repeated calls to stream-write-char.

stream-terpri  [Generic Function]
Arguments: stream
Summary: Writes an end-of-line character on stream and returns nil. This is used by terpri. The default method does stream-write-char of #\newline.

stream-fresh-line  [Generic Function]
Arguments: stream
Summary: Writes an end-of-line character on stream only if the stream is not at the beginning of the line. This is used by fresh-line. The default method uses stream-start-line-p and stream-terpri.

stream-finish-output  [Generic Function]
Arguments: stream
Summary: Ensures that all the output sent to stream has reached its destination, and only then return nil. This is used by finish-output. The default method does nothing.

stream-force-output  [Generic Function]
Arguments: stream
Summary: Like stream-finish-output, except that it may return nil without waiting for the output to complete. This is used by force-output. The default method does nothing.

stream-clear-output  [Generic Function]
Arguments: stream
Summary: Aborts any outstanding output operation in progress and returns nil. This is used by clear-output. The default method does nothing.

stream-advance-to-column  [Generic Function]
Arguments: stream column

<Bold>522CLIM User Guide
Summary: Writes enough blank space on stream so that the next character will be written at the position specified by column. Returns t if the operation is successful, or nil if it is not supported for this stream. This is intended for use by pprint and format ~T. The default method uses stream-line-column and repeated calls to stream-write-char with a \Space character; it returns nil if stream-line-column returns nil.

D.5 Binary Streams

Binary streams can be created by defining a class that includes either fundamental-binary-input-stream or fundamental-binary-output-stream (or both) and defining a method for stream-element-type and for one or both of the following generic functions.

stream-read-byte [Generic Function]
Arguments: stream
Summary: Returns either an integer or the symbol :eof if stream is at end-of-file. This is used by read-byte.

stream-write-byte [Generic Function]
Arguments: stream integer
Summary: Writes integer to stream and returns integer as the result. This is used by write-byte.

D.6 Hardcopy Streams in CLIM

CLIM supports hardcopy output through the macro with-output-to-postscript-stream.
with-output-to-postscript-stream  [Macro]
Arguments: (stream-var file-stream &key (:display-device clim::*postscript-device*) :header-comments :multi-page) &body body
Summary: Within body, stream-var is bound to a stream that produces PostScript code.

The following example writes a PostScript program that draws a square, a circle, and a triangle to a file named icons-of-high-tech.ps.

(defun print-icons-of-high-tech-to-file ()
  (with-open-file
    (file-stream "icons-of-high-tech.ps" :direction :output)
    (clim:with-output-to-postscript-stream
      (stream file-stream)
      (let* ((x1 150) (y 250) (size 100)
        (x2 (+ x1 size))
        (radius (/ size 2))
        (base-y (+ y (/ (* size (sqrt 3)) 2))))
        (clim:draw-rectangle* stream
          (- x1 size) (- y size)
          x1 y)
        (clim:draw-circle* stream
          (+ x2 radius) (- y radius)
          radius)
        (clim:draw-triangle* stream
          (+ x1 radius) y
          x1 base-y
          x2 base-y))))

The second example uses multi-page mode to draw a graph of the superclasses of the class window-stream by writing a PostScript program to the file some-pathname.

(with-open-file (file some-pathname :direction :output)
  (clim:with-output-to-postscript-stream
    (stream file :multi-page t)
    (clim:format-graph-from-root
      (clos:find-class 'clim-internals::window-stream)
      #'(lambda (object s)
        (write-string (string (clos:class-name object)) s))
      #'clos:class-direct-superclasses
      :stream stream)))

Note that with-output-to-postscript-stream is defined in the loadable module "clim-postscript". See section 1.5 on page 15 for details of how to load CLIM and associated modules.

<Bold>524CLIM User Guide
Appendix E  Windows

E.1  Window Stream Operations in CLIM

A window is a CLIM stream pane that supports all window and stream operations. Windows are primarily included for compatibility with CLIM 1.1, although it is sometimes useful to be able to perform operations directly on a window.

E.1.1  Clearing and Refreshing the Drawing Plane

CLIM supports the following operators for clearing and refreshing the drawing plane:

**window-clear**  [Generic Function]
Arguments: window
Summary: Clears the entire drawing plane of window, filling it with the background ink.

**window-erase-viewport**  [Generic Function]
Arguments: window
Summary: Clears the visible part of window’s drawing plane, filling it with background ink.

**window-refresh**  [Generic Function]
Arguments: window
Summary: Clears the visible part of the drawing plane of window, and then replays all of the output records in the visible part of the drawing plane.
E.1.2 The Viewport and Scrolling

A window stream **viewport** is the region of the drawing plane that is visible through the window. You can change the viewport by scrolling or by reshaping the window. The viewport does not change if the window is covered by another window (that is, the viewport is the region of the drawing plane that would be visible if the window were on top).

A window stream has an **end-of-line** action and an **end-of-page** action, which control what happens when the cursor position moves out of the viewport (**with-end-of-line-action** and **with-end-of-page-action**, respectively).

E.1.3 Viewport and Scrolling Operators

```lisp
(window-viewport*) [Generic Function]
Arguments: window
Summary: Returns a region that is the window’s current viewport, an object of type area. (See Subsection 10.2.3, “Composite Pane Generic Functions,” for the generic function pane-viewport, which returns a viewport.)
```

```lisp
(window-set-viewport-position*) [Generic Function]
Arguments: window x y
Summary: Moves the top-left corner of the window’s viewport. Use this to scroll a window.
```
E.2 Functions for Operating on Windows Directly

You can use `open-window-stream` to give you a CLIM window without incorporating it into a frame. After calling `open-window-stream`, call `window-expose` to make the resulting window stream visible.

The following operators are available for manipulating the CLIM primitive layer for window streams.

`open-window-stream` [Function]
Arguments: &key port left top right bottom width height borders console default-text-margin default-text-style depth display-device-type draw-p end-of-line-action end-of-page-action initial-cursor-visibility input-buffer label name output-record record-p save-under scroll-bars stream-background stream-foreground text-cursor text-margin viewport vsp window-class
Summary: A handy function for creating a CLIM window, but one not normally used. Most often windows are created by an application frame or by the menu and dialog functions.

`window-parent` [Generic Function]
Arguments: window
Summary: Returns the window that is the parent (superior) of window.

`window-children` [Generic Function]
Arguments: window
Summary: Returns a list of all of the windows that are children (inferiors) of window.

`window-label` [Generic Function]
Arguments: window
Summary: Returns the label (a string) associated with window, or `nil` if there is none.

`with-input-focus` [Macro]
Arguments: (stream) &body body
Summary: Temporarily gives the keyboard input focus to the given window, most often an interactor pane. By default, a frame will give the input focus to the `frame-query-io` pane.

The following functions are most usefully applied to the top level sheet of a frame. For example: `(clim:frame-top-level-sheet clim:*application-frame*)`.

- **window-expose** [Generic Function]
  Arguments: `window`
  Summary: Makes the `window` visible on the screen.

- **window-stack-on-bottom** [Generic Function]
  Arguments: `window`
  Summary: Puts the `window` underneath all other windows that it overlaps.

- **window-stack-on-top** [Generic Function]
  Arguments: `window`
  Summary: Puts the `window` on top of all the others it overlaps so that you can see all of it.

- **window-visibility** [Generic Function]
  Arguments: `stream`
  Summary: A predicate that returns `t` if the window is visible. You can use `setf` on `window-visibility` to expose or deexpose the window.

The following operators can be applied to a window to determine its position and size.

- **window-inside-edges** [Generic Function]
  Arguments: `window`
  Summary: Returns four values, the coordinates of the left, top, right, and bottom inside edges of the window `window`.

- **window-inside-left** [Function]
  Arguments: `window`
  Summary: Returns the coordinate of the left edge of the window `window`.
window-inside-top  [Function]
Arguments: window
Summary: Returns the coordinate of the top edge of the window window.

window-inside-right  [Function]
Arguments: window
Summary: Returns the coordinate of the right edge of the window window.

window-inside-bottom  [Function]
Arguments: window
Summary: Returns the coordinate of the bottom edge of the window window.

window-inside-size  [Generic Function]
Arguments: window
Summary: Returns the inside width and height of window as two values.

window-inside-width  [Function]
Arguments: window
Summary: Returns the inside width of window.

window-inside-height  [Function]
Arguments: window
Summary: Returns the inside height of window.
Symbols

(setf command-enabled) (generic function) 209
(setf cursor-visibility) (generic function) 326
(setf delegate-sheet-delegate) (generic function) 455
(setf frame-command-table) (generic function) 204
(setf frame-current-layout) (generic function) 207
(setf frame-manager) (generic function) 214
(setf frame-manager-dialog-view) (generic function) 310
(setf frame-manager-menu-view) (generic function) 303
(setf frame-pretty-name) (generic function) 204
(setf frame-properties) (generic function) 212
(setf gadget-client) (generic function) 246
(setf gadget-id) (generic function) 246
(setf gadget-label) (generic function) 250
(setf gadget-label-align-x) (generic function) 250
(setf gadget-label-align-y) (generic function) 251
(setf gadget-label-text-style) (generic function) 251
(setf gadget-max-value) (generic function) 251
(setf gadget-min-value) (generic function) 251
(setf graph-node-children) (generic function) 437
(setf graph-node-parents) (generic function) 437
(setf graph-root-nodes) (generic function) 435
(setf medium-background) (generic function) 67, 465
(setf medium-buffering-output-p) (generic function) 332
(setf medium-clipping-region) (generic function) 68, 466
(setf medium-default-text-style) (generic function) 69, 467
(setf medium-foreground) (generic function) 67, 465
(setf medium-ink) (generic function) 67, 465
(setf medium-line-style) (generic function) 68, 466
(setf medium-text-style) (generic function) 467
(setf medium-transformation) (generic function) 68, 465
(setf pointer-cursor) (generic function) 373
(setf pointer-sheet) (generic function) 373
(setf port-keyboard-input-focus) (generic function) 453
(setf port-properties) (generic function) 478
(setf presentation-object) (generic function) 123
(setf presentation-single-box) (generic function) 123
(setf presentation-type) (generic function) 123
(setf radio-box-current-selection) (generic function) 256, 257
(setf sheet-enabled-p) (generic function) 446
(setf sheet-region) (generic function) 448
(setf sheet-transformation) (generic function) 448
(setf space-requirement-height) (function) 232
(setf space-requirement-max-height) (function) 232
(setf space-requirement-max-width) (function) 231
(setf space-requirement-min-height) (function) 232
(setf space-requirement-min-width) (function) 231
(setf space-requirement-width) (function) 231
(setf stream-current-output-record) (generic function) 348
(setf stream-default-view) (generic function) 159
(setf stream-drawing-p) (generic function) 348
(setf stream-end-of-line-action) (generic function) 330
(setf stream-end-of-page-action) (generic function) 330
(setf stream-input-buffer) (generic function) 364
(setf stream-insertion-pointer) (generic function) 400
(setf stream-primary-pointer) (generic function) 365
(setf stream-recording-p) (generic function) 347
(setf stream-scan-pointer) (generic function) 401
(setf stream-text-cursor) (generic function) 326
(setf stream-text-margin) (generic function) 328
(setf text-style-mapping) (generic function) 101
(setf window-viewport-position) (generic function) 333
(setf* cursor-position) (generic function) 326
(setf* output-record-end-cursor-position) (generic function) 340

