Common Lisp Interface Manager (CLIM)

PREFACE

This book documents how to develop programs using the Common Lisp Interface Manager (CLIM), Release 2.0. It is intended for use by programmers who are experienced with Common Lisp and CLOS programming concepts.

This book is useful for a programmer writing CLIM 2.0 applications on any Lisp platform, but includes some material that is specific to CLIM on Genera and Cloe.

Organization

This document has these major parts:

"What is CLIM?"	Describes the roots of CLIM 2.0, gives a technical overview of it, and compares it to Genera's Dynamic Windows.
"CLIM Tutorial"	Gets you started developing programs in CLIM with extended examples that you can run. This section introduces the important concepts of CLIM via these examples.
"CLIM User's Guide"	Provides concept-oriented reference documentation covering topics in more detail than the Tutorial. The chapters in the User's Guide also include summaries of functions, classes, macros, etc., which are documented more fully in the Dictionary.
"CLIM Dictionary"	Provides detailed reference documentation of CLIM functions, classes, macros, and so forth, in alphabetic order. You can use the CLIM Dictionary when you need to know the syntax and semantics of a particular CLIM operator.
"Glossary of CLIM Terminology"	Provides a description of many CLIM terms.

How to Use This Book

Start with "What is CLIM?" for background information and a technical overview of CLIM.

When you are ready to start programming, proceed to the "CLIM Tutorial", and experiment with the examples. Code for these examples can be found in the directory `SYS:CLIM;REL-2;TUTORIAL;`. The directory `SYS:CLIM;REL-2;DEMO;` is also a rich source of example code.

When you want to learn about a topic in more detail, you can use the "CLIM User's Guide" and the "CLIM Dictionary").

Documentation Conventions

This documentation uses the following notation conventions:

cond, **clim:accept**	Printed representation of Lisp objects in running text.
`Return`, `Abort`, `control-F`	Keys on the keyboard.
`SPACE` or `Space`	The space bar.
`login`	Literal typein.
`(make-symbol "foo")`	Lisp code examples.
(**my-find** *item list* &optional *test* &key *:start :end*)	Syntax description of the invocation of the function **my-find**.
item, *list*, *test*	Arguments to the function **my-find**, usually expressed as a word that reflects the type of argument (for example, *string*).
:start, *:end*	Keyword arguments to the function **my-find**.
&optional, &key	Introduces optional or keyword argument(s).
Show File, Start	Command processor command names appear with the initial letter of each word capitalized.
`m-X` Insert File, Insert File (`m-X`)	Extended command names in Zmacs, Zmail, and Symbolics Concordia appear with the `m-X` notation either preceding the command name, or following it in parentheses. Both versions mean press `m-X` and then type the command name.
[Map Over]	Menu items. Click Left to select a menu item, unless other operations are indicated. (See the section "Mouse Command Conventions".)
Left, Middle, Right	Pointer gestures (also known as mouse clicks).
`shift`-Right, `c-m`-Middle	Modified mouse clicks. For example, `shift`-Right means hold down the `SHIFT` key while clicking Right on the mouse, and `c-m`-Middle means hold down the `CONTROL` and `META` keys while clicking Middle.

Modifier Key Conventions

Modifier keys are designed to be held down while pressing other keys. They do not themselves transmit characters. A combined keystroke like `Meta-X` is pronounced "meta x" and written as `m-X`. This notation means that you press the `META` key and, while holding it down, press the `X` key.

Modifier keys are abbreviated as follows:

`CONTROL`	`c-` or `control-`

META	m- or meta-
SUPER	s- or super-
HYPER	h- or hyper-
SHIFT	sh- or shift-

These modifier key names are based on the labels on a Symbolics keyboard, and most keyboards do not label their modifier key names the same way. This is just a convention; non-Symbolics machines will simply use differently labelled keys on the keyboard.

Modifier keys can be used in combination, as well as singly. For example, the notation c-m-Y indicates that you should hold down both the CONTROL and the META keys while pressing Y.

Modifier keys can also be used, both singly and in combination, to modify mouse commands. For example, the notation sh-Left means hold down the SHIFT key while clicking Left on the mouse and c-m-Middle means hold down CONTROL and META while clicking Middle.

The keys with white lettering (like X or SELECT) all transmit characters. Combinations of these keys should be pressed in sequence, one after the other (for example, SELECT L). This notation means that you press the SELECT key, release it, and then press the L key.

Trademarks

CLIM is a registered trademark of International Lisp Associates (ILA). Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Windows, Windows/286 and Windows/386 are trademarks of Microsoft Corporation. Intel and 386 are trademarks of Intel Corporation. Adobe and PostScript are registered trademarks of Adobe Systems Inc.

What is CLIM?

Technical Overview of CLIM

CLIM is an acronym for the Common Lisp Interface Manager. It is a portable, powerful, high-level user interface management system toolkit intended for Common Lisp software developers. The important things to understand about CLIM are:

- *How it helps you achieve a portable user interface* — how it fits into an existing host system; how you can achieve the look and feel of the target host system without implementing it directly, and without using the low-level implementation language of the host system.
- *The power inherent in its presentation model* — the advantages of having the visual representation of an object linked directly to its semantics.

- *The set of high-level programming techniques it provides* — capabilities that enable you to develop a user interface conveniently, including formatted output, graphics, windowing, and commands that are invoked by typing text or clicking a mouse (or ''pointer'') button, among other techniques.

CLIM 2.0 does not currently provide any high-level user interface building tools, nor does it provide any sort of high-level graphical or text editing substrate. These are areas into which future releases of CLIM may extend.

CLIM is also not suitable for high performance, very high quality graphics of the sort needed for sophisticated paint, animation, or video postproduction applications. Of course, it is possible to write the bulk of such an application's user interface using CLIM, and use lower level facilities for drawing in the main ''canvas'' of the application.

Introduction to CLIM's Presentation Model

A software application typically needs to interact with the person using it. The user interface is responsible for managing the interaction between the user and the application program. The user interface gets information from the user (commands, which might be entered by typing text or by clicking a mouse), gives that information to the application, and later presents information (the program's results) to the user. Figure 25 shows this common paradigm.

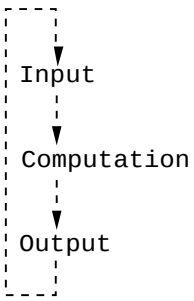


Figure 46. Cycle of Input/Computation/Output

We might describe one conventional approach to the input/computation/output cycle as follows. Invisibly to the user, the application takes the commands and interprets them in terms the program can handle. For example, if the application uses object-oriented techniques, it might build objects based on information garnered from the command and arguments, then manipulate those objects internally, finish its computation, and finally translate from the resulting objects to the appropriate response which is then given to the user. Figure 26 depicts this sequence of events.

The conventional approach uses object-oriented techniques within the computation phase, but the objects do not surface to the user interface. The program performs two translations: from user input to objects, and later from objects to output in-

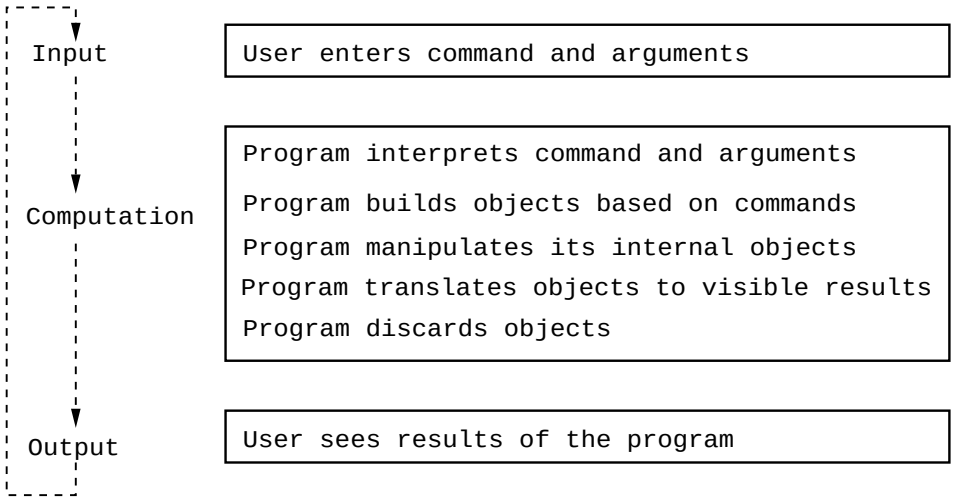


Figure 47. Conventional Approach to the Input/Computation/Output Cycle

tended for the user. CLIM revolutionizes the cycle by bringing the power of object-oriented programming to the surface, all the way up to the user interface.

CLIM recognizes that many applications manipulate internal objects which we call *application objects* and they have *display objects*, which are presented to the user. A display object can appear as text or a graphic picture. CLIM supports a direct linking between application objects and display objects. CLIM automatically maintains the association between application objects and display objects, so there is no need for the application to do any translation. Figure 27 shows how CLIM views the cycle of input/computation/output.

In effect, CLIM replaces some of the tedious and error-prone steps of the conventional user-interface model with higher-level object-oriented techniques. The advantages of the object-oriented user interface are subtle but extremely powerful (in fact, you might not recognize them at first glance, but they will grow on you gradually as you develop your CLIM applications):

- A command is structured so that the user interface understands something of the semantics of its arguments. That is, each argument must be an object of a specified type. This helps the user in several ways:
  - The user is prevented from entering invalid input, because the user interface automatically enforces the validity of each argument.
  - The user can get online help or prompting from the user interface, based on the type of the argument.
  - The user can enter input in creative and convenient ways, such as by clicking on object displayed on the screen by a previous command. The user interface knows which displayed objects are valid within the current context, and can make them *sensitive* (the objects are highlighted as the pointer passes over them).

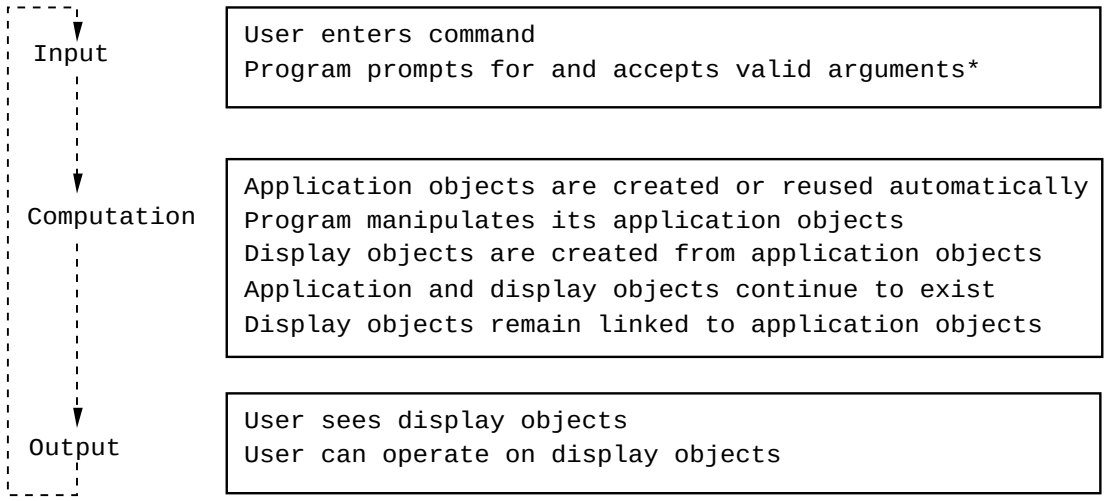


Figure 48. CLIM Approach to the Input/Computation/Output Cycle

  - The user has a flexible means of interacting with the application, and often can choose whether to use the mouse or keyboard to communicate with the application.

- In CLIM, the user interface directly reflects the application's structure, because the display objects stand for application objects. Unlike the conventional model, a CLIM user interface is not tacked on the application as a separate entity which can diverge from the application to ill effect. CLIM's direct linking between the application and display objects ensures a natural consistency between the application and its user interface.

- Display objects are organized in a type lattice in the usual object-oriented way, so inheritance can be used to good advantage. For example, when the user is entering an argument of a given type, objects of that type *and its subtypes* are valid as input. For example, an application might define display objects representing buildings, schools, and houses. When the application needs a building as input, the user can enter a school, because school is a subtype of building. CLIM also offers a library of predefined types, saving the application programmer some effort when dealing with commonly used display types.

- Objects can be shared freely among different applications. CLIM's ability to share objects directly contrasts to some conventional systems, in which data can be shared among applications only by reducing it to its lowest common denominator (usually text).

How CLIM Helps You Achieve a Portable User Interface

CLIM provides a consistent stream-oriented interface to window systems across a large set of hosts. When developing a portable user interface, you write your application in terms of CLIM windows and their operations. You can also use Common Lisp and CLOS. Figure 28 shows the elements on which your application depends.

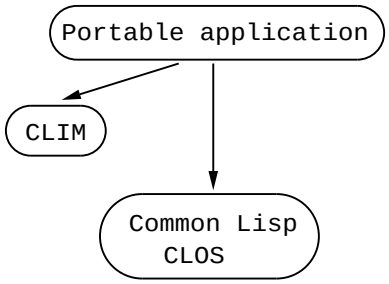


Figure 49. Foundation of a Portable Application

Your application is portable because it depends only on languages which have been standardized: Common Lisp, CLOS, and CLIM. Of course, porting is never entirely effortless, because different implementations of standardized languages can differ from one another in minor ways.

From the perspective of your application, the details of the host window system, host operating system, and host computer should be invisible. CLIM handles the interaction with the underlying window system. Figure 29 shows the elements of the host system from which CLIM shields your application.

CLIM shields you from the details of any one window system by abstracting out the concepts that many window systems have in common. Using CLIM, you specify the appearance of your application's output in general, high-level terms. CLIM turns your high-level description into the appropriate appearance for a given window system.

In some cases, you might prefer to control the appearance of your user interface more directly. You can bypass CLIM and use functions provided in the underlying window system or toolkit to achieve more explicit control, at the expense of portability.

Highlights of CLIM Tools and Techniques

CLIM offers the following tools and techniques:

Windows and events
: A portable layer for implementing *sheet* classes (types of window-like objects) that are suited to support particular high level facilities or interfaces. The windowing module of CLIM de-

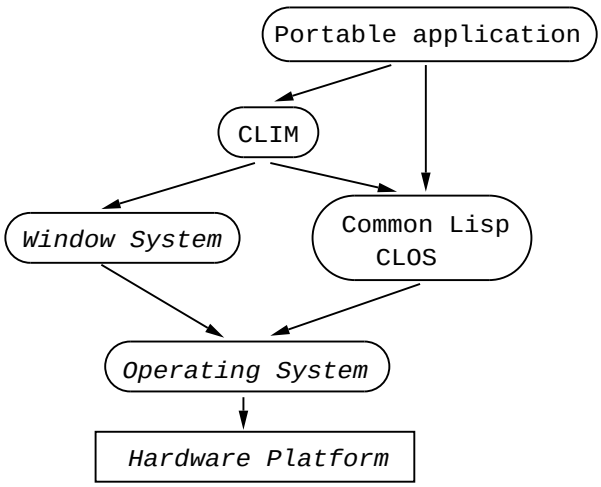


Figure 50. How CLIM is Layered Over the Host System

fines a uniform interface for creating and managing hierarchies of these objects regardless of their sheet class. This layer also provides event management.

Graphics — A rich set of drawing operations, including complex geometric shapes, a wide variety of drawing options (such as line thickness), and a sophisticated color inking model. CLIM provides full affine transforms, so that you can perform arbitrary translations, rotations, and scaling of drawings.

Output recording — A facility for capturing all output done to a stream, which provides the automatic support for scrollable windows. In many cases, programs produce output in the form of *display objects*, which can be manipulated directly by the user (see context-sensitive input). Thus, not only is the output recorded — whether textual or graphic — but it also retains its semantics and can be reused when appropriate.

Formatted output — High-level macros that enable you to produce neatly formatted tabular and graphical displays with minimal effort.

Context-sensitive input — Simple, direct means of using a displayed output object as input. As mentioned above, an application can produce output via objects, which retain their semantics. Users can recycle visible output into input for the same application or a different one. Each CLIM application sets up a context for what kind of input is expected at a given time. For example, a CAD program that supports designing electrical circuits might have a com-

mand called `Set Resistance` which sets up an input context in which a resistor object is expected. Any resistors appearing in the CAD programs display are automatically made mouse-sensitive in that context, so the user can click on one to enter it as input.

Adaptive toolkit
: A uniform interface to the standard compositional toolkits available in many environments. CLIM defines abstract panes that are analogous to the gadgets or widgets of a toolkit like Motif or OpenLook. CLIM fosters look and feel independence by specifying the interface of these abstract panes in terms of their function and not in terms of the details of their appearance or operation. If an application uses these interfaces, its user interface will adapt to use whatever toolkit is available in the host environment. By using this facility, application programmers can easily construct applications that will automatically conform to a variety of user interface standards. In addition, a portable CLIM-based implementation of the abstract panes is provided.

Application-building facilities
: High-level facilities for defining applications, helping you to lay out windows and gadgets, manage command menus and menu bars, and link user interface gestures (such as mouse clicks) with application operations. The application-building tools help you construct a flexible user interface that can grow from the prototype to the delivery phase.

Comparing and Contrasting DW and CLIM

This section describes CLIM in terms of how it is similar to and how it differs from Dynamic Windows. It also discusses conversion issues.

CLIM offers some advantages over Dynamic Windows. In brief, these are:

- Ability to develop a portable user interface.
- Support for using toolkits offered on various window systems to achieve the look-and-feel of a given system.
- Simplification of some Dynamic Windows functionality, resulting in greater ease of use and superior performance.
- Exposed underlying protocols, enabling you to modify or extend the behavior of CLIM.

Converting an Application From DW to CLIM

Genera users do not have to convert programs from Dynamic Windows to CLIM. Symbolics will continue to support DW in Genera. In fact, it is possible that the portions of Genera that use Dynamic Windows will not be reimplemented in CLIM.

One good reason to convert an existing program to CLIM is to take advantage of the portability benefit that CLIM provides. If your goal is to deliver an application with a user interface on a variety of Lisp platforms with different window systems, you will probably want to convert the application's user interface to CLIM.

Another good reason to convert a Dynamic Windows program to CLIM is that many of the high level facilities in CLIM are faster than in Dynamic Windows.

Symbolics provides a conversion tool to help automate the procedure of converting programs from DW to CLIM. For more information, see the section "Converting from Dynamic Windows to CLIM".

When developing a new application, you will need to decide whether the user interface should be programmed in Dynamic Windows or CLIM. Although both will coexist in Genera, there is no direct compatibility between them, and hence no mixed programming approach.

Converting an Application From TV to CLIM

Some Genera users have applications that depend on Release 6 window functions in the **tv** package. In some cases, these applications were not converted to Dynamic Windows because of performance reasons. CLIM's performance is superior to that of Dynamic Windows, so for performance reasons, users may want to convert programs that use Release 6 window functions to CLIM.

Note that the Release 6 window system has a very different architecture from DW or CLIM. For example, window programs typically define new flavors of the window with methods that handle very low-level events (such as refresh and mouse motion). Because of this architectural difference, converting from **tv** to CLIM usually requires a careful redesign of an application's user interface, and the usefulness of automatic conversion tools is limited.

Similarities Between Dynamic Windows and CLIM

- CLIM supports essentially the same presentation model as that in Dynamic Windows. CLIM captures the Dynamic Windows' philosophy that a program's user interface should reflect its semantics.
- CLIM provides a graphics model which is similar to that of Dynamic Windows, but is simplified and more uniform.
- CLIM includes an application-building tool similar to the **dw:define-program-framework** of Dynamic Windows.
- CLIM's command processor is virtually identical to that of Dynamic Windows.
- CLIM's input editor is a simplification of that of Dynamic Windows.
- CLIM supports a version of Genera's character styles. In Dynamic Windows characters, strings, and displayed text have style. In CLIM only displayed text has style.

- CLIM supports completion and context-sensitive help in the spirit of Dynamic Windows.

CLIM as a UIMS, and Not a Window System

Dynamic Windows plays two distinct roles. It is both a window system and a user interface management system. CLIM is not a window system; it is layered on top of some other window system, such as X, NeWS, or Microsoft Windows. Therefore, CLIM recasts the interfaces of Dynamic Windows related to being a window system in a portable manner. CLIM encompasses the functionality of many window systems; it acts as an ''abstract window system'' or a ''generic window system'' which can be layered on top of another window system. CLIM enables you to develop a portable user interface, whereas Dynamic Windows does not.

The portions of Dynamic Windows that are directly related to its role as a window system are not included in CLIM. For example, in Dynamic Windows, you can operate on a window by sending it a message because some dynamic windows are implemented as flavors that use the message-sending paradigm. CLIM does not support that paradigm.

CLIM, like the second role of Dynamic Windows, is a user interface management system. CLIM shares the philosophy that you as programmer should be able to express what you want to do in high-level terms, and the system should manage the details for you.

CLIM is Built up From Layered Protocols

Whereas Dynamic Windows includes a great deal of flexibility in its single documented interface, CLIM is a layered protocol in the spirit of CLOS. In this document, we refer to the higher level as the CLIM Application Programmer Interface (or API) and the underlying level as the CLIM Class Protocol.

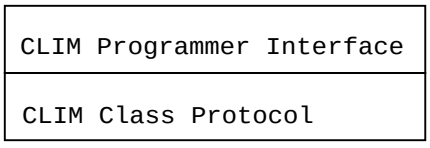


Figure 51. CLIM Protocol Layers

At the API level, an important design goal is that there should be one simple way to do something. There can be some exceptions to this goal; when a very common idiom is identified, it might be included even if there is another (more verbose) way to do the same thing. Where some Dynamic Windows functions and macros offer many keyword arguments, CLIM pares these down to a minimal set without sacrificing functionality.

The CLIM Class Protocol is exposed to allow advanced users to modify or extend CLIM in the object-oriented way. The API functions and macros are implemented in terms of the CLIM Class Protocol. The CLIM Class Protocol, for the most part, is not documented in this book. If you are interested in the CLIM Class Protocol, you should consult the **CLIM II Specification**.

Comparing the Presentation Type Systems

Dynamic Windows allows the presentation type lattice to be computed at run-time. In Dynamic Windows, using inheritance can get complicated, because you must specify what happens at run-time. In CLIM, the type lattice is fixed at load-time, as it is in CLOS. By fixing the type lattice at load-time, CLIM achieves a performance improvement and simplifies the conceptual model. In practice, this restriction has had no negative effects on any applications, and has the benefit of making CLIM's presentation type system far faster than the Dynamic Windows presentation type system.

The CLIM presentation type system is a straightforward extension of the CLOS type system. In CLIM, defining a presentation type is similar to defining a CLOS class. CLIM extends the CLOS type system by supporting parameterized types, such as integer ranges. This has the benefit of making the CLIM presentation type system ''feel'' almost exactly like CLOS.

CLIM's Unified Geometric Model

CLIM includes a unified geometric model which is used to represent windows, graphics, and widgets. In other words, everything from a window itself to the graphics drawn on it conforms to the same geometric model. This enables you to deal with windows and graphics in a uniform way. CLIM also provides a general model for transforming, rotating, and scaling geometric objects. CLIM's unified geometric model results in a simplification of some mechanisms used in Dynamic Windows.

CLIM and User Interface Appearance

It is an ambitious goal of CLIM to bridge a wide gap between two styles of user interface programming.

In Genera's style, the principal goal is for the user interface to convey the application's semantics. This goal leads to a natural consistency between the application and its user interface. However, Genera and Dynamic Windows have been weak in enabling programmers to specify a unique and attractive appearance of the user interface. In other words, Genera has tended to sacrifice form for content.

Many commercial toolkits have powerful means of controlling the visual appearance of a user interface. Traditionally, these toolkits offer no support at all for connecting the application's semantics to its user interface. The user interface is thus designed and implemented as a separate, add-on piece to the application. In other words, the toolkits tend to sacrifice content for form.

What's missing from each of these approaches is the connection between the semantics and the appearance of a user interface. CLIM enables the programmer to specify the semantics and appearance of the user interface in an integrated way. It provides the glue between the two.

For example, suppose that your user interface wants to use a dialog to read a real number in the range from zero to ten from the user. A conventional toolkit might make it easy to provide a visually attractive slider to prompt the user, but when the application receives the input, there are no semantics associated with it; the programmer must write some callback that handles events on the slider and converts them to the desired real number. In Genera, the straightforward way to get the number is to give a textual prompt such as "Enter a number from 0 to 10". The appearance of the prompt is not particularly appealing, but when the input arrives, Genera knows its semantics; it is a real number in the correct range. CLIM aims to include the strength of each of these paradigms. The presentation model maintains the link between the application's semantics and its user interface. The adaptive toolkit enables you to provide a visually attractive user interface. So, if you want to use a slider to get a real number in the range from 0 to 10 from the user, you can use the following:

```
(clim:accept '((real 0 10)) :view clim:+slider-view+)
```

A Tutorial on the Common Lisp Interface Manager (CLIM)

What is CLIM and Why Should I Learn About It?

The CLIM user interface manager provides a novel way to connect input and output to the semantics of the application. It is not a window system but rather allows you to write portable applications that use the underlying window system and/or toolkit interface.

This tutorial provides several examples of working code and then discuss the features of CLIM used in the code. While familiarity with Common Lisp programming is definitely a prerequisite for using CLIM, there is no need to be a ''wizard'' to read the examples. We have tried to shift the balance between simplicity and realism of the examples toward simplicity as much as possible.

Many of the facilities described in this tutorial are not fully documented here; for full documentation we refer you to the "CLIM User's Guide" and the "Dictionary of CLIM Operators". The intent of the tutorial is to familiarize you with the basic use of the CLIM application-building facilities; this will prepare you to understand the reference documentation.

> **Note for the advanced reader:** Throughout this tutorial you will see text like this. This is an indication to you that the material contained here is meant for the more knowledgeable reader. The beginning reader is encouraged to skip this material; you will not lose any of the basic lesson of the tutorial.
>
> This tutorial simplifies a lot of material. The purpose of these notes is to warn the advanced reader against reading too much into the simplified description of certain CLIM features presented

in the tutorial. In cases where a particularly simplified version of CLIM functionality is presented, a note like this will follow, indicating what simplifications have been made.

The Fifteen Puzzle — an Elementary Application

This chapter and the following two chapters show the evolutionary development of an elementary CLIM application. While the example is simple and short, it illustrates many of the features of a typical CLIM application. We present a series of complete, working programs. The programs can be short and yet complete because they make extensive use of high-level facilities provided by CLIM.

For this first example application, we have chosen the famous ''Fifteen Puzzle'', a sliding block puzzle which dates back more than a century. [Ref: *Sliding Piece Puzzles*, Edward Hordern, Oxford University Press, 1986.] The Fifteen Puzzle consists of a 4 by 4 array of spaces, fifteen of which are filled by consecutively numbered pieces. The remaining space is open, which allows a piece to slide into that space, effectively interchanging the piece and the space. Only pieces that are adjacent (horizontally or vertically) to the space can move.

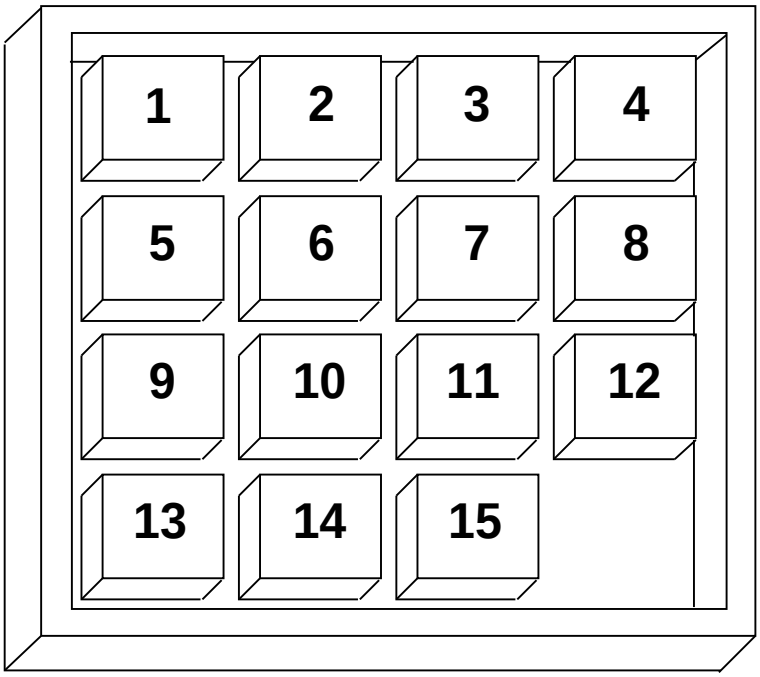


Figure 52. The Fifteen Puzzle

What is an Application?

We have already used the term ‘‘application’’ in this tutorial. Now we will make the use of this term a little more specific. An application is a program that contains a user interface component that interacts directly with the user (as opposed to, say, a library). CLIM lets you build what it calls an **application** which is a Lisp object that encapsulates much of the information necessary for a program to run.

Many applications fall into a simple framework: they consist of a ‘‘loop’’ doing three things, one after the other, repeatedly:

Input	Find out what to do;
Process	Do it;
Output	Show what happened.

Chances are, many programs that you already know fall into this general scheme. Consider an editor, a Lisp listener, or a spreadsheet, and see if you can see how they can be regarded in this manner.

> **Note for the advanced reader:** Of course, many applications *don't* fall into this simple framework. (Examples are applications that employ multiple processes, or applications that deal with asynchronous input from another source as well as the user.) That doesn't mean you can't use CLIM to build those applications too.

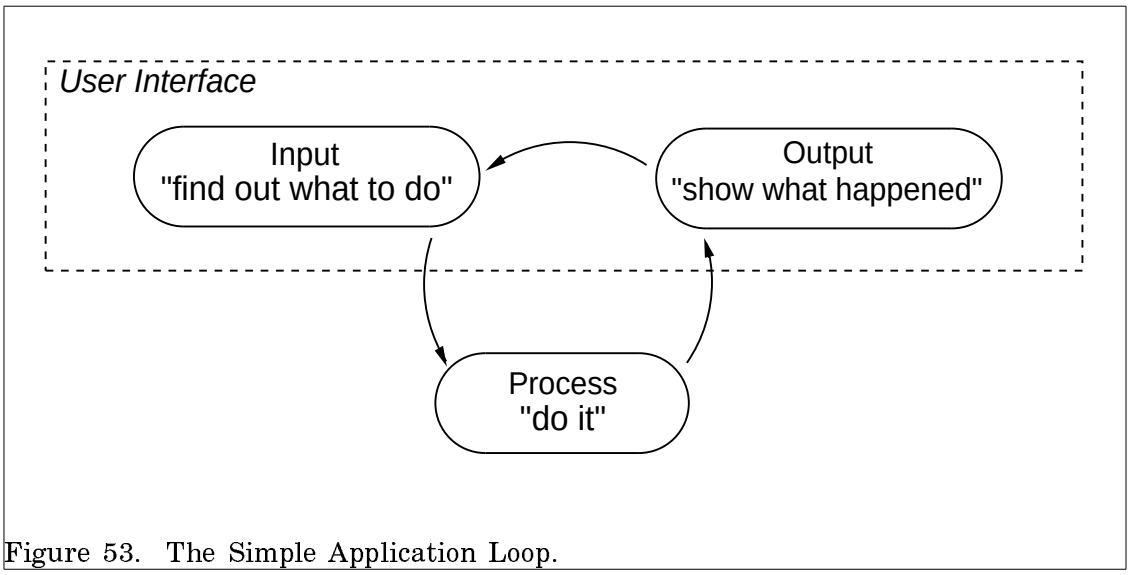


Figure 53. The Simple Application Loop.

Now it's time to look at the first version of the Fifteen Puzzle application. If you are reading this tutorial on-line, or if you have access to a machine, read the file `SYS:CLIM;REL-2;TUTORIAL;PUZZLE-1.LISP` into an editor buffer. In case you don't have on-line access, see the section "Code for Puzzle-1".

If you can do so, run the application right now. Read the file `SYS:CLIM;REL-2;TUTORIAL;PUZZLE-1.LISP` into an editor buffer. Then do the following:

1. Compile the buffer.

   ```
   m-X Compile Buffer
   ```

2. At the end of the file there are two or three forms commented out with `#|` and `|#`. Evaluate those forms by marking them, wiping them, and yanking them into a Listener and pressing `END`, or by marking them and pressing `c-sh-E`.

To exit from this first version of the puzzle, select the window in which you evaluated the **clim:run-frame-top-level** form. (This window should be visible immediately behind the puzzle.) You will need to press `ABORT` or otherwise tell the process to stop executing the application's top-level.

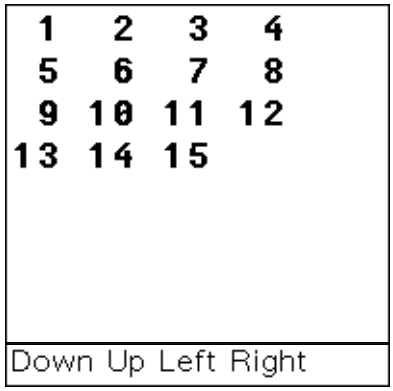


Figure 54. The First Version of the Fifteen Puzzle

The application occupies a small amount of space on the screen and the space is divided into two pieces — a display of the puzzle, and a small menu at the bottom with four menu items: Down, Up, Left, and Right. The whole area is the application's *top level sheet*; because it is subdivided we call it a *frame*, and the two pieces are each called *panes*.

Notice that the menu items *highlight* when the mouse points at them. This is CLIM's indication to you that something will happen when you click the mouse at that place. Try clicking the mouse on any of the four menu items. ''Right'' means ''try to move a piece right into the space'', and similarly for ''Left'', ''Up'', and ''Down''.

The most important defining form is at the beginning of the source file.

```
(clim:define-application-frame fifteen-puzzle-1 ()
    ((pieces :initform ...))
  (:panes
    (display ... )
    (menu ...)))
```

clim:define-application-frame is used here to define a *class* of applications called `fifteen-puzzle-1`. It is important to realize that this is the definition of the class, not the creation of a member of the class.

CLIM is implemented using the Common Lisp Object System — CLOS for short. You don't have to know a lot about CLOS to use CLIM, but if you do know about CLOS you may find that you can apply that knowledge to writing CLIM programs.

Note that the name `"fifteen-puzzle-1"` is used in some other places in the source file. You use the name of an application when making an instance of it or when you write a new command or method for it. (Defining commands for applications is discussed in the section "Application Commands".)

To understand this program there are several other concepts that we must introduce. In the following sections we will discuss:

- an application's *state*,
- an application's *commands*,
- the *panes*, or sub-windows that have already been mentioned,
- and the *display function* associated with a pane.

Application State

A dictionary definition of state is ''a mode or condition of being with respect to a set of circumstances''.

It is often useful for a program to remember things, even when the user is not running the program. Most operating systems let users switch their attention among several activities; users are more productive if they can return to a partly completed task and find the program's appearance and behavior reflecting the *state* of the task.

The Fifteen Puzzle has state. If you put it aside, and later return to it, you expect the pieces to be where you left them.

As the implementor of a program, you might choose to use global variables to store such state. Storing the state as part of the program itself is often a better choice than using global variables because:

- you might want more than one copy of the program around — each copy needs its own state;
- the values representing the state are less vulnerable to being inappropriately modified (for example, by another program in a shared Lisp environment);
- access to the state can be more efficient.

In short, we encourage the practice of storing state as part of the application object as good software engineering.

A CLIM *application* may contain *state variables*. State variables are CLOS slots. They are described in the **clim:define-application-frame** form in the same manner as slots are described in **clos:defclass**. For a full description of CLOS slots, see the section "Accessing Slots of CLOS Instances".

A typical specification of an application state variable might include:

- the name (mandatory),
- an initial value,
- reader, writer, or accessor functions, and
- documentation.

In the Fifteen Puzzle, we have a state variable called **pieces** which is an array containing the current arrangement of pieces in the puzzle.

Application Commands

In CLIM, a *command* is the way to tell an application what to do. A command is a lisp object with structure. It's important to think of a command as an object you (the user) may enter in several different ways. That's how CLIM supports *mixed-mode* interfaces where the user tells the application what to do in different ways.

Commands may be entered by:

- clicking on menu buttons
- clicking on items displayed in the interface (when a piece of a display responds to the user clicking on it, we say the item is *sensitive*)
- typing strings of characters (the facility that turns a string into a command is called a *command parser*)
- typing individual characters that each represent a command (such characters are called *keyboard accelerators*, they are often ''control'' or ''meta'' characters
- a combination of the above.

When you define a CLIM application by using **clim:define-application-frame**, one of the results is the definition of a *command-defining macro* for your application. Since the application is called **fifteen-puzzle-1**, the macro that gets defined is named **define-fifteen-puzzle-1-command**. This is one of the results of naming the class of applications as discussed in "What is an Application?".

> **Note for the advanced reader:** You can override the name that CLIM chooses for the command-defining macro if you wish. You can also decline to have one created at all if you wish.

Here is one of the commands of the Fifteen Puzzle:

```
(define-fifteen-puzzle-1-command (right :menu t) ()
  (with-slots (pieces) clim:*application-frame*
    (find-empty-piece-and-do (y x)
      (unless (zerop x)
        (rotatef (aref pieces y x) (aref pieces y (- x 1)))))))
```

When you write a command, you make the state variables of the application frame via **clos:with-slots**, since **clim:*application-frame*** will be bound to the application frame. In the Fifteen Puzzle command above, we refer to the variable **pieces** in just that way.

The name of the command is **right**. The option **:menu t** says that we want this command to appear in the command menu for the application.

Application Panes

It is often useful to partition your application's screen-area into functional divisions. A menu is such a partition — it groups all the menu buttons into one area. A drawing program may have an area reserved for displaying small versions of saved drawings, or for iconic representations of drawing tools. By placing like things near each other, the application designer can make the interface easier to use.

A *pane* is CLIM's name for a sub-window, that is, a window-like region within a window. The panes within a window form some configuration within a frame, and cannot be exposed or deexposed individually.

CLIM lets you describe how you want your application's window partitioned into panes. There are two parts to the description of an application's panes:

- describing each pane, individually;
- and describing how the panes are laid out to occupy the space available.

The description of an individual pane names the pane, and its type. The description may also contain other options that apply to the pane, such as whether the pane has scroll bars.

Descriptions of individual panes are introduced by the keyword **:panes** in the defining form. In the Fifteen Puzzle, the pane descriptions look like this:

```
(clim:define-application-frame fifteen-puzzle-1 ()
  ...
  (:panes
    (display :application ...)
    (menu :command-menu ...)))
```

Two panes are described, a pane named **display** and a pane named **menu**. Immediately following the pane is a keyword that assigns the type of the pane. The two types of panes here are:

- **:application** — a general purpose type of pane used for display;
- **:command-menu** — a pane used for displaying a menu of commands.

We will see other types of panes later.

We will discuss pane layout later in the tutorial. The definition for the Fifteen Puzzle does not explicitly specify a layout, which means that CLIM will supply a

simple default layout. The default layout consists of stacking all the panes vertically, in the order that the panes are mentioned in the **:panes** description.

Application Output

In the basic application loop that we discussed in "What is an Application?", one of the main functions that an application performed was to update the display to reflect the changed internal state of the application.

In the definition of the Fifteen Puzzle, there is a keyword **:display-function** in the pane description, followed by a name: **draw-the-display**.

```
(clim:define-application-frame fifteen-puzzle-1
           ...
  (:panes
    (display :application
             :text-style '(:fix :bold :very-large)
             :display-function 'draw-the-display
             :scroll-bars nil)
    ...))
```

This is how you tell CLIM how to draw the display you want. Since CLIM is providing the application loop itself, all you need to do is provide the name of a function, and CLIM's default loop will call it after executing a command.

> **Note for the advanced reader:** CLIM offers many options for modifying this behavior. You choose whether the pane should be cleared first or not. You can decline the facility completely and call display functions yourself as part of your commands. You can employ more advanced features of CLIM to incrementally redisplay only those parts of the display that have changed.

CLIM expects to call a function or method that takes two arguments: the application object and the stream on which the output should take place. Whatever function you write must have the correct argument list.

Here is the function that the Fifteen Puzzle uses for drawing the board.

> **Note for the advanced reader:** We have chosen to make the function a method on the application. This is a common choice in many circumstances: it may allow faster slot access to the application's state variables; it is also sometimes done to allow multiple classes of applications to share code — using, for example, **:after** methods.

```
(defmethod draw-the-display ((application fifteen-puzzle-1) stream
                             &key &allow-other-keys)
  (with-slots (pieces) application
    (dotimes (y 4)
      (dotimes (x 4)
        (let ((piece (aref pieces y x)))
          (if (zerop piece)
              (format stream "   ")
              (format stream "~2D " piece))))
      (terpri stream))))
```

Note that this function draws the entire display. In a later chapter, "Further Development of the Fifteen Puzzle", we will see examples of how to draw part of the display, while leaving part of the display unchanged.

Note also the use of **&key** and **&allow-other-keys**. This is because CLIM can pass addition keyword arguments (**:max-width** and **:max-height**) to display functions, so display functions must be prepared to handle them.

Summary

An application is a program that interacts with the user for a specific purpose. An application frame is an object that helps you implement an application. The contents of an application frame are specified by using **clim:define-application-frame**.

Applications have state. You can store values representing pieces of state in an application's state variables.

A command is an object that tells an application what to do. Commands can be entered in several different ways, including clicking on menu buttons.

Many applications run the standard application loop which (1) reads a command, (2) does what the command tells it to do, and (3) updates the display to reflect what it did.

Applications generally have their own window, called a frame, and often subdivide that window into smaller windows called panes. Common types of panes are menus and display panes.

You can tell CLIM the name of a function that draws the display for a particular pane by using the **:display-function** keyword. CLIM will run that function after executing each command.

In the next chapter you'll see how to make the pieces of the puzzle move when you click the mouse directly on them (instead of on menu items).

Using Presentations in the Fifteen Puzzle

If you have not already done so, edit the file `SYS:CLIM;REL-2;TUTORIAL;PUZZLE-2.LISP` and run the new version of the application.

1. Compile the buffer.

   ```
   m-X Compile Buffer
   ```

2. At the end of the file there are two or three forms commented out with `#|` and `|#`. Evaluate those forms by marking them, wiping them, and yanking them into a Listener and pressing `END`, or by marking them and pressing `c-sh-E`.

The biggest difference from the previous version is that now you can move pieces by clicking on them. Another difference is that the old menu commands Down,

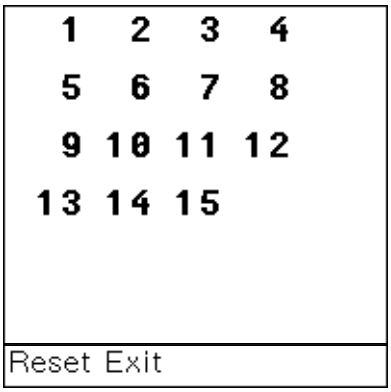


Figure 55. The second version of the Fifteen Puzzle lets the user move pieces by clicking on them.

Right, Left and Up are no longer needed; they have been replaced by Exit and Reset.

This new interface is more pleasant to use. This style of interface, that lets you move the piece you want to move by pointing at it, is called a *direct manipulation* interface. In this chapter you will see how CLIM helps you build such an interface with a very general and powerful mechanism based on the idea of a *presentation*.

The Concept of a Presentation

When CLIM does output it is doing more than just drawing bits on the screen and forgetting about them. CLIM remembers what was drawn on the window. This feature is called *output recording*. Several other features of CLIM make use of output recording; one of the most important features enabled by output recording is the *presentation*.

The purpose of a presentation is to associate some output with an object in the program. Any piece of output, whether text or graphics, can form the visual representation of any Lisp object.

It is important to appreciate the general power that presentations give you, the application writer. In a typical application, the majority of commands are concerned with performing some action upon some object. It is part of the power of direct manipulation interfaces that they eliminate, for the user, most of the difficulty of specifying ‘‘what object’’ to operate upon. By providing a direct link from something the user can point at (the output) to the recipient of the desired action (the application object), presentations let you implement direct manipulation with great ease.

To make all this specific, consider the Fifteen Puzzle. In the first version of the program that we saw in the previous chapter, we had a ‘‘Right’’ command, which meant ‘‘move the piece that can move right into the space’’. As you can see by running the second version of the program, it is much more preferable to move

''that piece there'' where the user points at the piece to indicate which piece is to be moved.

If you haven't already done so, you should now look at the source for the new version of the Fifteen Puzzle: SYS:CLIM;REL-2;TUTORIAL;PUZZLE-2.LISP.

The application frame is the same as the previous version, but immediately following the application definition is a new CLIM form. This is how you create a *presentation-type*: by using the form **clim:define-presentation-type**. In defining a type called **puzzle-piece** we are expressing our intent to represent a piece of the Fifteen Puzzle by drawing something on the screen.

```
(clim:define-presentation-type puzzle-piece ())
```

Note for the advanced reader: This is a *very* simple example of **clim:define-presentation-type**. We haven't even explained what makes this a type yet. Many more examples will appear later.

So far we have told CLIM that we want to do output which represents a piece of the puzzle. Now we have to do it. Look at the new version of the output routine **draw-the-display**. Notice that the code that writes the characters representing the piece is wrapped in **clim:with-output-as-presentation**. This form means: the following output (that is, within this extent) represents this object.

```
(clim:with-output-as-presentation (stream position 'puzzle-piece)
  (if (zerop piece)
      (format stream "  ")
      (format stream "~2D" piece)))
```

Here we are presenting the position, not the piece in that position.

clim:with-output-as-presentation needs to be given

- The stream on which you are presenting the object, in this case, the Fifteen Puzzle display pane that was passed as an argument to the method;
- The lisp object you are presenting, in this case the value of **position** which is a lisp object representing the position;
- The type of presentation you are presenting the object as, in this case, we are presenting it as a **puzzle-piece**.

Making Commands From Presentations

Now that we have presentations on the window, we have to use them.

You have already been introduced to the idea of a command, and to the idea that commands may be generated in several different ways, but the only method of entering a command you have seen so far is by clicking on a menu item.

In this chapter you see another, very useful, way of making a command — by translating a mouse-click on a presentation into a command. Look at the following form in the source:

```
(clim:define-presentation-to-command-translator move-a-piece
    (puzzle-piece move fifteen-puzzle-2)
    (object)
  (multiple-value-bind (yp xp) (floor object 4)
    (list yp xp)))
```

The form **clim:define-presentation-to-command-translator** does exactly what its name suggests: it says that ''if you click on a presentation of this type, translate that into the following command''. In the example, a click on anything presented as a **puzzle-piece** translates into a **move** command. The body of the presentation translator returns the argument list for the command.

The **move** command is new to this version of the Fifteen Puzzle, but you should have no difficulty in understanding what it does.

```
(define-fifteen-puzzle-2-command (move) ((yp 'integer) (xp 'integer))
  ...)
```

When you compare this command to, say, the Right command of the previous Fifteen Puzzle there are a number of differences you should notice. Most importantly, this command takes arguments (which are the X and Y coordinates of the piece to move). Notice that the argument list for a command is a little more complicated than the argument list to a function. Each element in the argument list is itself a list which describes one argument.

There is no **:menu** option mentioned in this command, because this is not a command we want to put in the menu.

Exiting an Application

The second version of the Fifteen Puzzle contains two new menu commands. One of them, Reset, simply resets the game board to its original state. You should be able to understand the implementation of this command with what you have learned already.

The other command, Exit, provides a way to ''cleanly'' exit an application, and its implementation illustrates some new features of the CLIM substrate. Here is the code for the Exit command:

```
(define-fifteen-puzzle-2-command (exit :menu t) ()
  (clim:frame-exit clim:*application-frame*))
```

Notice a variable that we haven't used before called **clim:*application-frame***. As part of running the application, CLIM binds this variable to the application object. There are many times when this value will be useful to you, and this Exit command shows one of them. The application object is passed as an argument to the **clim:frame-exit** function. This function causes the application loop (which we introduced in the previous chapter) to terminate.

After an application terminates, the application object is still around in your Lisp environment, but no process is running the application's command loop. That means you can start up the application again, but unless you do so, you can't give it commands. Even when it is not running, an application preserves its internal

state, so, for instance, if you restart the Fifteen Puzzle you will find the board in the same position that you left it.

Summary

A presentation provides an association from output on the window to any Lisp object. **clim:define-presentation-type** defines a class of presentations. Output done within **clim:with-output-as-presentation** forms the presentation.

By translating presentations into commands you can provide commands to an application by clicking on pieces of output. **clim:define-presentation-to-command-translator** is used to define how presentations of a particular type translate into a command.

The variable **clim:*application-frame*** is bound to the application object while CLIM runs the application loop.

Calling the function **clim:frame-exit** on the application causes the application loop to terminate.

In the next chapter we will fix one of the remaining major problems with the Fifteen Puzzle — that the entire display is redrawn when only a portion of it changes.

Further Development of the Fifteen Puzzle

In this chapter you will see how to make the Fifteen Puzzle redisplay only those pieces of the puzzle that have changed positions. You will also see how to make CLIM highlight only those pieces that can actually move.

The first of these topics is an introduction to a large area of user interface implementation which goes under the name *incremental redisplay*. In general, many applications need to change some small part of their display without spending the time redrawing those parts of the display that haven't changed. CLIM provides several facilities to help you do such redisplay.

Despite the title of the first section, ''Incremental Redisplay the Hard Way'', we recommend you read this section like any other. The ''hard'' way is not necessarily the ''bad'' way; in fact the ''hard'' way may be the best way under certain circumstances.

Incremental Redisplay the Hard Way

The example for this section is found in `SYS:CLIM;REL-2;TUTORIAL;PUZZLE-3.LISP` (also see the section "Code for Puzzle-3 "). As you run this example, notice that the display flickers less than previous examples when you move a piece; only those pieces that require redisplaying are drawn after each command. As explained in the introduction to this chapter, incremental redisplay is a common requirement of many applications.

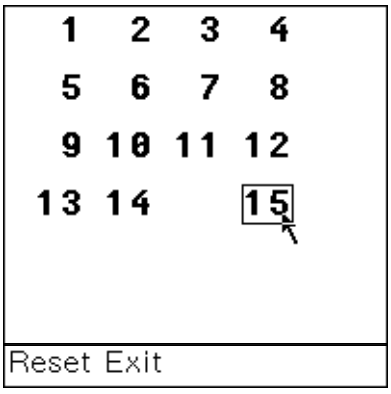


Figure 56. The third version of the Fifteen Puzzle uses Incremental Redisplay

We have already mentioned *output recording*: CLIM remembers what was written to the window. This implies you can't just overwrite old output with new — you have to clear pieces of the window.

As you read the source for this example you should notice several new state variables. The most important of these is **presentations** — a second array the same size as the board but instead of holding pieces it holds presentations. Presentations are returned from **clim:with-output-as-presentation**.

```
(:panes
  (display :application
           :text-style '(:fix :bold :very-large)
           :display-function 'draw-the-display
           :display-after-commands nil
           :scroll-bars nil
           :initial-cursor-visibility nil))
```

Notice that **draw-the-display** now calls a separate method, **draw-piece**, to draw each individual piece. That's because the **move** command also calls **draw-piece**. This is a significant difference. The command now explicitly requests the pieces of redisplay that it needs. Note the **:display-after-commands nil** in the application frame definition. Because the command requests redisplay, we don't need to ask for another complete redisplay.

```
(defmethod draw-piece ((application fifteen-puzzle-3)
                       piece position-y position-x stream)
  (with-slots (char-width line-height presentations) application
    (clim:stream-set-cursor-position
      stream (* position-x 3 char-width) (* position-y line-height))
    (when (aref presentations position-y position-x)
      (clim:erase-output-record
        (aref presentations position-y position-x) stream))
    (setf (aref presentations position-y position-x)
          (let ((position (+ (* position-y 4) position-x)))
            (write-string " " stream)
            (clim:with-output-as-presentation (stream position 'puzzle-piece)
              (if (zerop piece)
                  (format stream "  ")
                  (format stream "~2D" piece)))))))
```

If you examine **draw-piece** some more you'll see that not only does it draw the piece, storing the presentation in **presentations**, but it also calls **clim:erase-output-record** on the previous presentation. This clears the previous display of the piece.

In the Fifteen Puzzle, we have a simple situation in which none of the presentations overlap each other. You should realize that the task would be more complicated in the general case where presentations might overlap each other.

This is all rather cumbersome; we will see in the next section how CLIM provides a facility to hide most of these details from the programmer.

Incremental Redisplay the Easy Way

The example for this section is found in `SYS:CLIM;REL-2;TUTORIAL;PUZZLE-4.LISP`. (also see the section "Code for Puzzle-4").

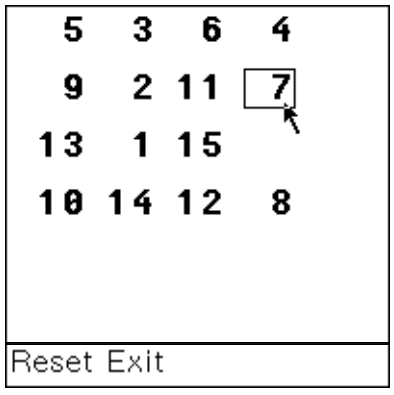


Figure 57. The fourth version of the Fifteen Puzzle uses the CLIM facility for Incremental Redisplay.

The fourth version of the Fifteen Puzzle runs exactly like the third version. However, when you examine the code, you should see that the code looks a lot simpler than the third version. In fact, it looks very much like the second version. Note that we don't need a state variable to keep track of the presentations any more.

```
(defmethod draw-the-display ((application fifteen-puzzle-4) stream
                             &key &allow-other-keys)
  (with-slots (pieces) application
    (dotimes (y 4)
      (dotimes (x 4)
        (clim:updating-output (stream :unique-id (+ (* y 4) x)
                                      :cache-value (aref pieces y x)
                                      :cache-test #'=)
          (draw-piece application (aref pieces y x) y x stream))))))
```

The extra piece of CLIM functionality that you are seeing for the first time is contained in the new version of the **draw-the-display** function; it is named **clim:updating-output**.

When you run a piece of code using **clim:updating-output** for the first time it:

- runs the enclosed code (which presumably produces some output),
- it associates that output with a tag called its **:unique-id**, (it is your job as programmer to ensure that the **:unique-id** *really* is unique in your program)
- pairs with the **:unique-id** a value, called a **:cache-value**.

When you run the same code again, CLIM:

- looks up the **:unique-id** (to find out if it has already run this output before, and if so, what **:cache-value** was supplied last time),
- examines the **:cache-value** supplied this time and compares it to the **:cache-value** supplied the last time,
- and if the two **:cache-value**s do *not* match, then CLIM must replace the old output with the new, so it:
  - erases the old output, and
  - runs the enclosed code to produce new output.

So, in the display function for this version of the Fifteen Puzzle:

```
(defmethod draw-the-display ((application fifteen-puzzle-4) stream
                             &key &allow-other-keys)
  (with-slots (pieces) application
    (dotimes (y 4)
      (dotimes (x 4)
        (clim:updating-output (stream :unique-id (+ (* y 4) x)
                                      :cache-value (aref pieces y x)
                                      :cache-test #'=)
          (draw-piece application (aref pieces y x) y x stream))))))
```

The **:unique-id** is a number that uniquely represents the position being displayed, while the **:cache-value** is the piece being drawn in that position. The contract of **clim:updating-output**, then, is to redraw that position in the puzzle whenever a new piece is found in that position — exactly what is needed to redraw only those parts of the puzzle that change.

Presentation Translator Testers

The example for this section is found in `SYS:CLIM;REL-2;TUTORIAL;PUZZLE-5.LISP`.

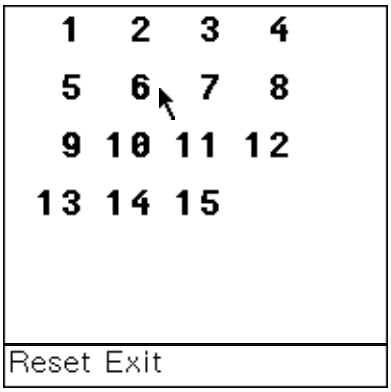


Figure 58. The last version of the Fifteen Puzzle uses a presentation tester so that non-movable pieces don't highlight.

The puzzle is essentially unchanged except that a presentation tester has been added to the CLIM presentation-to-command-translator.

A presentation translator tester is a predicate that verifies or refuses to verify the choice of object that CLIM makes. In this way, applications can further control the availability of a translator based on information that was not in the presentation.

It is important to realize that the tester can only ''filter out'' choices made by CLIM's presentation substrate. It cannot make choices available that would not be available without the tester. You should also realize that your code has to run every time the mouse moves over a presentation of the type specified in the translator. Testers should, therefore, be kept small and fast, or else you can adversely affect performance in your application.

```
(clim:define-presentation-to-command-translator move-a-piece
    (puzzle-piece move fifteen-puzzle-5
     :tester ((object)
              (multiple-value-bind (yp xp) (floor object 4)
                (which-way-to-move
                  yp xp (fifteen-puzzle-pieces clim:*application-frame*)))))
    (object)
  (multiple-value-bind (yp xp) (floor object 4)
    (list yp xp)))
```

In the example above, we use the **which-way-to-move** function, which already returns **nil** if the piece cannot move, as the core of the tester.

Summary

If your application needs to hold onto presentations explicitly, **clim:with-output-as-presentation** returns the presentation it created.

If you do not want the CLIM command loop to call its display function after every command, give the keyword **:display-after-commands** the value **nil** in your application frame definition.

This just about exhausts the Fifteen Puzzle as a teaching device. Once you have learned about graphics in the coming chapters, you might return to the Fifteen Puzzle and write another version yourself which displays the puzzle with graphics. Or you might implement similar games (the book referenced in "The Fifteen Puzzle — an Elementary Application" has hundreds of them) along similar lines.

Tic Tac Toe

This chapter describes the implementation of another small CLIM application — a program that plays Tic Tac Toe with the user. As with the Fifteen Puzzle, the program is relatively simple, but it illustrates several CLIM features in the context of a complete working application.

The code for this application is in `SYS:CLIM;REL-2;TUTORIAL;TIC-TAC-TOE.LISP`. If you have not already done so, edit the file and run the application.

1. Compile the buffer.

   ```
   m-X Compile Buffer
   ```

2. At the end of the file there are two or three forms commented out with `#|` and `|#`. Evaluate those forms by marking them, wiping them, and yanking them into a Listener and pressing `END`, or by marking them and pressing `c-sh-E`.

You can play against the program, with the program taking either player. To start a game, click on one of "Play (user X)" or "Play (program X)" in the menu. At this

point, empty board positions will be sensitive for your move. When the game ends, with either a win or a cat's game, the status window underneath the board will display the outcome of the game.

When a game is not in progress, you may edit the board to set up any arbitrary situation. In this state, a click on any board position will cycle through empty, "X" and "O". You may start a game from any situation.

Using a Presentation Type Hierarchy

In the Fifteen Puzzle of the previous chapters, we only used a single presentation type. As you might have guessed, there is much more to presentation types than simply naming something to be mouse-sensitive. Presentation types may have subtypes and supertypes, forming a hierarchy of types that can be very useful in expressing the intent of a program.

For example, the Tic Tac Toe game lets the user both play the game and edit the board at different times. The command that implements ''play in this position'' is only applicable to an empty position on the board, but the command that implements ''change the mark in this position'' is applicable (at the appropriate time) to any sort of position.

The relationship of presentation types to their supertypes is indicated using the **:inherit-from** keyword to **clim:define-presentation-type**. So the most general type is defined:

```
(clim:define-presentation-type board-position ())
```

and then the subtypes are defined as follows:

```
(clim:define-presentation-type empty-board-position ()
  :inherit-from 'board-position)

(clim:define-presentation-type x-board-position ()
  :inherit-from 'board-position)

(clim:define-presentation-type o-board-position ()
  :inherit-from 'board-position)
```

The command **com-user-move** is the command for playing a move in the game. Its argument must be an **empty-board-position**.

```
(define-tic-tac-toe-command com-user-move
    ((pos 'empty-board-position :gesture :select))
  ...)
```

The command **com-edit-position** is the command for editing the board when a game is not in progress. Its argument can be any **board-position**.

```
(define-tic-tac-toe-command com-edit-position
    ((pos 'board-position :gesture :select))
  ...)
```

You may have noticed another feature of the argument specifier to the above commands — there is an extra keyword and value after the name of the argument and its presentation-type. This is a convenience feature of CLIM. Saying

```
(define-something-command command-name
    ((argument presentation-type :gesture gesture))
  ...)
```

is a shorthand way of defining the command and a presentation-to-command-translator. It is equivalent to

```
(define-something-command command-name
    ((argument presentation-type))
  ...)

(clim:define-presentation-to-command-translator translator-name
    (empty-board-position command-name something :gesture :select)
    (object)
  (list object))
```

Enabling and Disabling Commands

An application may wish to temporarily prevent the use of certain commands. This can be done by calling **setf** on **clim:command-enabled** to set its value to **nil**; the command may be made available again by changing the value of **clim:command-enabled** back to **t**.

The Tic Tac Toe program enables the commands that are only used for playing the game alternately with the commands that are only used for editing the game. For example, when the application enters the state where someone is playing the game, the following method is called.

```
(defmethod go-to-playing-state ((frame tic-tac-toe))
  (setf (clim:command-enabled 'com-play-user-first frame) nil)
  (setf (clim:command-enabled 'com-play-program-first frame) nil)
  (setf (clim:command-enabled 'com-edit-position frame) nil)
  (setf (clim:command-enabled 'com-user-move frame) t))
```

When in this state, the menu items "Play (user X)" and "Play (program X)" are not sensitive, nor is the **com-edit-position** command available via the **:select** gesture. The **:select** gesture *does* translate to the **com-user-move** command, but only from empty positions, of course.

Command menus containing disabled commands may change their appearance; the disabled commands will often be ‘‘grayed out’’ and will not highlight.

> **Note for the advanced reader:** This behavior, like most of CLIM, can be changed by an application programmer if desired. The default behavior is chosen to provide minimum change in the appearance of the menu for minimum distraction of the user.

Too much use of command disabling can lead to an interface that has many ‘‘modes’’, which in general is not a good thing. Use command disabling with restraint.

Introduction to Using Graphics Transformations

A significant feature of CLIM is that it uses transformations as part of performing graphical output. Any function that performs graphical output may have its output transformed before reaching the output device.

The Tic Tac Toe application demonstrates one advantage of this facility — the writer of a graphical function may write it in simpler coordinates than would otherwise be possible. If you look at the method **display-board** in the Tic Tac Toe source, you will see that the internal function that draws the "X"s and "O"s of the board (**draw-element**) is written as if it is always going to draw the mark in the square bounded by (0,0) and (1,1). Yet the internal routine is used to draw marks in all nine positions of the board.

If you look at where **draw-element** is called, you will see how this is possible. The macros **clim:with-translation** and **clim:with-scaling** establish new coordinate systems, related to the existing coordinate system by a transformation. (CLIM supports a variety of types of transformation, they are documented more fully in the reference documentation. See the section "Transformations in CLIM".)

Summary

Presentation types form a hierarchy (or more precisely, a lattice). Use the **:inherit-from** keyword inside **clim:define-presentation-type** to indicate inheritance.

Graphics transformations let you abstract what you are drawing from where you are drawing it.

Calling **setf** on **clim:command-enabled** can be used to disable and enable an application's commands to allow only appropriate commands to be executable at a certain time.

Plotting Data

Here we describe a simple application for the plotting of scientific data. The user can enter a set of data points, perhaps the results of an experiment. The points are plotted on a two dimensional graph. An alternate view allows the examination of the data points sorted in a table. Least squares regression can be applied to fit a curve to the data and derive an equation which models the process which generated the data.

While studying this application, you will learn about presentation translators, pointer gestures, dialogs, transformations, and table formatting.

To try the application, read the file `SYS:CLIM;REL-2;TUTORIAL;LEAST-SQUARES-1.LISP` into an editor buffer.

1. Compile the buffer.

```
m-X Compile Buffer
```

2. At the end of the file there are two or three forms commented out with `#|` and `|#`. Evaluate those forms by marking them, wiping them, and yanking them into a Listener and pressing `END`, or by marking them and pressing `c-sh-E`.

Spend some time playing around with the application. Click the left mouse button on the graph, above the X axis (the horizontal line) and to the right of the Y axis (the vertical line). A point is plotted where you click the mouse. Plot a few points.

Now hold the `Shift` key down and move the mouse over a point. The point is highlighted by a small circle drawn around it. In the pointer documentation at the bottom of the screen it says something like "Sh-M: Delete Data Point". Click the middle mouse button to delete the data point.

Now hold the `Meta` key down and move the mouse over another point. The point is highlighted as before. Clicking the left mouse button on a point while the `Meta` key is held down will edit that point. A small window will pop up and display the X and Y values for the point. Clicking the left mouse button on one of these numbers will allow you to change it. Hit the `End` key when you have finished entering a number. If you decide not to change the point then hit `Abort`. When you have finished editing the coordinates of the point, hit `End`.

Once you have entered a few data points, try fitting a curve. Click on [Fit Curve] at the bottom of the application's window. A menu will pop up so that you can select whether to fit a linear, quadratic or cubic equation to the points you have plotted. Click on one with the mouse. It takes a little time for the curve to be calculated and drawn.

You can also look at your data arranged in a table. Click on [Switch Display] at the bottom of the window. You can now view your data arranged in a table with one row per datum. The X coordinate of a point is in the left column and the Y coordinate in the right. Note that you can edit and delete points with the mouse just as you could from the graphical display. If you have already fit a curve, you can see its equation and the correlation coefficient displayed in the pane below the one in which the data is tabulated. Try fitting a different curve to your data.

Input and the Mouse

Gestures

The mouse clicks you used to plot, delete and edit data points are called gestures. Not all systems have a three button mouse for an input device. Even if they do, there might be other user interface considerations which might preclude the use of certain mouse button combinations for input to a CLIM application. For this reason, CLIM adds a layer of abstraction between the actual *pointer gesture* as performed by the user and a *gesture name* representing the pointer gesture. The behavior of the application is defined in terms of gesture names. A system dependent mapping is provided between pointer gestures and gesture names.

For the Symbolics implementations of CLIM, the gesture name **:select** is associated with the mouse pointer gesture Left and the gesture named **:delete** with `shift`-Middle. The gesture `meta`-Left is named **:edit**.

You should resist the temptation to define gesture names called **:left**, **:middle**, and **:right**, because this conceptually adds non-portable gestures to your application.

Accept

When the application is not busy performing some task, it is waiting for input. The kind of input it is waiting for is referred to as the input context. For example, the application might be waiting for a command, for further arguments to a command, or for a menu choice. **clim:accept** is the function used to request input from the user.

clim:accept prompts the user for input which matches a specified presentation type. The user's input is parsed in accordance with the syntax of the printed representation of objects with the given presentation type. **clim:accept** returns what the user entered as a lisp object.

The call to **clim:accept** specifies a presentation type to be used as the input context. A prompt for the user can also be provided, as well as a default value to be offered. **clim:accept** can be used in the traditional mode of alternating requests for input from the application followed by input from the user, but today's user expects a more sophisticated interface.

Often, higher level facilities like the command processor or **clim:menu-choose** will invoke **clim:accept** rather than the applications programmer using it directly. The programmer will commonly use **clim:accept** within the context of **clim:accepting-values** to establish a dialog in which the user can respond to a number of inquiries in whatever order he feels comfortable.

Editing the Data Set Using Command Translators

The ability to add points is provided by a presentation translator from **clim:blank-area** to the command **com-create-data-point**. When the application is idle, it is waiting for the user to enter a command. When you click the **:select** gesture (the left mouse button) in a blank part of the plot, that gesture is translated to the **com-create-data-point**. The command takes two real numbers as arguments: the X and Y values of the piece of data.

```
(define-lsq-command com-create-data-point ((x 'real) (y 'real))
  (add-data-point (make-data-point x y) clim:*application-frame*))
```

Anyplace on the screen where there is no currently active presentation matches the input context for the presentation type **clim:blank-area**. There is a translator that will translate a gesture named **:select** on any **clim:blank-area** to this command.

```
(clim:define-presentation-to-command-translator new-point
    (clim:blank-area com-create-data-point lsq
     :gesture :select
     :tester
       ((x y window)
        (let ((frame clim:*application-frame*))
          (with-slots (data-left-margin data-top-margin
                       data-right-margin data-bottom-margin) frame
            (and (eql window (clim:get-frame-pane frame 'display))
                 (<= data-left-margin x data-right-margin)
                 (<= data-top-margin y data-bottom-margin))))))
    (x y)
  (with-slots (data-transform) clim:*application-frame*
    (multiple-value-bind (x y)
        (clim:untransform-position data-transform x y)
      (list x y))))
```

The value of the **:tester** keyword argument specifies the argument list and body of a function to be used to constrain the applicability of the translator. We don't want the user to be able to invoke the **com-create-data-point** command for just any blank area, only for blank area in the **display** pane which is within the confines of our graph's axes.

Note that the X and Y arguments to the command refer to coordinates in data-space. The translator receives its arguments in window coordinates, and transforms them to data coordinates so that they will be suitable as arguments to the command. This relationship between our data points and the points on the screen is discussed below.

The commands for deleting and editing data points are also provided through the use of command translators. There is a translator to the command **com-delete-data-point** which is invoked via the **:delete** gesture on a point which has been plotted.

```
(define-lsq-command com-delete-data-point
    ((point 'data-point :gesture :delete))
  (delete-data-point point clim:*application-frame*))
```

There is a similar translator from data points via the **:edit** gesture to the command **com-edit-data-point**. This command is described below in the section about dialogs.

Fitting a Curve: Menus

When you click on [Fit Curve], the application asks you what kind of curve you would like to fit. It does this through a pop-up menu. Menus can be used to ask the user to select from a number of choices.

```
(clim:menu-choose *known-curves*
  :label "Curve to Fit"
  :printer #'(lambda (curve stream)
               (write-string (curve-name curve) stream)))
```

The contract of **clim:menu-choose** is to allow the user to select one choice from a list of choices. The list of Lisp objects representing the choices is the only mandatory argument to **clim:menu-choose**. In the example above, the user is asked to select one curve from the list ***known-curves***.

The appearance of items in the menu is controlled by the optional **:printer** argument. If you don't supply one, **clim:menu-choose** will use a default printer for Lisp objects. If you supply one, it must be a function of two arguments — the object to be printed and the stream. In the example above, we supply a printer that prints the names of curve objects. We could have also supplied a printer that drew an icon representing the type of curve; this is left as an exercise for the interested reader.

The optional **:label** argument to **clim:menu-choose** allows the programmer to supply a descriptive label on the pop-up menu. Such a label helps add context to the user interaction.

Groups of Related Questions: Dialogs

Sometimes an application must ask the user several related questions. This is done through the use of dialogs. The application programmer describes a dialog in the body of an invocation of **clim:accepting-values**. The command for altering the coordinates of a data point provides us with an example of a dialog.

```
(define-lsq-command com-edit-data-point
    ((point 'data-point :gesture :edit))
  (let ((x (point-x point))
        (y (point-y point))
        (stream *standard-output*))
    (clim:accepting-values
        (stream
          :own-window '(:right-margin (20 :character))
          :label "New coordinates for the point")
      (fresh-line stream)
      (setq x (clim:accept 'real
                           :stream stream
                           :prompt "X: "
                           :default x))
      (fresh-line stream)
      (setq y (clim:accept 'real
                           :stream stream
                           :prompt "Y: "
                           :default y)))
    (alter-data-point point clim:*application-frame* x y)))
```

Within the dynamic context of **clim:accepting-values**, **clim:accept** is used to pose each individual query. The usual CLIM output facilities (tables, indented output, and so forth) could be used to describe the appearance of the dialog. In this example **fresh-line** is used so that each request for input will appear on its own line.

Within **clim:accepting-values**, the user can answer the queries in any order, by selecting with the mouse. Because he needn't answer all of the queries, calls to **clim:accept** from within **clim:accepting-values** must use **:default** to specify a default value to be returned by that call to **clim:accept**.

The [Set Axis Ranges] menu item implemented by the **com-set-axis-ranges** command is another example of the use of **clim:accepting-values** and **clim:accept** to construct a dialog. In addition, it uses **clim:accept-values-command-button** to create a button, appearing as the line of text "Set ranges to encompass all points", which, when clicked on, changes the minimum and maximum values for the X and Y axis ranges, such that the X axis extends from the point with the smallest X coordinate to the point with the largest X coordinate and the Y axis extends from the smallest Y coordinate to the largest Y coordinate.

```
(define-lsq-command (com-set-axis-ranges :menu t)
    ()
  (let ((frame clim:*application-frame*)
        (stream *standard-output*))
    (with-slots (data-x-min data-x-max data-y-min data-y-max
                 data-points data-transform data-points-tick) frame
      (incf data-points-tick)
      (let ((min-x data-x-min)
            (max-x data-x-max)
            (min-y data-y-min)
            (max-y data-y-max))
        (clim:accepting-values
            (stream
              :own-window '(:right-margin (20 :character))
              :label "Enter the ranges for the coordinate axes")
          (format stream "~&Range of X axis: ")
          (flet ((get-one (value id)
                   (clim:accept 'real
                                :stream stream
                                :default value
                                :query-identifier id
                                :prompt nil)))
            (setq min-x (get-one min-x 'x-min))
            (format stream " to ")
            (setq max-x (get-one max-x 'x-max))
            (format stream "~&Range of Y axis: ")
            (setq min-y (get-one min-y 'y-min))
            (format stream " to ")
            (setq max-y (get-one max-y 'y-max)))
          (fresh-line stream)
          (terpri stream)
          (clim:accept-values-command-button
            (stream :query-identifier 'all-of-them)
              "Set ranges to encompass all points"
            (multiple-value-setq (min-x min-y max-x max-y)
              (data-range frame)))
          (fresh-line stream)
          (terpri stream))
        (setq data-x-min min-x
              data-x-max max-x
              data-y-min min-y
              data-y-max max-y))
      (determine-data-transform frame))))
```

Also note the use of the **:query-identifier** keyword in the calls to **clim:accept**. Within **clim:accepting-values**, every call to **clim:accept** must have a unique query identifier associated with it. Under most circumstances, the value provided for the **:prompt** keyword argument would be sufficient for use as the **:query-identifier** as

well. In fact, if no **:query-identifier** is specified, CLIM will default to using the prompt as the query identifier.

Points, Transforms and Coordinate Spaces

Our application actually works with two coordinate spaces:

- an abstract coordinate space appropriate for the data being entered and manipulated
- the coordinate space of the **display** pane in which the points are plotted.

These two coordinate systems are related by a transform which maps from points in the abstract coordinate space of the data to the coordinate space of the **display** pane.

By establishing the transform as part of the drawing environment when we plot the points, we can plot the points using their abstract coordinates and have them drawn at the location corresponding to their **display** pane coordinates.

A Tabular Display of the Data

Layouts

Our user might want to examine his data in tabular form. We've added another pane to the application which displays the data in a table. Though the tabular display could be placed next to the plot, the user would probably want as much screen real estate as possible devoted to his graph. We can put the tabular display in a separate layout. CLIM applications can have their panes arranged in different layouts. A layout describes an arrangement of some (or all) of the application's panes. You have already seen one layout, named **drawing-layout**, with the command menu and the **display** pane. We can add a second layout containing the same command menu pane and a pane for the tabular display.

```
(clim:define-application-frame lsq ()
    (...)
  (:command-table (lsq :inherit-from (clim:accept-values-pane)))
  (:panes
    (display :application
             :display-function 'draw-data-display
             :incremental-redisplay t
             :display-after-commands t
             :scroll-bars nil)
    (table :application
           :incremental-redisplay t
           :scroll-bars :vertical
           :display-function 'tabulate-data-points)
    (equation :application
              :display-function 'print-equation-of-curve
              :display-after-commands t
              :incremental-redisplay t
              :scroll-bars nil))
  (:layouts
    (drawing-layout
      (clim:vertically () display))
    (tabular-layout
      (clim:vertically () (7/8 table) (1/8 equation)))))
```

Our pane for the tabular display is named **table**. It is visible in the layout named **tabular-layout**. The table of data is drawn by the pane's display function, **tabulate-data-points**, which is described below.