Index
(self* output-record-position) (generic function) 339
(self* output-start-cursor-position) (generic function) 340
(self* pointer-position) (generic function) 373
(self* stream-cursor-position) (generic function) 327
(self* stream-pointer-position) (generic function) 365
*abort-gestures* (variable) 368
*accelerator-gestures* (variable) 368
*activation-gestures* (variable) 388
*application-frame* (variable) 188, 204
*command-argument-delimiters* (variable) 297
*command-dispatchers* (variable) 282, 284
*command-name-delimiters* (variable) 297
*command-parser* (variable) 297
*command-unparser* (variable) 297
*completion-gestures* (variable) 393
*default-frame-manager* (variable) 214
*default-server-path* (variable) 478
*default-text-style* (variable) 94
*delimiter-gestures* (variable) 389
*help-gestures* (variable) 94
*input-context* (variable) 125
*input-wait-handler* (variable) 366
*input-wait-test* (variable) 366
*null-presentation* (variable) 131
*numeric-argument-marker* (variable) 297
*partial-command-parser* (variable) 297
*pointer-button-press-handler* (variable) 366
*pointer-documentation-output* (variable) 205
*possibilities-gestures* (variable) 393
*standard-activation-gestures* (variable) 388
*undefined-text-style* (variable) 95
*unsupplied-argument-marker* (variable) 297
*background-ink* (constant) 108
+control-key+ (constant) 462
+everywhere+ (constant) 41
+fill+ (constant) 225
+flipping-ink+ (constant) 109
+foreground-ink+ (constant) 108
+gadget-dialog-view+ (constant) 160
+gadget-menu-view+ (constant) 160
+hyper-key+ (constant) 462
+identity-transformation+ (constant) 83
+meta-key+ (constant) 462
+nowhere+ (constant) 41
+pointer-documentation-view+ (constant) 160
+pointer-left-button+ (constant) 462
+pointer-middle-button+ (constant) 462
+pointer-right-button+ (constant) 462
+shift-key+ (constant) 462
+super-key+ (constant) 462
+textual-dialog-view+ (constant) 160
+textual-menu-view+ (constant) 160
:activate-callback (initarg) 249
:align-x (initarg) 250, 432
:align-x (option) 226
:align-y (initarg) 250, 432
:align-y (option) 226
:armed-callback (initarg) 246
:background (initarg) 322
:background (option) 223
:button (initarg) 458
:calling-frame (initarg) 189
:client (initarg) 246
:clipping-region (option) 72
:command-table (initarg) 188
:contents (option) 224
:current-selection (initarg) 256, 257
:cutoff-depth (initarg) 435
:decimal-depth (initarg) 261
:default-text-style (initarg) 322
:default-view (initarg) 322
:disabled-commands (initarg) 188
:disarmed-callback (initarg) 246
;display-after-commands (option) 235
;display-function (option) 236
;display-string (option) 236
;display-time (option) 236
:drag-callback (initarg) 259, 261
:draw (option) 237
:editable-p (initarg) 263
:end-of-line-action (initarg) 322
:end-of-line-action (option) 237
:end-of-page-action (initarg) 322
:end-of-page-action (option) 237
:equalize-column-widths (initarg) 429
:foreground (initarg) 322
:foreground (option) 223
:generation-separation (initarg) 435
:help (initarg) 435

532CLIM User Guide
:height (option) 225
:id (initarg) 246
:incremental-redisplay (option) 236
:indicator-type (initarg) 264
:initial-spacing (initarg) 433
:ink (option) 71
:input-buffer (initarg) 364
:items (initarg) 253, 254
:key-name (initarg) 458
:label (initarg) 250
:line-cap-shape (option) 76
:line-dashes (option) 76
:line-joint-shape (option) 75
:line-style (option) 72
:line-thickness (option) 75
:line-unit (option) 75
:incremental-redisplay (option) 236
:id (initarg) 246
:initial-spacing (initarg) 433
:ink (option) 71
:input-buffer (initarg) 364
:items (initarg) 253, 254
:key-name (initarg) 458
:label (initarg) 250
:line-cap-shape (option) 76
:line-dashes (option) 76
:line-joint-shape (option) 75
:line-style (option) 72
:line-thickness (option) 75
:line-unit (option) 75
:max-height (initarg) 434
:max-height (option) 225
:max-value (initarg) 251
:max-width (initarg) 434
:max-width (option) 225
:menu-bar (initarg) 189
:merge-duplicates (initarg) 435
:min-height (initarg) 432
:min-height (option) 225
:min-value (initarg) 251
:min-width (initarg) 432
:min-width (option) 225
:mode (initarg) 253
:modifier (initarg) 124
:modifier-state (initarg) 457
:motif (server-path) 477
:ncolumns (initarg) 263
:ncolumns (initarg) 263
:n-rows (initarg) 434
:number-of-quanta (initarg) 262
:number-of-tick-marks (initarg) 262
:object (initarg) 124
:orientation (initarg) 249, 435
:output-record (option) 237
:port (initarg) 372
:pretty-name (initarg) 188
:properties (initarg) 189
:record (option) 237
:region (initarg) 461
:scroll-down-line-callback (initarg) 259
:scroll-down-page-callback (initarg) 259
:scroll-to-bottom-callback (initarg) 259
:scroll-to-top-callback (initarg) 259
:scroll-up-line-callback (initarg) 259
:scroll-up-page-callback (initarg) 259
:sheet (initarg) 325, 457
:show-as-default (initarg) 255
:show-value-p (initarg) 261
:single-box (initarg) 124
:size (initarg) 338
:spacing (option) 226
:state (initarg) 189
:test (initarg) 253, 254
:text-cursor (initarg) 364
:text-face (option) 97
:text-family (option) 96
:text-margin (initarg) 322
:text-margin (option) 236
:text-size (option) 97
:text-style (option) 73, 223
:timestamp (initarg) 457
:transformation (option) 72
:type (initarg) 124
:value (initarg) 248
:value-changed-callback (initarg) 248
:value-key (initarg) 253, 254
:vertical-spacing (initarg) 322
:vertical-spacing (option) 236
:view (initarg) 124
:width (option) 225
:win32 (server-path) 477
:within-generation-separation (initarg) 435
:x (initarg) 459
:x-position (initarg) 338
:x-spacing (initarg) 429, 433
:x-spacing (option) 226
:y (initarg) 459
:y-position (initarg) 338
:y-spacing (initarg) 429, 433
:y-spacing (option) 226

<Bold>533
A

abbreviations
  pane 190
  presentation type
    operators for 156
abort-gesture  (condition) 368
abort-gesture-event  (generic function) 368
*abort-gestures*  (variable) 368
abstract gadget classes 252
abstract gadgets 241
abstract panes 220
accelerator-gesture  (condition) 368
accelerator-gesture-event  (generic function) 368
accelerator-gesture-numeric-argument  (generic function) 369
*accelerator-gestures*  (variable) 368
accelerators, keystroke 293
accept  (function) 127
accept  (presentation method) 153
accept methods, errors and conditions in 390
accept-1  (function) 128
accept-from-string  (function) 129
accepting-values  (macro) 306
accept-values  (presentation method) 155
accept-values-command-button  (macro) 309
accept-values-pane  (command table) 278
accept-values-pane-displayer  (function) 308
accept-values-resynchronize  (generic function) 309
accessible (of commands) 276
accessing slots and components of application frames 200
accessors for
  application frames 203, 204
  ellipses 58
  polygons, polylines 50
action-gadget  (class) 249
actions 268
:activate-callback  (initarg) 249
activate-callback  (callback) 249
activate-gadget  (generic function) 247
activation gestures 388
activation-gesture-p  (function) 388
*activation-gestures*  (variable) 388
adaptive panes 220

adaptivetoolkit 10
add-character-output-to-text-record  (generic function) 345
add-command-to-command-table  (function) 277
add-gesture-name  (function) 370
add-keystroke-to-command-table  (function) 293
add-menu-item-to-command-table  (function) 288
add-output-record  (generic function) 342
add-pointer-gesture-name  (function) 169
add-presentation-translator-to-command-table  (function) 291
add-string-output-to-text-record  (generic function) 346
adjust-item-list-cells  (generic function) 434
adjust-multiple-columns  (generic function) 430
adjust-table-cells  (generic function) 430
adopted frames 212
adopted sheets 444
adopt-frame  (generic function) 215
affine transformations 79
:align-x  (initarg) 250, 432
:align-x  (option) 226
:align-y  (initarg) 250, 432
:align-y  (option) 226
allocate-pixmap  (generic function) 34
allocate-resource  (function) 510
allocate-space  (generic function) 234
all-processes  (function) 512
and  (presentation type) 137
application frames 6, 184
  accessing slots and components 200
  accessors for
    application frames 203, 204
    ellipses 58
    polygons, polylines 50
*application-frame*  (variable) 188, 204
application-frame  (protocol class) 188
application-frame-p  (function) 188
application-pane  (leaf pane) 238
applications