Our user will need a way to switch between the two layouts. We can define the [Switch Display] command in the command menu. It figures out what the currently displayed layout is by calling **clim:frame-current-layout** and then sets the current layout to the other layout by calling **setf** on **clim:frame-current-layout**.

```
(define-lsq-command (switch-configurations :menu "Switch Display") ()
  (let ((frame clim:*application-frame*))
    (let ((new-config
            (case (clim:frame-current-layout frame)
             (drawing-layout
              (setf (clim:command-enabled 'com-zoom-in frame) nil)
              (setf (clim:command-enabled 'com-zoom-out frame) nil)
              'tabular-layout)
             (tabular-layout
              (setf (clim:command-enabled 'com-zoom-in frame) t)
              (setf (clim:command-enabled 'com-zoom-out frame) t)
              'drawing-layout))))
      (setf (clim:frame-current-layout frame) new-config))))
```

The Table

The **table** pane displays the data in tabular form. The table has two columns: one for X coordinates and one for Y coordinates. The table has a row at the top for column headings (in italics). Each succeeding row represents a datum, with the datum's X and Y coordinates displayed in cells which fall under their respective columns.

The function **tabulate-data-points** displays this table of data. It is invoked when CLIM draws the contents of the **table** pane. **tabulate-data-points** uses the macros **clim:formatting-table**, **clim:formatting-row** and **clim:formatting-cell** to describe the contents of the table. The **clim:formatting-table** form describes the contents of a single table.

Within it, **clim:formatting-row** is used to describe each row of the table. The rows appear in the table in the same order in which their corresponding **clim:formatting-row** forms are evaluated. In our table of data, there is a row for column headings followed by one row for each point in the data set.

Within each row, **clim:formatting-cell** is used to describe the contents of each cell of that row. Each row of our table has one cell for the X coordinate of the data point and one cell for the Y coordinate. The **clim:formatting-cell** forms are evaluated in the order such that the cell will fall into the appropriate column.

We want each row of our table to remember what data point it displays. **clim:with-output-as-presentation** is used to associate the row with the datum it displays. This allows each row to be sensitive as a **data-point**. One benefit of this is that the **:delete** gesture will invoke the **com-delete-data-point** command for data points displayed as a row in the tabular view as well as for points displayed as dots in the plot view. In the invocation of **clim:with-output-as-presentation**, we specify **:single-box t** to emphasize that what is important is the datum, the entire row, rather than the individual cells containing the coordinates.

```
(defmethod tabulate-data-points ((frame lsq) pane)
  (fresh-line pane)
  (flet ((do-point (point stream)
           (clim:with-output-as-presentation
               (stream point 'data-point :single-box t)
             (clim:formatting-row (stream)
               (clim:formatting-cell (stream)
                 (format stream "~F" (point-x point)))
               (clim:formatting-cell (stream)
                 (format stream "~F" (point-y point)))))))
    (clim:formatting-table (pane)
      ;; print column headings
      (clim:formatting-row (pane)
        (clim:with-text-face (pane :italic)
          (clim:formatting-cell (pane :min-width 20
                                      :align-x :center)
            (write-string "X" pane))
          (clim:formatting-cell (pane :min-width 20
                                      :align-x :center)
            (write-string "Y" pane))))
      (with-slots (data-points) frame
        (dolist (point data-points)
          (do-point point pane))))))
```

Summary

CLIM's primary facility for requesting input from the user is **clim:accept**. It prompts the use and establishes an input context to help the user enter appropriate responses.

CLIM provides gesture names as a layer of abstraction between applications and pointer input devices.

A command can be invoked on an application object via a presentation translator from a presentation of the object, independent of the appearance of the object presented. The **:delete** gesture on a data point deletes the point whether it appears as a dot in the plotting pane or as a line of text in the table pane.

The presentation type **clim:blank-area** matches parts of the display where nothing is displayed. A translator from **clim:blank-area** to the **com-create-data-point** command was used to allow the entry of new data points by clicking on the plotting pane.

The application can prompt the use for input from a menu by calling **clim:menu-choose**. This was used to allow the user to select which type of function to fit the data to.

More complicated queries to the user take the form of *accepting values* dialogs. Within the context of **clim:accepting-values**, **clim:accept** and any output formatting facilities can be used to describe a set of related queries for the user to respond to.

Inside a dialog, a command button can be used to invoke a command. In the options dialog pane, a button was provided to widen the scope of the plotting pane to encompass all the data points.

Transformations can be used to map between the abstract coordinate space of an application and the coordinate space of the display.

The **:layouts** option to **clim:define-application-frame** can be used to describe the arrangement of your applications panes. These arrangements are called *layouts*. An application can have several such layouts. A pane can appear in more than one layout. You select which layout is the active one using **setf** on **clim:frame-current-layout**.

Output can be displayed in tabular form using **clim:formatting-table**, **clim:formatting-row** and **clim:formatting-cell**. The table facility arranges the output into regular rows and columns. **clim:with-output-as-presentation** can be used in conjunction with the table facility (or anyplace else) to make table rows mouse sensitive.

Appendices

Code for Puzzle-1

This code can be found in the file SYS:CLIM;REL-2;TUTORIAL;PUZZLE-1.LISP.

```
;;; -*- Mode: Lisp; Syntax: ANSI-Common-Lisp; Package: CLIM-USER; Base: 10 -*-

(define-application-frame fifteen-puzzle-1 ()
    ((pieces :initform (make-array '(4 4) :initial-contents '((1 2 3 4)
                                                              (5 6 7 8)
                                                              (9 10 11 12)
                                                              (13 14 15 0)))))
  (:menu-bar nil)
  (:panes
    (display :application
             :text-style '(:fix :bold :very-large)
             :display-function 'draw-the-display
             :scroll-bars nil)
    (menu :command-menu))
  (:layouts
    (main
      (vertically () display menu))))

;;; this draws the entire display
```

```
(defmethod draw-the-display ((application fifteen-puzzle-1) stream
                             &key &allow-other-keys)
  (with-slots (pieces) application
    (dotimes (y 4)
      (dotimes (x 4)
        (let ((piece (aref pieces y x)))
          (if (zerop piece)
              (format stream "   ")
              (format stream "~2D " piece))))
      (terpri stream))))

;;; useful macrology - the body will be run with x and y bound to
;;; the coordinates of the empty cell

(defmacro find-empty-piece-and-do ((y x) &body body)
  `(block find-empty-piece
     (dotimes (,y 4)
       (dotimes (,x 4)
         (when (zerop (aref pieces ,y ,x))
           ,@body
           (return-from find-empty-piece))))))

(define-fifteen-puzzle-1-command (down :menu t) ()
  (with-slots (pieces) *application-frame*
    (find-empty-piece-and-do (y x)
      (if (not (zerop y))
          (rotatef (aref pieces y x) (aref pieces (- y 1) x))))))

(define-fifteen-puzzle-1-command (up :menu t) ()
  (with-slots (pieces) *application-frame*
    (find-empty-piece-and-do (y x)
      (if (not (= y 3))
          (rotatef (aref pieces y x) (aref pieces (+ y 1) x))))))

(define-fifteen-puzzle-1-command (left :menu t) ()
  (with-slots (pieces) *application-frame*
    (find-empty-piece-and-do (y x)
      (if (not (= x 3))
          (rotatef (aref pieces y x) (aref pieces y (+ x 1)))))))

(define-fifteen-puzzle-1-command (right :menu t) ()
  (with-slots (pieces) *application-frame*
    (find-empty-piece-and-do (y x)
      (if (not (zerop x))
          (rotatef (aref pieces y x) (aref pieces y (- x 1)))))))
```

```
#||
()
(setq fp1 (make-application-frame 'fifteen-puzzle-1
            :left 200 :right 400 :top 150 :bottom 350))
(run-frame-top-level fp1)
||#
```

Code for Puzzle-2

This code can be found in the file SYS:CLIM;REL-2;TUTORIAL;PUZZLE-2.LISP.

```
;;; -*- Mode: Lisp; Syntax: ANSI-Common-Lisp; Package: CLIM-USER; Base: 10 -*-

(define-application-frame fifteen-puzzle-2 ()
    ((pieces :initform (make-array '(4 4) :initial-contents '((1 2 3 4)
                                                              (5 6 7 8)
                                                              (9 10 11 12)
                                                              (13 14 15 0)))))
  (:panes
    (display :application
             :text-style '(:fix :bold :very-large)
             :display-function 'draw-the-display
             :scroll-bars nil
             :initial-cursor-visibility nil))
  (:layouts
    (main
      (vertically () display))))

(define-presentation-type puzzle-piece ())

(defmethod draw-the-display ((application fifteen-puzzle-2) stream
                             &key &allow-other-keys)
  (with-slots (pieces) application
    (dotimes (y 4)
      (dotimes (x 4)
        (let ((piece (aref pieces y x))
              (position (+ (* y 4) x)))
          (write-string " " stream)
          (with-output-as-presentation (stream position 'puzzle-piece)
            (if (zerop piece)
                (format stream "  ")
                (format stream "~2D" piece)))))
      (terpri stream))))
```

```
;;; if the piece at (xp,yp) can be moved, five values are returned:
;;;  - the coordinates of the space in the puzzle,
;;;  - delta-y and delta-x representing the direction on the puzzle from
;;;     space towards the piece at (xp,yp)
;;;  - and the number of pieces to move
;;; if the piece at (xp,yp) cannot be moved, nil is returned

(defun which-way-to-move (yp xp pieces)
  (macrolet ((is-space (y x) `(zerop (aref pieces ,y ,x))))
    (loop for x from (+ xp 1) to 3 do
      (when (is-space yp x)
        (return-from which-way-to-move (values yp x 0 -1 (- x xp)))))
    (loop for x from (- xp 1) downto 0 do
      (when (is-space yp x)
        (return-from which-way-to-move (values yp x 0 1 (- xp x)))))
    (loop for y from (+ yp 1) to 3 do
      (when (is-space y xp)
        (return-from which-way-to-move (values y xp -1 0 (- y yp)))))
    (loop for y from (- yp 1) downto 0 do
      (when (is-space y xp)
        (return-from which-way-to-move (values y xp 1 0 (- yp y)))))))

(define-fifteen-puzzle-2-command (move) ((yp 'integer) (xp 'integer))
  (with-slots (pieces) *application-frame*
    (multiple-value-bind (start-y start-x dy dx n-moves)
        (which-way-to-move yp xp pieces)
      (when dx
        (loop repeat n-moves
              for x1 = start-x then x2
              for x2 = (+ x1 dx) then (+ x2 dx)
              for y1 = start-y then y2
              for y2 = (+ y1 dy) then (+ y2 dy)
              do (setf (aref pieces y1 x1) (aref pieces y2 x2))
              finally (setf (aref pieces yp xp) 0))))))

(define-presentation-to-command-translator move-a-piece
    (puzzle-piece move fifteen-puzzle-2)
    (object)
  (multiple-value-bind (yp xp) (floor object 4)
    (list yp xp)))
```

```
(define-fifteen-puzzle-2-command (reset :menu t) ()
  (with-slots (pieces) *application-frame*
    (loop for y from 0 to 3 do
      (loop with 4y+1 = (+ (* 4 y) 1)
            for x from 0 to 3 do
        (setf (aref pieces y x) (mod (+ 4y+1 x) 16))))))

(define-fifteen-puzzle-2-command (exit :menu t) ()
  (frame-exit *application-frame*))

#||
()
(setq fp2 (make-application-frame 'fifteen-puzzle-2
            :left 400 :right 600 :top 150 :bottom 350))
(run-frame-top-level fp2)
||#
```

Code for Puzzle-3

This code can be found in the file SYS:CLIM;REL-2;TUTORIAL;PUZZLE-3.LISP.

```
;;; -*- Mode: Lisp; Syntax: ANSI-Common-Lisp; Package: CLIM-USER; Base: 10 -*-

;;; things to fix - replace encoded position
;;;  - auto-size window , get line-height, char-width

(define-application-frame fifteen-puzzle-3 ()
    ((pieces :initform (make-array '(4 4) :initial-contents '((1 2 3 4)
                                                              (5 6 7 8)
                                                              (9 10 11 12)
                                                              (13 14 15 0))))
     (presentations :initform (make-array '(4 4)))
     (char-width :initform 12)
     (line-height :initform 30))
  (:panes
    (display :application
             :text-style '(:fix :bold :very-large)
             :display-function 'draw-the-display
             :display-after-commands nil
             :scroll-bars nil
             :initial-cursor-visibility nil))
  (:layouts
    (main
      (vertically () display))))
```

```
(define-presentation-type puzzle-piece ())

(defmethod draw-piece ((application fifteen-puzzle-3)
                       piece position-y position-x stream)
  (with-slots (char-width line-height presentations) application
    (stream-set-cursor-position stream (* position-x 3 char-width)
                                       (* position-y line-height))
    (when (aref presentations position-y position-x)
      (erase-output-record (aref presentations position-y position-x) stream))
    (setf (aref presentations position-y position-x)
          (let ((position (+ (* position-y 4) position-x)))
            (write-string " " stream)
            (with-output-as-presentation (stream position 'puzzle-piece)
              (if (zerop piece)
                  (format stream "  ")
                  (format stream "~2D" piece)))))))

(defmethod draw-the-display ((application fifteen-puzzle-3) stream
                             &key &allow-other-keys)
  (with-slots (pieces) application
    (dotimes (y 4)
      (dotimes (x 4)
        (draw-piece application (aref pieces y x) y x stream)))))

(defun which-way-to-move (yp xp pieces)
  (macrolet ((is-space (y x) `(zerop (aref pieces ,y ,x))))
    (loop for x from (+ xp 1) to 3 do
      (when (is-space yp x)
        (return-from which-way-to-move (values yp x 0 -1 (- x xp)))))
    (loop for x from (- xp 1) downto 0 do
      (when (is-space yp x)
        (return-from which-way-to-move (values yp x 0 1 (- xp x)))))
    (loop for y from (+ yp 1) to 3 do
      (when (is-space y xp)
        (return-from which-way-to-move (values y xp -1 0 (- y yp)))))
    (loop for y from (- yp 1) downto 0 do
      (when (is-space y xp)
        (return-from which-way-to-move (values y xp 1 0 (- yp y)))))))
```

```
(define-fifteen-puzzle-3-command (move) ((yp 'integer) (xp 'integer))
  (with-slots (pieces) *application-frame*
    (let ((display-pane (get-frame-pane *application-frame* 'display)))
      (flet ((update (y x new-piece)
               (setf (aref pieces y x) new-piece)
               (draw-piece *application-frame* new-piece y x display-pane)))
        (multiple-value-bind (start-y start-x dy dx n-moves)
            (which-way-to-move yp xp pieces)
          (when dx
            (loop repeat n-moves
                  for x1 = start-x then x2
                  for x2 = (+ x1 dx) then (+ x2 dx)
                  for y1 = start-y then y2
                  for y2 = (+ y1 dy) then (+ y2 dy)
                  do (update y1 x1 (aref pieces y2 x2))
                  finally (update yp xp 0)))))))))

(define-presentation-to-command-translator move-a-piece
    (puzzle-piece move fifteen-puzzle-3)
    (object)
  (multiple-value-bind (yp xp) (floor object 4)
    (list yp xp)))

(define-fifteen-puzzle-3-command (reset :menu t) ()
  (with-slots (pieces presentations) *application-frame*
    (loop for y from 0 to 3 do
      (loop with 4y+1 = (+ (* 4 y) 1)
            for x from 0 to 3 do
        (setf (aref pieces y x) (mod (+ 4y+1 x) 16))))
    (let ((display-pane (get-frame-pane *application-frame* 'display)))
      (window-clear display-pane)
      (dotimes (y 4)
        (dotimes (x 4)
          (setf (aref presentations y x) nil)))
      (draw-the-display *application-frame* display-pane))))

(define-fifteen-puzzle-3-command (exit :menu t) ()
  (frame-exit *application-frame*))

#||
()
(setq fp3 (make-application-frame 'fifteen-puzzle-3
            :left 400 :right 600 :top 150 :bottom 350))
(run-frame-top-level fp3)
||#
```

Code for Puzzle-4

This code can be found in the file SYS:CLIM;REL-2;TUTORIAL;PUZZLE-4.LISP.

```
;;; -*- Mode: Lisp; Syntax: ANSI-Common-Lisp; Package: CLIM-USER; Base: 10 -*-

(define-application-frame fifteen-puzzle-4 ()
    ((pieces :initform (make-array '(4 4) :initial-contents '((1 2 3 4)
                                                              (5 6 7 8)
                                                              (9 10 11 12)
                                                              (13 14 15 0))))
     (char-width :initform 12)
     (line-height :initform 30))
  (:panes
    (display :application
             :default-text-style '(:fix :bold :very-large)
             :display-function 'draw-the-display
             :incremental-redisplay t
             :scroll-bars nil
             :initial-cursor-visibility nil))
  (:layouts
    (main
      (vertically () display))))

(define-presentation-type puzzle-piece ())

(defmethod draw-piece ((application fifteen-puzzle-4)
                       piece position-y position-x stream)
  (with-slots (char-width line-height) application
    (stream-set-cursor-position stream (* position-x 3 char-width)
                                       (* position-y line-height)))
  (let ((position (+ (* position-y 4) position-x)))
    (write-string " " stream)
    (with-output-as-presentation (stream position 'puzzle-piece)
      (if (zerop piece)
          (format stream "  ")
          (format stream "~2D" piece)))))

(defmethod draw-the-display ((application fifteen-puzzle-4) stream
                             &key &allow-other-keys)
  (with-slots (pieces) application
    (dotimes (y 4)
      (dotimes (x 4)
        (updating-output (stream :unique-id (+ (* y 4) x)
                                 :cache-value (aref pieces y x)
                                 :cache-test #'=)
          (draw-piece application (aref pieces y x) y x stream))))))
```

```
(defun which-way-to-move (yp xp pieces)
  (macrolet ((is-space (y x) `(zerop (aref pieces ,y ,x))))
    (loop for x from (+ xp 1) to 3 do
      (when (is-space yp x)
        (return-from which-way-to-move (values yp x 0 -1 (- x xp)))))
    (loop for x from (- xp 1) downto 0 do
      (when (is-space yp x)
        (return-from which-way-to-move (values yp x 0 1 (- xp x)))))
    (loop for y from (+ yp 1) to 3 do
      (when (is-space y xp)
        (return-from which-way-to-move (values y xp -1 0 (- y yp)))))
    (loop for y from (- yp 1) downto 0 do
      (when (is-space y xp)
        (return-from which-way-to-move (values y xp 1 0 (- yp y)))))))

(define-fifteen-puzzle-4-command (move) ((yp 'integer) (xp 'integer))
  (with-slots (pieces) *application-frame*
    (multiple-value-bind (start-y start-x dy dx n-moves)
        (which-way-to-move yp xp pieces)
      (when dx
        (loop repeat n-moves
              for x1 = start-x then x2
              for x2 = (+ x1 dx) then (+ x2 dx)
              for y1 = start-y then y2
              for y2 = (+ y1 dy) then (+ y2 dy)
              do (setf (aref pieces y1 x1) (aref pieces y2 x2))
              finally (setf (aref pieces yp xp) 0))))))

(define-presentation-to-command-translator move-a-piece
    (puzzle-piece move fifteen-puzzle-4)
    (object)
  (multiple-value-bind (yp xp) (floor object 4)
    `(,yp ,xp)))

(define-fifteen-puzzle-4-command (reset :menu t) ()
  (with-slots (pieces) *application-frame*
    (loop for y from 0 to 3 do
      (loop with 4y+1 = (+ (* 4 y) 1)
            for x from 0 to 3 do
        (setf (aref pieces y x) (mod (+ 4y+1 x) 16))))))

(define-fifteen-puzzle-4-command (exit :menu t) ()
  (frame-exit *application-frame*))
```

```
#||
()
(setq fp4 (make-application-frame 'fifteen-puzzle-4
            :left 600 :right 800 :top 150 :bottom 350))
(run-frame-top-level fp4)
||#
```

Code for Puzzle-5

This code can be found in the file SYS:CLIM;REL-2;TUTORIAL;PUZZLE-5.LISP.

```
;;; -*- Mode: Lisp; Syntax: ANSI-Common-Lisp; Package: CLIM-USER; Base: 10 -*-

(define-application-frame fifteen-puzzle-5 ()
    ((pieces :initform (make-array '(4 4) :initial-contents '((1 2 3 4)
                                                              (5 6 7 8)
                                                              (9 10 11 12)
                                                              (13 14 15 0)))
             :reader fifteen-puzzle-pieces)
     (char-width :initform 12)
     (line-height :initform 30))
  (:panes
    (display :application
             :default-text-style '(:fix :bold :very-large)
             :display-function 'draw-the-display
             :incremental-redisplay t
             :scroll-bars nil
             :initial-cursor-visibility nil))
  (:layouts
    (main
      (vertically () display))))

(define-presentation-type puzzle-piece ())

(defmethod draw-piece ((application fifteen-puzzle-5)
                       piece position-y position-x stream)
  (with-slots (char-width line-height) application
    (stream-set-cursor-position stream (* position-x 3 char-width)
                                       (* position-y line-height)))
  (let ((position (+ (* position-y 4) position-x)))
    (write-string " " stream)
    (with-output-as-presentation (stream position 'puzzle-piece)
      (if (zerop piece)
          (format stream "  ")
          (format stream "~2D" piece)))))
```

```
(defmethod draw-the-display ((application fifteen-puzzle-5) stream
                             &key &allow-other-keys)
  (with-slots (pieces) application
    (dotimes (y 4)
      (dotimes (x 4)
        (updating-output (stream :unique-id (+ (* y 4) x)
                                 :cache-value (aref pieces y x)
                                 :cache-test #'=)
          (draw-piece application (aref pieces y x) y x stream))))))

(defun which-way-to-move (yp xp pieces)
  (macrolet ((is-space (y x) `(zerop (aref pieces ,y ,x))))
    (loop for x from (+ xp 1) to 3 do
      (when (is-space yp x)
        (return-from which-way-to-move (values yp x 0 -1 (- x xp)))))
    (loop for x from (- xp 1) downto 0 do
      (when (is-space yp x)
        (return-from which-way-to-move (values yp x 0 1 (- xp x)))))
    (loop for y from (+ yp 1) to 3 do
      (when (is-space y xp)
        (return-from which-way-to-move (values y xp -1 0 (- y yp)))))
    (loop for y from (- yp 1) downto 0 do
      (when (is-space y xp)
        (return-from which-way-to-move (values y xp 1 0 (- yp y)))))))

(define-fifteen-puzzle-5-command (move) ((yp 'integer) (xp 'integer))
  (with-slots (pieces) *application-frame*
    (multiple-value-bind (start-y start-x dy dx n-moves)
        (which-way-to-move yp xp pieces)
      (when dx
        (loop repeat n-moves
              for x1 = start-x then x2
              for x2 = (+ x1 dx) then (+ x2 dx)
              for y1 = start-y then y2
              for y2 = (+ y1 dy) then (+ y2 dy)
              do (setf (aref pieces y1 x1) (aref pieces y2 x2))
              finally (setf (aref pieces yp xp) 0))))))
```

```
(define-presentation-to-command-translator move-a-piece
    (puzzle-piece move fifteen-puzzle-5
     :tester ((object)
              (multiple-value-bind (yp xp) (floor object 4)
                (which-way-to-move
                  yp xp (fifteen-puzzle-pieces *application-frame*)))))
    (object)
  (multiple-value-bind (yp xp) (floor object 4)
    (list yp xp)))

(define-fifteen-puzzle-5-command (reset :menu t) ()
  (with-slots (pieces) *application-frame*
    (loop for y from 0 to 3 do
      (loop with 4y+1 = (+ (* 4 y) 1)
            for x from 0 to 3 do
        (setf (aref pieces y x) (mod (+ 4y+1 x) 16))))))

(define-fifteen-puzzle-5-command (exit :menu t) ()
  (frame-exit *application-frame*))

#||
()
(setq fp5 (make-application-frame 'fifteen-puzzle-5
            :left 600 :right 800 :top 150 :bottom 350))
(run-frame-top-level fp5)
||#
```

Code for Tic Tac Toe

This code can be found in the file SYS:CLIM;REL-2;TUTORIAL;TIC-TAC-TOE.LISP.

```
;;; -*- Mode: Lisp; Syntax: ANSI-Common-Lisp; Package: CLIM-USER; Base: 10 -*-

(defconstant *X* 1)
(defconstant *O* -1)
(defconstant *empty* 0)

(define-presentation-type board-position ())

(define-presentation-type empty-board-position ()
  :inherit-from 'board-position)

(define-presentation-type x-board-position ()
  :inherit-from 'board-position)
```

```
(define-presentation-type o-board-position ()
  :inherit-from 'board-position)

(clim:define-application-frame tic-tac-toe
                               ()
    ((board :initform (make-array '(3 3) :initial-element *empty*))
     (board-diagnosis :initform t)
     (whose-move :initform *X*)
     (user-plays :initform *X*))
  (:panes
    (board-display :application
                   :display-function 'display-board
                   :incremental-redisplay t
                   :scroll-bars nil)
    (status :application
            :display-function 'display-status
            :incremental-redisplay t
            :scroll-bars nil))
  (:layouts
    (main
      (vertically ()
        (9/10 board-display)
        (1/10 status)))))

(defmethod clos:initialize-instance :after ((frame tic-tac-toe) &rest args)
  (declare (ignore args))
  (go-to-stopped-state frame))

(define-tic-tac-toe-command (com-tic-tac-toe-exit :menu "Exit") ()
  (clim:frame-exit *application-frame*))

(define-tic-tac-toe-command (com-reset-tic-tac-toe :menu "Reset") ()
  (reset-board *application-frame*)
  (go-to-stopped-state *application-frame*))

(defmethod reset-board ((frame tic-tac-toe))
  (window-clear (get-frame-pane frame 'board-display))
  (with-slots (board board-diagnosis) frame
    (setf board-diagnosis t)
    (dotimes (ix 3)
      (dotimes (iy 3)
        (setf (aref board ix iy) *empty*)))))

;;; the display method for the board pane
```

```
(defmethod display-board ((frame tic-tac-toe) stream
                          &key &allow-other-keys)
  (with-slots (board) frame
    (multiple-value-bind (width height) (window-inside-size stream)
      (multiple-value-bind (size x-offset y-offset)
          (if (< width height)
              (values (floor width 3) 0 (floor (- height width) 2))
              (values (floor height 3) (floor (- width height) 2) 0))
        (flet ((draw-element (elt position stream)
                 (cond ((= elt *X*)
                        (with-output-as-presentation
                            (stream position 'x-board-position
                             :single-box t)
                          (draw-line* stream 0.1 0.1 0.9 0.9 :line-thickness 2)
                          (draw-line* stream 0.1 0.9 0.9 0.1 :line-thickness 2)))
                       ((= elt *O*)
                        (with-output-as-presentation
                            (stream position 'o-board-position
                             :single-box t)
                          (draw-circle* stream 0.5 0.5 0.45
                                        :filled nil :line-thickness 2)))
                       (t
                        (with-output-as-presentation
                            (stream position 'empty-board-position)
                          (draw-rectangle* stream 0.1 0.1 0.9 0.9 :ink +white+))))))
          (with-translation (stream x-offset y-offset)
            (with-scaling (stream size size)
              (dotimes (iy 3)
                (with-translation (stream 0 iy)
                  (dotimes (ix 3)
                    (let ((elt (aref board ix iy))
                          (position (+ (* 3 iy) ix)))
                      (updating-output (stream :unique-id position
                                               :cache-value elt)
                        (with-translation (stream ix 0)
                          (draw-element elt position stream)))))))
              (draw-line* stream 0.1 1.0 2.9 1.0)
              (draw-line* stream 0.1 2.0 2.9 2.0)
              (draw-line* stream 1.0 0.1 1.0 2.9)
              (draw-line* stream 2.0 0.1 2.0 2.9)))))))

;;; the display method for the status pane
```

```
(defmethod display-status ((frame tic-tac-toe) stream
                           &key &allow-other-keys)
  (with-slots (board-diagnosis) frame
    (updating-output (stream :cache-value board-diagnosis)
      (let ((string (cond ((null board-diagnosis) "Cat's game")
                          ((eql board-diagnosis *X*) "Win for X")
                          ((eql board-diagnosis *O*) "Win for O"))))
        (when string
          (write-string string stream))))))

;;; make a move in the game

(define-tic-tac-toe-command com-user-move
    ((pos 'empty-board-position :gesture :select))
  (with-slots (board whose-move board-diagnosis) *application-frame*
    (multiple-value-bind (iy ix) (floor pos 3)
      (make-move *application-frame* ix iy))
    (if (eql board-diagnosis t)
        (multiple-value-bind (ix iy) (find-next-ttt-move board whose-move)
          (make-move *application-frame* ix iy)
          (unless (eql board-diagnosis t)
            (go-to-stopped-state *application-frame*)))
        (go-to-stopped-state *application-frame*))))

(defmethod make-move ((frame tic-tac-toe) ix iy)
  (with-slots (board whose-move board-diagnosis) frame
    (if (= (aref board ix iy) *empty*)
        (progn (setf (aref board ix iy) whose-move)
               (setf whose-move (- whose-move))
               (setf board-diagnosis (diagnose-board board)))
        (error "Not an empty position"))))

;;; edit a position by cycling through possibilities

(define-tic-tac-toe-command com-edit-position
    ((pos 'board-position :gesture :select))
  (with-slots (board board-diagnosis) *application-frame*
    (multiple-value-bind (iy ix) (floor pos 3)
      (setf (aref board ix iy)
            (let ((old (aref board ix iy)))
              (cond ((= old *empty*) *X*)
                    ((= old *X*) *O*)
                    (t *empty*)))))
    (setf board-diagnosis (diagnose-board board))))
```

```
(define-tic-tac-toe-command (com-play-user-first :menu "Play (user X)") ()
  (with-slots (user-plays) *application-frame*
    (setf user-plays *X*)
    (start-play *application-frame*)))

(define-tic-tac-toe-command (com-play-program-first :menu "Play (program X)") ()
  (with-slots (user-plays) *application-frame*
    (setf user-plays *O*)
    (start-play *application-frame*)))

(defmethod start-play ((frame tic-tac-toe))
  (with-slots (board whose-move user-plays board-diagnosis) *application-frame*
    (unless (eql board-diagnosis t)
      (reset-board *application-frame*))
    (go-to-playing-state *application-frame*)
    (setf whose-move (determine-whose-move board))
    (when (/= user-plays whose-move)
      (multiple-value-bind (ix iy) (find-next-ttt-move board whose-move)
        (make-move *application-frame* ix iy)))))

;;; enable and disable appropriate commands

(defmethod go-to-playing-state ((frame tic-tac-toe))
  (setf (command-enabled 'com-play-user-first frame) nil)
  (setf (command-enabled 'com-play-program-first frame) nil)
  (setf (command-enabled 'com-edit-position frame) nil)
  (setf (command-enabled 'com-user-move frame) t))

(defmethod go-to-stopped-state ((frame tic-tac-toe))
  (setf (command-enabled 'com-play-user-first frame) t)
  (setf (command-enabled 'com-play-program-first frame) t)
  (setf (command-enabled 'com-edit-position frame) t)
  (setf (command-enabled 'com-user-move frame) nil))

;;; these are the game-playing functions
;;; they have no user-interface component

;;; picks the next move for the program
```

```
(defun find-next-ttt-move (board whose-move)
  (let ((me whose-move) (you (- whose-move)))
    (let (target (points nil))
      (labels ((pushnu (x y)
                 (unless (find-if #'(lambda (xy)
                                      (and (= (first xy) x)
                                           (= (second xy) y)))
                                  points)
                   (push (list x y) points)))
               (point-= (p1 p2)
                 (and (= (first p1) (first p2)) (= (second p1) (second p2))))
               (pick-a-choice-if-ive-got-one ()
                 (when points
                   (let ((ran (random (length points))))
                     (return-from find-next-ttt-move
                       (values-list (nth ran points))))))
               (almost-complete-row (elt0 x0 y0 elt1 x1 y1 elt2 x2 y2)
                 (cond ((and (= elt0 *empty*) (= elt1 elt2 target))
                        (pushnu x0 y0))
                       ((and (= elt1 *empty*) (= elt0 elt2 target))
                        (pushnu x1 y1))
                       ((and (= elt2 *empty*) (= elt0 elt1 target))
                        (pushnu x2 y2))))
               (row-with-one-only (elt0 x0 y0 elt1 x1 y1 elt2 x2 y2)
                 (cond ((and (= elt0 elt1 *empty*) (= elt2 target))
                        (pushnu x0 y0)
                        (pushnu x1 y1))
                       ((and (= elt1 elt2 *empty*) (= elt0 target))
                        (pushnu x1 y1)
                        (pushnu x2 y2))
                       ((and (= elt0 elt2 *empty*) (= elt1 target))
                        (pushnu x0 y0)
                        (pushnu x2 y2))))
               (pair-of-rows-to-fork (elt0 x0 y0
                                      elt1a x1a y1a
                                      elt2a x2a y2a
                                      elt1b x1b y1b
                                      elt2b x2b y2b)
                 (declare (ignore x1a y1a x2a y2a x1b y1b x2b y2b))
                 (when (and (= elt0 *empty*)
                            (or (and (= elt1a *empty*) (= elt2a target))
                                (and (= elt2a *empty*) (= elt1a target)))
                            (or (and (= elt1b *empty*) (= elt2b target))
                                (and (= elt2b *empty*) (= elt1b target))))
                   (pushnu x0 y0))))
        ;; look for immediate win
        (setq target me)
```

```
        (map-over-ttt-board-rows board #'almost-complete-row)
        (pick-a-choice-if-ive-got-one)
        ;; look for immediate loss unless I block
        (setq target you)
        (map-over-ttt-board-rows board #'almost-complete-row)
        (pick-a-choice-if-ive-got-one)
        ;; look for my fork
        (setq target me)
        (map-over-pairs-of-ttt-board-rows board #'pair-of-rows-to-fork)
        (pick-a-choice-if-ive-got-one)
        ;; look for opponent's fork
        (setq target you)
        (map-over-pairs-of-ttt-board-rows board #'pair-of-rows-to-fork)
        (when points
          (if (= (length points) 1)
              ;; block the fork
              (pick-a-choice-if-ive-got-one)
              ;; two fork points - have to force
              (let ((fork-points points))
                (setq points nil)
                (map-over-ttt-board-rows board #'row-with-one-only)
                (setq points (set-difference points fork-points :test #'point-=))
                (pick-a-choice-if-ive-got-one))))))
    ;; an opening move in a corner requires a reply in the center
    ;; else prefer a corner
    (when (= (aref board 1 1) *empty*)
      (return-from find-next-ttt-move (values 1 1)))
    (loop for ix from 0 to 2 by 2 do
      (loop for iy from 0 to 2 by 2 do
        (when (= (aref board ix iy) *empty*)
          (return-from find-next-ttt-move (values ix iy)))))))

;;; returns *X* or *O* for a winning board
;;; T for a board still playable, nil for a cats game
```

```
(defun diagnose-board (board)
  (flet ((diagnose-row (elt0 x0 y0 elt1 x1 y1 elt2 x2 y2)
           x0 y0 x1 y1 x2 y2
           ;; returns *X* or *O* for a winning row
           ;; T for a row still playable, nil for a blocked row
           (cond ((= elt0 elt1 elt2)
                  (if (= elt0 *empty*) t elt0))
                 ((= elt0 elt1)
                  (if (or (= elt0 *empty*) (= elt2 *empty*)) t nil))
                 ((= elt0 elt2)
                  (if (or (= elt0 *empty*) (= elt1 *empty*)) t nil))
                 ((= elt1 elt2)
                  (if (or (= elt0 *empty*) (= elt1 *empty*)) t nil))
                 (t nil))))
    (let ((x-win nil) (o-win nil) (cats-game t))
      (map-over-ttt-board-rows board
        #'(lambda (elt0 x0 y0 elt1 x1 y1 elt2 x2 y2)
            (let ((q (diagnose-row elt0 x0 y0 elt1 x1 y1 elt2 x2 y2)))
              (cond ((eql q *X*) (setq x-win t))
                    ((eql q *O*) (setq o-win t))
                    ((eql q t) (setq cats-game nil))))))
      (cond (x-win *X*)
            (o-win *O*)
            (cats-game nil)
            (t t)))))

;;; used when starting the program in the middle of a game

(defun determine-whose-move (board)
  (let ((xs 0) (os 0))
    (dotimes (x 3)
      (dotimes (y 3)
        (let ((elt (aref board x y)))
          (cond ((= elt *X*) (incf xs))
                ((= elt *O*) (incf os))))))
    (if (> xs os) *O* *X*)))
```

```
(defun map-over-ttt-board-rows (board function)
  (macrolet ((dorow (x0 y0 dx dy)
               (let* ((x1 (+ x0 dx)) (y1 (+ y0 dy))
                      (x2 (+ x1 dx)) (y2 (+ y1 dy)))
                 (setq x2 (mod x2 3) y2 (mod y2 3))
                 `(funcall function
                           (aref board ,x0 ,y0) ,x0 ,y0
                           (aref board ,x1 ,y1) ,x1 ,y1
                           (aref board ,x2 ,y2) ,x2 ,y2))))
    (dorow 0 0 0 1)
    (dorow 1 0 0 1)
    (dorow 2 0 0 1)
    (dorow 0 0 1 0)
    (dorow 0 1 1 0)
    (dorow 0 2 1 0)
    (dorow 0 0 1 1)
    (dorow 0 2 1 -1)))
```

```
(defun map-over-pairs-of-ttt-board-rows (board function)
  (macrolet ((dorows (x0 y0 dxa dya dxb dyb)
               (let* ((x1a (+ x0 dxa))  (y1a (+ y0 dya))
                      (x2a (+ x1a dxa)) (y2a (+ y1a dya))
                      (x1b (+ x0 dxb))  (y1b (+ y0 dyb))
                      (x2b (+ x1b dxb)) (y2b (+ y1b dyb)))
                 (setq x2a (mod x2a 3) y2a (mod y2a 3)
                       x2b (mod x2b 3) y2b (mod y2b 3))
                 `(funcall function
                           (aref board ,x0 ,y0) ,x0 ,y0
                           (aref board ,x1a ,y1a) ,x1a ,y1a
                           (aref board ,x2a ,y2a) ,x2a ,y2a
                           (aref board ,x1b ,y1b) ,x1b ,y1b
                           (aref board ,x2b ,y2b) ,x2b ,y2b))))
    (dorows 0 0 0 1 1 0)
    (dorows 0 0 0 1 1 1)
    (dorows 0 0 1 0 1 1)
    (dorows 0 1 1 0 0 1)
    (dorows 0 2 0 -1 1 0)
    (dorows 0 2 0 -1 1 -1)
    (dorows 0 2 1 0 1 -1)
    (dorows 1 0 0 1 1 0)
    (dorows 1 1 0 1 1 0)
    (dorows 1 1 0 1 1 1)
    (dorows 1 1 0 1 1 -1)
    (dorows 1 1 1 0 1 1)
    (dorows 1 1 1 0 1 -1)
    (dorows 1 1 1 1 1 -1)
    (dorows 1 2 0 -1 1 0)
    (dorows 2 0 0 1 -1 0)
    (dorows 2 0 0 1 -1 1)
    (dorows 2 0 -1 0 -1 1)
    (dorows 2 1 -1 0 0 1)
    (dorows 2 2 0 -1 -1 0)
    (dorows 2 2 0 -1 -1 -1)
    (dorows 2 2 -1 0 -1 -1)))

#||
()
(setq ttt (clim:make-application-frame 'tic-tac-toe
            :left 400 :right 800 :top 150 :bottom 400))
(clim:run-frame-top-level ttt)
||#
```

Code for Plotting Data

This code can be found in the file SYS:CLIM;REL-2;TUTORIAL;LEAST-SQUARES-1.LISP.

```
;;; -*- Mode: Lisp; Syntax: ANSI-Common-Lisp; Package: CLIM-USER; Base: 10 -*-

(defclass data-point ()
    ((x :accessor point-x :initarg :x)
     (y :accessor point-y :initarg :y)))

(defun make-data-point (x y)
  (make-instance 'data-point :x x :y y))

(define-presentation-type data-point ())
```

```
(define-application-frame lsq ()
    ((data-points :initform nil)
     (data-points-tick :initform 0)
     (data-x-min :initform 0.0)
     (data-x-max :initform 1.0)
     (data-y-min :initform 0.0)
     (data-y-max :initform 1.0)
     (data-left-margin)
     (data-top-margin)
     (data-right-margin)
     (data-bottom-margin)
     (data-transform)
     (current-curve :initform nil)
     (curve-correlation)
     (current-coefficients :initform (make-array 25 :fill-pointer t))
     (x-is-function-of-y :initform nil))
  (:command-table (lsq :inherit-from (accept-values-pane)))
  (:panes
    (display :application
             :display-function 'draw-data-display
             :incremental-redisplay t
             :display-after-commands t
             :scroll-bars nil)
    (table :application
           :incremental-redisplay t
           :scroll-bars :vertical
           :display-function 'tabulate-data-points)
    (equation :application
              :display-function 'print-equation-of-curve
              :display-after-commands t
              :incremental-redisplay t
              :scroll-bars nil))
  (:layouts
    (drawing-layout
      (vertically () display))
    (tabular-layout
      (vertically () (7/8 table) (1/8 equation)))))


;;; initializations

(defmethod frame-standard-output ((frame lsq))
  (if (eql (frame-current-layout frame) 'drawing-layout)
      (get-frame-pane frame 'display)
      (get-frame-pane frame 'table)))
```

```
(defmethod frame-standard-input ((frame lsq))
  (frame-standard-output frame))

(defmethod frame-query-io ((frame lsq))
  (frame-standard-output frame))

(defmethod run-frame-top-level :before ((frame lsq) &key)
  (enable-frame frame)
  (assign-margins-for-axes frame)
  (determine-data-transform frame))

;;; switching layouts

(define-lsq-command (switch-configurations :menu "Switch Display") ()
  (let ((frame *application-frame*))
    (let ((new-config (case (frame-current-layout frame)
                        (drawing-layout
                          (setf (command-enabled 'com-zoom-in frame) nil)
                          (setf (command-enabled 'com-zoom-out frame) nil)
                          'tabular-layout)
                        (tabular-layout
                          (setf (command-enabled 'com-zoom-in frame) t)
                          (setf (command-enabled 'com-zoom-out frame) t)
                          'drawing-layout))))
      (setf (frame-current-layout frame) new-config))))

;;; adding new points, deleting points
;;; points are maintained in sorted order:

(defun point< (point1 point2)
  (let ((x1 (point-x point1)) (x2 (point-x point2)))
    (cond ((< x1 x2) t)
          ((= x1 x2) (< (point-y point1) (point-y point2))))))

(defmethod add-data-point ((point data-point) (frame lsq))
  (with-slots (data-points-tick data-points) frame
    (setf data-points (merge 'list data-points (list point) #'point<))
    (incf data-points-tick)))

(defmethod delete-data-point ((point data-point) (frame lsq))
  (with-slots (data-points-tick data-points dummy-data-points) frame
    (setf data-points (delete point data-points))
    (incf data-points-tick)))
```

```
(defmethod alter-data-point ((point data-point) (frame lsq) x y)
  (with-slots (data-points-tick data-points dummy-data-points) frame
    (setf (point-x point) x)
    (setf (point-y point) y)
    (setf data-points (sort data-points #'point<))
    (incf data-points-tick)))

;;; table display

(defmethod tabulate-data-points ((frame lsq) pane &key &allow-other-keys)
  (fresh-line pane)
  (flet ((do-point (point stream)
           (with-output-as-presentation (stream point 'data-point
                                         :single-box t)
             (formatting-row (stream)
               (formatting-cell (stream)
                 (format stream "~F" (point-x point)))
               (formatting-cell (stream)
                 (format stream "~F" (point-y point)))))))
    (formatting-table (pane)
      ;; print column headings
      (formatting-row (pane)
        (with-text-face (pane :italic)
          (formatting-cell (pane :min-width 20
                                 :align-x :center)
            (write-string "X" pane))
          (formatting-cell (pane :min-width 20
                                 :align-x :center)
            (write-string "Y" pane))))
      (with-slots (data-points) frame
        (dolist (point data-points)
          (do-point point pane))))))

;;; graphic display
```

```
(defmethod draw-data-display ((frame lsq) pane &key &allow-other-keys)
  (with-slots (current-curve current-coefficients x-is-function-of-y
               data-x-min data-x-max data-y-min data-y-max data-points-tick
               data-right-margin data-left-margin data-transform)
              frame
    (draw-data-axes frame pane)
    (updating-output (pane :unique-id ':the-curve
                           :cache-value data-points-tick
                           :cache-test #'=)
      (draw-data-points frame pane)
      (when current-curve
        (with-drawing-options (pane :transformation data-transform)
          (draw-fitted-curve current-curve current-coefficients pane
                             data-x-min data-x-max data-y-min data-y-max
                             (- data-right-margin data-left-margin)
                             x-is-function-of-y))))))

;;; methods on the data points

(defmethod data-range ((frame lsq))
  (with-slots (data-points) frame
    (let* ((min-x (point-x (first data-points)))
           (max-x min-x)
           (min-y (point-y (first data-points)))
           (max-y min-y))
      (dolist (point (rest data-points))
        (macrolet ((maxminf (val min max)
                     `(let ((v ,val))
                        (cond ((< v ,min) (setq ,min v))
                              ((> v ,max) (setq ,max v))))))
          (maxminf (point-x point) min-x max-x)
          (maxminf (point-y point) min-y max-y)))
      (values min-x min-y max-x max-y))))
```

```
(defmethod determine-data-transform ((frame lsq)
                                     &optional left top right bottom)
  (with-slots (data-left-margin data-top-margin
               data-right-margin data-bottom-margin
               data-x-min data-x-max data-y-min data-y-max
               data-transform data-points-tick) frame
    (when (null left) (setq left data-left-margin))
    (when (null top) (setq top data-top-margin))
    (when (null right) (setq right data-right-margin))
    (when (null bottom) (setq bottom data-bottom-margin))
    (setf data-transform
          (make-3-point-transformation*
            data-x-min data-y-min
            data-x-min data-y-max
            data-x-max data-y-min
            left bottom
            left top
            right bottom))
    (incf data-points-tick)))
```

```
(define-lsq-command (com-set-axis-ranges :menu t)
    ()
  (let ((frame *application-frame*)
        (stream *standard-output*))
    (with-slots (data-x-min data-x-max data-y-min data-y-max
                 data-points data-transform data-points-tick) frame
      (incf data-points-tick)
      (let ((min-x data-x-min)
            (max-x data-x-max)
            (min-y data-y-min)
            (max-y data-y-max))
        (accepting-values
            (stream
              :own-window '(:right-margin (20 :character))
              :label "Enter the ranges for the coordinate axes")
          (format stream "~&Range of X axis: ")
          (flet ((get-one (value id)
                   (accept 'real
                           :stream stream
                           :default value
                           :query-identifier id
                           :prompt nil)))
            (setq min-x (get-one min-x 'x-min))
            (format stream " to ")
            (setq max-x (get-one max-x 'x-max))
            (format stream "~&Range of Y axis: ")
            (setq min-y (get-one min-y 'y-min))
            (format stream " to ")
            (setq max-y (get-one max-y 'y-max)))
          (fresh-line stream)
          (terpri stream)
          (accept-values-command-button
            (stream :query-identifier 'all-of-them)
              "Set ranges to encompass all points"
            (multiple-value-setq (min-x min-y max-x max-y)
              (data-range frame)))
          (fresh-line stream)
          (terpri stream))
        (setq data-x-min min-x
              data-x-max max-x
              data-y-min min-y
              data-y-max max-y))
      (determine-data-transform frame))))
```

```
(defmethod draw-data-points ((frame lsq) pane)
  (with-slots (data-points data-points-tick data-transform
               data-x-min data-x-max data-y-min data-y-max) frame
    (flet ((do-point (point window)
             (let ((x (point-x point)) (y (point-y point)))
               (when (and (<= data-x-min x data-x-max)
                          (<= data-y-min x data-y-max))
                 (with-output-as-presentation (window point 'data-point)
                   (multiple-value-bind (x y)
                       (transform-position data-transform x y)
                     (draw-circle* window x y 3)))))))
      (dolist (point data-points)
        (do-point point pane)))))

(define-lsq-command com-create-data-point ((x 'real) (y 'real))
  (add-data-point (make-data-point x y) *application-frame*))

(define-presentation-to-command-translator new-point
    (blank-area com-create-data-point lsq
     :gesture :select
     :tester
       ((x y window)
        (let ((frame *application-frame*))
          (with-slots (data-left-margin data-top-margin
                       data-right-margin data-bottom-margin) frame
            (and (eql window (get-frame-pane frame 'display))
                 (<= data-left-margin x data-right-margin)
                 (<= data-top-margin y data-bottom-margin))))))
    (x y)
  (with-slots (data-transform) *application-frame*
    (multiple-value-bind (x y)
        (untransform-position data-transform x y)
      (list x y))))

(define-lsq-command com-delete-data-point
    ((point 'data-point :gesture :delete))
  (delete-data-point point *application-frame*))
```

```
(define-lsq-command com-edit-data-point
    ((point 'data-point :gesture :edit))
  (let ((x (point-x point))
        (y (point-y point))
        (stream *standard-output*))
    (accepting-values
        (stream
          :own-window '(:right-margin (20 :character))
          :label "New coordinates for the point")
      (fresh-line stream)
      (setq x (accept 'real
                      :stream stream
                      :prompt "X: "
                      :default x))
      (fresh-line stream)
      (setq y (accept 'real
                      :stream stream
                      :prompt "Y: "
                      :default y)))
    (alter-data-point point *application-frame* x y)))

;;; axes

(defmethod draw-data-axes ((frame lsq) pane)
  (with-slots (data-x-min data-x-max data-y-max data-y-min
                          data-transform
                          data-left-margin data-top-margin
                          data-right-margin data-bottom-margin)
              frame
    (flet ((pair= (pair1 pair2)
             (and (= (car pair1) (car pair2))
                  (= (cdr pair1) (cdr pair2)))))
      (declare (dynamic-extent #'pair=))
      (updating-output (pane :unique-id 'x-axis
                             :cache-value (cons data-x-min data-x-max)
                             :cache-test #'pair=)
        (draw-horizontal-axis pane data-x-min data-x-max
                              data-transform
                              data-left-margin data-right-margin
                              data-bottom-margin))
      (updating-output (pane :unique-id 'y-axis
                             :cache-value (cons data-y-max data-y-min)
                             :cache-test #'pair=)
        (draw-vertical-axis pane data-y-min data-y-max data-transform
                            data-left-margin data-top-margin
                            data-bottom-margin)))))
```

```
(defvar *horizontal-scale-mark-density* 80)

(defun draw-horizontal-axis (window data-min data-max data-transform
                             view-x-min view-x-max view-y)
  (let* ((tick-end view-y)
         (tick-start (+ tick-end 8))
         (text-top (+ tick-start 2)))
    (draw-line* window view-x-min view-y view-x-max view-y)
    (flet ((drawer (x)
             (multiple-value-bind (vx vy)
                 (transform-position data-transform x 0.0)
               vy
               (draw-line* window vx tick-start vx tick-end)
               (draw-text* window (format nil "~7F" x) vx text-top
                           :align-x :center :align-y :top))))
      (declare (dynamic-extent #'drawer))
      (axis-iterator data-min
                     data-max
                     #'drawer
                     (max (round (- view-x-max view-x-min)
                                 *horizontal-scale-mark-density*)
                          1)))))

(defvar *vertical-scale-mark-density* 50)

(defun draw-vertical-axis (window data-min data-max data-transform
                           view-x view-y-min view-y-max)
  (let* ((tick-end view-x)
         (tick-start (- tick-end 8))
         (text-end (- tick-start 2)))
    (draw-line* window view-x view-y-min view-x view-y-max)
    (flet ((drawer (y)
             (multiple-value-bind (vx vy)
                 (transform-position data-transform 0.0 y)
               vx
               (draw-line* window tick-start vy tick-end vy)
               (draw-text* window (format nil "~7F" y) text-end vy
                           :align-x :right :align-y :center))))
      (declare (dynamic-extent #'drawer))
      (axis-iterator data-min
                     data-max
                     #'drawer
                     (max (round (- view-y-max view-y-min)
                                 *vertical-scale-mark-density*)
                          1)))))
```

```
(defun axis-iterator (min-val max-val drawit approx-number-of-steps)
  (let* ((step (float (stepsize-for-scale min-val max-val
                                          approx-number-of-steps)))
         (i0 (ceiling min-val step))
         (i1 (floor max-val step)))
    (loop for i from i0 to i1 do (funcall drawit (* i step)))))

(defun stepsize-for-scale (datamin datamax approx-number-of-steps)
  (declare (values stepsize mantissa))
  (assert (< datamin datamax))
  (let* ((step (/ (- datamax datamin) approx-number-of-steps))
         (step-exponent (expt 10 (floor (log step 10))))
         (step-mantissa (/ step step-exponent)))
    ;; at this point 1 <= step-mantissa < 10
    ;; we choose nearest (logarithmically) to 1, 2, or 5
    (cond ((< step-mantissa (sqrt 2)) (values step-exponent 1))
          ((< step-mantissa (sqrt 5/2)) (values (* 2 step-exponent) 2))
          ((< step-mantissa (* 5.0 (sqrt 2)))
           (values (* 5 step-exponent) 5))
          (t (values (* 10 step-exponent) 1)))))

(defmethod assign-margins-for-axes ((frame lsq))
  (with-slots (data-left-margin data-top-margin
               data-right-margin data-bottom-margin) frame
    (let* ((display (get-frame-pane frame 'display))
           (typical-text (with-output-to-output-record (display)
                           (format display "~7F" 123.456))))
      (multiple-value-bind (width height)
          (bounding-rectangle-size (window-viewport display))
        (setf data-left-margin
              (+ 10 (bounding-rectangle-width typical-text)))
        (setf data-top-margin
              (bounding-rectangle-height typical-text))
        (setf data-right-margin
              (- width (bounding-rectangle-width typical-text)))
        (setf data-bottom-margin
              (- height
                 (+ 10 (bounding-rectangle-height typical-text)))))))))

;;; linear algebra utilities

;;; solves a lower triangular system
```

```
(defun solve-lower-tri (l x)
  (let ((ii (length x)))
    (dotimes (i ii)
      (let ((xi (aref x i)))
        (dotimes (j i)
          (decf xi (* (aref l i j) (aref x j))))
        (setf (aref x i) (/ xi (aref l i i)))))
    x))

;;; solves an upper triangular system stored in transposed form

(defun solve-upper-tri-trans (u x)
  (let ((ii (length x)))
    (do* ((i (1- ii) (1- i))) ((< i 0))
      (let ((xi (aref x i)))
        (do ((j (1+ i) (1+ j))) ((= j ii))
          (decf xi (* (aref u j i) (aref x j))))
        (setf (aref x i) (/ xi (aref u i i)))))
    x))

;;; in-place Cholesky decomposition of a positive definite matrix

(defun cholesky (a)
  (let ((n (array-dimension a 0)))
    (dotimes (k n)
      (let ((akk (aref a k k)))
        (dotimes (j k) (decf akk (expt (aref a k j) 2)))
        (setq akk (sqrt akk))           ;told you it must be positive definite
        (setf (aref a k k) akk)
        (do ((i (1+ k) (1+ i))) ((= i n))
          (let ((aik (aref a i k)))
            (dotimes (j k) (decf aik (* (aref a i j) (aref a k j))))
            (setf (aref a i k) (/ aik akk)))))))
  a)

;;; least squares
```

```
(defun general-linear-regression (value-mapper functions &optional
                                  result (compute-correlation t))
  (let* ((n-coef (length functions))
         (fx (make-array n-coef))
         (dims (list n-coef n-coef))
         (a (make-array dims :initial-element 0)))
    (declare (dynamic-extent fx dims a))
    (if (and (arrayp result)
             (or (= (array-dimension result 0) n-coef)
                 (and (> (array-dimension result 0) n-coef)
                      (array-has-fill-pointer-p result))))
        (progn (dotimes (i n-coef) (setf (aref result i) 0.0))
               (if (array-has-fill-pointer-p result)
                   (setf (fill-pointer result) n-coef)))
        (setq result (make-array n-coef :initial-element 0.0)))
    (flet ((total-up-values (yi &rest xs)
             (let ((functions functions))
               (dotimes (i n-coef)
                 (setf (aref fx i) (apply (pop functions) xs))))
             (dotimes (i n-coef)
               (let ((fxi (aref fx i)))
                 (incf (aref a i i) (* fxi fxi))
                 (incf (aref result i) (* fxi yi))
                 (do ((j (1+ i) (1+ j)))
                     ((= j n-coef))
                   (let* ((fxj (aref fx j)) (fxi-fxj (* fxi fxj)))
                     (incf (aref a i j) fxi-fxj)
                     (incf (aref a j i) fxi-fxj)))))))
      (declare (dynamic-extent #'total-up-values))
      (funcall value-mapper #'total-up-values))
    (cholesky a)
    (solve-lower-tri a result)
    (solve-upper-tri-trans a result)
    (if compute-correlation
        (let ((n 0)
              (sum-y 0.0)
              (sum-y2 0.0)
              (sum-fx 0.0)
              (sum-fx2 0.0))
          (flet ((calc-correlation (yi &rest xs)
                   (let ((fxi (do ((j 0 (1+ j))
                                   (functions functions)
                                   (sum 0.0))
                                  ((null functions) sum)
                                (incf sum (* (aref result j)
                                             (apply (pop functions)
                                                    xs))))))
```

```
                    (incf n)
                    (incf sum-y yi)
                    (incf sum-y2 (* yi yi))
                    (incf sum-fx fxi)
                    (incf sum-fx2 (* fxi fxi)))))
          (declare (dynamic-extent #'calc-correlation))
          (funcall value-mapper #'calc-correlation))
        (values result (/ (+ sum-fx2 (* (/ sum-y n)
                                        (- sum-y (* 2 sum-fx))))
                          (- sum-y2 (/ (* sum-y sum-y) n)))))
      result)))

;;; curves to fit

(defclass fit-curve ()
    ((name :accessor curve-name :initarg :name)
     (n-coefs :accessor curve-n-coefs :initarg :n-coefs)
     (component-functions :initarg :component-functions)
     (printer :initarg :printer :accessor curve-printer)))

(defclass linear-fit-curve (fit-curve) ())

(defmethod function-value ((fit-curve fit-curve) coefficients
                           &rest independent-variable-values)
  (declare (dynamic-extent independent-variable-values))
  (with-slots (component-functions n-coefs) fit-curve
    (let ((total 0.0) (components component-functions))
      (dotimes (i n-coefs)
        (incf total
              (* (aref coefficients i)
                 (apply (pop components) independent-variable-values))))
      total)))

(defmethod least-squares-fit ((fit-curve fit-curve) value-mapper coefficients)
  (with-slots (component-functions n-coefs) fit-curve
    (assert (>= (array-dimension coefficients 0) n-coefs))
    (general-linear-regression value-mapper
                               component-functions
                               coefficients)))

;;; utility component-functions

(defun unity (&rest args) (declare (ignore args)) 1)

(defun square (x) (* x x))

(defun cube (x) (expt x 3))
```

```
;;; the curves that are fitted

(defvar *known-curves* nil)

(defun def-fit-curve (name &key component-functions printer
                                (curve-class 'fit-curve))
  (setq *known-curves* (delete name *known-curves*
                               :test #'string-equal
                               :key #'curve-name))
  (assert (listp component-functions))
  (let ((n-coefs (length component-functions)))
    (push (make-instance curve-class
                         :name name
                         :n-coefs n-coefs
                         :component-functions component-functions
                         :printer printer)
          *known-curves*)))

(def-fit-curve "Cubic"
  :component-functions (list #'cube #'square #'identity #'unity)
  :printer #'(lambda (stream var coefs)
               (format stream "~7F ~A^3 + ~7F ~A^2 + ~7F ~A + ~7F"
                       (elt coefs 0) var (elt coefs 1) var
                       (elt coefs 2) var (elt coefs 3))))

(def-fit-curve "Quadratic"
  :component-functions (list #'square #'identity #'unity)
  :printer #'(lambda (stream var coefs)
               (format stream "~7F ~A^2 + ~7F ~A + ~7F"
                       (elt coefs 0) var (elt coefs 1) var
                       (elt coefs 2))))

(def-fit-curve "Linear"
  :curve-class 'linear-fit-curve
  :component-functions (list #'identity #'unity)
  :printer #'(lambda (stream var coefs)
               (format stream "~7F ~A + ~7F" (elt coefs 0) var
                       (elt coefs 1))))


;;; curve display

;;; the default method of drawing curves
```

```
(defmethod draw-fitted-curve ((curve fit-curve) coefficients pane
                              data-x-min data-x-max data-y-min data-y-max
                              n-plotting-steps x-is-function-of-y)
  (labels ((plotter (umin umax vmin vmax drawer)
             (let ((du (/ (- umax umin) n-plotting-steps)))
               (do* ((u0 nil u1)
                     (v0 nil v1)
                     (u1 umin (+ u1 du))
                     (v1 (function-value curve coefficients u1)
                         (function-value curve coefficients u1)))
                    ((> u1 umax))
                 (when (and u0
                            (<= vmin v0 vmax)
                            (<= vmin v1 vmax))
                   (funcall drawer u0 v0 u1 v1)))))
           (y=fx-drawer (u0 v0 u1 v1)
             (draw-line* pane u0 v0 u1 v1))
           (x=fy-drawer (u0 v0 u1 v1)
             (draw-line* pane v0 u0 v1 u1)))
    (declare (dynamic-extent #'y=fx-drawer #'x=fy-drawer))
    (if x-is-function-of-y
        (plotter data-y-min data-y-max data-x-min data-x-max #'x=fy-drawer)
        (plotter data-x-min data-x-max data-y-min data-y-max #'y=fx-drawer))))

;;; the linear-fit-curve class has a faster method of drawing
```

```
(defmethod draw-fitted-curve ((curve linear-fit-curve) coefficients pane
                              data-x-min data-x-max data-y-min data-y-max
                              n-plotting-steps x-is-function-of-y)
  (declare (ignore n-plotting-steps))
  (labels ((linterp (x x0 x1 y0 y1)
             (let ((d0 (- x x0)) (d1 (- x1 x)))
               (/ (+ (* d0 y1) (* d1 y0)) (+ d0 d1))))
           (plotter (umin umax vmin vmax drawer)
             (let ((u0 umin)
                   (v0 (function-value curve coefficients umin))
                   (u1 umax)
                   (v1 (function-value curve coefficients umax)))
               (macrolet ((v-clip (<< vlimit)
                            `(if (,<< v0 ,vlimit)
                                 (if (,<< v1 ,vlimit)
                                     (return-from plotter)
                                     (setq u0 (linterp ,vlimit v0 v1 umin umax)
                                           v0 ,vlimit))
                                 (if (,<< v1 ,vlimit)
                                     (setq u1 (linterp ,vlimit v0 v1 umin umax)
                                           v1 ,vlimit)))))
                 (v-clip < vmin)
                 (v-clip > vmax))
               (funcall drawer u0 v0 u1 v1)))
           (y=fx-drawer (u0 v0 u1 v1)
             (draw-line* pane u0 v0 u1 v1))
           (x=fy-drawer (u0 v0 u1 v1)
             (draw-line* pane v0 u0 v1 u1)))
    (declare (dynamic-extent #'y=fx-drawer #'x=fy-drawer))
    (if x-is-function-of-y
        (plotter data-y-min data-y-max data-x-min data-x-max #'x=fy-drawer)
        (plotter data-x-min data-x-max data-y-min data-y-max #'y=fx-drawer))))


;;; curve printing
```

```
(defmethod print-equation-of-curve ((frame lsq) pane &key &allow-other-keys)
  (with-slots (current-curve x-is-function-of-y data-points-tick
               current-coefficients curve-correlation) frame
    (updating-output (pane :unique-id ':printed-equation
                           :cache-value (cons current-curve data-points-tick))
      (when current-curve
        (multiple-value-bind (dep-var ind-var)
            (if x-is-function-of-y (values "X" "Y") (values "Y" "X"))
          (format pane "~A = " dep-var)
          (funcall (curve-printer current-curve) pane ind-var current-coefficients)
          (format pane "~& Correlation: ~7F ~& " curve-correlation))))))

;;; interface to least-squares

(define-lsq-command (com-fit-curve :menu "Fit Curve")
    ()
  (fit-curve *application-frame*))

;;; modifying the data set invalidates the least squares fit

(defmethod add-data-point :after ((point data-point) (frame lsq))
  (with-slots (current-curve) frame
    (setf current-curve nil)))

(defmethod delete-data-point :after ((point data-point) (frame lsq))
  (with-slots (current-curve) frame
    (setf current-curve nil)))

;;; here's where we can control y-as-function-of-x vs x-as-function-of-y
;;; and limited data-sets, and other variables, etc by constructing the
;;; appropriate mapper
```

```
(defmethod fit-curve ((frame lsq))
  (with-slots (current-curve current-coefficients curve-correlation
               data-points data-points-tick x-is-function-of-y) frame
    (incf data-points-tick)
    (setf current-curve
          (menu-choose *known-curves*
                       :label "Curve to Fit"
                       :printer #'(lambda (curve stream)
                                    (write-string (curve-name curve) stream))))
    (when current-curve
      (if (>= (length data-points) (curve-n-coefs current-curve))
          (flet ((y-as-function-of-x-value-mapper (function)
                   (dolist (point data-points)
                     (funcall function (point-y point) (point-x point))))
                 (x-as-function-of-y-value-mapper (function)
                   (dolist (point data-points)
                     (funcall function (point-x point) (point-y point)))))
            (declare (dynamic-extent #'y-as-function-of-x-value-mapper
                                     #'x-as-function-of-y-value-mapper))
            (multiple-value-setq (current-coefficients curve-correlation)
              (least-squares-fit current-curve
                                 (if x-is-function-of-y
                                     #'x-as-function-of-y-value-mapper
                                     #'y-as-function-of-x-value-mapper)
                                 current-coefficients)))
          (progn
            (notify-user frame
                         (format nil "Not enough data points to fit a ~A function"
                                 (curve-name current-curve)))
            (setq current-curve nil))))))

(define-lsq-command (com-exit-lsq :menu "Exit")
    ()
  (frame-exit *application-frame*))

#||
()
(setq *lsq* (make-application-frame 'lsq
              :height 500 :width 500))
(run-frame-top-level *lsq*)
||#
```

Using Symbolics CLIM

Using CLIM in Genera

For more information on getting started, see the section "Running a CLIM Application".

Also, see the section "Using CLIM in CLOE Developer", and see the section "Using the CLIM Demos".

First, you must load the CLIM systems. Note that you must load the system `CLIM` before you can load `GENERA-CLIM`, `CLX-CLIM`, or `POSTSCRIPT-CLIM`.

```
Command: Load System CLIM
Command: Load System GENERA-CLIM     ;if you want the Genera port
Command: Load System CLX-CLIM        ;if you want the CLX port
Command: Load System POSTSCRIPT-CLIM ;if you want the PostScript port
```

Note: The CLX port of CLIM is *not* currently a supported port of CLIM under Genera. It exists only as a sample port for people who are interested in perhaps implementing a port of their own.

Using the Debugger with CLIM

If you enter the Debugger with a full-screen CLIM window, you will see a pop-up notification such as `"Process CAD Demo 1 got an error"`. This is normal, since the Debugger cannot display itself on the CLIM window. To enable the Debugger display (so you can find out about the error), you should select the Listener window. Usually this is straightforward (for example, by clicking on the pop-up window).

Once you have finished debugging and want to return to the CLIM window, you can press `FUNCTION S`. Another way to return to the CLIM window is to use the `[Select]` command from the System Menu. CLIM windows made by application frames appear in the menu with the name of the application.

In some cases, after you have finished experimenting with your application in the Debugger, you need to restore the application to a running state before reselecting the CLIM window. If you are offered a top-level restart for the application, that can work well. Otherwise, you can try to return from a com-*your-command* frame, if one happens to be in your stack.

For small experiments (such as running some example code from the documentation), you might try creating a small Lisp Listener so that both the experimental CLIM window and the Lisp Listener on which the Debugger will appear are both exposed and non-overlapping.

To do this, use the System Menu commands (available by clicking `sh`-Right) to move, reshape, and expand the windows so that both are exposed. For example, when you are in Lisp Listener, choose `[Reshape]` from the System Menu to make the Lisp Listener window smaller. Then create the CLIM application frame. You can use the `[Move]` command to move one of the windows so it does not overlap the other. You can use the `[Expand]` command to make one of the windows take up

the space not occupied by the other window, so that the windows together take up the whole screen.

You can easily switch from one window to the other by clicking Left on a window. You can use the Lisp Listener window to use the Debugger, and use the CLIM window to experiment with CLIM code.

Using CLIM with a Color Screen in Genera

In order to use CLIM on a ''two headed'' color system under Genera, you must first find the color screen to use. Use **color:find-color-screen** to do this. Then call **clim:find-port** to make a CLIM port, using the screen returned by **color:find-color-screen**.

When you create an application frame, specify the port that corresponds to the color screen as the port. For example, you can run the CLIM Lisp Listener on a color screen as follows:

```
(clim:run-frame-top-level
  (clim-demo::do-lisp-listener
    :port (clim:find-port
            :server-path `(:genera :screen ,(color:find-color-screen)))))
```

If you are using CLIM on a machine that has ''native'' color support (such as a color MacIvory or a color DEC Alpha AXP or UX400/UX1200), you don't need to do anything special. CLIM will notice that the display supports color, and it will simply work as expected.

Using CLIM in CLOE

This section describes how to use CLIM in Cloe Runtime, and in Cloe Developer. It also describes how to run the CLIM demos in Cloe.

For information on how to run CLIM in Genera, see the section "Using CLIM in Genera".

Using CLIM in CLOE Runtime

Cloe and CLIM runs with MS-DOS 3.3 or later, and with Microsoft Windows 3.0 in STANDARD mode. The Cloe Developer runs with Genera 8.1 and uses CLIM 1.0.

Getting Started

Use the following procedure to start up CLIM. Note that you must start up CLIM from MS-DOS (that is, you cannot start up CLIM from within Windows). Also note that you can save these commands in a Lisp startup file (INIT.L is loaded automatically at startup).

1. Get into the DOS directory containing the CLIM files.

2. Enter the following command to start up Cloe with CLIM embedded.

```
cloeclim
```

3. (Optional) Set **win::*winfe-exe*** to a string that points to the WINFE.EXE file. For example:

```
(setq win::*winfe-exe* "c:\\clim\\winfe.exe")
```

   If `winfe.exe` is in your current working directory, or in a directory specified in your `PATH` shell variable, you do not need to set this special variable in Cloe.

   If you do not set this variable, Cloe will search for the file based on the path directory for MS-DOS.

4. To start up MS Windows evaluate the following Lisp form:

```
(win:start-windows)
```

   or create a CLIM port:

```
(clim:find-port :server-path '(:cloe))
```

   You will be in a terminal WINFE (Window Front End) window. Note that this window wraps when you fill the screen (it does not scroll).

You can now run your CLIM application. Note that you can compile, write, or load CLIM files anytime, but you must have MS Windows running to run the CLIM code.

Running CLIM Applications

Note that when you are running a CLIM application in Cloe:

- CLIM applications usually fill the whole screen.
- You must have the input focus to type to a window. (The window title bar is highlighted for a window with input focus.) Click in a window to give it the input focus.
- You can type `c-C` anytime to cause a break and set the input focus back to the Cloe Front End window. This causes the Cloe Front End window to come to the top, and to be de-iconified if necessary. (To continue after a break use the debugger Continue option.)
- With Windows running, you can switch between the MS-DOS executive and CLIM using the standard Windows commands. You can also run any other Windows program, as long as it doesn't use memory already used by Cloe (most Windows programs don't). See *Microsoft Windows User Guide* by Microsoft Corporation.

For more information, see the section "Running a CLIM Application".

Exiting from CLIM

Use one of the following methods to exit from CLIM after Windows has started up.

- Use the [Close] option from the leftmost pulldown menu
- Press ALT-F4 once (to exit Cloe), then again (to exit from Windows), and press RETURN (to confirm).

Do not use (exit). Windows acts as a subprocess of Cloe (since it was activated after Cloe). If Cloe exits, the Windows process hangs and you must reboot the machine.

Using CLIM in CLOE Developer

CLIM is not a system that application developers need to migrate. Symbolics supplies a Cloe Runtime image with CLIM built in. Therefore, we recommend Cloe Developer users load CLIM into a regular Genera listener, and then access CLIM from Cloe.

Getting Started

First, in a Genera Lisp Listener, give the following commands:

```
Load System CLIM
Load System GENERA-CLIM     ;if you want the Genera port
Load System POSTSCRIPT-CLIM ;if you want the PostScript port
Load System CLIM-DEMO       ;if you want the CLIM demos
```

Next, in the Cloe Listener, give the following command:

```
Make CLIM Available
```

Using the Debugger with CLIM under the Cloe Developer

While experimenting with CLIM or debugging a CLIM application, if you go into the Debugger, the Debugger cannot be displayed on a CLIM window. Assuming that a CLIM window is exposed when the error happens, we recommend the following two-window approach.

For small experiments (such as running some example code from the documentation), we recommend you resize the Cloe Listener so that both the experimental CLIM window and the Cloe Listener (on which the Debugger will appear) are both exposed and non-overlapping.

To do this, use the System Menu commands (available by clicking sh-Right) to move, reshape, and expand the windows so that both are exposed. For example, when you are in the Cloe Listener, choose Reshape from the System Menu to make the Cloe Listener window smaller. Then create the CLIM window. You can use the

Move command to move one of the windows so it does not overlap the other. You can use the Expand command to make one of the windows take up the space not occupied by the other window, so that the windows together take up the whole screen.

You can easily switch from one window to the other by clicking Left on a window. You can use the Cloe Lisp Listener window to use the Debugger, and use the CLIM window to experiment with CLIM code.

For experiments with program frames (where you usually want to see a full-screen version of your program), the two-window strategy may not be appropriate. For these applications, you should consider the size of the window on the 386 PC. A standard VGA screen is 640 horizontal by 480 vertical pixels. If you use these values to size the application, you can still use the two-window approach. If you expect your 386-based users to use 800x600 or 1024x768 resolution, you should use the full screen technique described in "Using CLIM in Genera"

Setting up Your Packages to Use CLIM

You can use any of the following approaches for setting up your packages to use CLIM. Using these approaches will give a high likelihood that your application can be ported from one Common Lisp platform to another with minimal effort.

- Use an explicit **clim:** package prefix whenever referencing a symbol in CLIM, and define your package as usual.

- Define your package to inherit from the **clim** and **clim-lisp** packages.

  For example,

  ```
  (defpackage clim-user
     (:use clim-lisp clim))
  ```

  Note that you must inherit from both packages. The **clim-lisp** package provides an implementation of Common Lisp that is similar to the draft ANSI standard, with modifications to work with CLIM.

- Define a package that inherits from the Common Lisp dialect of your choice and from the **clim** package.