<Bold>534CLIM User Guide
basic sheet 443
CLIM events, figure of 456
extended stream pane 237
gadgets
  abstract 252
  basic 245
  output records 343
  panes
layout 226
repaît protocol 471
sheet genealogy 447
sheet geometry 450
sheet input protocol 454
sheet output protocol 467
stream 516
structure of regions, figure of 39
class-presentation-type-name (function) 142
clear-output-record (generic function) 342
clear-resource (function) 511
:client (initarg) 246
clients 241
clim-stream-pane (leaf pane) 237
:clipping-region (option) 72
close (generic function) 519
color (protocol class) 105
color-ihs (generic function) 107
colorp (function) 105
color-rgb (generic function) 107
colors 105
  background
    ink 108
  concepts 104
  examples of drawing in 109
  foreground
    ink 108
  objects 105
  operators 106
  predefined names 108
  rendering 105
column-output-record (protocol class) 431
column-output-record-p (function) 431
command
  objects 272
command (presentation type) 139, 280
command line names 270
command line processors 292
  input editing 292
command loops 13, 184, 282
command menus 287
command names 270
command processors 282, 296
  input editor 282
command tables 268, 275, 285
  conditions 279
  predefined 278
command translators 13
command-accessible-in-command-table-p (function) 286
command-already-present (error condition) 279
*command-argument-delimiters* (variable) 297
command-arguments (function) 272
*command-dispatchers* (variable) 282, 284
command-enabled (generic function) 209
  (self command-enabled) (generic function) 209
command-line-command-parser (function) 296
command-line-command-unparser (function) 296
command-line-name-for-command (function) 286, 293
command-line-read-remaining-arguments-for-partial-command (function) 296
command-menu-item-options (function) 290
command-menu-item-type (function) 290
command-menu-item-value (function) 290
command-menu-pane (leaf pane) 238
command-name (function) 272
command-name (presentation type) 139, 281
*command-name-delimiters* (variable) 297
command-not-accessible (error condition) 279
command-not-present (error condition) 279
command-or-form (presentation type) 139, 281
*command-parser* (variable) 297
command-present-in-command-table-p (function) 286
commands 268
  accessible 276
  defined 272
  defining 269
  input editor 385
  present 275
  presentation types for 280
  processor 282, 296
:command-table (initarg) 188
command-table (protocol class) 276
command-table-already-exists (error condition) 279
command-table-complete-input (function) 287
command-table-error (error condition) 279
command-table-inherit-from (generic function) 276
command-table-not-found (error condition) 279
command-table-p (function) 276
*command-unparser* (variable) 297
complete-from-generator (function) 395
complete-from-possibilities (function) 396
complete-input (function) 393
completing-from-suggestions (macro) 396
completion (presentation type) 134
completion, string 393
*completion-gestures* (variable) 393
complex (presentation type) 132
compose-rotation-with-transformation (function) 86
compose-scaling-with-transformation (function) 86
compose-space (generic function) 234
compose-transformations (generic function) 85
compose-transformation-with-rotation (function) 86
compose-transformation-with-scaling (function) 86
compose-transformation-with-translation (function) 86
compose-translation-with-transformation (function) 86
composites of ellipses and elliptical arcs 57
polygons and polylines 49
transformations 80
:contents (option) 224
contrasting-dash-pattern-limit (generic function) 77
contrasting-inks-limit (generic function) 107
+control-key+ (constant) 462
coordinate (type) 40
coordinate system, local 22
figure of 22
copy-area (generic function) 35
copy-from-pixmap (generic function) 35
copy-to-pixmap (generic function) 34
current-process (function) 512
current-selection (initarg) 256, 257
cursor (protocol class) 325
cursorp (function) 325
cursor-position (generic function) 326
(cursor* cursor-position) (generic function) 326
cursors
stream text
protocol 326
text
protocol 325
cursors, text 323
cursor-sheet (generic function) 325
cursor-visibility (generic function) 326
(cursor* cursor-visibility) (generic function) 326
:cutoff-depth (initarg) 435

D
DAG (directed acyclic graph) 418
decompose-gadget (generic function) 247
deallocate-pixmap (generic function) 34
deallocate-resource (function) 511
:decimal-places (initarg) 261
default-describe-presentation-type (function) 142
*default-frame-manager* (variable) 214
default-frame-top-level (generic function) 208
*default-server-path* (variable) 478
*default-text-style* (variable) 94
:default-text-style (initarg) 322
:default-view (initarg) 322
defgeneric* (macro) 514
define-application-frame (macro) 186, 203
define-border-type (macro) 427
define-command (macro) 272
define-command-table (macro) 276
define-default-presentation-method (macro) 161
define-drag-and-drop-translator (macro) 173
define-gesture-name (macro) 370
define-graph-type (macro) 435
define-presentation-action (macro) 172
define-presentation-generic-function (macro) 161
define-presentation-method (macro) 152

537
define-presentation-to-command-translator (macro) 172, 271
define-presentation-translator (macro) 170
define-presentation-type (macro) 150
define-presentation-type-abbreviation (macro) 157
defining
  application frames 186
  commands 269
  pane types 239
  presentation methods 152
  presentation translators 175
  examples 175
  operators for 170
presentation types
  abbreviations, operators for 156
  concepts 146
  examples 148
  operators for 150
defmethod* (macro) 514
defresource (generic function) 509
degrafted sheets 444
delegate-sheet-delegate (generic function) 455
  (self delegate-sheet-delegate) (generic function) 455
delegate-sheet-input-mixin (class) 455
delete-gesture-name (function) 371
delete-output-record (generic function) 342
delimiter gestures 388
delimiter-gesture-p (function) 389
*delimiter-gestures* (variable) 389
derived bounding rectangles 59
describe-presentation-type (function) 139
describe-presentation-type (presentation method) 154
destroy-port (generic function) 479
destroy-process (function) 511
device events 456
device events, standard 456
device-event (class) 457
dialogs
  concepts 300
  examples 310
  operators for 306
directed acyclic graphs 418
  figure of 418
disable-command (function) 284
disabled frames 212
disabled sheets 444
:disabled-commands (initarg) 188
disable-frame (generic function) 215
:disarmed-callback (initarg) 246
disarmed-callback (callback) 247
disowned frames 212
disowned sheets 444
disown-frame (generic function) 215
dispatch-event (generic function) 453
;display-after-commands (option) 235
display-command-menu (generic function) 210,
  288
display-command-table-menu (generic function)
  287
display-cursor (generic function) 326
displayed-output-record (protocol class) 338
displayed-output-record-p (function) 338
display-exit-boxes (generic function) 308
;display-function (option) 236
;display-string (option) 236
;display-time (option) 236
distribute-event (generic function) 453
do-command-table-inheritance (macro) 285
document-presentation-translator (function) 179
:drag-callback (initarg) 259, 261
drag-callback (callback) 260, 261
dragging-output (macro) 378
drag-output-record (generic function) 377
:draw (option) 237
draw-arrow (function) 30
draw-arrow* (function) 30
draw-circle (function) 29
draw-circle* (function) 29
draw-ellipse (function) 28
draw-ellipse* (function) 28
drawing functions 24
  compound 30
    examples 23
    figure of 23
    general behavior of 36
    medium-specific 37
    spread version 46
drawing options, using 70
drawing plane 20
    figure of 20
draw-line (function) 25
draw-line* (function) 25
draw-lines (function) 26
draw-oval* (function) 26
draw-oval (function) 31
draw-oval* (function) 31
draw-pattern* (function) 32
draw-point (function) 24
draw-point* (function) 25
draw-points (function) 25
draw-points* (function) 25
draw-polygon (function) 26
draw-polygon* (function) 26
draw-rectangle (function) 27
draw-rectangle* (function) 27
draw-rectangles (function) 27
draw-rectangles* (function) 27
draw-standard-menu (function) 304
draw-text (function) 29
draw-text* (function) 29

E

:editable-p (initarg) 263
editing, input 382
ellipse
bounding parallelogram, table of 55
ellipse (protocol class) 56
ellipse-center-point (generic function) 58
ellipse-center-point* (generic function) 58
ellipse-end-angle (generic function) 59
ellipses (function) 56
ellipse-radii (generic function) 58
ellipses 54
accessors for 58
as specified by parallelograms, figure of 59
constructors for 57
ellipse-start-angle (generic function) 59
eccentric arcs 54
constructors for 57
elliptical-arc (protocol class) 56
elliptical-arc-p (function) 56
enable-command (function) 284
enabled frames 212
enabled sheets 444
enable-frame (generic function) 215
:end-of-line-action (initarg) 322
:end-of-line-action (option) 237
:end-of-page-action (initarg) 322
:end-of-page-action (option) 237
:equalize-column-widths (initarg) 449
erase-input-buffer (generic function) 402
erase-output-record (generic function) 341
even-scaling-transformation-p (generic function) 85
event (protocol class) 456
event classes, figure of 456
event-listen (generic function) 454
event-modifier-state (generic function) 458
eventp (function) 457
event-peek (generic function) 454
event-read (generic function) 453
event-read-no-hang (generic function) 454
events 9, 456
   client 452
   defined 168
   dispatching 452
   distributing 452
   keyboard 452
   pointer 452
   standard device 456
event-sheet (generic function) 457
event-timestamp (generic function) 457
event-type (generic function) 457
event-unread (generic function) 454
+everywhere+ (constant) 41
execute-frame-command (generic function) 209, 283
exiting an application 201
expand-presentation-type-abbreviation (function) 158
expand-presentation-type-abbreviation-1 (function) 158
expression (presentation type) 138
extended input streams 363
   conditions 368
   protocol 364
extended output streams 321
extended stream panes 184, 220, 234
   classes 237
   making 238
   options 234
extended stream protocol 516
extended-input-stream (protocol class) 363
extended-input-stream-p (function) 363
extended-output-stream (protocol class) 322
extended-output-stream-p (function) 322

F

+fill+ (constant) 225
filled-in areas 73
filling-output (macro) 425
find-applicable-translators (function) 178
find-command-from-command-line-name (function) 286, 292
find-command-table (function) 277
find-frame-manager (function) 213
find-graft (function) 480
find-innermost-applicable-presentation (function) 180
find-keystroke-item (function) 294
find-menu-item (function) 290
find-pane-for-frame (generic function) 216
find-pane-named (generic function) 206
find-port (function) 477
find-presentation-translator (function) 292
find-presentation-translators (function) 177
find-presentation-type-class (function) 142
flipping ink 109
example 111
+flipping-ink+ (constant) 109
float (presentation type) 132
:foreground (initarg) 322
:foreground (option) 223
foreground 104
ink 108
+foreground-ink+ (constant) 108
form (presentation type) 138
format-graph-from-roots (function) 419
format-items (function) 410
format-textual-list (function) 410, 424
formatting
  cells
    protocol 432
  graphs 418
    concepts 418
    examples 422
    operators for 419
    protocol 434
  item lists 405
  protocol 433

protocols for tables, item lists, and graphs 428
rows and columns
  protocol 431
tables 404
  calendar month example, figure of 414
  examples 411
  figure of 415, 417
  from a list, figure of 412
  operators for 405
  output records of, figure of 337
  protocol 429
text 424

formatting-cell (macro) 408
formatting-column (macro) 407
formatting-item-list (macro) 408
formatting-row (macro) 407
formatting-table (macro) 405
frame managers 10, 184, 212
  finding 213
  layout protocol 230
  operators 214
frame-calling-frame (generic function) 206
frame-command-table (generic function) 204
  (setf frame-command-table) (generic function) 204
frame-current-layout (generic function) 207
  (setf frame-current-layout) (generic function) 207
frame-current-panes (generic function) 206
frame-document-highlighted-presentation (generic function) 217
frame-drag-and-drop-feedback (generic function) 218
frame-drag-and-drop-highlighting (generic function) 218
frame-error-output (generic function) 205
frame-exit (generic function) 210
frame-exit (restart) 210
frame-find-innermost-applicable-presentation (generic function) 217
frame-input-context-button-press-handler (generic function) 217
frame-maintain-presentation-histories (generic function) 217
frame-manager (generic function) 214
  (setf frame-manager) (generic function) 214
frame-manager (protocol class) 212
frame-manager-dialog-view (generic function) 310
frame-manager-frames (generic function) 215
frame-manager-menu-choose (generic function) 303
frame-manager-menu-view (generic function) 303
frame-manager-notify-user (generic function) 211
frame-manager-p (function) 213
frame-name (generic function) 204
frame-panes (generic function) 206
frame-parent (generic function) 206
frame-pointer-documentation-output (generic function) 205
frame-pretty-name (generic function) 204
(setf frame-pretty-name) (generic function) 204
frame-properties (generic function) 212
(setf frame-properties) (generic function) 212
frame-query-io (generic function) 205
frame-replay (generic function) 211
frames 6, 184
  adopted 212
  application 6, 184
    accessing slots and components 200
    accessors for 203, 204
    defining 186
    examples 201
    initializing 197
    operators for 203
    protocol 188
  disabled 212
  disowned 212
  enabled 212
  shrunk 212
frame-standard-input (generic function) 204
frame-standard-output (generic function) 205
frame-state (generic function) 215
frame-top-level-sheet (generic function) 207
funcall-presentation-generic-function (macro) 161
functions
  composite pane 229
  drawing 24
  compound 30
  examples 23
  figure of 23
  general behavior of 36
  medium-specific 37
spread versions of 46
low-level, for presentation translators 177
mirrored sheet 483
mirrors 483
presentation type 139
repaint protocol 470
sheet geometry 448
sheet input protocol 452
sheet output protocol 467
stream 518
text style 97
transformation 85
window stream pane 332
fundamental-binary-input-stream (class) 518
fundamental-binary-output-stream (class) 518
fundamental-binary-stream (class) 517
fundamental-character-input-stream (class) 517
fundamental-character-output-stream (class) 518
fundamental-character-stream (class) 517
fundamental-input-stream (class) 517
fundamental-output-stream (class) 517
fundamental-stream (class) 517

G

gadget (protocol class) 245
gadget id 242
gadget-activate-callback (generic function) 249
gadget-active-p (generic function) 247
gadget-armed-callback (generic function) 246
gadget-client (generic function) 246
  (setf gadget-client) (generic function) 246
+gadget-dialog-view+ (constant) 160
gadget-dialog-view (class) 160
gadget-disarmed-callback (generic function) 246
gadget-id (generic function) 246
  (setf gadget-id) (generic function) 246
gadget-label (generic function) 250
  (setf gadget-label) (generic function) 250
gadget-label-align-x (generic function) 250
  (setf gadget-label-align-x) (generic function) 250
gadget-label-align-y (generic function) 250
  (setf gadget-label-align-y) (generic function) 251
gadget-label-text-style (generic function) 251
  (setf gadget-label-text-style) (generic function) 251
<Bold>541
geometric objects 38
geometry, sheet 448
functions 448
notifications 472
gesture, sheet classes 450
gesture names 168, 369
standard 371
gestures 369
activation 388
defined 168
delimiter 388
keyboard 369
pointer 168, 369
get-frame-pane (generic function) 206
global-command-table (command table) 278
graft (generic function) 481
grafted sheets 444
graft-height (generic function) 481
graft-orientation (generic function) 481
graft-pixels-per-inch (function) 482
graft-pixels-per-millimeter (function) 482
grafts 8, 440, 480
graft-units (generic function) 481
graft-width (generic function) 481
graphics
mixing with text 329
output recording 349
output records 344
protocols 36
graphics-displayed-output-record (protocol class) 344
graphics-displayed-output-record-p (function) 344
graph-node-children (generic function) 437
(setf graph-node-children) (generic function) 437
graph-node-object (generic function) 437
graph-node-output-record (protocol class) 436
graph-node-output-record-p (function) 436
graph-node-parents (generic function) 437
(setf graph-node-parents) (generic function) 437
graph-output-record (protocol class) 434
graph-output-record-p (function) 435
graph-root-nodes (generic function) 435
(setf graph-root-nodes) (generic function) 435
graphs

<Bold>542CLIM User Guide
acyclic 418
directed 418
directed acyclic 418
figure of 418
formatting 418
concepts 418
examples 422
operators for 419
protocol 434
horizontal
figure of 422
vertical
figure of 423

H

handle-event (generic function) 453
handle-repaint (generic function) 470
hardcopy streams 523
:hash-table (initarg) 435
hbox-pane (composite pane) 226
:height (option) 226
*help-gestures* (variable) 393
hierarchies of interactive regions 440
highlight-applicable-presentation (function) 181
highlight-output-record (generic function) 341
highlight-presentation (presentation method) 156
horizontally (macro) 226
hrack-pane (composite pane) 228
+hyper-key+ (constant) 462

I

:id (initarg) 246
+identity-transformation+ (constant) 83
identity-transformation-p (generic function) 84
immediate-repainting-mixin (class) 471
immediate-rescan (generic function) 401
immediate-sheet-input-mixin (class) 454
implementing gadgets 243
incremental redisplay
concepts 354
defined 354
example 359
operators for 355
using updating-output 357
:incremental-redisplay (option) 236
indenting-output (macro) 424
:indicator-type (initarg) 264
indirect inks 108
inheritance (in presentation types) 116, 147
initializing application frames 197
:initial-spacing (initarg) 433
ink (option) 71
inks
background 108
flipping 109
example 111
foreground 108
indirect 108
input
accepting, operators for 125
by means of gadgets 124
contexts 167
nested 167
from users 124
operators 125
sheet protocol classes 454
sheet protocol functions 452
sheet protocols 452
standard 204, 237
input buffers, reading and writing tokens in 391
input contexts 12, 115, 125
input editing 292, 382
input editing stream protocol 400
input editor commands 385
table of 386
input editors 282
input of presentation types 124
input streams
basic 362
character 519
editing 382
extended 363
conditions 368
protocol 364
:input-buffer (initarg) 364
*input-context* (variable) 125
input-editing-stream (protocol class) 384
input-editing-stream-p (function) 384
input-not-of-required-type (condition) 390
input-not-of-required-type (function) 390
input-stream-p (generic function) 517

<Bold>543
interacting via
  command line 292
  command menus 287
  keystroke accelerators 293
  translators 291
interaction styles 280
  command line 292
  command menus 287
  keystroke accelerators 293
  mouse 291
interactive regions, hierarchies of 440
interactive-stream-p (generic function) 384
interactor-pane (leaf pane) 237
invalidate-cached-regions (generic function) 485
invalidate-cached-transformations (generic function) 485
invertible-transformation-p (generic function) 84
inverted-transformation (generic function) 85
invoke-accept-values-command-button (generic function) 309
invoke-updating-output (generic function) 355
invoke-with-drawing-options (generic function) 71
invoke-with-new-output-record (generic function) 352
invoke-with-output-recording-options (generic function) 351
invoke-with-output-to-output-record (generic function) 353
invoke-with-text-style (generic function) 100
item lists
  formatting protocol 433
item-list-output-record (protocol class) 433
item-list-output-record-p (function) 433
:items (initarg) 253, 254

K

keyboard events 452
keyboard gestures 369
keyboard-event (class) 458
keyboard-event-character (generic function) 458
keyboard-event-key-name (generic function) 458
key-modifier-state-match-p (macro) 462