  For example,

```
(defpackage another-clim-user
    (:use clim common-lisp)
    #-Cloe-Runtime
    (:shadowing-import
      clim:input-stream-p
      clim:output-stream-p
      clim:open-stream-p
      clim:streamp
      clim:stream-element-type
      clim:close
      clim:pathname
      clim:truename))
```

Note that you may need to use shadowing to resolve conflicts (the exact set of conflicts will vary depending on what other package you inherit from).

Converting from Dynamic Windows to CLIM

The Dynamic Windows to CLIM conversion tool is a series of special-purpose Zmacs commands that can save you time and effort in editing large pieces of Dynamic Windows code in ways that can be done semi-automatically.

Note: these conversion tools are supported only in Genera.

Depending on the complexity of your Dynamic Windows code, these tools may or may not perform the conversion completely. You will probably need to perform additional editing of your Dynamic Windows programs in order to generate working CLIM programs. The closer your Dynamic Windows source program is to standard Dynamic Windows, the more automatic the conversion process will be.

For more information on conversion tools in general, see the section "Conversion Tools".

Though using the conversion tool is simple, the conversion task is not. For this reason you must use the conversion tool with knowledge and care.

Prerequisites

- You should be sufficiently comfortable with both Dynamic Windows and CLIM systems to be aware of function equivalences and to understand the possible effects of a conversion (such as a changed order of argument evaluation for functions whose calling sequence is different).
- You should be familiar with the basic workings of the Zmacs editor.

Note: these conversion commands are intended as an *aid* to conversion, not as a fully automatic conversion tool. Used properly, they will save you time and effort, but you must monitor the results carefully after each step and be aware that you might have to do some manual work after conversion. For instance, comments might not end up exactly in the right place in the rearranged program, indentation might change, converted functions might need some additions to the code, and so

on. Many CLIM functions support fewer options than their Dynamic Windows counterparts, so conversion often removes these options from the program or leaves them unconverted, which will cause a run-time error. Either way, you'll need to decide what you want the CLIM version of the program to do and adjust it accordingly.

Getting Started

Load the CLIM and Conversion Tools systems (in either order).

```
Command: Load System CLIM
Command: Load System GENERA-CLIM     ;if you want the Genera port
Command: Load System POSTSCRIPT-CLIM ;if you want the PostScript port
Command: Load System Conversion Tools
```

Note: The CLX port of CLIM is *not* currently a supported port of CLIM under Genera. It exists only as a sample port.

Conversion Procedures

1. Most Dynamic Windows programs are written in Symbolics Common Lisp. If yours is written in Zetalisp, first convert it to Symbolics Common Lisp using the Zetalisp to Common Lisp conversion tool (see the section "Zetalisp to Common Lisp Conversion").

2. Convert your program to a new package using either of these commands:

   ```
   m-X Convert Package of Buffer
   m-X Convert Package of Tag Table
   ```

   This step is optional, but it is useful because it allows you to run the Dynamic Windows and CLIM versions of the program side by side without them interfering with each other. When you test and fix the CLIM version, it's often useful to have the original Dynamic Windows version available for comparison.

   If you plan to convert your program to a more portable dialect of Common Lisp, along with converting to CLIM, you should select a package in the CLtL, CLtL-Only, or Cloe package universe.

3. Compile any macro definitions that your program uses. This step is optional, but helps the next step work better. Some conversions, for example of command definitions, analyze the source code of your program and having all the macro definitions available results in more accurate analysis.

4. Convert the Dynamic Windows functions to CLIM using either of these commands with the ''DW to CLIM'' conversion set:

```
m-X Convert Functions of Buffer
m-X Convert Functions of Tag Table
```

Read the queries carefully before answering them, and don't type ahead. It is often useful to accept conversions that are not quite correct; you will probably need to do some manual conversion afterwards anyway, so it may be useful to accept an automatic conversion that gets you closer to the goal (even if it may not be exactly what you want).

5. Review the warnings printed during the conversion, check over the code by hand, and compile it. You can use `c-m-Scroll` to review the warnings printed in the typeout window. While checking over the code, you might want to fix up the indentation and formatting; some of the more complex conversions tend to mangle the indentation and line divisions, producing code that works but is difficult to read.

6. Note that since a Dynamic Windows *program-framework* is a Flavors object, and a CLIM *application frame* is a CLOS object, you may need to run the Flavors to CLOS conversion tool before your CLIM program will compile.

   See the section "Flavors to CLOS Conversion".

7. You might want to run the ''Symbolics Common Lisp to Portable Common Lisp'' and ''Common Lisp to Common Lisp Developer'' conversion tools at this time, especially if you selected a package universe other than Common Lisp in step 2. See the section "Symbolics Common Lisp to Portable Common Lisp Conversion".

When you are satisfied with your CLIM program, and it compiles without errors or warnings, you can run it.

For example, to run your CLIM program under Genera use **clim:define-genera-application** and the Select Activity command.

You can also use the following procedure:

```
(setq *port* (clim:find-port))
(setq *frame* (clim:make-application-frame 'your-frame-type :parent *port*))
(clim:run-frame-top-level *frame*)
```

Summary of Differences Between CLIM 1.1 and CLIM 2.0

CLIM 2.0 is a more robust product than CLIM 1.1. Many customer-requested features have been added, and many bugs have been fixed.

However, the main reason for the existence of CLIM 2.0 is to provide integrated support for gadgets and event management so that CLIM applications can use many of the facilities found in standard toolkits. Some of the new features of CLIM 2.0 include the following:

- A new window and event management model that supports use of standard user interface toolkits when running on standard platforms, such as Motif under Allegro or Lucid Common Lisp. Since Genera does not support toolkits like Motif, CLIM 2.0 includes a set of gadgets, including scroll bars, push buttons, toggle buttons, radio and check boxes, pull-down menus, sliders, text editing panes, and list and option panes.
- Integration between CLIM's gadgets and **clim:accepting-values**.
- A set of drawing functions that draw multiple graphics, for example, **clim:draw-lines*** and **clim:draw-rectangles***.
- CLIM 2.0 supports use of pixmaps, and has **clim:with-output-to-pixmap** and **clim:copy-area** functions. This can be used to cache portions of a display that needs to be rapidly, repeatedly drawn.
- Functions to read X11 bitmap files and convert them to CLIM patterns, such as **clim:make-pattern-from-bitmap-file**.
- Keyboard gestures are now specified in a more portable fashion. For example, what would have been `#\control-X` in CLIM 1.1 is now specified as `(:x :control)`. This allows greater portability, but it is an incompatible change from CLIM 1.1.
- A new form for defining drag-and-drop translators, called **clim:define-drag-and-drop-translator**.
- The completion presentation types now support **:printer** and **:highlighter** options.
- The input editor has a much richer set of editing commands. Type control-Help to see the entire set of commands.
- The appearance of the mouse cursor can be changed, either directly by calling **setf** on **clim:pointer-cursor**, or by changing the cursor associated with a CLIM sheet by calling **setf** on **clim:sheet-pointer-cursor**.
- Command tables may now inherit menu items from superior command tables.
- The command processor now tells you what the defaults are for keyword arguments when you type Help while reading a command.
- **clim:surrounding-output-with-border** now takes drawing options.
- The graph formatter is now more sophisticated.
- The new **clim-sys** package contains a number of generally useful utilities.
- A number of new demos, including a ''color chooser'', a simple bitmap editor, a simple graphical editor, a Peek-like utility, a data plotting program, and a graphical browser. Note that these are demo programs; they are not intended to be of product quality, but are meant to be instructive in the use of CLIM 2.0. The CLIM Lisp Listener is particularly useful when you are debugging fragments of CLIM code; type `Select Lambda (symbol-shift-L)` in Genera to use it.

Converting From CLIM 1.1 to CLIM 2.0

The CLIM 1.1 to CLIM 2.0 conversion tools are another set of commands like the Dynamic Windows to CLIM conversion tools. Like the Dynamic Windows to CLIM conversion tools, the CLIM 1.1 to CLIM 2.0 tools will not perform the conversion completely. You will need to perform additional editing of your CLIM 1.1 programs in order to generate working CLIM 2.0 programs.

Again, these conversion commands are intended as an *aid* to conversion to CLIM 2.0, not as a fully automatic conversion tool. Used properly, they will save you time and effort, but you must monitor the results carefully after each step and be aware that you might have to do some manual work after conversion.

Note: these conversion tools are supported only in Genera.

For more information on conversion tools in general, see the section "Conversion Tools" and see the section "Converting from Dynamic Windows to CLIM".

Getting Started

Load the CLIM and Conversion Tools systems (in either order).

```
Command: Load System CLIM
Command: Load System GENERA-CLIM     ;if you want the Genera port
Command: Load System POSTSCRIPT-CLIM ;if you want the PostScript port
Command: Load System Conversion Tools
```

Conversion Procedures

1. Convert the CLIM 1.1 functions to CLIM 2.0 using either of these commands with the ‘‘CLIM 1.1 to CLIM 2.0’’ conversion set:

   ```
   m-X Convert Functions of Buffer
   m-X Convert Functions of Tag Table
   ```

   Read the queries carefully before answering them, and don’t type ahead. It is often useful to accept conversions that are not quite correct; you will probably need to do some manual conversion afterwards anyway, so it may be useful to accept an automatic conversion that gets you closer to the goal (even if it may not be exactly what you want).

2. Review the warnings printed during the conversion, check over the code by hand, and compile it. You can use `c-m-Scroll` to review the warnings printed in the typeout window. While checking over the code, you might want to fix up the indentation and formatting; some of the more complex conversions tend to mangle the indentation and line divisions, producing code that works but is difficult to read.

3. The CLIM 1.1 to CLIM 2.0 conversion tools do not convert the **:panes** and **:layouts** options from **clim:define-application-frame**. You must do this yourself.

When you are satisfied with your CLIM 2.0 program, and it compiles without errors or warnings, you can run it.

Here are some of the incompatible changes from CLIM 1.1 to CLIM 2.0. Most of these are picked up by the CLIM 1.1 to CLIM 2.0 conversion tools.

- The **:panes** and **:layouts** clauses to **clim:define-application-frame** are now completely different. The conversion tools do not handle this, since it is not clear what should be done in many cases.
- **clim:run-frame-top-level** now takes keyword arguments. You must include **&key** in your methods for this function.
- **clim:set-frame-layout** has been removed in favor of using **setf** on **clim:frame-current-layout**.
- **clim:frame-top-level-window** is now called **clim:frame-top-level-sheet**.
- **clim:command-enabled-p** is now called **clim:command-enabled**. **clim:disable-command** and **clim:enable-command** have been removed in favor of using **setf** on **clim:command-enabled**.
- **clim:open-root-window** has been removed. Its closest replacement is **clim:find-port**, although you may find that you rarely need to explicitly specify a port.
- The **:stream**, **:object**, and **:type** keyword arguments to **clim:with-output-as-presentation** are now required arguments, since it was always necessary to supply these arguments.
- **clim:+background+** is now called **clim:+background-ink+**, and **clim:+foreground+** is now called **clim:+foreground-ink+**. This was done to be consistent with **clim:+flipping-ink+**.
- **clim:make-color-rgb** is now called **clim:make-rgb-color**, and **clim:make-color-ihs** is now called **clim:make-ihs-color**.
- **clim:draw-character**, **clim:draw-character***, **clim:draw-string**, and **clim:draw-string*** have all been removed in favor of using **clim:draw-text** and **clim:draw-text***.
- **clim:draw-icon** and **clim:draw-icon*** are now called **clim:draw-pattern***.
- The argument order to **clim:with-text-style**, **clim:with-text-family**, **clim:with-text-face**, and **clim:with-text-size** has been changed so that the stream argument is first.
- **clim:stream-cursor-position*** is now called **clim:stream-cursor-position**, **clim:stream-set-cursor-position*** is now called **clim:stream-set-cursor-position**, and **clim:stream-increment-cursor-position*** is now called **clim:stream-increment-cursor-position**.
- **clim:cursor-position*** is now called **clim:cursor-position**, and **clim:cursor-set-position*** is now called **clim:cursor-set-position**.
- The argument order to **clim:with-end-of-line-action** and **clim:with-end-of-page-action** has been changed so that the stream argument is first.
- **clim:stream-pointer-position*** is now called **clim:stream-pointer-position**, and **clim:stream-set-pointer-position*** is now called **clim:stream-set-pointer-position**.

- **clim:pointer-position*** is now called **clim:pointer-position**, and **clim:pointer-set-position*** is now called **clim:pointer-set-position**.
- **clim:event-window** is now called **clim:event-sheet**.
- **clim:pointer-event-shift-mask** is now called **clim:event-modifier-state**.
- The **:inter-column-spacing**, **:inter-row-spacing**, and **:multiple-columns-inter-column-spacing** keyword arguments to **clim:formatting-table** have been renamed to **:x-spacing**, **:y-spacing**, and **:multiple-columns-x-spacing**.
- The **:minimum-width** and **:minimum-height** keyword arguments to **clim:formatting-cell** have been renamed to **:min-width** and **:min-height**.
- The **:inter-column-spacing**, **:inter-row-spacing**, and **:no-initial-spacing** keyword arguments to **clim:formatting-item-list** and **clim:format-items** have been renamed to **:x-spacing**, **:y-spacing**, and **:initial-spacing**.
- The **:inter-column-spacing** and **:inter-row-spacing** keyword arguments to **clim:menu-choose** and **clim:draw-standard-menu** have been renamed to **:x-spacing** and **:y-spacing**.
- The **:draw-p** and **:record-p** keyword arguments to **clim:with-output-recording-options** have been renamed to **:draw** and **:record**.
- **clim:*unsupplied-argument*** is now called **clim:*unsupplied-argument-marker***.
- The **:inter-column-spacing** and **:inter-row-spacing** keyword arguments to **clim:display-command-table-menu** has been renamed to **:x-spacing** and **:y-spacing**.
- The **:test** keyword argument has been removed from **clim:add-command-to-command-table**, **clim:add-keystroke-to-command-table**, and **clim:remove-keystroke-from-command-table**.
- The **:keystroke-test** keyword argument has been removed from **clim:read-command** and **clim:read-command-using-keystrokes**.
- **clim:window-viewport-position*** is now called **clim:window-viewport-position**, and **clim:window-set-viewport-position*** is now called **clim:window-set-viewport-position**.
- **clim:position-window-near-carefully** is now called **clim:position-sheet-carefully**.
- **clim:position-window-near-pointer** is now called **clim:position-sheet-near-pointer**.
- **clim:size-menu-appropriately** is now called **clim:size-frame-from-contents**.
- **clim:stream-draw-p** is now called **clim:stream-drawing-p**, and **clim:stream-record-p** is now called **clim:stream-recording-p**.
- **clim:output-record-position*** is now called **clim:output-record-position**, and **clim:output-record-set-position*** is now called **clim:output-record-set-position**.
- **clim:output-record-element-count** is now called **clim:output-record-count**.
- **clim:output-record-elements** is now called **clim:output-record-children**.

- **clim:output-record-refined-sensitivity-test** is now called **clim:output-record-refined-position-test**.
- **clim:output-recording-stream-output-record** is now called **clim:stream-output-history**.
- **clim:output-recording-stream-current-output-record-stack** is now called **clim:stream-current-output-record**.
- **clim:output-recording-stream-replay** is now called **clim:stream-replay**.
- **clim:add-output-record** is now called **clim:stream-add-output-record**.
- **clim:add-output-record-element** is now called **clim:add-output-record**, and **clim:delete-output-record-element** is now called **clim:delete-output-record**.
- **clim:map-over-output-record-elements** is now called **clim:map-over-output-records**, **clim:map-over-output-record-elements-containing-point*** is now called **clim:map-over-output-records-containing-position**, and **clim:map-over-output-record-elements-overlapping-region** is now called **clim:map-over-output-records-overlapping-region**.
- **clim:dragging-output-record** is now called **clim:drag-output-record**.
- The *frame* argument to **clim:find-presentation-translators** is now a *command-table* argument.
- The **:shift-mask** keyword argument to **clim:test-presentation-translator**, **clim:find-applicable-translators**, **clim:presentation-matches-context-type**, and **clim:find-innermost-applicable-presentation** is now a **:modifier-state** argument.
- **clim:define-gesture-name** uses a completely different syntax for specifying the gesture, and **clim:add-pointer-gesture-name** has been replace by **clim:add-gesture-name**.
- **clim:remove-pointer-gesture-name** is now called **clim:delete-gesture-name**.
- **clim:dialog-view** is now called **clim:textual-dialog-view**, and **clim:+dialog-view+** is now called **clim:+textual-dialog-view+**.
- **clim:menu-view** is now called **clim:textual-menu-view**, and **clim:+menu-view+** is now called **clim:+textual-menu-view+**.
- **clim:call-presentation-generic-function** has been replaced by a pair of functions, **clim:apply-presentation-generic-function** and **clim:funcall-presentation-generic-function**.
- The **:activation-characters**, **:additional-activation-characters**, **:blip-characters**, and **:additional-blip-characters** to **clim:accept** are now called **:activation-gestures**, **:additional-activation-gestures**, **:delimiter-gestures**, and **:additional-delimiter-gestures**.
- **clim:*activation-characters*** is now called **clim:*activation-gestures***, and **clim:*standard-activation-characters*** is now called **clim:*standard-activation-gestures***.
- **clim:*blip-characters*** is now called **clim:*delimiter-gestures***.
- **clim:activation-character-p** is now called **clim:activation-gesture-p**, **clim:blip-character-p** is now called **clim:delimiter-gesture-p**.

- **clim:with-activation-characters** is now called **clim:with-activation-gestures**, and **clim:with-blip-characters** is now called **clim:with-delimiter-gestures**.
- **clim:*abort-characters*** is now called **clim:*abort-gestures***.
- **clim:*complete-characters*** is now called **clim:*completion-gestures***, **clim:*help-characters*** is now called **clim:*help-gestures***, and **clim:*possibilities-characters*** is now called **clim:*possibilities-gestures***.
- **clim:input-position** is now called **clim:stream-scan-pointer**, and **clim:insertion-pointer** is now called **clim:stream-insertion-pointer**, and **clim:rescanning-p** is now called **clim:stream-rescanning-p**.
- The *right* and *bottom* arguments to **clim:with-bounding-rectangle*** are now required.
- **clim:point-position*** is now called **clim:point-position**.
- **clim:region-contains-point*-p** is now called **clim:region-contains-position-p**.
- **clim:bounding-rectangle-position*** is now called **clim:bounding-rectangle-position**, and **clim:bounding-rectangle-set-position*** is now called **clim:bounding-rectangle-set-position**.
- The argument order to **clim:make-3-point-transformation** and **clim:make-3-point-transformation*** has been changed.
- **clim:compose-rotation-transformation** is now called **clim:compose-rotation-with-transformation**, **clim:compose-scaling-transformation** is now called **clim:compose-scaling-with-transformation**, and **clim:compose-translation-transformation** is now called **clim:compose-translation-with-transformation**.
- **clim:transform-point*** is now called **clim:transform-position**, and **clim:untransform-point*** is now called **clim:untransform-position**.

Using the CLIM Demos

The CLIM Demo system provides a number of examples of small and medium-sized applications that use CLIM. They all run as part of a demo loop that uses a CLIM menu to let you choose which demo to run. The source files for the CLIM demos in included in `SYS:CLIM;REL-2;DEMO;*.LISP`. You may find that reading the source code for the CLIM demos will provide many useful hints and techniques for writing your own CLIM applications.

This section describes how to run the CLIM demos.

To run the demos:

- Ensure that the `CLIM` system is loaded, plus an appropriate back-end (such as `Genera-CLIM`). Then load the system `CLIM-DEMOS`.
- To run the demos, evaluate the form **(clim-demo:start-demo)**. If you are running Genera, you can also use the `:Demonstrate CLIM` command. This will pop up a menu of the demos to run.

If you are using Genera, you may also want to try the CLIM demos using the ''gadget menu bar'' style of frame manager. You can do this by evaluating the following:

```
(clim-demo:start-demo
  :framem (clim:find-frame-manager :gadget-menu-bar t))
```

The following sections describe the operation of each of the demos.

The CLIM Lisp Listener

This is a simple Lisp Listener (in SYS:CLIM;REL-2;DEMO;LISTENER.LISP). It includes a basic Lisp read-eval-print loop to which you can type either Lisp forms or CLIM commands. It includes a mouse documentation line and provides a scrollable history. The available commands include Show Directory, Show File, Compile File, Load File, and so forth. You may find the CLIM Lisp Listener very useful when debugging small CLIM programs.

Some of the techniques used in the CLIM Lisp Listener include:

- Use of a custom top-level loop that reads both commands and Lisp forms, and uses keystroke accelerators.
- Use of patterns as a prompt.
- Use of some interesting presentation translators.

Note that, by default, ***debug-io*** is not rebound to the CLIM Lisp Listener window, so any error needs to be dealt with in some other window.

You can use the Quit command to exit from the CLIM Lisp Listener back to the demo menu. Under Genera, you can use Select λ (that is, lambda or symbol-shift-L) to create a CLIM Lisp Listener that runs in its own process.

The Graphics Demo

This is a simple application (in SYS:CLIM;REL-2;DEMO;GRAPHICS-DEMOS.LISP) that demonstrates some of the graphics facilities in CLIM. This demo uses color, which appears as stipple patterns on a monochrome system. It also uses transformations.

The CAD Demo

This demo is a very simple CAD system (in SYS:CLIM;REL-2;DEMO;CAD-DEMO.LISP) that lets you build up logic circuits using ''and'' and ''or'' gates. When you run it, click on the [Setup] command menu button, and it will display an example circuit. You can select gates or gate nodes with the mouse; look at the mouse documentation line to see what operations are available. You can also create new components and connect them together.

Some of the techniques used in the CAD Demo include:

- Use of a custom output records that implement the logic components.

- Use of custom highlighting.
- Use of a purely menu-driven command interaction style.
- Use of **clim:drag-output-record**.

Note: feedback circuits will cause the CAD demo program to crash.

The Flight Planning Demo

This example (in SYS:CLIM;REL-2;DEMO;NAVFUN.LISP) sets up an application frame with a map of most of the airports in eastern Massachusetts. The airports and visual references are selectable for various activities such as querying distance between airports and building up a flight plan. This application demonstrates the power of inheritance of presentation types for accepting objects.

Some of the techniques used in the Flight Planner include:

- Use of a number presentation types, using specialization and inheritance to control user interface behavior.
- Use of custom highlighting.
- Use of dialogs.
- Use of table formatting.

The 15 Puzzle

This example (in SYS:CLIM;REL-2;DEMO;PUZZLE.LISP) is an implementation of the 15 puzzle described in the CLIM Tutorial.

The 15 Puzzle uses a simple direct-manipulation interaction style, and uses incremental redisplay in conjunction with table formatting.

The Address Book Demo

This example (in SYS:CLIM;REL-2;DEMO;ADDRESS-BOOK.LISP) is an implementation of a simple address book. You can look up, add, or remove addresses, and edit existing entries.

The Thinkadot Demo

This example (in SYS:CLIM;REL-2;DEMO;THINKADOT.LISP) is an implementation of the mechanical ‘‘Thinkadot’’ game.

The Plotting Demo

This example (in SYS:CLIM;REL-2;DEMO;PLOT.LISP) is a simple data plotting application. There are a number of interesting techniques used in the application:

- Using **clim:tracking-pointer** to select a region on a window, including modification of the pointer cursor.
- Sophisticated use of formatted output facilities.

- Use of a modeless dialog pane.
- Use of a presentation action to implement ‘‘drag scrolling’’.

The Color Chooser

This example (in SYS:CLIM;REL-2;DEMO;COLOR-EDITOR.LISP) uses slider gadgets to implement a color chooser. You can drag a slider in either the RGB or IHS panes to change the color of the color swatch. The main techniques used in this program are:

- A custom pane to implement the color swatch.
- An ‘‘exit’’ push button.
- Two sets of ‘‘linked’’ sliders that track each other.

The Boxes-and-Wires Editor

The boxes-and-wires editor (in SYS:CLIM;REL-2;DEMO;GRAPHICS-EDITOR.LISP) is a program that allows you to create diagrams consisting of boxes optionally connected by wires. As you drag a box around, the wires track the boxes. You can also select a box and perform some operations on the selected box.

The interesting techniques in the program are:

- Use of incremental redisplay and ‘‘redisplay ticks’’ that control when redisplay should occur.
- Use of **clim:tracking-pointer**.
- Use of CLIM’s bounding rectangle functions.
- Use of multiple command tables to group related commands.
- Use of a presentation type with a graphical interface to represent line thickness and style.

The Bitmap Editor

The bitmap editor (in SYS:CLIM;REL-2;DEMO;BITMAP-EDITOR.LISP) allows you to create a little ‘‘bitmap’’ (actually, a CLIM pattern). The main interesting thing in this program is the way the grid is built up, the way cells in the grid are modified, and how the grid is connected to the display of the pattern represented by the grid pane.

The Icosahedron Demo

The file SYS:CLIM;REL-2;DEMO;ICO.LISP implements the traditional ‘‘Ico’’ demo that can be found on almost everywhere. You can ‘‘throw’’ or ‘‘catch’’ an icosahedral ‘‘ball’’, which bounces around the display pane.

The CLIM Browser

The CLIM Browser (in SYS:CLIM;REL-2;DEMO;BROWSER.LISP) implements an extensible, graphical browser. To use it, first select the type of thing you want to browse (classes, packages, or whatever), then the primary ''axis'' along which you want browse (subclasses, superclasses, or whatever). Then use the Show Graph command to get an initial display. Once there is an initial display, you can click on nodes in the graph to add, remove, or hide nodes, ''ellipsize'' a node, or expand an ellipsis. You can also get a hardcopy of a graph.

The interesting techniques in this program are:

- Sophisticated use of incremental redisplay in conjunction with graph formatting, and use of ''redisplay ticks'' that control when redisplay should occur.
- Use of multiple command tables to group related commands.

The ''Peek'' Demo

The ''Peek'' Demo (in SYS:CLIM;REL-2;DEMO;PEEK-FRAME.LISP) is a small process status program. It demonstrates another way to use incremental redisplay.

The Custom Output Records Demo

The Customer Output Records Demo (in SYS:CLIM;REL-2;DEMO;CUSTOM-RECORDS.LISP) shows the benefit of using special-purpose output records when you can take advantage of detailed knowledge of your application's data. This program displays the same data set three different ways. The simplest (and slowest) way simply creates a new presentation for each point in the data set and display that. A somewhat better technique is to use each data point as its own presentation; this is possible because each point already has knowledge about where is will be displayed. The best technique (at least for this application) is to create a ''container'' output record that holds all of the data points, and then display that. For this application, the third technique is about three times faster than the first.

Drawing Graphics in CLIM

Concepts of Drawing Graphics in CLIM

Drawing Functions and Options

CLIM offers a set of *drawing functions* for drawing points, lines, polygons, rectangles, ellipses, circles, and text. You can affect the way the geometric objects are drawn by supplying *drawing options* to the drawing functions. The drawing options support clipping, transformation, line style, text style, ink, and other aspects of the graphic to be drawn (see the section "Using CLIM Drawing Options").

In many cases, it is convenient to use **clim:with-drawing-options** to surround several calls to drawing functions, each using the same options. You can override one

or more drawing options in any given call by supplying keywords to the drawing functions themselves. Using **clim:with-drawing-options** around several drawing functions that would otherwise use the same options is also more efficient than supplying the drawing options separately to each drawing function.

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). A line that you draw on the drawing plane is infinitely thin. The drawing plane provides an idealized version of the graphics you draw; it has no material existence and cannot be viewed directly.

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

Figure 38 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.

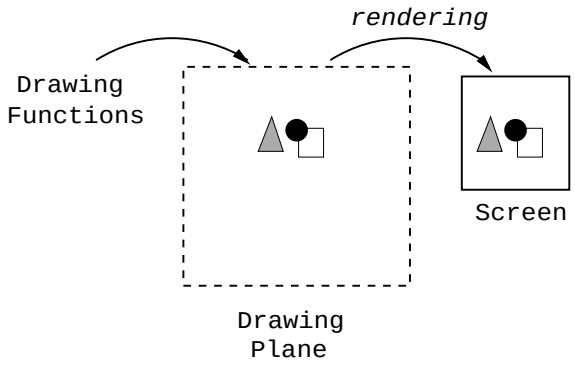


Figure 59. Rendering from Drawing Plane to Window

The distinction between the idealized drawing plane and the real window enables you to develop programs 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 resolution 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. (**Note:** Currently, this feature is not fully implemented.)

Coordinates

When producing graphic output on the drawing plane, you indicate where to place the output 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 39 shows the orientation of the drawing plane, X extends toward the right, and Y extends downward.

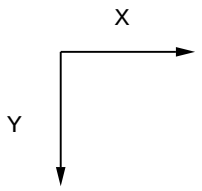


Figure 60. X and Y Axes of Drawing Plane

As the window scrolls downward, the origin of the drawing plane moves above the top edge of the window. Because windows maintain an output history, the Y-axis can extend to a great length. In many cases, it is burdensome to keep track of the coordinates of the drawing plane, and it can be easier to think in terms of a *local coordinate system*.

For example, you might want to draw some business graphics as shown in Figure 40. 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* which informs CLIM how to map from the local coordinate system to the coordinates of the drawing plane. For more information, see **clim:with-room-for-graphics**.

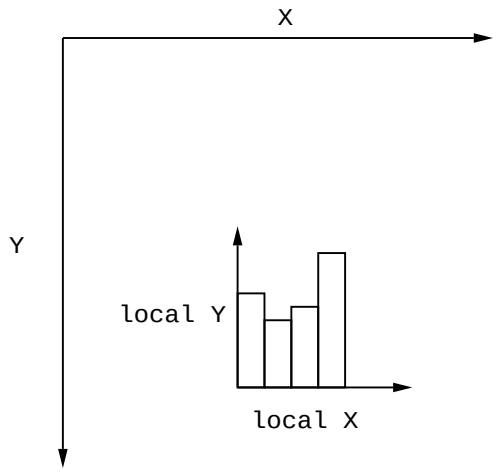


Figure 61. Using a Local Coordinate System

Sheets and Streams, and Mediums

A *sheet* is the most basic window-like object supported by CLIM. It has two primary properties: a region, and a transformation that relates its coordinate system to the coordinate system of its parent. A *stream* is a special kind of sheet that implements the stream protocol; streams include additional state such as the current text cursor position (which is some point in the drawing plane).

A *medium* is an object on which drawing takes place. A medium has as attributes: a drawing plane, the medium's foreground and background, a drawing ink, a transformation, a clipping region, a line style, a text style, and a default text style. Sheets and streams that support output have a medium as one of their attributes.

The drawing functions take a *medium* argument that specifies the destination for output. The drawing functions are specified to be called on mediums, but they can be called on most sheets and streams as well.

The medium keeps track of default drawing options, so if drawing functions are called and some options are unspecified, they default to the values maintained by the medium.

Different medium classes are provided to allow users to draw on different sorts of devices, such as displays and printers.

Examples of Using CLIM Drawing Functions

Figure 41 shows the result of evaluating the following forms:

```
(clim:draw-rectangle* *my-window* 10 10 200 150
                      :filled nil :line-thickness 2)
```

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

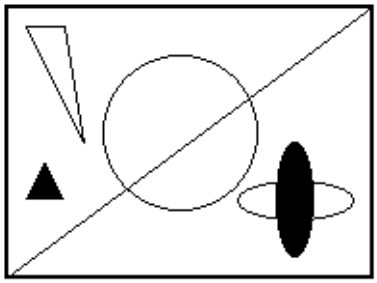


Figure 62. Simple Use of the Drawing Functions

CLIM Drawing Functions

Most of CLIM's drawing functions come in pairs. One function takes two arguments to specify a point by its X and Y coordinates; the corresponding function takes one argument, a point object. The function accepting a point object has a name without an asterisk (*), and the function accepting coordinates of the point has the same name with an asterisk appended to it. For example, **clim:draw-point** accepts a point object, and **clim:draw-point*** accepts coordinates of a point. We expect that using the functions that take spread point arguments and specifying points by their coordinates will be more convenient in most cases. (If you prefer to create and use point objects, see the section "CLIM Point Objects").

The drawing functions take keyword arguments specifying drawing options. For information on the drawing options, see the section "Using CLIM Drawing Options".

The following drawing functions operate on either either mediums, or sheets and streams. When called on an output recording stream that has output recording enabled, these functions all record their output.

clim:draw-point *medium point* &key *:line-style :line-thickness :line-unit :ink :clipping-region :transformation*
Draws a point on *medium* at the position indicated by *point*.

clim:draw-point* *medium x y* &key *:line-style :line-thickness :line-unit :ink :clipping-region :transformation*
Draws a point on *medium* at the position indicated by *x* and *y*.

clim:draw-points *medium point-seq* &key *:line-style :line-thickness :line-unit :ink :clipping-region :transformation*
Draws a set of points on *medium*. *point-seq* is a sequence of point objects specifying where a point is to be drawn.

clim:draw-points* *medium coord-seq* &key *:line-style :line-thickness :line-unit :ink :clipping-region :transformation*
Draws a set of points on *medium*. *coord-seq* is a sequence of pairs of X/Y pairs. Each pair specifies a point to be drawn.

clim:draw-line *medium point-1 point-2* &key *:line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws a line segment on *medium*. The line starts at the position specified by *point-1* and ends at the position specified by *point-2*, two point objects.

clim:draw-line* *medium x1 y1 x2 y2* &key *:line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws a line segment on *medium*. The line starts at the position specified by (*x1*, *y1*), and ends at the position specified by (*x2*, *y2*).

clim:draw-lines *medium point-seq* &key *:line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws a set of disconnected line segments onto *medium*. *point-seq* is a sequence of pairs of points. Each point pair specifies the starting and ending point of a line.

clim:draw-lines* *medium coord-seq* &key *:line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws a set of disconnected line segments onto *medium*. *coord-seq* is a sequence of pairs of X and Y positions. Each pair of X/Y pairs specifies the starting and ending point of a line.

clim:draw-arrow *medium start-point end-point* &key *:from-head (:to-head* **t***) (:head-length* **10***) (:head-width* **5***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws an arrow on *medium*. The arrow starts at the position specified by *start-point* and ends with the arrowhead at the position specified by *end-point*, two point objects.

clim:draw-arrow* *medium x1 y1 x2 y2* &key *:from-head (:to-head* **t***) (:head-length* **10***) (:head-width* **5***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws an arrow on *medium*. The arrow starts at the position specified

by *(x1,y1)* and ends with the arrowhead at the position specified by *(x2,y2)*.

clim:draw-polygon *medium point-seq* &key *(:closed* **t***) (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-joint-shape :line-cap-shape :ink :clipping-region :transformation*
Draws a polygon, or sequence of connected lines, on *medium*. The keyword arguments control whether the polygon is closed (each segment is connected to two other segments) and filled. *point-seq* is a sequence of points that indicate the start of a new line segment.

clim:draw-polygon* *medium coord-seq* &key *(:closed* **t***) (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-joint-shape :line-cap-shape :ink :clipping-region :transformation*
Draws a polygon, or sequence of connected lines, on *medium*. The keyword arguments control whether the polygon is closed (each segment is connected to two other segments) and filled. *coord-seq* is a sequence of alternating X and Y positions that indicate the start of a new line segment.

clim:draw-rectangle *medium point1 point2* &rest *args* &key *:line-style :line-thickness :line-unit :line-dashes :line-joint-shape :ink :clipping-region :transformation (:filled* **t***)*
Draws an axis-aligned rectangle on *medium*. The boundaries of the rectangle are specified by the two points *point1* and *point2*.

clim:draw-rectangle* *medium x1 y1 x2 y2* &key *(:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-joint-shape :ink :clipping-region :transformation*

Draws an axis-aligned rectangle on *medium*. The boundaries of the rectangle are specified by *x1*, *y1*, *x2*, and *y2*.

clim:draw-rectangles *medium point-seq* &rest *args* &key *:line-style :line-thickness :line-unit :line-dashes :line-joint-shape :ink :clipping-region :transformation (:filled* **t***)*
Draws a set of axis-aligned rectangles on *medium*. *point-seq* is a sequence of pairs of points.

clim:draw-rectangles* *medium coord-seq* &rest *args* &key *:line-style :line-thickness :line-unit :line-dashes :line-joint-shape :ink :clipping-region :transformation (:filled* **t***)*
Draws a set of axis-aligned rectangles on *medium*. *coord-seq* is a sequence of 4-tuples *x1*, *y1*, *x2*, and *y2*, with *(x1,y1)* at the upper left and *(x2,y2)* at the lower right in the standard +Y-downward coordinate system.

clim:draw-ellipse *medium point radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws an ellipse or elliptical arc on *medium*. The center of the ellipse is specified by *point*.

clim:draw-ellipse* *medium center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws an ellipse or elliptical arc on *medium*. The center of the ellipse is specified by *center-x* and *center-y*.

clim:draw-circle *medium center radius* &key *:start-angle :end-angle (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws a circle or arc on *medium*. The center of the circle is specified by *center*, and the radius is specified by *radius*.

clim:draw-circle* *medium center-x center-y radius* &key *:start-angle :end-angle (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws a circle or arc on *medium*. The center of the circle is specified by *center-x* and *center-y*, and the radius is specified by *radius*.

clim:draw-oval *medium point x-radius y-radius* &key *(:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws an oval, that is, a ''race-track'' shape, centered on *point*, a point object, with the specified X and Y radii.

clim:draw-oval* *medium center-x center-y x-radius y-radius* &key *(:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation*
Draws an oval, that is, a ''race-track'' shape, centered on (*center-x center-y*), with the specified X and Y radii.

clim:draw-text *medium text point* &key *(:start* **0***) :end (:align-x* **:left***) (:align-y* **:baseline***) :towards-point :text-style :text-family :text-face :text-size :ink :clipping-region :transformation*
Draws *text* onto *medium* starting at the position specified by *point*. *text* can be either a character or a string.

clim:draw-text* *medium text x y* &key *(:start* **0***) :end (:align-x* **:left***) (:align-y* **:baseline***) :towards-x :towards-y :text-style :text-family :text-face :text-size :ink :clipping-region :transformation*
Draws *text* onto *medium* starting at the position specified by *x* and *y*. *text* can be either a character or a string.

clim:draw-design *design stream* &key *:ink :clipping-region :transformation :line-style :unit :thickness :joint-shape :cap-shape :dashes :text-style :text-family :text-face :text-size*
Draws *design* onto *medium*.

clim:draw-pattern* *stream pattern x y* &key *:clipping-region :transformation*
Draws the pattern *pattern* on *stream* at the position (*x*,*y*).

clim:draw-pixmap *medium pixmap point* &rest *args* &key *:ink :clipping-region :transformation (:function* **boole-1***)*

Draws the pixmap *pixmap* on *medium* at the position *point*, creating a ''pixmap output record'' if *medium* is an output recording stream. *:function* is a boolean operation that controls how the source and destination bits are combined.

clim:draw-pixmap* *medium pixmap x y* &rest *args* &key *:ink :clipping-region :transformation (:function* **boole-1***)*
Draws the pixmap *pixmap* on *medium* at the position (*x*,*y*), creating a ''pixmap output record'' if *medium* is an output recording stream. *:function* is a boolean operation that controls how the source and destination bits are combined.

CLIM also provides some drawing functions that operate only on mediums. These functions take no drawing options directly, but instead use the drawing state in the medium. You may want to use these functions when you need higher performance than the ordinary drawing functions provide, but they are somewhat less convenient and do not support output recording.

clim:medium-draw-point* *medium x y*
Draws a point on the medium *medium*. The point is drawn at (*x*,*y*), transformed by the medium's current transformation.

clim:medium-draw-points* *medium position-seq*
Draws a set of points on the medium *medium*. *position-seq* is a sequence of coordinate pairs, which are real numbers. The coordinates in *position-seq* are transformed by the medium's current transformation.

clim:medium-draw-line* *medium x1 y1 x2 y2*
Draws a line on the medium *medium*. The line is drawn from (*x1*,*y1*) to (*x2*,*y2*), with the start and end positions transformed by the medium's current transformation.

clim:medium-draw-lines* *medium position-seq*
Draws a set of disconnected lines on the medium *medium*. *position-seq* is a sequence of coordinate pairs, which are real numbers. The coordinates in *position-seq* are transformed by the medium's current transformation.

clim:medium-draw-polygon* *medium position-seq closed filled*
Draws a polygon or polyline on the medium *medium*. *position-seq* is a sequence of coordinate pairs, which are real numbers. The coordinates in *position-seq* are transformed by the medium's current transformation.

clim:medium-draw-rectangle* *medium x1 y1 x2 y2 filled*
Draws a rectangle on the medium *medium*. The corners of the rectangle are at (*x1*,*y1*) and (*x2*,*y2*), with the corner positions transformed by the medium's current transformation. If *filled* is **t**, the rectangle is filled, otherwise it is not.

clim:medium-draw-rectangles* *medium position-seq filled*
Draws a set of rectangles on the medium *medium*. *position-seq* is a sequence of coordinate pairs, which are real numbers. The coordinates in *position-seq* are transformed by the medium's current transformation. If *filled* is **t**, the rectangles are filled, otherwise they are not.

clim:medium-draw-ellipse* *medium center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy start-angle end-angle filled*
Draws an ellipse on the medium *medium*. The center of the ellipse is at (*x,y*), and the radii are specified by the two vectors (*radius-1-dx,radius-1-dy*) and (*radius-2-dx,radius-2-dy*). The center point and radii are transformed by the medium's current transformation.

clim:medium-draw-text* *medium string-or-char x y start end align-x align-y towards-x towards-y transform-glyphs*
Draws a character or a string on the medium *medium*. The text is drawn starting at (*x,y*), and towards (*toward-x,toward-y*); these positions are transformed by the medium's current transformation.

If you need even higher performance, you can draw directly on the underlying host window system object, but using this technique sacrifices portability. The following functions provide the necessary low-level access to the components of the medium and its owning sheet.

clim:with-medium-state-cached *(medium)* &body *body*
Declares that all of the drawing operations within *body* will use exactly the same drawing options. This allows CLIM back-ends to cache the state of the medium so that the medium does not need to be ''decoded'' for each drawing operation.

clim:medium-sheet *medium*
Returns the sheet with which the medium *medium* is associated.

clim:medium-drawable *medium*
Returns the host window system object (or ''drawable'') that is drawn on by the CLIM drawing functions when they are called on *medium*.

clim:sheet-device-region *sheet*
Returns a region object that describes the region that *sheet* occupies on the display device. The coordinates are in the host's native window coordinate system.

clim:sheet-device-transformation *sheet*
Returns a transformation that converts coordinates in *sheet*'s coordinate system into native coordinates on the display device.

For example, the following code might be used by a CLX-based port of CLIM to draw lines (where **convert-to-device-coordinates** transforms and ''fixes'' its coordinates, and **decode-ink-and-region** ''decodes'' the ink and region into a X Windows gcontext).

```
(defmethod medium-draw-line* ((medium clx-medium) x1 y1 x2 y2)
  (let* ((sheet (clim:medium-sheet medium))
         (transform (clim:sheet-device-transformation sheet))
         (region (clim:sheet-device-region sheet))
         (ink (clim:medium-ink medium))
         (line-style (clim:medium-line-style medium))
         (drawable (clim:medium-drawable medium)))
    (convert-to-device-coordinates transform x1 y1 x2 y2)
    (xlib:draw-line drawable
                    (decode-ink-and-region ink region)
                    x1 y1 x2 y2)))
```

General Geometric Objects in CLIM

Regions in CLIM

A *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 *bounded region* is 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*.

An *unbounded region* either contains no points or contains points that are arbitrarily far apart.

Another way to describe a region is that it maps every *(x,y)* pair into either true or false (meaning member or not a member, respectively, of the region).

The following classes are what CLIM uses to classify the various types of regions. All regions are a subclass of region, and all bounded regions are also a subclass of either **clim:point**, **clim:path**, or **clim:area**.

clim:region
: The protocol class that corresponds to a set of points.

clim:point
: The protocol class that corresponds to a mathematical point.

clim:path
: This is a subclass of **clim:region** that denotes regions that have dimensionality 1. If you want to create a new class that obeys the path protocol, it must be a subclass of **clim:path**.

clim:area
: This is a subclass of **clim:region** that denotes regions that have dimensionality 2 (that is, have area). If you want to create a new class that obeys the area protocol, it must be a subclass of **clim:area**.

These two constants represent the regions that correspond, respectively, to all of the points on the drawing plane and none of the points on the drawing plane.

clim:+everywhere+
: The region that includes all the points on the infinite drawing plane.

clim:+nowhere+
: The empty region (the opposite of **clim:+everywhere+**).

Region Predicates in CLIM

The following functions can be used to examine certain aspects of regions, such as whether two regions are equal or if they overlap.

clim:region-equal *region1 region2*
: Returns **t** if *region1* and *region2* contain exactly the same set of points, otherwise returns **nil**.

clim:region-contains-region-p *region1 region2*
: Returns **t** if all points in *region2* are members of *region1*, otherwise returns **nil**.

clim:region-contains-position-p *region x y*
: Returns **t** if the point (x,y) is contained in *region*, otherwise returns **nil**. This is a special case of **clim:region-contains-region-p**.

clim:region-intersects-region-p *region1 region2*
: Returns **nil** if **clim:region-intersection** of the two regions would be **clim:+nowhere+**, otherwise returns **t**.

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 **clim:+nowhere+**. (For instance, the intersection of two lines that cross each other at a point is empty.) 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, **clim: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*, which is an object that represents one or more region objects related by some region operation, usually a union.

clim:region-union *region1 region2*
: Returns a region that contains all points that are in either *region1* or *region2* (possibly with some points removed to satisfy the dimensionality

rule). The result of **clim:region-union** always has dimensionality that is the maximum dimensionality of *region1* and *region2*.

clim:region-intersection *region1 region2*
Returns a region that contains all points that are in both *region1* and *region2* (possibly with some points removed to satisfy the dimensionality rule). The result of **clim:region-intersection** has dimensionality that is the minimum dimensionality of *region1* and *region2*, or is **clim:+nowhere+**.

clim:region-difference *region1 region2*
Returns a region that contains all points in *region1* that are not in *region2* (plus additional boundary points to make the result closed). The result of **clim:region-difference** has the same dimensionality as *region1*, or is **clim:+nowhere+**.

clim:region-set
The class that represents region sets; a subclass of *region*.

clim:region-set-function *region*
Returns a symbol representing the operation that relates the regions in *region*. This will be one of the symbols **union**, **intersection**, or **set-difference**.

clim:region-set-regions *region* &key *:normalize*
Returns a sequence of the regions in *region*. *Region* can be either a *region-set* or any member of **region**, in which case the result is simply a sequence of one element: *region*.

clim:map-over-region-set-regions *function region* &key *:normalize*
Call *function* on each region in *region*. This is often more efficient than calling **clim:region-set-regions**.

CLIM Point Objects

A *point* is a mathematical point in the drawing plane, which 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 **clim:draw-point** use **clim:draw-point***, which takes two arguments that specify the point by its coordinates. (We generally recommend the use of the spread versions.)

The operations for creating and dealing with points are:

clim:point
The protocol class that corresponds to a mathematical point.

clim:standard-point
: The standard class CLIM uses to implement points. This is the class that **clim:make-point** instantiates.

clim:make-point *x y*
: Creates and returns a point object whose coordinates are *x* and *y*.

clim:point-position *point*
: Returns two values, the X and Y coordinates of *point*.

clim:point-x *point*
: Returns the X coordinate of *point*.

clim:point-y *point*
: Returns the Y coordinate of *point*.

Other Region Types in CLIM

The other types of regions are polylines, polygons, elliptical arcs, and ellipses. All of these region types are closed under affine transformations.

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 a single 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*; you should not confuse this use of the term ''closed'' 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.

The classes that correspond to polylines and polygons are:

clim:polyline
: The protocol class that corresponds to a polyline. This is a subclass of **clim:path**.

clim:polygon
: The protocol class that corresponds to a mathematical polygon. This is a subclass of **clim:area**.

clim:standard-polyline
: The standard class CLIM uses to implement polylines. This is a subclass of **clim:polyline**. This is the class that **clim:make-polyline** and **clim:make-polyline*** instantiate.

clim:standard-polygon
: The standard class CLIM uses to implement polygons. This is a subclass

of **clim:polygon**. This is the class that **clim:make-polygon** and **clim:make-polygon*** instantiate.

Constructors for CLIM Polygons and Polylines

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

clim:make-polygon *point-seq*
Makes an object of class **clim:standard-polygon** consisting of the area contained in the boundary that is specified by the segments connecting each of the points in *point-seq*.

clim:make-polygon* *coord-seq*
Makes an object of class **clim:standard-polygon** consisting of the area contained in the boundary that is specified by the segments connecting each of the points represented by the coordinate pairs in *coord-seq*.

clim:make-polyline *point-seq* &key *:closed*
Makes an object of class **clim:standard-polyline** consisting of the segments connecting each of the points in *point-seq*.

clim:make-polyline* *coord-seq* &key *:closed*
Makes an object of class **clim:standard-polyline** consisting of the segments connecting each of the points represented by the coordinate pairs in *coord-seq*.

Accessors for CLIM Polygons and Polylines

The following functions can be used to access the components of polygons and polylines:

clim:polyline-closed *polyline*
Returns **t** if *polyline* is closed, otherwise returns **nil**.

clim:polygon-points *polygon*
Returns a sequence of points that specify the segments in *polygon*.

clim:map-over-polygon-coordinates *function polygon*
Applies *function* to all of the coordinates of the vertices of *polygon*. The *function* takes two arguments, the X and Y coordinates.

clim:map-over-polygon-segments *function polygon*
Applies *function* to the line segments that compose *polygon*. The *function* takes four arguments, the X and Y coordinates of the start of the line segment, and the X and Y coordinates of the end of the line segment.

Lines in CLIM

A line is a special case of a polyline that has only a single segment. The functions for making and dealing with line are the following:

clim:line The protocol class that corresponds to a mathematical line-segment, that is, a polyline with only a single segment. This is a subclass of **clim:polyline**.

clim:standard-line
The standard class CLIM uses to implement lines. This is a subclass of **clim:line**. This is the class that **clim:make-line** and **clim:make-line*** instantiate.

clim:make-line *start-point end-point*
Makes an object of class **clim:standard-line** that connects *start-point* to *end-point*.

clim:make-line* *start-x start-y end-x end-y*
Makes an object of class **clim:standard-line** that connects (*start-x*, *start-y*) to (*end-x, end-y*).

clim:line-start-point *line*
Returns the starting point of *line*.

clim:line-end-point *line*
Returns the ending point of *line*.

clim:line-start-point* *line*
Returns the starting point of *line* as two values representing the coordinate pair.

clim:line-end-point* *line*
Returns the ending point of *line* as two values representing the coordinate pair.

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. CLIM uses rectangles extensively for various purposes, particularly in optimizations related to output recording.

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

clim:rectangle
The protocol class that corresponds to an axis-aligned mathematical rectangle, that is, rectangular polygons whose sides are parallel to the coordinate axes. This is a subclass of **clim:polygon**.

clim:standard-rectangle
The standard class CLIM uses to implement rectangles. This is a subclass of **clim:rectangle**. This is the class that **clim:make-rectangle** and **clim:make-rectangle*** instantiate.

clim:make-rectangle *min-point max-point*
Makes an object of class **clim:standard-rectangle** whose edges are parallel to the coordinate axes.

clim:make-rectangle* *min-x min-y max-x max-y*
Makes an object of class **clim:standard-rectangle** whose edges are parallel to the coordinate axes.

clim:rectangle-min-point *rectangle*
Returns the minimum point of *rectangle*. The position of a rectangle is specified by its minimum point.

clim:rectangle-max-point *rectangle*
Returns the maximum point of *rectangle*. (The position of a rectangle is specified by its minimum point).

clim:rectangle-edges* *rectangle*
Returns the coordinate of the minimum X and Y and maximum X and Y of *rectangle* as four values.

clim:rectangle-min-x *rectangle*
Returns the coordinate of the minimum X of *rectangle*.

clim:rectangle-min-y *rectangle*
Returns the coordinate of the minimum Y of *rectangle*.

clim:rectangle-max-x *rectangle*
Returns the coordinate of the maximum X of *rectangle*.

clim:rectangle-max-y *rectangle*
Returns the coordinate of the maximum Y of *rectangle*.

clim:rectangle-width *rectangle*
Returns the width of *rectangle*. The width of a rectangle is the difference between the maximum X and the minimum X.

clim:rectangle-height *rectangle*
Returns the height of *rectangle*. The height is the difference between the maximum Y and the minimum Y.

clim:rectangle-size *rectangle*
Returns two values, the width and the height of *rectangle*.

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.

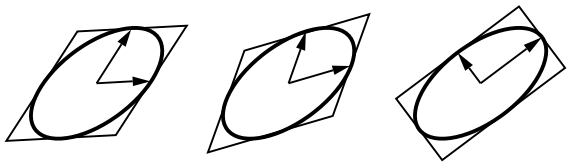
An ellipse is specified in a manner that is easy to transform, and treats all ellipses on an equal basis. An ellipse is specified by its center point and two vectors that describe a bounding parallelogram of the ellipse. The bounding parallelogram is

made by adding and subtracting the vectors from the center point in the following manner:

	x coordinate	y coordinate
Center of Ellipse	x_c	y_c
Vectors	dx_1 dx_2	dy_1 dy_2
Corners of Paralellogram	$x_c + dx_1 + dx_2$ $x_c + dx_1 - dx_2$ $x_c - dx_1 - dx_2$ $x_c - dx_1 + dx_2$	$y_c + dy_1 + dy_2$ $y_c + dy_1 - dy_2$ $y_c - dy_1 - dy_2$ $y_c - dy_1 + dy_2$

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:



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.

clim:ellipse *Class*

The protocol class that corresponds to a mathematical ellipse. This is a subclass of **clim:area**. If you want to create a new class that obeys the ellipse protocol, it must be a subclass of **clim:ellipse**.

clim:elliptical-arc *Class*

The protocol class that corresponds to a mathematical elliptical arc. This is a subclass of **clim:path**. If you want to create a new class that obeys the elliptical arc protocol, it must be a subclass of **clim:elliptical-arc**.

clim:standard-ellipse *Class*

The standard class CLIM uses to implement an ellipse. This is a subclass of **clim:ellipse**. This is the class that **clim:make-ellipse** and **clim:make-ellipse*** instantiate.

clim:standard-elliptical-arc *Class*

The standard class CLIM uses to implement an elliptical arc. This is a subclass of **clim:elliptical-arc**. This is the class that **clim:make-elliptical-arc** and **clim:make-elliptical-arc*** instantiate.

Constructor Functions for Ellipses and Elliptical Arcs in CLIM

clim:make-ellipse *center-point radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle*
Makes an object of class **clim:standard-ellipse**. The center of the ellipse is *center-point*.

clim:make-ellipse* *center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle*
Makes an object of class **clim:standard-ellipse**. The center of the ellipse is (*center-x*, *center-y*).

clim:make-elliptical-arc *center-point radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle*
Makes an object of class **clim:standard-elliptical-arc**. The center of the ellipse is *center-point*.

clim:make-elliptical-arc* *center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle*
Makes an object of class **clim:standard-elliptical-arc**. The center of the ellipse is (*center-x*, *center-y*).

Accessors for CLIM Elliptical Objects

The following accessor functions apply to both ellipses and elliptical arcs. In all cases, the name *ellipse* means that the argument is an ellipse or an elliptical arc.

clim:ellipse-center-point *ellipse*
Returns the center point of *ellipse*.

clim:ellipse-center-point* *ellipse*
Returns the center point of *ellipse* as two values representing the coordinate pair.

clim:ellipse-radii *ellipse*
Returns four values corresponding to the two radius vectors of *ellipse*.

clim:ellipse-start-angle *ellipse*
Returns the start angle of *ellipse*.

clim:ellipse-end-angle *ellipse*
Returns the end angle of *ellipse*.

Bounding Rectangles

Every bounded region in CLIM has a derived *bounding rectangle*, which is the smallest rectangle that contains every point in the region, and may contain additional points as well. Unbounded regions do not have any bounding rectangle. For example, all sheets and output records have bounding rectangles whose coordinates are relative to the bounding rectangle of the parent of the sheet or output record. See the section "Bounding Rectangles in CLIM".

The following functions can be used to access the bounding rectangle of a region.

clim:bounding-rectangle* *region*
Returns the bounding rectangle of *region* as four real numbers that specify the left, top, right, and bottom edges of the bounding rectangle.

clim:bounding-rectangle *region*
Returns a new bounding rectangle for *region* as a **clim:standard-bounding-rectangle** object.

clim:bounding-rectangle-left *region*
Returns the coordinate of the left edge of the bounding rectangle of *region*.

clim:bounding-rectangle-top *region*
Returns the coordinate of the top edge of the bounding rectangle of *region*.

clim:bounding-rectangle-right *region*
Returns the coordinate of the right edge of the bounding rectangle of *region*.

clim:bounding-rectangle-bottom *region*
Returns the coordinate of the bottom edge of the bounding rectangle of *region*.

clim:bounding-rectangle-position *region*
Returns the position of the bounding rectangle of *region* as two values, the left and top coordinates of the bounding rectangle.

clim:bounding-rectangle-set-position *region x y*
Changes the position of the bounding rectangle of *region* to the new position *x* and *y*.

clim:bounding-rectangle-size *region*
Returns the size (as two values, width and height) of the bounding rectangle of *region*.

clim:bounding-rectangle-width *region*
Returns the width of the bounding rectangle of *region*.

clim:bounding-rectangle-height *region*
Returns the height of the bounding rectangle of *region*.

For example, the size of a the output generated by *body* can be determined by calling **clim:bounding-rectangle-size** on the output record:

```
(let ((record (clim:with-output-to-output-record (s) body)))
  (multiple-value-bind (width height)
      (clim:bounding-rectangle-size record)
    (format t "~&Width is ~D, height is ~D" width height)))
```

Pixmaps in CLIM

A *pixmap* can be thought of as an ''off-screen window'', that is, a medium that can be used for graphical output, but 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 **clim:draw-pixmap** or **clim:copy-from-pixmap** to draw the part as needed.

The exact representation of a pixmap is explicitly unspecified. Some mediums may not support pixmaps (such as PostScript mediums); in this case, CLIM will signal an error.

Note that there is no interaction between most of the pixmap copying operations and output recording. That is, copying a pixmap onto an output recording is a pure drawing operation that affects only the display, not the output history.

The following functions are provided for managing pixmaps:

clim:copy-from-pixmap *pixmap pixmap-x pixmap-y width height medium medium-x medium-y* &optional *(op* **boole-1***)*
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*). *op* is a boolean operation that controls how the source and destination bits are combined.

clim:copy-to-pixmap *medium medium-x medium-y width height* &optional *pixmap (pixmap-x* **0***) (pixmap-y* **0***) (op* **boole-1***)*
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*). *op* is a boolean operation that controls how the source and destination bits are combined.

clim:copy-area *medium from-x from-y width height to-x to-y* &optional *(op* **boole-1***)*
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. *op* is a boolean operation that controls how the source and destination bits are combined.

clim:allocate-pixmap *medium width height*
Allocates and returns a pixmap object that can be used on any medium that shares the same characteristics as *medium*.

clim:deallocate-pixmap *pixmap*
Deallocates the pixmap *pixmap*.

clim:with-output-to-pixmap *(medium-var medium* &key *:width :height)* &body *body*

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.

clim:draw-pixmap *medium pixmap point* &rest *args* &key *:ink :clipping-region :transformation (:function* **boole-1***)*
Draws the pixmap *pixmap* on *medium* at the position *point*, creating a ''pixmap output record'' if *medium* is an output recording stream. *:function* is a boolean operation that controls how the source and destination bits are combined.

clim:draw-pixmap* *medium pixmap x y* &rest *args* &key *:ink :clipping-region :transformation (:function* **boole-1***)*
Draws the pixmap *pixmap* on *medium* at the position (*x*,*y*), creating a ''pixmap output record'' if *medium* is an output recording stream. *:function* is a boolean operation that controls how the source and destination bits are combined.

Try the following example, which creates a pixmap and then draws it on a stream wherever you click the pointer.

```
(defun test-pixmaps (&optional (function boole-1) (stream *standard-output*))
  (let* ((medium (clim:sheet-medium stream))
         (pixmap
           (clim:with-output-to-pixmap (mv medium)
             (clim:draw-circle* mv 50 50 20
                                :filled t :ink (make-gray-color 1/2))
             (clim:draw-rectangle* mv 0 0 100 100
                                   :filled nil))))
    (block get-position
      (loop
        (clim:tracking-pointer (stream)
          (:pointer-button-press (x y)
           (clim:draw-pixmap* stream pixmap x y :function function))
          (:key-press ()
           (return-from get-position)))))
    pixmap))
```

The CLIM Drawing Environment

There are a number of factors that affect drawing in CLIM. Drawing is affected by transformations, style options, clipping, and by the ink which is used. All of these are controlled by the drawing environment.

When you draw in CLIM, you do so on a medium. A medium can be thought of as a drawing surface. The medium also keeps track of its drawing environment, the current transformation, text style, foreground and background inks, etc.

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 function **draw-stuff** (as well as any drawing performed by anything it calls) below will be affected by the scaling transformation.

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

The drawing environment is controlled through the use of drawing options.

Components of CLIM Mediums

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 for reading and writing the values of these components. Also, these components are temporarily bound within a dynamic context by using **clim:with-drawing-options**, **clim:with-text-style**, and related forms. Using **setf** of a component while it is temporarily bound takes effect immediately, but is undone when the dynamic context is exited. For convenience, these accessors generally work on sheets and streams as well as on mediums.

clim:medium-foreground *medium*
: Returns the current foreground design of the *medium*. You can use **setf** on **clim:medium-foreground** to change the foreground design.

clim:medium-background *medium*
: Returns the current background design of the *medium*. You can use **setf** on **clim:medium-background** to change the background design.

clim:medium-ink *medium*
: Returns the current drawing ink of the *medium*. You can use **setf** on **clim:medium-ink** to change the current ink.

clim:medium-transformation *medium*
: Returns the current transformation of the *medium*. You can use **setf** on **clim:medium-transformation** to change the current transformation.

clim:medium-clipping-region *medium*
: Returns the current clipping region of the *medium*. You can use **setf** on **clim:medium-clipping-region** to change the clipping region.

clim:medium-line-style *medium*
Returns the current line style of the *medium*. You can use **setf** on **clim:medium-line-style** to change the line style.

clim:medium-text-style *medium*
Returns the current text style of the *medium*. You can use **setf** on **clim:medium-text-style** to change the current text style.

clim:medium-default-text-style *medium*
The default text style for *medium*. You can use **setf** on **clim:medium-default-text-style** to change the default text style, but the text style must be a fully specified text style.

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 section "Components of CLIM Mediums".
- You can use **clim:with-drawing-options** and **clim:with-text-style** to temporarily bind the drawing options of the medium. In many cases, it is convenient to use **clim: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 **clim: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 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 **clim:with-drawing-options** or a drawing function, normally you supply either the **:text-style** option or a text style suboption (or more than one suboption), but you would not supply 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.

clim:with-drawing-options *(medium* &key *:ink :clipping-region :transformation :line-style :line-unit :line-thickness :line-dashes :line-joint-shape :line-cap-shape :text-style :text-family :text-face :text-size)*
Binds the state of *medium* to correspond to the supplied drawing *options*,

and evaluates the *body* with the new drawing options in effect. Each option causes binding of the corresponding component of the medium for the dynamic extent of the body.

Set of CLIM Drawing Options

The drawing options can be any of the following, plus any of the suboptions for line styles and text styles.

:clipping-region *Clim Drawing Option*

Specifies the region of the drawing plane on which the drawing functions can draw.

The clipping region must be an **clim:area**; furthermore, an error might be signalled if the clipping region is not a rectangle or a **clim:region-set** composed of rectangles. Drawing 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 **clim:+everywhere+**, which means that no clipping occurs in the drawing plane, only in the viewport.

The **:clipping-region** drawing option temporarily changes the value of **clim:medium-clipping-region** to **clim:region-intersection** of the argument and the previous value. If both a clipping region and a transformation are supplied in the same set of drawing options, the clipping region is transformed by the newly composed transformation.

:ink *Clim Drawing Option*

A design used as the ink for drawing operations. The drawing functions draw with the color and pattern specified by the **:ink** option, which can have any of the following values:

- **clim:+foreground-ink+**, **clim:+background-ink+**, or **clim:+flipping-ink+**.
- A color (created by **clim:make-rgb-color** or **clim:find-named-color**, for example).
- An opacity (including **clim:+transparent-ink+**).
- A more general design, such as a pattern (created by **clim:make-pattern**) or a tile (created by **clim:make-rectangular-tile**).

The default value is **clim:+foreground-ink+**.

The **:ink** drawing option temporarily changes the value of **clim:medium-ink** and replaces the previous ink; the new and old inks are not combined in any way.

For more information on how to use the **:ink** drawing option, see the section "Drawing in Color in CLIM".

:transformation *Clim Drawing Option*

Transforms the coordinates used as arguments to drawing functions to the coordinate system of the drawing plane. The default value is **clim:+identity-transformation+**.

The **:transformation** drawing option temporarily changes the value of **clim:medium-transformation** to **clim:compose-transformations** of the argument and the previous value.

:text-style *Clim Drawing Option*

Controls how text is displayed, both for the graphic drawing functions and ordinary stream output. The value of the **:text-style** option is a text style object.

This drawing option temporarily changes the value of **clim:medium-text-style** to the result of merging the value of **:text-style** with the prevailing text style.

If text style suboptions are also specified, they temporarily change the value of **clim:medium-text-style** to the result of merging the specified suboptions with the **:text-style** drawing options, which is then merged with the previous value of **clim:medium-text-style**.

See the section "CLIM Text Style Suboptions".

:line-style *Clim Drawing Option*

Controls how lines and arcs are drawn. The value of the **:line-style** option is a line style object.

This drawing option temporarily changes the value of **clim:medium-line-style**.

See the section "CLIM Line Style Suboptions".

For the set of line and text style options, see the section "CLIM Line Style Suboptions" , and see the section "CLIM Text Style Suboptions".

Using the :filled Option to Certain CLIM Drawing Functions

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 strokes the outline of 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 **clim:with-drawing-options**.

These are functions that have a **:filled** keyword argument:

clim:draw-rectangle
clim:draw-rectangle*
clim:draw-rectangles

clim:draw-rectangles*
clim:draw-polygon
clim:draw-polygon*
clim:draw-circle
clim:draw-circle*
clim:draw-ellipse
clim:draw-ellipse*

Line Styles in CLIM

A line or other path is a one-dimensional object. In order to be visible, the rendering of a line must, however, occupy some non-zero area on the display hardware. A *line style* object is used to represent the advice that CLIM supplies to the rendering substrate on how to perform the rendering.

CLIM Line Style Objects

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.

The class of a line style object is **clim:line-style**. You create a line style object with **clim:make-line-style**.

clim:make-line-style &key *(:unit* **:normal***) (:thickness* **1***) :dashes (:joint-shape* **:miter***) (:cap-shape* **:butt***)*
Creates a line style object with the supplied characteristics.

The following readers are provided for the components of line styles:

clim:line-style-thickness *line-style*
Returns the thickness component of a line style object, which is an integer.

clim:line-style-dashes *line-style*
Returns the dashes component of a line style object.

clim:line-style-joint-shape *line-style*
Returns the joint shape component of a line style object.

clim:line-style-cap-shape *line-style*
Returns the cap shape component of a line style object.

clim:line-style-unit *line-style*
Returns the unit component of a line style object, which will be one of **:normal** or **:point**.

CLIM Line Style Suboptions

Each line style suboption has a reader function which returns the value of that component from a line style object.

The line style suboptions are listed as follows:

:line-thickness *Clim Drawing Option*

The thickness (an integer in the units described by **clim:line-style-unit**) of the lines or arcs drawn by a drawing function. The default is 1, which combined with the default unit of **:normal**, means that the default line drawn is the ''comfortably visible thin line''.

You can call **clim:line-style-thickness** on a line style object to get the value of the **:line-thickness**, or **:thickness** component.

:line-dashes *Clim Drawing Option*

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 substrate. This allows the underlying display substrate to provide a default dashed line for the user whose only requirement is to draw a line that is visually distinguished from the default solid line. Using the default dashed line can be more efficient than specifying customized dashes.

sequence Specifies a sequence of integers, 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 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 **clim:line-style-unit**. You can use **clim:make-contrasting-dash-patterns** to create a a sequence for the **:dashes** option.

See the function **clim:make-contrasting-dash-patterns**.

You can call **clim:line-style-dashes** on a line style object to get the value of the **:line-dashes**, or **:dashes** component.

:line-joint-shape *Clim Drawing Option*

Specifies the shape of joints between line segments of closed, unfilled figures, when the **:line-thickness** or **:thickness** option to a drawing function is greater than 1. 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.

You can call **clim:line-style-joint-shape** on a line style object to get the value of the **:line-joint-shape**, or **:joint-shape** component.

:line-cap-shape *Clim Drawing Option*

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.

You can call **clim:line-style-cap-shape** on a line style object to get the value of the **:line-cap-shape**, or **:cap-shape** component.

:line-unit *Clim Drawing Option*

The units in which the thickness, dash pattern, and dash phase are measured. Possible values are **:normal** and **:point**, described as follows:

:normal A relative measure in terms of the usual or ''normal'' line thickness. The normal line thickness is the thickness of the ''comfortably visible thin line'', which is a property of the underlying rendering substrate. This is the default.

:point An absolute measure in terms of printer's points (approximately 1/72 of an inch).

You can call **clim:line-style-unit** on a line style object to get the value of the **:line-unit** or **:unit** component.

This function can be used to generate a value for the **:dashes** line style suboption.

clim:make-contrasting-dash-patterns *n* &optional *k*

Makes a simple vector of *n* dash patterns with recognizably different appearances. If *k* (an integer between 0 and *n*-1) is supplied, **clim:make-contrasting-dash-patterns** returns the *k*'th dash pattern. If the implementation does not have *n* different contrasting dash patterns, **clim:make-contrasting-dash-patterns** signals an error.

clim:contrasting-dash-patterns-limit *port*

Returns the number of contrasting dash patterns that the port *port* can generate.

Text Styles in CLIM

CLIM's model for the appearance of text follows the same principle as the model for creating formatted output. This principle holds 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.

Concepts of CLIM Text Styles

In CLIM, 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.

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:

clim:*default-text-style*
The default text style used by all streams.

The sizes **:smaller** and **:larger** are treated specially in that they are merged with the default text style size to result in a size that is discernably 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**.

Some systems include color in their notion of a text style. This is not the case in CLIM. If you want to change the color of textual output, use the **:ink** option to **clim:draw-text***, or if you are using functions like **write-string** or **format**, use the **:ink** option to **clim:with-drawing-options**.

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, is neither **:smaller** nor **:larger**).

When text is displayed 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 media. 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.

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 sub-options.

For example, you might want to have a completely different family, face and size for menus. You could make a text style object and make it be the value of ***menu-text-style***.

You create text style objects using **clim:make-text-style**.

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

     Here is a text-style example.
=>
```

In the current implementation of CLIM, text style objects are interned. That is, two different invocations of **clim: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.

CLIM Text Style Suboptions

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 text style suboptions are:

:text-family *Clim Drawing Option*

Specifies the family of the text style. The reader function is **clim:text-style-family**.

:text-face *Clim Drawing Option*

Specifies the face of the text style. The reader function is **clim:text-style-face**.

:text-size *Clim Drawing Option*

Specifies the size of the text style. The reader function is **clim:text-style-size**.

CLIM Text Style Functions

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

clim:parse-text-style *text-style*
Returns the text style object representing the text style described by *text-style*.

clim:merge-text-styles *style1 style2*
Merges *style1* against the defaults provided by *style2*.

clim:text-style-components *text-style medium*
Returns the components of *text-style* as three values (family, face, and size).

clim:text-style-family *text-style*
Returns the family component of the *text-style*.

clim:text-style-face *text-style*
Returns the face component of the *text-style*.

clim:text-style-size *text-style*
Returns the size component of the *text-style*.

clim:text-style-ascent *text-style medium*
The ascent (a real number) of *text-style* as it would be rendered on *medium*.

clim:text-style-descent *text-style medium*
The descent (a real number) of *text-style* as it would be rendered on *medium*.

clim:text-style-height *text-style medium*
Returns the height (a real number) of the ''usual character'' in *text-style* on *medium*.

clim:text-style-width *text-style medium*
Returns the width (a real number) of the ''usual character'' in *text-style* on *medium*.

clim:text-style-fixed-width-p *text-style medium*
Returns **t** if *text-style* will map to a fixed-width font on *medium*, otherwise returns **nil**.

clim:text-style-mapping *port style* &optional *character-set*
Returns the font object that will be used if characters in *character-set* in the text style *style* are drawn on any medium on the port *port*.

clim:text-style-mapping-exists-p *port style* &optional *character-set exact-size-required*

Returns **t** if there is a font associated with the text style *style* on the port *port*, otherwise returns **nil**.

clim:make-text-style *family face size*
Creates a text style object with the given family, face, and style.

The following forms can be used to change the current text style for a stream by merging the specified style with the stream's current text style. They are intended as abbreviations to be used instead of **clim:with-drawing-options**.

clim:with-text-style *(medium style)* &body *body*
Binds the current text style of *medium* to correspond to the new text style, within the *body*. *style* is a text style object.

clim:with-text-face *(medium face)* &body *body*
Binds the current text face of *medium* to correspond to the new text face *face*, within the *body*.

clim:with-text-family *(medium family)* &body *body*
Binds the current text family of *medium* to correspond to the new text family *family*, within the *body*.

clim:with-text-size *(medium size)* &body *body*
Binds the current text size of *medium* to correspond to the new text size *size*, within the *body*.

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 or medium. 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 Figure42.

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

```
(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))
```

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:

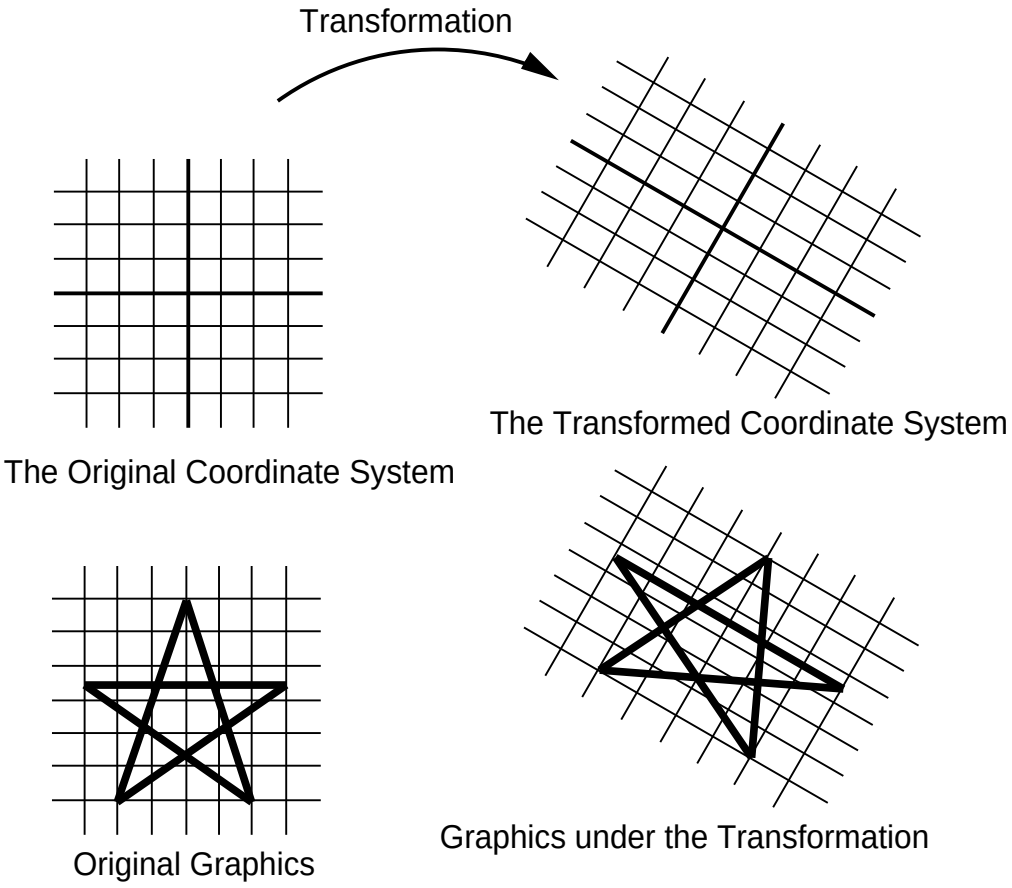


Figure 63. Graphic Transformation

```
(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 the lower half of Figure 63 on *stream*.