:key-name (initarg) 458
key-press-event (class) 458
key-release-event (class) 458
keystroke accelerators 293

table of 386
keyword (presentation type) 131

L

:label (initarg) 250
label gadgets 252
labelled-gadget-mixin (class) 250
labelling (macro) 252
label-pane (leaf pane) 252
layering CLIM over the host system, figure of 5
layout panes 184, 224
  classes 226
  options 224
layout, protocol 230
layout-frame (generic function) 207
layout-graph-edges (generic function) 436
layout-graph-nodes (generic function) 436
leaf panes 220
  defining 239
line (protocol class) 50
  line protocol 51
  line styles 74
    options 74
line wrapping (text) 329
:line-cap-shape (option) 76
:line-dashes (option) 76
line-end-point (generic function) 52
line-end-point* (generic function) 51
:line-joint-shape (option) 75
linep (function) 51
lines 47, 50
  cap shapes, figure of 76
  joint shapes, figure of 76
line-start-point (generic function) 52
line-start-point* (generic function) 51
:line-style (option) 72
line-style (protocol class) 74
line-style-cap-shape (generic function) 76
line-style-dashes (generic function) 76
line-style-joint-shape (generic function) 75
line-style-p (function) 74
line-style-thickness (generic function) 75
line-style-unit (generic function) 75

[line-thickness (option) 75
[line-unit (option) 75
list-pane (class) 253
list-pane gadgets 253
local coordinate system 22
figure of 22
lookup-keystroke-command-item (function) 295
lookup-keystroke-item (function) 295

M

make-3-point-transformation (function) 82
make-3-point-transformation* (function) 82
make-application-frame (function) 187, 203
make-bounding-rectangle (function) 61
make-clim-application-pane (function) 239
make-clim-interactor-pane (function) 239
make-clim-stream-pane (function) 238
make-command-table (function) 277
make-contrasting-dash-patterns (function) 77
make-contrasting-inks (function) 106
make-design-from-output-record (generic function) 353
make-device-font-text-style (function) 102
make-ellipse (function) 57
make-ellipse* (function) 57
make-elliptical-arc (function) 57
make-elliptical-arc* (function) 57
make-flipping-ink (function) 109
make-gray-color (function) 106
make-ihs-color (function) 106
make-line (function) 51
make-line* (function) 51
make-line-style (function) 74
make-lock (function) 513
make-pane (function) 222
make-pane-1 (generic function) 222
make-pattern (function) 31
make-point (function) 47
make-polygon (function) 49
make-polygon* (function) 49
make-polyline (function) 49
make-polyline* (function) 49
make-presentation-type-specifier (function) 143, 158
make-process (function) 511

make-rectangle (function) 53
make-rectangle* (function) 53
make-rectangular-tile (function) 32
make-recursive-lock (function) 513
make-reflection-transformation (function) 81
make-reflection-transformation* (function) 81
make-rgb-color (function) 106
make-rotation-transformation (function) 80
make-rotation-transformation* (function) 80
make-scaling-transformation (function) 81
make-scaling-transformation* (function) 81
make-space-requirement (function) 231
make-text-style (function) 96
make-transformation (function) 82
make-translation-transformation (function) 80
managers, frame 10, 184, 212
finding 213
layout protocol 230
operators for 214
map-over-column-cells (generic function) 432
map-over-command-table-commands (function) 285
map-over-command-table-keystrokes (function) 294
map-over-command-table-menu-items (function) 290
map-over-command-table-names (function) 285, 293
map-over-command-table-translators (function) 292
map-over-ducts (function) 481
map-over-item-list-cells (generic function) 434
map-over-output-records-containing-position (generic function) 343
map-over-output-records-overlapping-region (generic function) 343
map-over-polygon-coordinates (generic function) 50
map-over-polygon-segments (generic function) 50
map-over-ports (function) 479
map-over-presentation-type-supertypes (function) 142
map-over-presentation-type-supertypes (presentation method) 155
map-over-region-set-regions (generic function) 44
map-over-row-cells (generic function) 431
map-over-table-elements (generic function) 430

<Bold>545
mappings
text style 101
map-resource (function) 511
map-sheet-position-to-child (generic function) 449
map-sheet-position-to-parent (generic function) 449
map-sheet-rectangle* to-child (generic function) 449
map-sheet-rectangle* to-parent (generic function) 449
matching (presentation translators) 166
:max-height (initarg) 434
:max-height (option) 225
:max-value (initarg) 251
:max-width (initarg) 434
:max-width (option) 225
medium (protocol class) 464
medium-background (generic function) 67, 465
(setf medium-background) (generic function) 67, 465
medium-buffering-output-p (generic function) 332
(setf medium-buffering-output-p) (generic function) 332
medium-clipping-region (generic function) 68, 466
(setf medium-clipping-region) (generic function) 68, 466
medium-current-text-style (generic function) 69
medium-default-text-style (generic function) 69, 467
(setf medium-default-text-style) (generic function) 69, 467
medium-draw-ellipse* (generic function) 38
medium-draw-line* (generic function) 37
medium-draw-lines* (generic function) 37
medium-draw-point* (generic function) 37
medium-draw-points* (generic function) 37
medium-draw-polygon* (generic function) 37
medium-draw-rectangle* (generic function) 37
medium-draw-text* (generic function) 38
medium-foreground (generic function) 67, 465
(setf medium-foreground) (generic function) 67, 465
medium-ink (generic function) 67, 465
(setf medium-ink) (generic function) 67, 465
medium-line-style (generic function) 68, 466
(setf medium-line-style) (generic function) 68, 466
medium-merged-text-style (generic function) 467
mediumump (function) 464
mediums 9, 22, 94, 464
associating with a sheet 468
components 66
defined 66
medium-text-style (generic function) 69, 466
(setf medium-text-style) (generic function) 69, 467
medium-transformation (generic function) 68, 465
(setf medium-transformation) (generic function) 68, 465
member (presentation type abbreviation) 135
member-alist (presentation type abbreviation) 135
member-sequence (presentation type abbreviation) 135
:menu-bar (initarg) 189
menu-button (class) 255
menu-button gadgets 255
menu-button-pane (class) 255
menu-choose (generic function) 300
menu-choose-command-from-command-table
(function) 288
menu-choose-from-drawer (generic function) 303
menu-command-parser (function) 296
menu-item-display (function) 305
menu-item-options (function) 305
menu-item-value (function) 305
menu-read-remaining-arguments-for-partial-command
(function) 296
menus
concepts 300
examples 310
:merge-duplicates (initarg) 435
merge-text-styles (generic function) 97
+meta-key+ (constant) 462
methods
presentation 152
Microsoft Windows
themes 11
:min-height (initarg) 432
:min-height (option) 225
:min-value (initarg) 251
:min-width (initarg) 432
:min-width (option) 225
mirrored sheets 483
mirrors 483
functions 483
:mode (initarg) 253
:modifier (initarg) 124
:modifier-state (initarg) 457
:motif (server-path) 477
move-and-resize-sheet (generic function) 449
move-sheet (generic function) 448
multiple-columns-x-spacing (initarg) 429
multiple-value setf 514
multi-processing 511
mute-repainting-mixin (class) 471
mute-sheet-input-mixin (class) 455
mute-sheet-output-mixin (class) 468

N

:name (initarg) 188
:name (option) 223
:name-key (initarg) 253, 254
:n-columns (initarg) 434
:n-columns (initarg) 263
nested input context 167
nested presentations 168
nil (presentation type) 130
:nils (initarg) 263
note-command-disabled (generic function) 216
note-command-enabled (generic function) 216
note-frame-state-changed (generic function) 216
note-gadget-activated (generic function) 247
note-gadget-deactivated (generic function) 247
note-sheet-adopted (generic function) 472
note-sheet-degrafted (generic function) 472
note-sheet-disabled (generic function) 472
note-sheet-disowned (generic function) 472
note-sheet-enabled (generic function) 472
note-sheet-grafted (generic function) 472
note-sheet-region-changed (generic function) 472
note-sheet-transformation-changed (generic function) 472
note-space-requirements-changed (generic function) 233
notify-user (generic function) 211
+nowhere+ (constant) 41
:n-rows (initarg) 434
null (presentation type) 131
null-or-type (presentation type abbreviation) 138
*null-presentation* (variable) 131

number (presentation type) 131
:number-of-quanta (initarg) 262
:number-of-tick-marks (initarg) 262
*numeric-argument-marker* (variable) 297

O

:object (initarg) 124
objects
  application 113
    figure of 114
  color 105
  command 272
  controlling sensitivity 164
  geometric 38
  inheritance in presentation types 116, 147
  line style 74
  point 46
  text style 95
open-stream-p (generic function) 518
open-window-stream (function) 527
operators for
  application frames 203
  defining presentation translators 170
  defining presentation types 150
  dialogs 306
  drawing in color 106
  formatting graphs 419
  formatting tables 405
  frame managers 214
  incremental redisplay 355
  input 125
  output 120
  output recording 337
  pointer gestures 169
  presentation type abbreviations 156
  presentation types 139
  primitive window layer 527
  running applications 207
  viewport and scrolling in windows 526
  views of presentation types 159
option-pane (class) 254
option-pane gadgets 253
options
  application frames
    :layouts 190
example 192

figure of 193, 194, 196

:pane 189

example 192

:panes 190

example 192

drawing

using 70

extended stream pane 234

layout panes 224

line style 74

pane initialization 222

text style 96

or (presentation type) 137

:orientation (initarg) 249, 435

oriented-gadget-mixin (class) 249

origin 21

outlined-pane (composite pane) 228

outlining (macro) 228

output

bordered 426

elements of 426

buffered 331

character streams 521

flushing 331

sheet properties 464

sheet protocol classes 467

sheet protocol functions 467

sheet protocols 464

standard 205, 238

with attached semantics 115

output operators 120

output recording

concepts 336

graphics 349

operators for 337

protocol 339

text 349

utilities 350

output recording streams 347

protocol 347

output records 108, 336

bounding rectangle of, figure of 60

classes 343

graphics-displayed 344

history 336

integrating with gadgets 264

presentations 337

protocol

database 342

replaying 108

text-displayed 345

top-level 346

tree structure of, figure of 336

output streams

basic 320

extended 321

:output-record (option) 237

output-record (protocol class) 338

output-record-children (generic function) 342

output-record-count (generic function) 343

output-record-end-cursor-position (generic function) 340

(setf* output-record-end-cursor-position)