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 length, angle, and orientation of all geometric entities.

A rotation is a transformation that preserves 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, and 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 transforming a region creates a new region; it does not side-effect the *region* argument.

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

CLIM Transformation Constructors

The following functions can be used to create a transformation object that can be used, for instance, in a call to **clim:compose-transformations**.

clim:make-translation-transformation *delta-x delta-y*

> Makes a transformation that translates all points by *delta-x* in the X direction and *delta-y* in the Y direction.

clim:make-rotation-transformation *angle* &optional *origin*

Makes a transformation that rotates all points clockwise by *angle* around the point *origin*. The angle is specified in radians. If *origin* is supplied it must be a point; if not supplied it defaults to (0,0).

clim:make-rotation-transformation* *angle origin-x origin-y*

Makes a transformation that rotates all points clockwise by *angle* around the point, (*origin-x, origin-y*). The angle is specified in radians.

clim:make-scaling-transformation *mx my* &optional *origin*

Makes a transformation that multiplies the X-coordinate distance of every point from *origin* by *mx* and the Y-coordinate distance of every point from *origin* by *my*. If *origin* is supplied it must be a point; if not supplied it defaults to (0,0).

clim:make-scaling-transformation* *mx my origin-x origin-y*

Makes a transformation that multiplies the X-coordinate distance of every point from *origin-x* by *mx* and the Y-coordinate distance of every point from *origin-y* by *my*.

clim:make-reflection-transformation *point-1 point-2*

Makes a transformation that reflects every point through the line passing through the points *point-1* and *point-2*.

clim:make-reflection-transformation* *x1 y1 x2 y2*

Makes a transformation that reflects every point through the line passing through the points (*x1*, *y1*) and (*x2*, *y2*).

clim:make-transformation *mxx mxy myx myy tx ty*

Makes a general transformation whose effect is:

$$x' = m_{xx}\, x + m_{xy}\, y + t_x$$

$$y' = m_{yx}\, x + m_{yy}\, y + t_y$$

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.

clim:make-3-point-transformation *point-1 point-2 point-3 point-1-image point-2-image point-3-image*

Makes a transformation that takes *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 the points are collinear, the **clim:transformation-underspecified** condition is signalled.

clim:make-3-point-transformation* *x1 y1 x2 y2 x3 y3 x1-image y1-image x2-image y2-image x3-image y3-image*

Makes a transformation that takes (*x1*, *y1*) into (*x1-image*, *y1-image*), (*x2*, *y2*) into (*x2-image*, *y2-image*) and (*x3*, *y3*) into (*x3-image*, *y3-image*). (Three non-collinear points and their images under the transformation are enough to specify any affine transformation.) If the points are collinear, the **clim:transformation-underspecified** condition is signalled.

CLIM Transformation Protocol

clim:transformation *Class*

The protocol class for all transformations. There are one or more subclasses of **clim:transformation** with implementation-dependent names that implement transformations. If you want to create a new class that obeys the transformation protocol, it must be a subclass of **clim:transformation**.

clim:+identity-transformation+ *Constant*

An instance of a transformation that is guaranteed to be an identity transformation, that is, the transformation that ''does nothing''.

CLIM Transformation Predicates

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

clim:transformation-equal *transform1 transform2*
Returns **t** if the two transformations have equivalent effects (that is, are mathematically equal), otherwise returns **nil**.

clim:identity-transformation-p *transform*
Returns **t** if *transform* is equal (in the sense of **clim:transformation-equal**) to the identity transformation, otherwise returns **nil**.

clim:translation-transformation-p *transform*
Returns **t** if *transform* 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 returns **nil**.

clim:invertible-transformation-p *transform*
Returns **t** if *transform* has an inverse, otherwise returns **nil**.

clim:reflection-transformation-p *transform*
Returns **t** if *transform* inverts the ''handedness'' of the coordinate system, otherwise returns **nil**.

clim:rigid-transformation-p *transform*
Returns **t** if *transform* transforms the coordinate system as a rigid object, that is, as a combination of translations, rotations, and pure reflections. Otherwise, it returns **nil**.

clim:even-scaling-transformation-p *transform*
Returns **t** if *transform* multiplies all X-lengths and Y-lengths by the same magnitude, otherwise returns **nil**. This includes pure reflections through vertical and horizontal lines.

clim:scaling-transformation-p *transform*
Returns **t** if *transform* multiplies all X-lengths by one magnitude and all

Y-lengths by another magnitude, otherwise returns **nil**. This category includes even scalings as a subset.

clim:rectilinear-transformation-p *transform*
Returns **t** if *transformation* will always transform any axis-aligned rectangle into another axis-aligned rectangle, otherwise returns **nil.** This category includes scalings as a subset, and also includes 90 degree rotations.

CLIM Transformation Functions

The following functions can be used to compose transformations. The ''compose with'' functions have exactly the same effect as **clim:compose-transformations**, except that they are more efficient.

clim:compose-transformations *transform1 transform2*
Returns a transformation that is the composition of its arguments. Composition is in right-to-left order, that is the resulting transformation represents the effects of applying *transform2* followed by *transform1.*

clim:compose-translation-with-transformation *transform dx dy*
Creates a new transformation by composing *transform* with a given translation, as specified by *dx* and *dy*. The order of composition is that the translation ''transformation'' is first, followed by *transform*.

clim:compose-rotation-with-transformation *transform angle* &optional *origin*
Creates a new transformation by composing *transform* with a given rotation, as specified by *angle* and *origin*. The order of composition is that the rotation ''transformation'' is first, followed by *transform*.

clim:compose-scaling-with-transformation *transform mx my* &optional *origin*
Creates a new transformation by composing *transform* with a given scaling, as specified by *mx*, *my*, and *origin*. The order of composition is that the scaling ''transformation'' is first, followed by *transform*.

clim:compose-transformation-with-translation *transform dx dy*
Creates a new transformation by composing the translation given by *dx* and *dy* with *transform*. The order of composition is that *transform* is first, followed by the translation ''transformation''.

clim:compose-transformation-with-rotation *transform angle* &optional *origin*
Creates a new transformation by composing the rotation given by *angle* and *origin* with *transform*. The order of composition is that *transform* is first, followed by the rotation ''transformation''.

clim:compose-transformation-with-scaling *transform mx my* &optional *origin*
Creates a new transformation by composing the scaling given by *mx*, *my*, and *origin* with *transform*. The order of composition is that *transform* is first, followed by the scaling ''transformation''.

clim:invert-transformation *transform*
Returns a transformation that is the inverse of *transform*. The result of

composing a transformation with its inverse is the identity transformation. If *transform* is singular, **clim:invert-transformation** signals the **clim:singular-transformation** condition.

The following three forms can be used to compose a transformation into the current transformation of a stream. They are intended as abbreviations for calling **clim:compose-transformations** and **clim:with-drawing-options** directly.

clim:with-rotation *(medium angle* &optional *origin)* &body *body*
Establishes a rotation on *medium* that rotates clockwise by *angle* (in radians), and then evaluates *body* with that transformation in effect. If *origin* is supplied, the rotation is about that point. The default for *origin* is (0,0).

clim:with-translation *(medium dx dy)* &body *body*
Establishes a scaling transformation on *medium* that scales by *dx* in the X direction and *dy* in the Y direction, and then evaluates *body* with that transformation in effect.

clim:with-scaling *(medium sx* &optional *sy)* &body *body*
Establishes a scaling transformation on *medium* that scales by *sx* in the X direction and *sy* in the Y direction, and then evaluates *body* with that transformation in effect. If *sy* is not supplied, it defaults to *sx*.

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

clim:with-room-for-graphics *(*&optional *stream* &key *:height (:first-quadrant* **t***) (:move-cursor* **t***) :record-type)* &body *body*
Binds the dynamic environment to establish a local coordinate system for doing graphics output.

clim:with-local-coordinates *(*&optional *stream x y)* &body *body*
Binds the dynamic environment to establish a local coordinate system with the positive X-axis extending to the right and the positive Y-axis extending downward, with (0,0) at the current cursor position of *stream*.

clim:with-first-quadrant-coordinates *(*&optional *stream x y)* &body *body*
Binds the dynamic environment to establish a local coordinate system with the positive X-axis extending to the right and the positive Y-axis extending upward, with (0,0) at the current cursor position of *stream*.

Applying CLIM Transformations

The following functions can be used to apply a transformation to some sort of a geometric object, such as a region or a distance. Calling **clim:transform-position** or **clim:untransform-position** on a spread points is generally more efficient than calling **clim:transform-region** or **clim:untransform-region** on the unspread point object.

clim:transform-region *transformation region*
Applies *transformation* to *region*, and returns a new transformed region.

clim:untransform-region *transformation region*
Applies the inverse of *transformation* to *region* and returns a new transformed region.

clim:transform-position *transform x y*
Applies *transform* to the point whose coordinates are *x* and *y*, and returns two values, the transformed X-coordinate and the transformed Y-coordinate.

clim:untransform-position *transform x y*
Applies the inverse of *transform* to the point whose coordinates are *x* and *y,* and returns two values, the transformed X-coordinate and the transformed Y-coordinate.

clim:transform-rectangle* *transform x1 y1 x2 y2*
Applies the transformation *transform* to the rectangle specified by the four coordinate arguments, which are real numbers. One corner of the rectangle is at *(x1,y1)* and the opposite corner is at *(x2,y2)*.

clim:untransform-rectangle* *transform x1 y1 x2 y2*
Applies the inverse of *transform* to the rectangle specified by the four coordinate arguments, and returns four values that specify the minimum and maximum points of the transformed rectangle.

clim:transform-distance *transform dx dy*
Applies *transform* to the distance represented by *dx* and *dy*, and returns two values, the transformed *dx* and the transformed *dy*.

clim:untransform-distance *transform dx dy*
Applies the inverse of *transform* to the distance represented by *dx* and *dy*, and returns two values, the transformed *dx* and the transformed *dy*.

clim:translate-coordinates *x-delta y-delta* &body *coordinate-pairs*
Translates each of the X and Y coordinate pairs in *coordinate-pairs* by *x-delta* and *y-delta*.

Drawing in Color in CLIM

Concepts of Drawing in Color in CLIM

To draw in color, you can supply the **:ink** drawing option to CLIM's drawing functions when using streams opened on a color port (see the section "Functions for Operating on Windows Directly")

The drawing functions work by selecting a region of the drawing plane and painting it with color.

The region to be painted is the intersection of the shape specified by the drawing function and the **:clipping-region** drawing option, which is then transformed by the **:transformation** drawing option. 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 letterforms of text.

Use the **:ink** drawing option to specify how to color this region. The value for **:ink** is usually a color, but you can also specify a design for **:ink**. When you use a design for **:ink**, you can control the coloring-in process by specifying a new color of the drawing plane for each ideal point in the shape being drawn. (This can depend on the coordinates of the point, and on the current color at that point in the drawing plane). For more information, see the section "Drawing with Designs in CLIM".

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.

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 **clim:make-rgb-color**, **clim:make-ihs-color**, or **clim:make-gray-color**, or by using one of the predefined colors listed in "Predefined Color Names in CLIM". 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.

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 **clim:make-contrasting-inks** rather than a fixed palette of colors.

The region of the display device that gets colored when rendering a path or text is controlled by the line-style or text-style, respectively.

CLIM Operators for Drawing in Color

clim:make-ihs-color *intensity hue saturation*
Creates a color object with the specified *intensity*, *hue*, and *saturation*.

clim:make-rgb-color *red green blue*
Creates a color object with color components *red*, *green*, and *blue*.

clim:make-gray-color *luminosity*
Creates a color object. *luminosity* is 0 for black, 1 for white, in between for gray.

clim:color-ihs *color*
Returns three values, the *intensity*, *hue*, and *saturation* components of *color*.

clim:color-rgb *color*
Returns three values, the *red*, *green*, and *blue* components of *color*. The values are real numbers between 0 and 1 (inclusive).

clim:make-contrasting-inks *n* &optional *k*
Returns a simple vector of *n* inks with different appearances. If *k* (an integer between 0 and *n*-1) is supplied, **clim:make-contrasting-inks** returns the *k*'th design.

clim:contrasting-inks-limit *port*
Returns the number of contrasting inks that the port *port* can generate.

Predefined Color Names in CLIM

The color corresponding to the color names can be found by calling **clim:find-named-color**, which looks the color name up in an object called a *palette*. A palette is a table that maps color names to color objects. On some platforms, the palette has a bounded size (typically around 256 entries), and serves as a way to allocate a colormap resource.

clim:find-named-color *name palette* &key *:errorp*
Finds the color named *name* in the palette *palette*.

clim:frame-palette *frame*
Returns the palette associated with the application frame *frame*.

clim:frame-manager-palette *frame-manager*
Returns the palette that will be used, by default, by all the frames managed by *frame-manager*, if those frame's don't have a palette of their own.

clim:port-default-palette *port*
Returns the palette associated with the port *port*.

clim:palette-color-p *palette*

Returns **t** if the palette supports color, otherwise returns **nil**.

The following table lists the basic set of named colors in the default palette. You can look up one of these colors, for example, by doing

```
(clim:find-named-color "lavender"
  (clim:frame-palette clim:*application-frame*))
```

Applications can define other colors, but these are provided because they are commonly used in the X Windows community (not because there is anything special about these particular colors). This table is a subset of the colors listed in the file `/X11/R4/mit/rgb/rgb.txt`, from the X11 R4 distribution.

alice-blue	antique-white	aquamarine
azure	beige	bisque
black	blanched-almond	blue
blue-violet	brown	burlywood
cadet-blue	chartreuse	chocolate
coral	cornflower-blue	cornsilk
cyan	dark-goldenrod	dark-green
dark-khaki	dark-olive-green	dark-orange
dark-orchid	dark-salmon	dark-sea-green
dark-slate-blue	dark-slate-gray	dark-turquoise
dark-violet	deep-pink	deep-sky-blue
dim-gray	dodger-blue	firebrick
floral-white	forest-green	gainsboro
ghost-white	gold	goldenrod
gray	green	green-yellow
honeydew	hot-pink	indian-red
ivory	khaki	lavender
lavender-blush	lawn-green	lemon-chiffon
light-blue	light-coral	light-cyan
light-goldenrod	light-goldenrod-yellow	light-gray
light-pink	light-salmon	light-sea-green
light-sky-blue	light-slate-blue	light-slate-gray
light-steel-blue	light-yellow	lime-green
linen	magenta	maroon
medium-aquamarine	medium-blue	medium-orchid
medium-purple	medium-sea-green	medium-slate-blue
medium-spring-green	medium-turquoise	medium-violet-red
midnight-blue	mint-cream	misty-rose
moccasin	navajo-white	navy-blue
old-lace	olive-drab	orange
orange-red	orchid	pale-goldenrod
pale-green	pale-turquoise	pale-violet-red
papaya-whip	peach-puff	peru
pink	plum	powder-blue
purple	red	rosy-brown
royal-blue	saddle-brown	salmon
sandy-brown	sea-green	seashell
sienna	sky-blue	slate-blue
slate-gray	snow	spring-green
steel-blue	tan	thistle
tomato	turquoise	violet
violet-red	wheat	white
white-smoke	yellow	yellow-green

In addition to these named colors, CLIM also provides constants for the primary colors: **clim:+black+**, **clim:+white+**, **clim:+red+**, **clim:+green+**, **clim:+blue+**, **clim:+cyan+**, **clim:+yellow+**, and **clim:+magenta+**.

Drawing with Designs in CLIM

Concepts of Designs in CLIM

A design is an object that represents a way of arranging colors and opacities in the drawing plane. The simplest kind of design is a color, which simply places a constant color at every point in the drawing plane. See the section "Drawing in Color in CLIM".

This chapter describes more complex kinds of design, which place different colors at different points in the drawing plane or compute the color from other information, such as the color previously at that point in the drawing plane. Not all of the features described in this chapter are supported in the present implementation.

Recall that the drawing functions work by selecting a region of the drawing plane and painting it with color, and that the **:ink** drawing option specifies how to color this region. The value of the **:ink** drawing option can be any kind of design, any member of the class **clim:design**. The values of **clim:medium-foreground**, **clim:medium-background**, and **clim:medium-ink** are also designs. Not all designs are supported as the arguments to the **:ink** drawing option, or as a foreground or background in the present implementation.

A design can be characterized in several different ways:

All designs are either *bounded* or *unbounded*. Bounded designs are transparent everywhere beyond a certain distance from a certain point. Drawing a bounded design has no effect on the drawing plane outside that distance. Unbounded designs have points of non-zero opacity arbitrarily far from the origin. Drawing an unbounded design affects the entire drawing plane.

All designs are either *uniform* or *non-uniform*. Uniform designs have the same color and opacity at every point in the drawing plane. Uniform designs are always unbounded, unless they are completely transparent.

All designs are either *solid* or *translucent*. At each point a solid design is either completely opaque or completely transparent. A solid design can be opaque at some points and transparent at others. In translucent designs, at least one point has an opacity that is intermediate between completely opaque and completely transparent.

All designs are either *colorless* or *colored*. Drawing a colorless design uses a default color specified by the medium's foreground design. This is done by drawing with `(clim:compose-in clim:+foreground-ink+ clim:+transparent-ink+)`.

A variety of designs are available. See

"Concepts of Drawing in Color in CLIM"
"Indirect Ink in CLIM"
"Flipping Ink in CLIM"
"Concepts of Patterned Designs in CLIM"
"Concepts of Translucent Ink in CLIM"
"Complex Designs in CLIM"

Indirect Ink in CLIM

Drawing with an *indirect ink* is the same as drawing another design named directly. For example, **clim:+foreground-ink+** is a design that draws the medium's foreground design.

Indirect inks exist for the benefit of output recording. For example, one can draw with **clim:+background-ink+**, change to a different **clim:medium-background**, and replay the output record; the replayed output will come out with a new background color. If the current background is the color red, drawing with **clim:+background-ink+** means to draw with the background, whatever it is. On the other hand, drawing with **clim:+red+** means to draw with the color red, even if the background is later changed to green.

You can change the foreground or background design 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 design is drawn all over the 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.

If an infinite recursion is created using an indirect ink, an error is signalled when the recursion is created, when the design is used for drawing, or both.

clim:+foreground-ink+ is the default value of the **:ink** drawing option.

In the current implementation, the foreground and background must be colors.

Two indirect inks are defined:

clim:+foreground-ink+
: An indirect ink that uses the medium's foreground design.

clim:+background-ink+
: An indirect ink that uses the medium's background design.

Flipping Ink in CLIM

You can use a flipping ink to interchange occurrences of two colors. The purpose of flipping is to allow the use of ''XOR hacks'' for temporary changes to the display. For example, CLIM uses **clim:+flipping-ink+** when drawing highlighting boxes.

In the present implementation, both designs must be colors.

clim:make-flipping-ink *design1 design2*
: Returns a design that interchanges occurrences of two designs.

clim:+flipping-ink+
: A flipping ink that flips **clim:+foreground-ink+** and **clim:+background-ink+**.

Concepts of Patterned Designs in CLIM

Patterned designs are non-uniform designs that have a certain regularity. These include patterns, stencils, tiled designs, and transformed designs.

In the present implementation, patterned designs are not fully supported as a foreground or background, and the only patterned designs supported as the **:ink** drawing option are tilings of patterns of **clim:+background-ink+** (or **clim:+transparent-ink+**) and **clim:+foreground-ink+**. In Cloe there is an additional restriction that the X offset and Y offset of the tiling must be 8.

Patterns and Stencils

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 a pattern, use **clim:make-pattern**. To repeat a pattern so it fills the drawing plane, apply **clim:make-rectangular-tile** to a pattern.

A stencil is a special kind of pattern that contains only opacities. The name ''stencil'' refers to their use with **clim:compose-in** and **clim:compose-over**.

Tiling

Tiling repeats a rectangular portion of a design throughout the drawing plane. This is most commonly used with patterns. Use **clim:make-rectangular-tile** to make a tiled design.

Transforming Designs

The functions **clim:transform-region** and **clim:untransform-region** accept any design as their second argument and apply a coordinate transformation to the design. The result is a design that might be freshly constructed or might be an existing object.

Transforming a uniform design simply returns the argument. Transforming a composite, flipping, or indirect design applies the transformation to the component design(s). Transforming a pattern, tile, or output record design is described in the sections on those designs.

Operators for Patterned Designs in CLIM

clim:make-pattern *array designs*

Creates a pattern design that has `(array-dimension 2d-array 0)` cells in the vertical direction and `(array-dimension 2d-array 1)` cells in the horizontal direction.

clim:make-pattern-from-bitmap-file *pathname* &rest *args* &key *(:type* **:x11***) :designs :format* &allow-other-keys

Reads the bitmap file specified by *pathname* and creates a CLIM pattern object from it.

clim:make-stencil *array*

Make a pattern of opacities from a two-dimensional array.

clim:make-rectangular-tile *design width height*

Creates a design that tiles the specified rectangular portion of *design* across the entire drawing plane.

Concepts of Translucent Ink in CLIM

Translucent ink supports the following drawing techniques:

- Controlling opacity
- Blending colors
- Compositing

Controlling Opacity

Opacity controls how new output covers previous output. Intermediate opacity values result in color blending so that the earlier picture shows through what is drawn on top of it.

An opacity is a real number between 0 and 1; 0 is completely transparent, 1 is completely opaque, and fractions are translucent. The opacity of a design is the degree to which it hides the previous contents of the drawing plane when it is drawn. Opacity can vary from totally opaque to totally transparent.

Use **clim:make-opacity** or **clim:make-stencil** to specify opacity.

Note: Opacity values that are not either fully transparent or fully opaque are not currently fully supported.

Color Blending

Drawing a design that is not completely opaque at all points allows the previous contents of the drawing plane to show through. The simplest case is drawing a *solid* design. Where the design is opaque, it replaces the previous contents of the drawing plane. Where the design is transparent, it leaves the drawing plane unchanged.

In the more general case of drawing a translucent design, the resulting color is a blend of the design's color and the previous color of the drawing plane. For purposes of color blending, the drawn design is called the foreground and the drawing plane is called the background.

The function **clim:compose-over** performs a similar operation. It combines two designs to produce a design, rather than combining a design and the contents of the drawing plane to produce the new contents of the drawing plane. For purposes of color blending, the first argument to **clim:compose-over** is called the foreground and the second argument is called the background.

Color blending is defined by an ideal function: $F(r_1,g_1,b_1,o_1,r_2,g_2,b_2,o_2)$ to (r_3,g_3,b_3,o_3) that operates on the color and opacity at a single point.

(r_1,g_1,b_1,o_1) are the foreground color and opacity.

(r_2, g_2, b_2, o_2) are the background color and opacity.

(r_3, g_3, b_3, o_3) are the resulting color and opacity.

The color blending function is conceptually applied at every point in the drawing plane.

The function F performs linear interpolation on all four components:

$$o_3 = o_1 + (1 - o_1) * o_2$$
$$r_3 = (o_1 * r_1 + (1 - o_1) * o_2 * r_2) / o_3$$
$$g_3 = (o_1 * g_1 + (1 - o_1) * o_2 * g_2) / o_3$$
$$b_3 = (o_1 * b_1 + (1 - o_1) * o_2 * b_2) / o_3$$

Note that if o_3 is zero, these equations would divide zero by zero. In that case r_3, g_3, and b_3 are defined to be zero.

CLIM requires that F be implemented exactly if o_1 is zero or one or if o_2 is zero. If o_1 is zero, the result is the background. If o_1 is one or o_2 is zero, the result is the foreground. For fractional opacity values, an implementation can deviate from the ideal color blending function either because the implementation has limited opacity resolution or because the implementation can compute a different color blending function much more quickly.

If a medium's background design is not completely opaque at all points, the consequences are unspecified. Consequently, a drawing plane is always opaque and drawing can use simplified color blending that assumes $o_2 = 1$ and $o_3 = 1$. However, **clim:compose-over** must handle a non-opaque background correctly.

Note that these *(r,g,b,o)* quadruples of real numbers between 0 and 1 are mathematical and an implementation need not store information in this form. Most implementations are expected to use a different representation.

Compositing

Compositing creates a design whose appearance at each point is a composite of the appearances of two other designs at that point. Three varieties of compositing are provided: composing *over*, composing *in*, and composing *out*.

You can use **clim:compose-over**, **clim:compose-in**, or **clim:compose-out** to create CLIM composite designs.

In the present implementation compositing is not fully supported.

Operators for Translucent Ink in CLIM

The following functions can be used to create an opacity object, and to compose a new ink from a color and an opacity. (The three composition operators can also be used to compose more complex designs.)

The present implementation of CLIM only fully supports opacities that are either fully opaque or fully transparent. It uses stipples for translucent opacities, and composition of translucent opacities does not work well.

clim:make-opacity *value* Creates a member of class **clim:opacity** whose opacity is *value*, which is a real number in the range from 0 to 1 (inclusive), where 0 is fully transparent and 1 is fully opaque.

clim:+transparent-ink+ When you draw a design that has areas of **clim:+transparent-ink+**, the former background shows through in those areas.

clim:opacity-value *opacity* Returns the *value* of *opacity*, which is a real number in the range from 0 to 1 (inclusive).

clim:compose-over *design1 design2*
Composes a design that is equivalent to *design1* drawn on top of *design2*. Drawing the resulting design produces the same visual appearance as drawing *design2* and then drawing *design1*, but might be faster and might not allow the intermediate state to be visible on the screen.

clim:compose-in *design1 design2*
Composes a design by using the color (or ink) of *design1* and clipping to the inside of *design2* (that is, *design2* specifies the mask to use for changing the shape of the design).

clim:compose-out *design1 design2*
Composes a design by using the color (or ink) of *design1* and clipping to the outside of *design2* (that is, *design2* specifies the mask to use for changing the shape of the design).

Complex Designs in CLIM

Note that the designs described in this section are not supported as the **:ink** drawing option in the present implementation, but you can use **clim:draw-design** to draw them.

You can use **clim:make-design-from-output-record** to make a design that replays the output record when the design is drawn using **clim:draw-design**.

clim:make-design-from-output-record *record*
Makes a design that replays *record* when the design is drawn by **clim:draw-design**.

Since designs are a generalization of regions that include color, any member of the class **clim:region** acts as a solid, colorless design. The design is opaque at points in the region and transparent elsewhere.See the section "Regions in CLIM".

Achieving Different Drawing Effects in CLIM

Here are some examples of how to achieve a variety of commonly used drawing effects:

- **Drawing in the foreground color**

Use the default, or specify `:ink clim:+foreground-ink+`

- **Erasing**

  Specify `:ink clim:+background-ink+`

- **Drawing in color**

  Specify `:ink clim:+green+`, or `:ink (clim:make-color-rgb 0.6 0.0 0.4)`

- **Painting a gray or colored wash over a display**

  Specify a translucent design as the ink, such as

```
:ink (clim:compose-in clim:+black+ (clim:make-opacity 0.25))
:ink (clim:compose-in clim:+red+ (clim:make-opacity 0.1))
:ink (clim:compose-in clim:+foreground-ink+ (clim:make-opacity 0.75))
```

  The last example can be abbreviated as `:ink (clim:make-opacity 0.75)`. On a non-color, non-grayscale display this will probably turn into a stipple.

- **Drawing an opaque gray**

  Specify `:ink (clim:make-gray-color 0.25)` to draw in a shade of gray independent of the window's foreground color. On a non-color, non-grayscale display this will probably turn into a stipple.

- **Drawing a faded but opaque version of the foreground color**

  Specify `:ink (clim:compose-over (clim:compose-in clim:+foreground-ink+ (clim:make-opacity 0.25)) clim:+background-ink+)` to draw at 25% of the normal contrast.

  This technique is not fully supported in the present implementation. The design will generally be displayed as a stippled pattern.

- **Drawing a stipple of little bricks**

  Specify `:ink bricks`, where `bricks` is defined as

```
(clim:make-rectangular-tile
  (clim:make-pattern #2a((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 clim:+background-ink+
                           clim:+foreground-ink+))
  8 8)
```

- **Drawing a tiled pattern**

  Specify `:ink (clim:make-rectangular-tile (clim:make-pattern` *array colors*`))`

- **Drawing a pattern**

  Use `(clim:draw-pattern*` *medium* `(clim:make-pattern` *array colors*`)` *x y*`)`

Presentation Types in CLIM

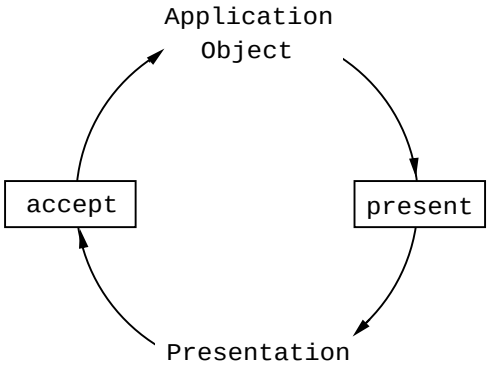
Concepts of CLIM Presentation Types

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 drawings.

Users need to interact with the application objects. A CLIM user interface enables users to see a visual representation of the application objects, and to operate on them. The objects that appear on the screen are not the application objects themselves; they are one step removed. The visual representation of an object is a stand-in for the application object itself, in the same sense that the word ‘‘cat’’ (or a picture of a cat) is a stand-in for a real cat.

A fundamental part of designing a CLIM user interface is to specify how users will interact with the 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, **clim:present** and **clim:accept**, and some related functions.



Presentations

CLIM directly couples the visual representation of an object with 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

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, or a graphical representation (such as a form with name, address, 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 the object is known, CLIM knows which operations are appropriate to perform on it. For example, when a student is displayed, it is possible to perform operations such as ‘‘send-tuition-bill’’ or ‘‘show-transcript’’.

Input Context

Presentations are the basis of many of the higher-level application-building tools, which use **clim:accept** to get input and **clim:present** to do output. A command that takes arguments as input states the presentation type of each argument. This sets up an *input context*, in which presentations of that type are sensitive (they are highlighted when the pointer passes over them). When the user gives the **send-tuition-bill** command, the input context is looking for a student, so any displayed students are sensitive. Presentations that have been output in previous user interactions retain their semantics. In other words, CLIM has recorded the fact that a student has been displayed, and has saved this information so that whenever the input context expects a student, all displayed students are sensitive.

Inheritance

CLIM presentation types can be 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.

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 presented object, such as a student. In the absence of a presentation translator, the student is not sensitive because the user must enter a command and cannot enter anything else to this input context.

In the presence of a presentation translator that translates from students to commands, however, the presented student would be sensitive. In one scenario, the presented student is highlighted, and the middle pointer button does ‘‘Show Transcript’’ of that student.

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:

- Decide what types of application objects will be presented to the user as output and accepted from the user as input.
- 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. Usually the presentation type is the same as the class of the application object.
- 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.

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 the section "Predefined Presentation Types in CLIM".

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. Not all presentation types are Common Lisp types (such as the **clim:boolean** presentation type) and not all Common Lisp types are presentation types, but there is a lot of overlap.

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

name
(*name parameters...*)
((*name parameters...*) *options...*)

Each presentation type has a name, which is usually a symbol naming the presentation type. The name can also be a CLOS class object; this usage provides the support for anonymous CLOS classes.

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. The parameters state a restriction on the presentation type, so a parameterized presentation type is a specialization, or a subset, of the presentation type of that name with no parameters.

Examples of the second pattern are:

(integer 0 10)	A parameterized type indicating an integer in the range of zero through ten.
(string 25)	A parameterized type indicating a string whose length is 25.
(member :yes :no :maybe)	A parameterized type which can be one of the three given values, **:yes**, **:no**, and **:maybe**.

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

Some presentation type options may appear as an option in any presentation type specifier. Currently, the only such option is **:description**.

Using CLIM Presentation Types for Output

The reason for using presentations for program output is so that the objects presented will be acceptable to input functions. Suppose, for example, you present an object, such as **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 **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 this information remains available, previously presented objects are themselves available for input to functions for accepting objects.

CLIM Operators for Presenting Typed Output

An application can use the following operators to produce output that will be associated with a given Lisp object and be 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.

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

clim:with-output-as-presentation *(stream object type* &key *:modifier :single-box (:allow-sensitive-inferiors* **t***) :parent :record-type)* &body *body*
Gives separate access to the two aspects of **clim:present**: recording the presentation and drawing the visual representation. This macro generates a presentation from the output done in the *body* to the *stream*. The presentation's underlying object is *object*, and its presentation type is *type*.

clim:present *object* &optional *(presentation-type* **(clim:presentation-type-of object)***)* &key *(:stream* ***standard-output****) (:view* **(clim:stream-default-view stream)***) :modifier :acceptably (:for-context-type* **presentation-type***) :single-*

box :allow-sensitive-inferiors :sensitive (:record-type **'clim:standard-presentation***)* Creates a presentation on the stream of the specified object, using the given type and view to determine visual appearance. The manner in which the object is displayed depends on the presentation type of the object; the display is done by the type's **clim:present** method for the given *view*.

clim:present-to-string *object* &optional *(presentation-type* **(clim:presentation-type-of object)***)* &key *(:view* **clim:+textual-view+***) :acceptably (:for-context-type* **presentation-type***) :string :index*
Presents an object into a string in such a way that it can subsequently be accepted as input by **clim:accept-from-string**.

Additional Functions for Operating on Presentations in CLIM

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

clim:presentationp *object* Returns **t** if and only if *object* is of type **clim:presentation**.

clim:presentation-object *presentation*
Returns the application object represented by the presentation *presentation*. You can use **setf** on **clim:presentation-object** to change the object associated with the presentation.

clim:presentation-type *presentation*
Returns the presentation type of the presentation *presentation*. You can use **setf** on **clim:presentation-type** to change the presentation type associated with the presentation.

clim:presentation The protocol class that corresponds to a presentation.

clim:standard-presentation The standard class used by CLIM to represent presentations.

Using CLIM Presentation Types for Input

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

Examples:

```
(clim:accept 'string) ==>
Enter a string: abracadabra
"abracadabra"

(clim:accept 'string) ==>
Enter a string [default abracadabra]: abracadabra
"abracadabra"
```

In the first call to **clim:accept**, "abracadabra" was typed at the keyboard. In the second call to **clim:accept**, the user clicked on the keyboard-entered string of the first function. In both cases, the same (in the sense of **eq**) string object "abracadabra" was returned.

Typically, not just any kind of object is acceptable as input. Only an object of the presentation type specified in the current **clim:accept** function (or one of its subtypes) can be input. In other words, the **clim:accept** function establishes the current *input context*. For example, if the call to **clim:accept** specifies an integer presentation type, only a typed-in or a 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. Thus, **clim:accept** controls the input context and thereby the sensitivity of displayed presentations.

Clicking on a presentation of a type different from the input context may cause translation to an acceptable object, if there is an appropriate presentation translator defined. 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 the section "Presentation Translators in CLIM".

Typically, the range of acceptable input is restricted. How restricted it is, is strictly up to you, the programmer. Using compound presentation types like **and** and **or**, and other predefined or specially devised presentation types gives you a high degree of flexibility and control over the input context.

CLIM Operators for Accepting Input

CLIM provides the following top-level operators for accepting typed input.

clim:with-input-context is the most general operator, and **clim:accept** and **clim:accept-from-string** support common idioms.

Note that **clim:accept** does not insert newlines. If you want each call to **clim:accept** to appear on a new line, use **terpri**.

clim:accept *type* &rest *accept-args* &key *(:stream* ***standard-input***) *(:view* **(clim:stream-default-view stream)**) *:default (:default-type* **type**) *(:history* **type**) *:provide-default (:prompt* **t**) *(:prompt-mode* **':normal**) *(:display-default* **prompt**) *:query-identifier :activation-gestures :additional-activation-gestures :delimiter-gestures :additional-delimiter-gestures :insert-default (:replace-input* **t**) *(:active-p* **t**)

Requests input of the *type* from the *stream*. **clim:accept** returns two values, the object and its presentation type. **clim:accept** works by prompting, then establishing an input context via **clim:with-input-context**, and then calling the **clim:accept** CLIM presentation method for *type* and *:view*.

clim:accept-from-string *type string* &key *(:view* **clim:+textual-view+***) :default (:default-type* **type***) :activation-gestures :additional-activation-gestures :delimiter-gestures :additional-delimiter-gestures (:start* **0***) :end*
Reads a printed representation of an object of type *type* from *string*. This function is like **clim:accept**, except that the input is taken from *string*, starting at *:start* and ending at *:end*. This function is analogous to **read-from-string**.

clim:with-input-context *(type* &key *:override) (*&optional *object-var type-var event-var options-var) form* &body *clauses*
Establishes an input context of type *type*, evaluates *form* in the new context, and (optionally) evaluate one of *clauses*.

clim:*input-context* The current input context, which describes the presentation type(s) currently being input by CLIM.

clim:input-context-type *context-entry*
Given one element from **clim:*input-context***, *context-entry*, this returns the presentation type of the context entry.

Predefined Presentation Types in CLIM

This section documents predefined CLIM presentation types, presentation type options, and parameters. For more information on how to use these presentation types, see the section "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).