(generic function) 340

output-recording-stream (protocol class) 347

output-recording-stream-p (function) 347

output-record-p (function) 338

output-record-parent (generic function) 340

output-record-position (generic function) 339

(setf* output-record-position)

(generic function) 339

output-record-refined-sensitivity-test (generic function) 341

output-record-start-cursor-position (generic function) 339

(setf* output-start-cursor-position)

(generic function) 340

output-stream-p (generic function) 517

packages, user 507

pane (protocol class) 221

pane hierarchy 230

pane-background (generic function) 223

pane-foreground (generic function) 223

pane-frame (generic function) 223

pane-name (generic function) 223

pane-needs-redisplay (generic function) 210

panep (function) 221

:panes (initarg) 189

panes, 6, 184, 220

<Bold>548CLIM User Guide
abbreviation 190
abstract 220
adaptive 220
composite 220
functions 229
constructing 221
defining 239
element 239
example 239
extended stream 184, 220, 234
classes 237
making 238
options 234
gadget 184
hierarchy 230
initialization options 222
layout 184, 224, 226
leaf 220
properties of 223
space allocation 230
space composition 230
space requirement 230
using :accept-values in application frames 197
window 332
stream, functions 332
pane-scroller (generic function) 229
panes-need-redisplay (generic function) 210
pane-viewport (generic function) 229
pane-viewport-region (generic function) 229
:parent (initarg) 338
parent sheets 440
parse-error (condition) 390
parse-error (function) 390
parse-text-style (generic function) 97
partial-command-p (function) 272
*partial-command-parser* (variable) 297
path (protocol class) 39
pathname (presentation type) 133
pathp (function) 40
patterns 31
permanent-medium-sheet-output-mixin (class) 468
pixmap-depth (generic function) 34
pixmap-height (generic function) 34
pixmaps 33
pixmap-width (generic function) 34
plane, drawing 20
plist 316
point
coordinates 21
point (protocol class) 46
:pointer (initarg) 364, 458
pointer
documentation 170, 190
events 452
gestures 168, 369
operators for 169
protocol 372
tracking 373
pointer (protocol class) 372
pointer-button-click-and-hold-event (class) 460
pointer-button-click-event (class) 460
pointer-button-double-click-event (class) 460
pointer-button-event (class) 460
pointer-button-hold-event (class) 460
pointer-button-press-event (class) 460
*pointer-button-press-handler* (variable) 366
pointer-button-release-event (class) 460
pointer-button-state (generic function) 373
pointer-cursor (generic function) 373
(self pointer-cursor) (generic function) 373
*pointer-documentation-output* (variable) 205
pointer-documentation-pane (leaf-pane) 238
+pointer-documentation-view+ (constant) 160
pointer-documentation-view (class) 160
pointer-enter-event (class) 460
pointer-event (class) 458
pointer-event-button (generic function) 459
pointer-event-native-x (generic function) 459
pointer-event-native-y (generic function) 459
pointer-event-pointer (generic function) 459
pointer-event-shift-mask (generic function) 460
pointer-event-x (generic function) 459
pointer-event-y (generic function) 459
pointer-exit-event (class) 460
pointer-input-rectangle (function) 380
pointer-input-rectangle* (function) 379
+pointer-left-button+ (constant) 462
+pointer-middle-button+ (constant) 462
pointer-motion-event (class) 460
pointerp (function) 372
pointer-place-rubber-band-line* (function) 379
pointer-port (generic function) 372
pointer-position (generic function) 373
(setf* pointer-position) (generic function) 373

<Bold>549
presentations 12, 114, 337
  finding applicable 180
  nested 168
  sensitive 164
presentation-single-box (generic function) 123
presentation-subtypep (function) 141
presentation-subtypep (presentation method) 155
presentation-type (generic function) 123
  (self presentation-type) (generic function) 123
presentation-type-direct-supertypes (function) 142
presentation-type-history (presentation method) 156
presentation-type-name (function) 140
presentation-type-of (function) 141
presentation-type-options (function) 140
presentation-typep (function) 140
presentation-typep (presentation method) 154
presentation-type-parameters (function) 140
presentation-type-specifier-p (function) 141
presentation-type-specifier-p (presentation method) 154
present-to-string (function) 122
:pretty-name (initarg) 188
print-menu-item (function) 305
process-interrupt (function) 512
process-wait (function) 512
process-wait-with-timeout (function) 512
process-yield (function) 512
prompt-for-accept (generic function) 129
prompt-for-accept-1 (function) 130
:properties (initarg) 189
properties of panes 223
protocols
  application frame 188
  basic stream 516
  bounding rectangle 62
  cell formatting 432
  extended input stream 364
  extended stream 516
  formatting
    for tables, item lists, and graphs 428
    item lists 433
  graph formatting 434
  graphics 36
  input editing stream 400
  layout 230
line 51
output record 339
output record database 342
output recording stream 347
point 47
pointer 372
rectangle 53
repaint 470
repaint classes 471
repaint functions 470
row and column formatting 431
sheet 441, 444
sheet input 452
sheet input classes 454
sheet input functions 452
sheet notification 472
sheet output 464
sheet output classes 467
sheet output functions 467
stream text cursor 326
table formatting 429
text 327
text cursor 325
transformation 83
push-button (class) 255
push-button gadgets 255
push-button-pane (class) 256
push-button-show-as-default (generic function) 256
Q
query identifier 306
queue-event (generic function) 453
queue-repaint (generic function) 470
queue-rescan (generic function) 401
quitting an application 201
R
radio-box (class) 256
radio-box gadgets 256
radio-box-current-selection (generic function) 256
  (self radio-box-current-selection) (generic function) 256
radio-box-pane (class) 257
radio-box-selections (generic function) 257
raise-sheet (generic function) 445
range-gadget-mixin (class) 251
ratio (presentation type) 132
rational (presentation type) 132
read-command (function) 282
read-command-using-keystrokes (function) 284
read-frame-command (generic function) 209, 283
read-gesture (function) 366
reading tokens 391
read-token (function) 391
realize-mirror (generic function) 484
:record (option) 237
rectangle (protocol class) 52
rectangle-edges* (generic function) 53
rectangle-height (generic function) 54
rectangle-max-point (generic function) 53
rectangle-max-x (generic function) 53
rectangle-max-y (generic function) 53
rectangle-min-point (generic function) 53
rectangle-min-x (generic function) 53
rectangle-min-y (generic function) 53
rectanglep (function) 52
rectangles 52
bounding 59
derived bounding 59
protocol 53
rectangle-size (generic function) 54
rectangle-width (generic function) 54
rectilinear-transformation-p (generic function) 85
redisplay (function) 356
redisplay, incremental
  defined 354
  example 359
  operators for 355
  using updating-output 357
redisplay-frame-pane (generic function) 210
redisplay-frame-panes (generic function) 211
redisplay-output-record (generic function) 357
redraw-input-buffer (generic function) 402
reflection 79
reflection-transformation-p (generic function) 84
reflection-underspecified (error condition) 83
:region (initarg) 461
region (protocol class) 39
region composition 42
region set 42
examples, figure of 46
normalization of rectangular, figure of 44
region-contains-position-p (generic function) 41
region-contains-region-p (generic function) 41
region-difference (generic function) 45
region-equal (generic function) 41
region-intersection (generic function) 45
region-intersects-region-p (generic function) 41
regionp (function) 39
regions 38
  bounded 38
class structure of, figure of 39
examples, figure of 46
normalization 43
  figure of 44
  predicates 41
  unbounded 38
region-set (protocol class) 42
region-set-p (function) 42
region-set-regions (generic function) 43
region-union (generic function) 44
remove-command-from-command-table (function) 278
remove-keystroke-from-command-table (function) 294
remove-menu-item-from-command-table (function) 291
remove-pointer-gesture-name (function) 169
remove-presentation-translator-from-command-table (function) 291
rendering 20, 105
  figure of 20
reorder-sheets (generic function) 446
repaint protocol 470
  classes 471
  functions 470
repaint-sheet (generic function) 470
replace-input (generic function) 391
replay (function) 340
replay-output-record (generic function) 341
rescan-if-necessary (generic function) 402
reset-scan-pointer (generic function) 401
resize-sheet (generic function) 449
resources 509
restart 210
restart-port (generic function) 479
restraining (macro) 229
restraining-pane (composite pane) 229
rigid-transformation-p (generic function) 84
rotation 79
row-output-record (protocol class) 431
row-output-record-p (function) 431
rows and columns, formatting protocol 431
run-frame-top-level (around method) 208
run-frame-top-level (generic function) 208
running applications 200
operators for 207

S

scaling transformations 79
scaling-transformation-p (generic function) 85
scroll-bar (class) 258
scroll-bar gadgets 258
scroll-bar-drag-callback (generic function) 259
scroll-bar-pane (class) 261
scroll-bar-scroll-down-line-callback (generic function) 259
scroll-bar-scroll-down-page-callback (generic function) 259
scroll-bar-scroll-to-bottom-callback (generic function) 259
scroll-bar-scroll-to-top-callback (generic function) 259
scroll-bar-scroll-up-line-callback (generic function) 259
scroll-bar-scroll-up-page-callback (generic function) 259
scroll-bar-scroll-up-page-callback (initarg) 259
scroll-down-line-callback (initarg) 259
scroll-down-line-callback (callback) 260
scroll-down-page-callback (initarg) 259
scroll-down-page-callback (callback) 260
scroller-pane (composite pane) 228
scroll-extent (generic function) 229
scrolling 526
scrolling (macro) 228
:scroll-to-bottom-callback (initarg) 259
scroll-to-bottom-callback (callback) 260
:scroll-to-top-callback (initarg) 259
scroll-to-top-callback (callback) 260
:scroll-up-line-callback (initarg) 259
scroll-up-line-callback (callback) 260
:scroll-up-page-callback (initarg) 259
scroll-up-page-callback (callback) 260
sensitive 164
sensitivity, controlling 164
sequence (presentation type) 136
sequence-enumerated (presentation type) 137
server paths 477
set, region 42
set-application-themed (function) 11
(setf command-enabled) (generic function) 209
(setf cursor-visibility) (generic function) 326
(setf delegate-sheet-delegate) (generic function) 455
(setf frame-command-table) (generic function) 204
(setf frame-current-layout) (generic function) 207
(setf frame-manager) (generic function) 214
(setf frame-manager-dialog-view) (generic function) 310
(setf frame-manager-menu-view) (generic function) 303
(setf frame-pretty-name) (generic function) 204
(setf frame-properties) (generic function) 212
(setf gadget-client) (generic function) 246
(setf gadget-id) (generic function) 246
(setf gadget-label) (generic function) 250
(setf gadget-label-align-x) (generic function) 250
(setf gadget-label-align-y) (generic function) 251
(setf gadget-label-text-style) (generic function) 251
(setf gadget-max-value) (generic function) 251
(setf gadget-min-value) (generic function) 251
(setf gadget-value) (generic function) 248
(setf graph-node-children) (generic function) 437
(setf graph-node-parents) (generic function) 437
(setf graph-root-nodes) (generic function) 435
(setf medium-background) (generic function) 67, 465
(setf medium-buffering-output-p) (generic function) 332
(setf medium-clipping-region) (generic function) 68, 466
(setf medium-default-text-style) (generic function) 69, 467
(setf medium-foreground) (generic function) 67, 465
(setf medium-ink) (generic function) 67, 465
(setf medium-line-style) (generic function) 68, 466
(setf medium-text-style) (generic function) 467
(setf medium-transformation) (generic function) 68, 465

<Bold>553
(setf pointer-cursor) (generic function) 373
(setf pointer-sheet) (generic function) 373
(setf port-keyboard-input-focus) (generic function) 453
(setf port-properties) (generic function) 478
(setf presentation-object) (generic function) 123
(setf presentation-type) (generic function) 123
(setf radio-box-current-selection) (generic function) 256
(setf sheet-enabled-p) (generic function) 446
(setf sheet-region) (generic function) 448
(setf sheet-transformation) (generic function) 448
(setf space-requirement-height) (function) 232
(setf space-requirement-max-height) (function) 232
(setf space-requirement-max-width) (function) 231
(setf space-requirement-min-height) (function) 232
(setf space-requirement-min-width) (function) 231
(setf stream-current-output-record) (generic function) 348
(setf stream-default-view) (generic function) 159
(setf stream-drawing-p) (generic function) 348
(setf stream-end-of-line-action) (generic function) 330
(setf stream-end-of-page-action) (generic function) 330
(setf stream-input-buffer) (generic function) 364
(setf stream-insertion-pointer) (generic function) 400
(setf stream-primary-pointer) (generic function) 365
(setf stream-recording-p) (generic function) 347
(setf stream-scann-pointer) (generic function) 401
(setf stream-text-cursor) (generic function) 326
(setf stream-text-margin) (generic function) 328
(setf text-style-mapping) (generic function) 101
(setf window-viewport-position) (generic function) 333
(setf* cursor-position) (generic function) 326
(setf* output-record-end-cursor-position) (generic function) 340
(setf* output-start-cursor-position) (generic function) 340
(setf* pointer-position) (generic function) 373
(setf* stream-cursor-position) (generic function) 327
(setf* stream-pointer-position) (generic function) 365
(setf* defined) 514
set-highlighted-presentation (function) 181
:sheet (initarg) 325, 457
sheet (protocol class) 443
sheet-adopt-child (generic function) 445
sheet-allocated-region (generic function) 450
sheet-ancestor-p (generic function) 445
sheet-children (generic function) 444
sheet-delta-transformation (generic function) 450
sheet-device-region (generic function) 485
sheet-device-transformation (generic function) 484
sheet-direct-mirror (generic function) 483
sheet-disown-child (generic function) 445
sheet-enabled-children (generic function) 445
sheet-enabled-p (generic function) 446
(sheet* enabled-p) (generic function) 446
sheet-grafted-p (generic function) 480
sheet-identity-transformation-mixin (class) 450
sheet-leaf-mixin (class) 447
sheet-medium (generic function) 469
sheet-mirror (generic function) 484
sheet-mirrored-ancestor (generic function) 483
sheet-multiple-child-mixin (class) 447
sheet-native-region (generic function) 484
sheet-native-transformation (generic function) 484
sheet-occluding-sheets (generic function) 446
sheetp (function) 443
sheet-parent (generic function) 444
sheet-parent-mixin (class) 447
sheet-region (generic function) 448
sheets 8, 23, 440
adopted 444
associating with a medium 468
basic classes 443
child 440
degrafted 444
disabled 444
disowned 444
enabled 444

554CLIM User Guide
genealogy classes 447
geometry 448
geometry classes 450
geometry functions 448
genealogy notifications 472
grafted 444
input protocol 452
input protocol classes 454
input protocol functions 452
mediums and output properties 464
mirrored 483
functions 483
mirrors
functions 483
notification protocol 472
output protocol 464
output protocol classes 467
output protocol functions 467
parent 440
properties of 441
protocol 441, 444
relationships between 444
sheet-siblings (generic function) 445
sheet-single-child-mixin (class) 447
sheet-transformation (generic function) 448
(setf sheet-transformation) (generic function) 448
sheet-transformation-mixin (class) 451
sheet-translation-mixin (class) 451
sheet-viewable-p (generic function) 446
sheet-y-inverting-transformation-mixin (class) 451
+shift-key+ (constant) 462
:show-as-default (initarg) 255
:show-value-p (initarg) 261
shrink-frame (generic function) 215
shrunken frames 212
simple-parse-error (condition) 390
simple-parse-error (function) 390
:single-box (initarg) 124
singular-transformation (error condition) 83
:size (initarg) 338
slider (class) 261
slider gadgets 261
slider-drag-callback (generic function) 261
slider-pane (class) 262
solid shapes 73
space
allocation 230
composition 230
requirement 230
space-requirement (protocol class) 231
space-requirement+ (function) 232
space-requirement+* (function) 232
space-requirement-combine (function) 232
space-requirement-components (generic function) 232
(space-requirement-height (function) 231
(setf space-requirement-height) (function) 232
space-requirement-max-height (function) 232
(setf space-requirement-max-height) (function) 232
space-requirement-max-width (function) 231
(setf space-requirement-max-width) (function) 231
space-requirement-min-height (function) 232
(setf space-requirement-min-height) (function) 232
space-requirement-min-width (function) 231
(setf space-requirement-min-width) (function) 231
space-requirement-width (function) 231
(setf space-requirement-width) (function) 231
:spacing (option) 226
spacing (macro) 227
spacing-pane (composite pane) 227
specifiers, presentation type 117
spread versions of drawing functions 46
standard input 204, 237
standard output 205, 238
*standard-activation-gestures* (variable) 388
standard-application-frame (class) 189
standard-bounding-rectangle (class) 61
standard-cell-output-record (class) 433
standard-column-output-record (class) 432
standard-command-table (class) 276
standard-ellipses (class) 56
standard-elliptical-arc (class) 56
standard-extended-input-stream (class) 364
standard-extended-output-stream (class) 322
standard-graph-node-output-record (class) 436
standard-input-editing-stream (class) 384

<Bold>555
standard-input-stream (class) 362
standard-item-list-output-record (class) 434
standard-line (class) 51
standard-line-style (class) 74
standard-output-recording-stream (class) 347
standard-output-stream (class) 320
standard-point (class) 47
standard-pointer (class) 372
standard-polygon (class) 48
standard-polyline (class) 48
standard-presentation (class) 124
standard-rectangle (class) 52
standard-rectangle-set (class) 42
standard-region-difference (class) 42
standard-region-intersection (class) 42
standard-region-union (class) 42
standard-repainting-mixin (class) 471
standard-row-output-record (class) 431
standard-sequence-output-record (class) 343
standard-sheet-input-mixin (class) 454
standard-sheet-output-mixin (class) 467
standard-table-output-record (class) 429
standard-text-cursor (class) 325
standard-text-style (class) 96
standard-tree-output-history (class) 346
standard-tree-output-record (class) 344
:state (initarg) 189
stencils 31
stream-accept (generic function) 127
stream-add-character-output (generic function) 350
stream-add-output-record (generic function) 348
stream-add-string-output (generic function) 350
stream-advance-to-column (generic function) 321, 522
stream-baseline (generic function) 329
stream-character-width (generic function) 327
stream-clear-input (generic function) 363, 520
stream-clear-output (generic function) 321, 522
stream-close-text-output-record (generic function) 350
stream-current-output-record (generic function) 348
(stream stream-current-output-record) (generic function) 348
stream-cursor-position (generic function) 327
(stream stream-cursor-position) (generic function) 327
stream-default-view (generic function) 159
(stream stream-default-view) (generic function) 159
stream-drawing-p (generic function) 348
(stream stream-drawing-p) (generic function) 348
stream-element-type (generic function) 518
stream-end-of-line-action (generic function) 329
(stream stream-end-of-line-action) (generic function) 330
stream-end-of-page-action (generic function) 330
(stream stream-end-of-page-action) (generic function) 330
stream-finish-output (generic function) 321, 522
stream-force-output (generic function) 321, 522
stream-fresh-line (generic function) 321, 522
stream-increment-cursor-position (generic function) 327
stream-input-buffer (generic function) 364, 400
(stream stream-input-buffer) (generic function) 364
stream-input-wait (generic function) 367
stream-insertion-pointer (generic function) 400
(stream stream-insertion-pointer) (generic function) 400
stream-line-column (generic function) 320, 521
stream-line-height (generic function) 328
stream-listen (generic function) 363, 520
stream-output-history (generic function) 348
stream-output-history-mixin (class) 346
stream-pointer-position (generic function) 365
(stream stream-pointer-position) (generic function) 365
stream-pointers (generic function) 364
stream-present (generic function) 121
stream-primary-pointer (generic function) 365
(stream stream-primary-pointer) (generic function) 365
stream-process-gesture (generic function) 402
stream-read-byte (generic function) 523
stream-read-char (generic function) 362, 519
stream-read-char-no-hang (generic function) 362, 520

<Bold>556CLIM User Guide
stream-read-gesture (generic function) 366, 402
stream-read-line (generic function) 363, 520
stream-recording-p (generic function) 347
(stream-recording-p) (generic function) 347
stream-replay (generic function) 348
stream-rescanning-p (generic function) 401
stream-restore-input-focus (generic function) 365
streams 11, 23, 516
  basic
    protocol 516
    basic input 362
    basic output 320
    binary 523
    character input 519
    character output 521
    classes 516
  extended
    panes 184
    protocol 516
    extended input 363
    conditions 368
    protocol 364
    extended output 321
    functions 518
    hardcopy 523
    input editing 382
    protocol 400
    output recording 347
    protocol 347
    window 525
    window operations 525
    window, pane functions 332
stream-scan-pointer (generic function) 401
(stream-scan-pointer) (generic function) 401
stream-set-input-focus (generic function) 365
stream-start-line-p (generic function) 320, 521
stream-string-width (generic function) 328
stream-terpri (generic function) 321, 522
stream-text-cursor (generic function) 326
(stream-text-cursor) (generic function) 326
stream-text-margin (generic function) 328
(stream-text-margin) (generic function) 328
stream-text-output-record (generic function) 349
stream-truename (generic function) 519
stream-unread-char (generic function) 362, 519
stream-unread-gesture (generic function) 368, 402
stream-vertical-spacing (generic function) 328
stream-write-byte (generic function) 523
stream-write-char (generic function) 320, 521
stream-write-string (generic function) 320, 522
string (presentation type) 133
string completion 393
styles
  interaction 280
    command line 292
    command menus 287
    keystroke accelerators 293
    mouse 291
  line 74
subset (presentation type abbreviation) 136
subset-alist (presentation type abbreviation) 136
subset-completion (presentation type) 135
subset-sequence (presentation type abbreviation) 136
substitute-numeric-argument-marker (function) 295
suggest (function) 397
+super-key+ (constant) 462
surrounding-output-with-border (macro) 426
symbol (presentation type) 131
T
t (presentation type) 130
table-output-record (protocol class) 429
table-output-record-p (function) 429
table-pane (composite pane) 227
tables
  command 275, 285
  predefined 278
formatting 404
  calendar month example, figure of 414
  examples 411
  figure of 415, 417
  from a list, figure of 412
  operators for 405
  output records of, figure of 337
  protocol 429
tabling (macro) 227
template for application frame 15
temporary-medium-sheet-output-mixin (class) 468
test (initarg) 253, 254
test-presentation-translator (function) 177
text-style (option) 223
text cursors 323
    protocol 325
    stream, protocol 326
formatting 424
line wrapping 329
mixing with graphics 329
output recording 349
output records 345
protocol 327
text styles 94
    ascent 98
    binding forms 100
    descent 98
    face 94, 96, 101
    family 94, 96
    functions 97
    height 99
    mapping 101
    objects 95
    options 96
    size 94, 96
:text-cursor (initarg) 364
text-displayed-output-record (protocol class) 345
text-displayed-output-record-p (function) 345
text-displayed-output-record-string (generic function) 346
text-editor (class) 263
text-editor gadgets 262
text-editor-pane (class) 263
:text-face (option) 97
:text-family (option) 96
text-field (class) 262
text-field gadgets 262
text-field-pane (class) 263
:text-margin (initarg) 322
:text-margin (option) 236
:text-size (option) 97
text-size (generic function) 99
:text-style (option) 73
text-style (protocol class) 96
text-style-ascent (generic function) 98
text-style-components (generic function) 98
text-style-descent (generic function) 98
text-style-face (generic function) 97, 98
text-style-family (generic function) 96, 98
text-style-fixed-width-p (generic function) 99
text-style-height (generic function) 99
text-style-mapping (generic function) 101
    (setf text-style-mapping) (generic function) 101
text-style-p (function) 96
text-style-size (generic function) 97, 98
text-style-width (generic function) 99
+textual-dialog-view+ (constant) 160
textual-dialog-view (class) 160
+textual-menu-view+ (constant) 160
textual-menu-view (class) 159
throw-highlighted-presentation (function) 180
timer-event (class) 461
:timestamp (initarg) 457
title-pane (leaf pane) 238
toggle-button (class) 264
toggle-button gadgets 263
toggle-button-indicator-type (generic function) 264
toggle-button-pane (class) 264
token-or-type (presentation type abbreviation) 138
tokens, reading and writing 391
toolkit, adaptive 10
top-level output records 346
tracking-pointer (macro) 373
:transformation (option) 72
transformation (protocol class) 83
transformation-equal (generic function) 84
transformation-error (error condition) 83
transformationp (function) 83
transformations 78
    affine 79
    applying 89
    composition 80
    constructors 80
    functions 85
    graphic, example, figure of 78
    predicates 84
    protocol 83
    reflection 79
    rotation 79
<Bold>558CLIM User Guide
scaling 79
translation 79
transformation-underspecified (error condition) 83
transform-distance (generic function) 90
transform-position (generic function) 90
transform-rectangle* (generic function) 90
transform-region (generic function) 89
translate 125
translation 79
translation-transformation-p (generic function) 84
translators, presentation 116, 125, 176, 291
applicability 166
concepts 164
defining
examples 175
operators for 170
low-level functions 177
matching 166
using, figure of 14
:type (initarg) 124
type-or-string (presentation type abbreviation) 138
types of output records 343

U
unbounded regions 38
*undefined-text-style* (variable) 95
unhighlight-highlighted-presentation (function) 181
unread-gesture (function) 367
*unsupplied-argument-marker* (variable) 297
untransform-distance (generic function) 90
untransform-position (generic function) 90
untransform-rectangle* (generic function) 91
untransform-region (generic function) 89
updating-output (macro) 355
user packages 507
user-command-table (command table) 278
using drawing options 70
using gadgets 241
using-resource (macro) 510

V
:value (initarg) 248
:value-changed-callback (initarg) 248
value-changed-callback (callback) 248
value-gadget (class) 248
:value-key (initarg) 253, 254
vbox-pane (composite pane) 227
vertically (macro) 227
:vertical-spacing (initarg) 322
:vertical-spacing (option) 236
:view (initarg) 124
view (protocol class) 159
viewp (function) 159
viewports 228
defined 526
views
of presentation types
operators for 159
with presentation types 159
vrack-pane (composite pane) 228

W
:width (option) 225
:win32 (server-path) 477
window-children (generic function) 527
window-clear (generic function) 332, 525
window-configuration-event (class) 461
window-erase-viewport (generic function) 333, 525
window-event (class) 461
window-event-mirrored-sheet (generic function) 461
window-event-native-region (generic function) 461
window-event-region (generic function) 461
window-expose (generic function) 528
windowing relationships 440
window-inside-bottom (function) 529
window-inside-edges (generic function) 528
window-inside-height (function) 529
window-inside-left (function) 528
window-inside-right (function) 529
window-inside-size (generic function) 529
window-inside-top (function) 529
window-inside-width (function) 529
window-label (generic function) 527
window-parent (generic function) 527
window-refresh (generic function) 332, 525
window-repaint-event (class) 461
windows 332, 525
concepts 440
functions for direct operation 527
operators for viewport and scrolling 526
origin 21
primitive layer operators 527
stream operations 525
stream pane functions 332
streams 525
Windows themes 11
window-set-viewport-position* (generic function) 526
window-stack-on-bottom (generic function) 528
window-stack-on-top (generic function) 528
window-viewport (generic function) 333, 526
window-viewport-position (generic function) 333
(setf window-viewport-position) (generic function) 333
window-viewport-position* (generic function) 526
window-visibility (generic function) 528
with-accept-help (macro) 397
with-activation-gestures (macro) 388
with-application-frame (macro) 188
with-bounding-rectangle* (macro) 63
with-command-table-keystrokes (macro) 283
with-delimiter-gestures (macro) 389
with-drawing-options (macro) 70
with-end-of-line-action (macro) 330
with-end-of-page-action (macro) 331
with-first-quadrant-coordinates (macro) 89
with-frame-manager (macro) 214
with-graft-locked (macro) 481
:within-generation-separation (initarg) 435
with-input-context (macro) 126
with-input-editing (macro) 385
with-input-editor-typeout (macro) 385
with-input-focus (macro) 366, 527
with-local-coordinates (macro) 88
with-lock-held (macro) 513
with-look-and-feel-realization (macro) 222
with-menu (macro) 305
with-new-output-record (macro) 351
with-output-as-gadget (macro) 265
with-output-as-presentation (macro) 120
with-output-buffered (macro) 332
with-output-recording-options (macro) 351
with-output-to-output-record (macro) 352
with-output-to-pixmap (macro) 35
with-output-to-postscript-stream (macro) 524
without-scheduling (macro) 512
with-port-locked (macro) 478
with-presentation-type-decoded (macro) 140
with-presentation-type-options (macro) 141
with-presentation-type-parameters (macro) 141
with-radio-box (macro) 258
with-recursive-lock-held (macro) 513
with-room-for-graphics (macro) 329
with-rotation (macro) 88
with-scaling (macro) 88
with-sheet-medium (macro) 468
with-sheet-medium-bound (macro) 469
with-text-face (macro) 101
with-text-family (macro) 101
with-text-size (macro) 101
with-text-style (macro) 100
with-translation (macro) 87
wrapping text lines 329
write-token (function) 392
writing tokens 391
X
:x (initarg) 459
X resources
  defaults 11
  file 11
:x-position (initarg) 338
:x-spacing (initarg) 429, 433
:x-spacing (option) 226
Y
:yy (initarg) 459
:yy-position (initarg) 338
:yy-spacing (initarg) 429, 433
:yy-spacing (option) 226
</Bold>560CLIM User Guide