Basic Presentation Types in CLIM

Here are basic presentation types that correspond to the Common Lisp types having the same name.

t *Clim Presentation Type*

The supertype of all other presentation types.

Note that the **clim:accept** method for this type allows input only via the pointer; if the user types anything on the keyboard, the **clim:accept** method just beeps.

null *Clim Presentation Type*

The presentation type that represents ‘‘nothing’’. The single object associated with this type is **nil**, and its printed representation is "None".

clim:boolean *Clim Presentation Type*

The presentation type that represents **t** or **nil**. The textual representation is "Yes" and "No", respectively.

symbol *Clim Presentation Type*

The presentation type that represents a symbol.

keyword *Clim Presentation Type*

The presentation type that represents a symbol in the **keyword** package. It is a subtype of **symbol**.

Numeric Presentation Types in CLIM

The following presentation types represent the Common Lisp numeric types having the same names.

number *Clim Presentation Type*

The presentation type that represents a general number. It is the supertype of all the number types.

complex &optional *type* *Clim Presentation Type*

The presentation type that represents a complex number. It is a subtype of **number**.

type is the type to use for the components. It must be a subtype of **real**.

future-common-lisp:real &optional *low high* *Clim Presentation Type*

The presentation type that represents either a ratio, an integer, or a floating point number between *low* and *high*. *low* and *high* can be inclusive or exclusive, as in Common Lisp type specifiers.

Options to this type are **:base** and **:radix**, which are the same as for the **integer** type. This type is a subtype of **number**.

rational &optional *low high* *Clim Presentation Type*

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. It is a subtype of **real**.

integer &optional *low high* *Clim Presentation Type*

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 &optional *low high* *Clim Presentation Type*

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 &optional *low high* *Clim Presentation Type*

The presentation type that represents a floating point number between *low* and *high*. This type is a subtype of **clim:real**.

Character and String Presentation Types in CLIM

These two presentation types can be used for reading and writing character and strings.

character *Clim Presentation Type*

The presentation type that represents a Common Lisp character object.

string &optional *length* *Clim Presentation Type*

The presentation type that represents a string. If *length* is specified, the string must have exactly that many characters.

Pathname Presentation Type in CLIM

clim-lisp:pathname *Clim Presentation Type*

The presentation type that represents a pathname.

The options are **:default-type** (which defaults to **nil**), **:default-version** (which defaults to **:newest**), and **:merge-default** (which defaults to **t**). If **:merge-default** is **nil**, **clim:accept** returns the exact pathname that was entered, otherwise **clim:accept** merges against the default provided to **clim:accept** and **:default-type**

and **:default-version**, using **merge-pathnames**. If no default is specified, it defaults to ***default-pathname-defaults***.

One-of and Some-of Presentation Types in CLIM

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.

Args	*Single*	*Multiple*	
most general		**clim:completion**	**clim:subset-completion**
&rest elements		**member**	**clim:subset**
sequence	**clim:member-sequence**		**clim:subset-sequence**
alist	**clim:member-alist**	**clim:subset-alist**	

clim:completion *sequence* &key *:test :value-key* *Clim Presentation Type*

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 **clim:completion** type, including **clim:token-or-type**, **clim:null-or-type**, **member**, **clim:member-sequence**, and **clim:member-alist**.

The presentation type parameters are:

sequence 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. **clim:accept** reads a name and returns a value. **clim:present** is given a value and outputs a name.

:test A function that compares two values for equality. The default is **eql**.

:value-key A function that returns a value given an element of *sequence*. The default is **identity**.

The following presentation type options are available:

:name-key
A function that returns a name, as a string, given an element of *sequence*. The default is a function that behaves as follows:

Argument	*Returned Value*
string	the string
null	the string **"NIL"**
cons	**string** of the **car**
symbol	**string-capitalize** of its name
otherwise	**princ-to-string** of it

:documentation-key
A function that returns **nil** or a descriptive string, given an element of *sequence*. The default always returns **nil**.

:partial-completers
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 &rest *elements* *Clim Presentation Type Abbreviation*

The presentation type that specifies one of *elements*. The options (**:name-key**, **:value-key**, and **:partial-completers**) are the same as for **clim:completion**.

clim:member-sequence *sequence* &key *:test* *Clim Presentation Type Abbreviation*

Like **member**, except that the set of possibilities is the sequence *sequence*. The parameter *:test* and the options (**:name-key**, **:value-key**, and **:partial-completers**) are the same as for **clim:completion**.

clim:member-alist *alist* &key *:test* *Clim Presentation Type Abbreviation*

Like **member**, except that the set of possibilities is the alist *alist*. Each element of *alist* is either an atom (as in **clim: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 containing one or more of the following properties:

  :value — the value
  :documentation — a descriptive string

The *:test* parameter and the options are the same as for **clim:completion** except that the **:value-key** and **:documentation-key** options default to functions that support the specified alist format.

clim:subset-completion *sequence* &key *:test :value-key* *Clim Presentation Type*

The presentation 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 **clim:completion** with the following additional options:

:separator The character that separates members of the set of possibilities in the printed representation when there is more than one. The default is comma.

:echo-space (**t** or **nil**) Whether to insert a space automatically after the separator. The default is **t**.

The other subset types (**clim:subset**, **clim:subset-sequence**, and **clim:subset-alist**) are implemented in terms of the **clim:subset-completion** type.

clim:subset &rest *elements* *Clim Presentation Type Abbreviation*

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 the separator character. The options (**:name-key**, **:value-key**, **:partial-completers**, **:separator**, and **:echo-space**) are the same as for **clim:subset-completion**.

clim:subset-sequence *sequence* &key *:test* *Clim Presentation Type Abbreviation*

Like **clim:subset**, except that the set of possibilities is the sequence *sequence*. The parameter *:test* and the options (**:name-key**, **:value-key**, **:partial-completers**, **:separator**, and **:echo-space**) are the same as for **clim:subset-completion**.

clim:subset-alist *alist* &key *:test* *Clim Presentation Type Abbreviation*

Like **clim:subset**, except that the set of possibilities is the alist *alist*. The parameter *:test* and the options (**:name-key**, **:value-key**, **:partial-completers**, **:separator**, and **:echo-space**) are the same as for **clim:subset-completion**. The parameter *alist* has the same format as **clim:member-alist**.

Sequence Presentation Types in CLIM

The following two presentation types can be used to accept and present a sequence of objects.

sequence *type* *Clim Presentation Type*

The presentation type that represents a sequence of elements of type *type*. The printed representation of a **sequence** type is the elements separated by the separator character. It is unspecified whether **clim:accept** returns a list or a vector. You can specify the following options:

:separator The character that separates members of the set of possibilities in the printed representation when there is more than one. The default is comma (`#\,`).

:echo-space If this is **t**, then CLIM will insert a space automatically after the separator, otherwise it will not. The default is **t**.

type can be a presentation type abbreviation.

clim:sequence-enumerated &rest *types* *Clim Presentation Type*

clim:sequence-enumerated is like **sequence**, except that the type of each element in the sequence is individually specified. It is unspecified whether **clim:accept** returns a list or a vector. You can specify the following options:

:separator The character that separates members of the set of possibilities in the printed representation when there is more than one. The default is comma (`#\,`).

:echo-space If this is **t**, then CLIM will insert a space automatically after the separator, otherwise it will not. The default is **t**.

The elements of *types* can be presentation type abbreviations.

Meta Presentation Types in CLIM

or &rest *types* *Clim Presentation Type*

The presentation type that is used to specify one of several types, for example,

```
(or (member :all :none) integer)
```

clim:accept returns one of the possible types as its second value, not the original **or** presentation type specifier.

The elements of *types* can be presentation type abbreviations.

The **clim:accept** method for the **or** type works by iteratively calling **clim:accept** on each of the presentation types in *types*. It establishes a condition handler for **user::parse-error**, calls **clim:accept**, and returns the result if no condition is signalled. If a **user::parse-error** condition is signalled, CLIM calls the **clim:accept** method for the next type. If all of the calls to **clim:accept** fail, the **clim:accept** method for **or** signals a **user::parse-error**.

and &rest *types* *Clim Presentation Type*

The presentation type that is used for multiple inheritance. **and** is usually used in conjunction with **satisfies**. For example,

```
(and integer (satisfies oddp))
```

The elements of *types* can be presentation type abbreviations.

The first type in *types* is in charge of accepting and presenting. The remaining elements of *types* are used for type checking (for example, filtering applicability of presentation translators).

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.

Compound Presentation Types in CLIM

The following compound presentation types are provided because they implement some common idioms.

clim:token-or-type *tokens type* *Clim Presentation Type Abbreviation*

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 *alist* parameter to **clim:member-alist**; typically it is a list of keyword symbols.

type can be a presentation type abbreviation.

For example, the following is a common way of using **clim:token-or-type**:

```
(clim:accept '(clim:token-or-type (:all :none) integer)
             :prompt "How many?")
```

clim:null-or-type *type* *Clim Presentation Type Abbreviation*

A compound type that is used to select **nil**, whose printed representation is the special token "None", or an object of type *type*.

type can be a presentation type abbreviation.

clim:type-or-string *type* *Clim Presentation Type Abbreviation*

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 **clim:accept** cannot parse as the representation of an object of type *type* is returned as a string.

type can be a presentation type abbreviation.

Command and Form Presentation Types in CLIM

The command and form presentation types are complex types provided primarily for use by the top level interactor of an application.

clim:expression *Clim Presentation Type*

The presentation type used to represent any Lisp object. The textual view of this type looks like what the standard **prin1** and **read** functions produce and accept.

This type has one option, **:auto-activate**, which controls whether the expression terminates on a delimiter gestures, or when the Lisp expression ''balances'' (for example, you type enough close parentheses to complete the expression). The default for **:auto-activate** is **nil**, meaning that the user must use an activation gesture to terminate the input.

clim:form *Clim Presentation Type*

The presentation type used to represent a Lisp form. This type is a subtype of **clim:expression**. It has one option, **:auto-activate**, which is treated the same way as the **:auto-activate** option to **clim:expression**.

clim:command &key *:command-table* *Clim Presentation Type*

The presentation type used to represent a CLIM command processor command and its arguments. *:command-table* can be either a command table or a symbol that names a command table.

If *:command-table* is not supplied, it defaults to the command table for the current application, that is, **(clim:frame-command-table clim:*application-frame*)**.

When you call **clim:accept** on this presentation type, the returned value is a list; the first element is the command name, and the remaining elements are the command arguments. You can use **clim:command-name** and **clim:command-arguments** to access the name and arguments of the command object.

For more information about CLIM command objects, see the section "Command Objects in CLIM".

clim:command-name &key *:command-table* *Clim Presentation Type*

The presentation type used to represent the name of a CLIM command processor command in the command table *:command-table*.

:command-table may be either a command table or a symbol that names a command table. If *:command-table* is not supplied, it defaults to the command table for the current application. The textual representation of a **clim:command-name** object is the command-line name of the command, while the internal representation is the command name.

clim:command-or-form &key *:command-table* *Clim Presentation Type*

The presentation type used to represent either a Lisp form or a CLIM command processor command and its arguments. In order for the user to indicate that he wishes to enter a command, a command dispatch character must be typed as the first character of the command line.

See the variable **clim:*command-dispatchers***.

:command-table may be either a command table or a symbol that names a command table. If *:command-table* is not supplied, it defaults to the command table for the current application, that is, **(clim:frame-command-table clim:*application-frame*)**.

Defining a New Presentation Type in CLIM

Concept of Defining a New Presentation Type in CLIM

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.

By defining a presentation type, you define all of the user interface components of the entity:

- A displayed representation, for example, textual or graphical
- A textual representation, for user input via the keyboard (a textual representation is optional)
- Pointer sensitivity, for user input via the pointer

In other words, by defining a presentation type, you describe in one place all the information about an object necessary to display it to the user and interact with the user for getting input.

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 **clim:define-presentation-type** macro.

   - Name the new presentation type.
   - Supply parameters that further restrict the type (if appropriate).
   - Supply options that affect the appearance of the type (if appropriate).
   - State the supertypes of this type, to make use of inheritance (if appropriate).

2. Define CLIM presentation methods.

   - Specify how objects are displayed with a **clim:present** presentation method. (You must define a **clim:present** method, unless the new presentation type inherits a method that is appropriate for it.)

- Specify how objects are parsed with a **clim:accept** presentation method. (In most cases, you must define a **clim:accept** method, unless the new presentation type inherits a method that is appropriate for it. If it is never necessary to enter the object by typing its representation on the keyboard, you don't need to provide this method.)
- Specify the type/subtype relationships of this type and its related types, if necessary, with **clim:presentation-typep** and **clim:presentation-subtypep** presentation methods. (You must define or inherit these methods when defining a presentation type that has parameters.)

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 the same as the CLOS class.

When the class is a subclass of **clos:built-in-class**, the presentation type is a built on a special class that is internal to CLIM. 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 (**clim:presentation-type** *name*).

IMPORTANT NOTE: If the same name is defined with both **clos:defclass** (or **defstruct**) and **clim: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 **clim: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 inheritance that the built-in class already has; you may want to do this in order to add presentation 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 **clim:presentation-typep** presentation method for it. If you define a presentation type that has parameters, you must define or inherit a **clim:presentation-subtypep** for it.

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 **clim: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.

Example of Defining a New CLIM Presentation Type

This example shows how to define a new presentation type, and how to define the presentation methods for the new type. First we define the application objects themselves and create some test data. Then we define a simple presentation type, and gradually add enhancements to it to show different CLIM techniques.

This example models a university. The application objects are students, courses, and departments. This is such a simple example that there is no need to use inheritance.

Note that this example must be run in a package, such as **clim-user**, that has access to symbols from the **clim** and **clos** packages.

These are the definitions of the application objects:

```
(defclass student ()
    ((name :reader student-name :initarg :name)
     (courses :accessor student-courses :initform nil)))

(defclass course ()
    ((name :reader course-title :initarg :title)
     (department :reader course-department :initarg :department)))

(defclass department ()
    ((name :reader department-name :initarg :name)))
```

The following code provides support for looking up objects by name.

```
(defvar *student-table* (make-hash-table :test #'equal))
(defvar *course-table* (make-hash-table :test #'equal))
(defvar *department-table* (make-hash-table :test #'equal))

(defun find-student (name &optional (errorp t))
  (or (gethash name *student-table*)
      (and errorp (error "There is no student named ~S" name))))

(defun find-course (name &optional (errorp t))
  (or (gethash name *course-table*)
      (and errorp (error "There is no course named ~S" name))))

(defun find-department (name &optional (errorp t))
  (or (gethash name *department-table*)
      (and errorp (error "There is no department named ~S" name))))
```

```
(defmethod initialize-instance :after ((student student) &key)
  (setf (gethash (student-name student) *student-table*) student))

(defmethod initialize-instance :after ((course course) &key)
  (setf (gethash (course-title course) *course-table*) course))

(defmethod initialize-instance :after ((department department) &key)
  (setf (gethash (department-name department) *department-table*) department))

(defmethod print-object ((student student) stream)
  (print-unreadable-object (student stream :type t)
    (write-string (student-name student) stream)))

(defmethod print-object ((course course) stream)
  (print-unreadable-object (course stream :type t)
    (write-string (course-title course) stream))
  (format stream " (~A)" (department-name (course-department course))))

(defmethod print-object ((department department) stream)
  (print-unreadable-object (department stream :type t)
    (write-string (department-name department) stream)))
```

Here we create some test data:

```
(flet ((make-student (name &rest courses)
         (setf (student-courses (make-instance 'student :name name))
               (copy-list courses)))
       (make-course (title department)
         (make-instance 'course :title title :department department))
       (make-department (name)
         (make-instance 'department :name name)))
  (let* ((english (make-department "English"))
         (physics (make-department "Physics"))
         (agriculture (make-department "Agriculture"))
         (englit (make-course "English Literature" english))
         (mabinogion (make-course "Deconstructing the Mabinogion" english))
         (e&m (make-course "Electricity and Magnetism II" physics))
         (beans (make-course "The Cultivation and Uses of Beans" agriculture))
         (horses (make-course "Horse Breeding for Track and Field" agriculture))
         (corn (make-course "Introduction to Hybrid Corn" agriculture)))
    (make-student "Susan Charnas" englit e&m)
    (make-student "Orson Card" englit beans)
    (make-student "Roberta MacAvoy" horses mabinogion)
    (make-student "Philip Farmer" corn beans horses)))
```

You can evaluate the following forms to test what you have done so far. A printed representation of each object will be displayed.

```
(find-student "Philip Farmer")
=>#<STUDENT Philip Farmer>

(find-course "The Cultivation and Uses of Beans")
=>#<COURSE The Cultivation and Uses of Beans (Agriculture)>

(find-department "Agriculture")
=>#<DEPARTMENT Agriculture
```

If you try to evaluate a form that has not yet been defined (for example, if you try to look up a student that ''doesn't exist''), you might see something like this:

```
Command: (find-student "Jill Parker")
Error: There is no student named "Jill Parker"

FIND-STUDENT
   Arg 0 (NAME): "Jill Parker"
   --Defaulted args:--
   Arg 1 (ERRORP): T
s-A, <ABORT>:    Return to Breakpoint ZMACS in Editor Typeout Window 1
s-B:             Editor Top Level
s-C:             Restart process Zmacs Windows
→
```

Now we are ready to develop a user interface. This first example defines presentations of students, represented by their names. This simple presentation type does not provide parameters or options. A real program would also provide presentation types for courses and departments, but this example shows students only.

```
(clim:define-presentation-type student ())

(clim:define-presentation-method clim:present
                                 (student (type student) stream
                                  (view clim:textual-view) &key)
  (write-string (student-name student) stream))

(clim:define-presentation-method clim:accept
                                 ((type student) stream
                                  (view clim:textual-view) &key)
  (let* ((token (clim:read-token stream))
         (student (find-student token nil)))
    (when student
      (return-from clim:accept student))
    (clim:input-not-of-required-type token type)))
```

Test this by evaluating the following forms in a CLIM Lisp Listener. Note that there is no completion and **find-student** is case-sensitive, so the student's name must be entered exactly to be accepted.

```
(clim:describe-presentation-type 'student *standard-output*)
(clim:describe-presentation-type 'student *standard-output* 5)
(clim:present (find-student "Philip Farmer") 'student)
(clim:accept 'student :default (find-student "Philip Farmer"))
```

We can improve the input interface by using completion over elements of ***student-table***.

```
(clim:define-presentation-method clim:accept
                                 ((type student) stream
                                  (view textual-view) &key)
  (values               ;suppress values after the first
    (clim:completing-from-suggestions
        (stream :partial-completers '(#\space))
      ;; SUGGEST takes arg of name, object
      (maphash #'clim:suggest *student-table*))))
```

Test this by evaluating the following form in a CLIM Lisp Listener.

```
(clim:accept 'student :default (find-student "Philip Farmer"))
```

Try the `Help` key, and try entering just the initials of a student, separated by a space; they complete to the full name.

It would be useful to be able to select students in a particular department. We can revise the presentation type for **student** by adding a parameter for the department. A student is in a department if the student is taking any course in that department.

```
(defun student-in-department-p (student department)
  (find department (student-courses student) :key #'course-department))

(clim:define-presentation-type student (&optional department))
```

When a presentation type has parameters, the defaults for the **clim:presentation-typep** and **clim:presentation-subtypep** presentation methods are not sufficient. Therefore, we need to define these presentation methods. We also define a new **clim:describe-presentation-type** method.

```
(clim:define-presentation-method clim:presentation-typep
                                 (object (type student))
  (or (eq department '*)
      (student-in-department-p object department)))
```

```
(clim:define-presentation-method clim:presentation-subtypep
                                 ((type1 student) type2)
  (let ((department1 (clim:with-presentation-type-parameters
                       (student type1) department))
        (department2 (clim:with-presentation-type-parameters
                       (student type2) department)))
    (values (or (eq department1 department2)
                (eq department2 '*))
            t)))

(clim:define-presentation-method clim:describe-presentation-type
                                 ((type student) stream plural-count)
  (when (eql plural-count 1)
    (write-string (if (or (eq department '*)
                          (not (find (char (department-name department) 0) "aeiou"
                                     :test #'char-equal)))
                      "a "
                      "an ")
                  stream))
  (format stream
      (if (and (integerp plural-count) (> plural-count 1))
          (if (eq department '*) "~R student~:P" "~R ~A student~:*~:P")
          (if (eq department '*) "student~P" "~*~A student~:*~:P"))
    (typecase plural-count
      (integer plural-count)
      (null 1)
      (otherwise 2))
    (unless (eq department '*)
      (department-name department))))
```

Evaluate the following forms to test these methods:

```
(clim:presentation-typep (find-student "Philip Farmer")
                         `(student ,(find-department "Agriculture")))

(clim:presentation-typep (find-student "Philip Farmer")
                         `(student ,(find-department "English")))

(clim:presentation-typep "Philip Farmer"
                         `(student ,(find-department "Agriculture")))

(clim:presentation-subtypep `(student ,(find-department "Agriculture"))
                            `(student *))

(clim:presentation-subtypep `(student ,(find-department "Agriculture"))
                            `(student ,(find-department "English")))
```

```
(clim:describe-presentation-type '(student ,(find-department "Physics")))

(clim:describe-presentation-type '(student *))
```

The existing method for **clim:accept** suggests all the students, even the ones in the wrong department, so we provide the following **:around** method to check that we are returning a student in the right department.

```
(clim:define-presentation-method clim:accept :around
                                 ((type student) stream view &key)
  (declare (ignore stream view))
  (multiple-value-bind (object actual-type)
      (call-next-method)
    (unless (clim:presentation-typep object type)
      (clim:input-not-of-required-type object type))
    (values object actual-type)))
```

Evaluate the following form in a CLIM Lisp Listener before and after defining the above method.

```
(clim:accept '(student ,(find-department "Agriculture")))
```

Type the following at the prompt:

```
susan RETURN
```

Before defining the above method, **clim:accept** returns a student that is not in the specified department. After defining the above method, CLIM asks the user to try again. But if you press `HELP`, Susan Charnas is still listed as one of the possibilities.

Another way to do this would be to filter out students in other departments before calling **clim:suggest.** To do that, define this method instead of the preceding method. This way works better because the completion possibilities won't include any extra students.

```
(clim:define-presentation-method clim:accept
                                 ((type student) stream
                                  (view clim:textual-view) &key)
  (values                       ;suppress values after the first
    (clim:completing-from-suggestions
        (stream :partial-completers '(#\space))
      (maphash (if (eq department '*)
                   #'clim:suggest
                   #'(lambda (name student)
                       (when (student-in-department-p student department)
                         (clim:suggest name student))))
               *student-table*))))
```

On Genera, this gets rid of the **:around** method that we don't want any more. (On other platforms, you can use **clos:remove-method**.)

```
(scl:fundefine '(clos:method clim:accept-method (student t t t t t) :around))
```

Evaluate these forms in the CLIM Lisp Listener again. Try entering the names of agricultural and non-agricultural students.

```
(clim:accept `(student ,(find-department "Agriculture")))
(clim:accept `student)
```

You can also try the HELP key.

It is easy to define an abbreviation for a presentation type. Here we define **aggie** as an abbreviation for a student in the Agriculture department:

```
(clim:define-presentation-type-abbreviation aggie ()
  `(student ,(find-department "Agriculture")))
```

Evaluate these forms to test it.

```
(clim:describe-presentation-type 'aggie)
(clim:accept 'aggie)
```

Now we refine our example by providing an option that controls the printing of the student's name.

```
(clim:define-presentation-type student (&optional department)
  :options (last-name-first))

(clim:define-presentation-method clim:present
                                 (student (type student) stream
                                  (view clim:textual-view) &key)
  (let* ((name (student-name student))
         (index (and last-name-first (position #\space name :from-end t))))
    (cond ((null index)
           (write-string name stream))
          (t
           (write-string name stream :start (1+ index))
           (write-string ", " stream)
           (write-string name stream :end index)))))
```

```
(clim:define-presentation-method clim:accept
                                 ((type student) stream
                                  (view clim:textual-view) &key)
  (values                       ;suppress values after the first
    (clim:completing-from-suggestions
        (stream :partial-completers '(#\space #\,))
      (maphash #'(lambda (name student)
                   (when (clim:presentation-typep student type)
                     (clim:suggest
                       (or (and last-name-first
                                (let ((index (position #\space name
                                                       :from-end t)))
                                  (and index
                                       (concatenate 'string
                                                    (subseq name (1+ index))
                                                    ", "
                                                    (subseq name 0 index)))))
                           name)
                       student)))
               *student-table*))))
```

Evaluate these forms to test it.

```
(clim:present (find-student "Philip Farmer") 'student)

(clim:present (find-student "Philip Farmer") '((student) :last-name-first t))

(clim:accept `((student) :last-name-first t))

(clim:accept `((student ,(find-department "Physics")) :last-name-first t))
```

Since presentation type options are not automatically inherited by subtypes and abbreviations, the following example doesn't work.

```
(clim:accept '((aggie) :last-name-first t))
```

This example works if you redefine **aggie** to accept the **:last-name-first** option:

```
(clim:define-presentation-type-abbreviation aggie ()
  `((student ,(find-department "Agriculture"))
    :last-name-first ,last-name-first)
  :options (last-name-first))
```

You can override the presentation type's description:

```
(clim:accept `((student ,(find-department "English"))
               :description "English major"))
```

CLIM Operators for Defining New Presentation Types

clim:define-presentation-type *name parameters* &key *:options :inherit-from :description :history :parameters-are-types*
Defines a CLIM presentation type.

clim:define-presentation-method *presentation-function-name [qualifiers]* *specialized-lambda-list* &body *body*
Defines a presentation method for the function named *presentation-function-name* on the presentation type named in *specialized-lambda-list*.

For the names and lambda-lists of CLIM presentation methods, see the section "Presentation Methods in CLIM".

Under rare circumstances, you may wish to define or call a new presentation generic function. The following forms may be used to accomplish this.

clim:define-presentation-generic-function *generic-function-name presentation-function-name lambda-list* &rest *options*
Defines a new presentation named *presentation-function-name* whose methods are named by *generic-function-name*. *lambda-list* and *options* are as for **clos:defgeneric**.

clim:define-default-presentation-method *presentation-function-name [qualifiers]* *specialized-lambda-list* &body *body*
This is like **clim:define-presentation-method**, except that it is used to define a default method that will be used if there are no more specific methods.

clim:funcall-presentation-generic-function *presentation-function-name* &body *arguments*
Funcalls the presentation generic function *presentation-function-name* with arguments *arguments* using **funcall**.

clim:apply-presentation-generic-function *presentation-function-name* &body *arguments*
Applies the presentation generic function *presentation-function-name* to arguments *arguments* using **apply**.

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.

clim:define-presentation-type-abbreviation *name parameters expansion* &key *:options*
Defines a presentation type that is an abbreviation for the presentation type specifier that is the value of *expansion*.

This example defines a presentation type to read an octal integer:

```
(clim:define-presentation-type-abbreviation octal-integer
                                            (&optional low high)
  `((integer ,low ,high) :base 8 :description "octal integer"))
```

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.

clim:expand-presentation-type-abbreviation *type* &optional *environment*
: **clim:expand-presentation-type-abbreviation** is like **clim:expand-presentation-type-abbreviation-1**, except that *type* is repeatedly expanded until all presentation type abbreviations have been expanded.

clim:expand-presentation-type-abbreviation-1 *type* &optional *environment*
: If the presentation type specifier *type* is a presentation type abbreviation, or is an **and**, **or**, **sequence**, or **clim:sequence-enumerated** that contains a presentation type abbreviation, then **clim: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.

clim:make-presentation-type-specifier *type-name-and-parameters* &rest *options*
: Given a presentation type name and its parameters *type-name-and-parameters* and some presentation type options, make a new presentation type specifier that includes all of the type parameters and options.

Presentation Methods in CLIM

You define presentation methods using **clim:define-presentation-method**.

clim:define-presentation-method *presentation-function-name [qualifiers]* specialized-lambda-list* &body *body*
: Defines a presentation method for the function named *presentation-function-name* on the presentation type named in *specialized-lambda-list*.

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 **clim:presentation-subtypep** have lexical access to the parameters from the presentation type specifier. Presentation methods for the functions **clim:accept**, **clim:present**, **clim:describe-presentation-type**, **clim:presentation-type-specifier-p**, and **clim: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. The reason presentation methods are not exactly the same as ordinary CLOS methods revolves around 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.

Here are the names of the various presentation methods defined by **clim:define-presentation-method** , along with the lambda-list for each method. For a complete description of these presentation methods,see the section "Dictionary of CLIM Operators".

clim:accept *type-key parameters options type stream view* &key *:default :default-type* &allow-other-keys
The presentation method responsible for ''parsing'' the representation of *type* for a particular *view*.

clim:present *type-key parameters options object type stream view* &key *:acceptably :for-context-type*
The presentation method responsible for displaying the representation of *object* having type *type* for a particular view *view*.

clim:describe-presentation-type *type-key parameters options type stream plural-count*
The presentation method that is responsible for textually describing the type *type*.

clim:default-describe-presentation-type *description stream plural-count*
Given a string *description* that describes a presentation type (such as ''integer'') and *plural-count* (either **nil** or an integer), this function pluralizes the string if necessary, prepends an indefinite article if appropriate, and outputs the result onto *stream*.

clim:presentation-typep *type-key parameters object type*
The presentation method called when the **clim:presentation-typep** function requires type-specific knowledge.

clim:presentation-subtypep *type-key type putative-supertype*
The presentation method called when the **clim:presentation-subtypep** function requires type-specific knowledge.

clim:presentation-type-specifier-p *type-key parameters options type*
The presentation method that is responsible for checking the validity of the parameters and options.

clim:accept-present-default *type-key parameters options type stream view default default-supplied-p present-p query-identifier* &key *(:prompt* **t***) (:active-p* **t***)* &allow-other-keys
The presentation method called when **clim:accept** turns into **clim:present** inside of **clim:accepting-values**.

clim:highlight-presentation *type-key parameters options type record stream state*
The presentation method responsible for drawing a highlighting box for the presentation *record* on the stream *stream*.

Utilities for clim:accept Presentation Methods

The utilities documented in this section are typically useful with **clim:accept** (and sometimes **clim:present**) presentation methods.

The following two functions are used to read or write a token (that is, a string):

clim:read-token *stream* &key *:timeout :input-wait-handler :pointer-button-press-handler :click-only*
Reads characters from *stream* until it encounters an activation gesture, a delimiter gesture, or a pointer gesture. Returns the accumulated string that was delimited by an activation or delimiter gesture, leaving the delimiter unread.

clim:write-token *token stream* &key *:acceptably*
Given the string *token*, **clim:write-token** writes it to the stream *stream*.

Sometimes, an **clim:accept** method may wish to signal an error while it is parsing the user's input, or a nested call to **clim:accept** may signal such an error itself. The following functions and conditions may be used:

clim:simple-parse-error *format-string* &rest *format-arguments*
Signals an error of type **clim:simple-parse-error** while parsing an input token. This function does not return.

clim:input-not-of-required-type *object type*
Reports that input does not satisfy the specified type.

clim:simple-parse-error
This condition is signalled when CLIM does not know how to parse some sort of user input while inside of **clim:accept**.

clim:input-not-of-required-type
This condition is signalled when CLIM gets input that does not satisfy the specified type while inside of **clim:accept**.

Some **clim:accept** methods will want to allow for the completion of partial input strings by the user. The following functions are useful for doing that:

clim:complete-input *stream function* &key *:partial-completers :allow-any-input :possibility-printer (:help-displays-possibilities* **t***)*
Reads input from *stream*, completing from a set of possibilities.

clim:complete-from-generator *string generator delimiters* &key *(:action* **:complete***) :predicate*
Given an input string *string* and a list of delimiter characters *delimiters* that act as partial completion characters, **clim:complete-from-generator** completes against the possibilities that are generated by the function *generator*.

clim:complete-from-possibilities *string completions delimiters* &key *(:action* **:complete***) :predicate (:name-key* **#'first***) (:value-key* **#'second***)*
Given an input string *string* and a list of delimiter characters *delimiters* that act as partial completion characters, **clim:complete-from-possibilities** completes against the possibilities in the sequence *completions*.

clim:completing-from-suggestions *(stream* &rest *options* &key *:partial-completers :allow-any-input :possibility-printer (:help-displays-possibilities* **t***))* &body *body*
Reads input from *stream*, completing from a set of possibilities generated by calls to **clim:suggest** in *body*. Returns three values: *object*, *success*, and *string*.

clim:suggest *name* &rest *objects*
Specifies one possibility for **clim:completing-from-suggestions**. *completion* is a string, the printed representation. *object* is the internal representation. This function has lexical scope and is defined only inside the body of **clim:completing-from-suggestions**.

clim:*completion-gestures*
A list of gesture names that cause **clim:complete-input** to complete the input as fully as possible.

clim:*possibilities-gestures*
A list of gesture names that cause **clim:complete-input** to display a help message and the list of possibilities.

clim:*help-gestures*
A list of gesture names that cause **clim:accept** and **clim:complete-input** to display a help message, and, for some presentation types, the list of possibilities.

Sometimes after an **clim: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:

clim:replace-input *stream new-input* &key *:start :end :rescan :buffer-start*
Replaces *stream*'s input buffer with the string *new-input*.

clim:presentation-replace-input *stream object type view* &key *:rescan :buffer-start*
Like **clim:replace-input**, except that the new input to insert into the input buffer is gotten by presenting the object *object* with the presentation type *type* and view *view*.

For example, the following **clim:accept** method reads a token followed by a ''system'' or a pathname, but if the user clicks on either a ''system'' or a pathname, it inserts that object into the input buffer and returns:

```
(clim:define-presentation-method clim:accept
                                 ((type library) stream (view clim:textual-view)
                                  &key default)
  (clim:with-input-context ('(or system pathname)) (object type)
       (let ((system (clim:accept '(clim:token-or-type (:private) system)
                                  :stream stream :view view
                                  :prompt nil :display-default nil
                                  :default default
                                  :additional-delimiter-gestures '(#\space)))
             file)
         (let ((char (clim:read-gesture :stream stream)))
           (unless (eql char #\space)
             (clim:unread-gesture char :stream stream))
           (when (eql system ':private)
             (setq file (clim:accept 'pathname
                                     :stream stream :view view
                                     :prompt "library pathname"
                                     :display-default t)))
           (if (eql system ':private) file system)))
       (t (clim:presentation-replace-input stream object type view)
          (values object type))))
```

Occasionally, **clim:accept** methods will want to change the conditions under which input fields (or the entire input line) should be terminated. The following macros are useful for this:

clim:with-activation-gestures *(additional-gestures* &key *:override)* &body *body*
: Specifies gestures that terminate input during the evaluation of *body*. *additional-gestures* is a gesture spec or a form that evaluates to a list of gesture specs.

clim:with-delimiter-gestures *(additional-gestures* &key *:override)* &body *body*
: Specifies gestures that terminate an individual token but not the entire input sentence during the evaluation of *body*. *additional-gestures* is a gesture spec or a form that evaluates to a list of gesture specs.

clim:*standard-activation-gestures*
: A list of gesture names that cause the current input to be activated.

clim:accept tries to generate meaningful help messages based on the name of the presentation type, but sometimes this is not adequate. You can use **clim:with-accept-help** to create more complex help messages.

clim:with-accept-help *options* &body *body*
: Binds the local environment to control `HELP` and `c-?` documentation for input to **clim:accept**.

Here are some examples of the use of **clim:with-accept-help**:

```
(clim:with-accept-help ((:subhelp "This is a test."))
  (clim:accept 'pathname))

==> You are being asked to enter a pathname. [ACCEPT did this for you]
    This is a test.                          [You did this via :SUBHELP]

(clim:with-accept-help ((:top-level-help "This is a test."))
  (clim:accept 'pathname))

==> This is a test.                          [You did this via :TOP-LEVEL-HELP

(clim:with-accept-help (((:subhelp :override) "This is a test."))
  (clim:accept 'pathname))

==> You are being asked to enter a pathname. [ACCEPT did this]
    This is a test.                          [You did this via :SUBHELP]

(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:
```

clim:accept uses the input editor to read textual input from the user. If you want an **clim:accept** method to do any sort of typeout, you must coordinate it with the input editor via the **clim:with-input-editor-typeout** macro. The input editor is discussed in more detail in "The Structure of the CLIM Input Editor".

clim:with-input-editing *(*&optional *stream* &key *:input-sensitizer :initial-contents :class)* &body *body*
Establishes a context in which the user can edit the input he or she types in on the stream *stream*. *body* is then evaluated in this context, and the values returned by *body* are returned as the values of **clim:with-input-editing**.

clim:with-input-editor-typeout *(*&optional *stream* &key *:erase)* &body *body*
If, when you are inside of a call to **clim:with-input-editing**, you want to perform some sort of typeout, it should be done inside **clim:with-input-editor-typeout**.

clim:input-editor-format *input-editing-stream format-string* &rest *format-args*
This function is like **format**, except that it is intended to be called on input editing streams. It arranges to insert ‘‘noise strings’’ in the input editor’s input buffer.

Input Editing and Built-in Keystroke Commands in CLIM

This is a list of the keystrokes that are built into CLIM's input editor, the specific keys assigned in Genera and in Cloe, and how to change them. For more detail on the input editor, see the section "The Structure of the CLIM Input Editor".

Activation Gestures

Activation gestures terminate an input ''sentence'', such as a command or anything else being read by **clim:accept**. When you enter an activation gesture, CLIM ceases reading input and executes the input that has been entered.

The default activation gesture is `#\Newline` (also known as `#\Return`). On Genera, `#\End` is also an activation gesture.

clim:with-activation-gestures	Macro
:activation-gestures to **clim:accept**	Option
:additional-activation-gestures to **clim:accept**	Option
clim:*standard-activation-gestures*	Variable
clim:activation-gesture-p	Function

Delimiter Gestures

Delimiter gestures terminate an input ''word'', such as a recursive call to **clim:accept**. There are no global default delimiter gestures; each presentation type that recursively calls **clim:accept** specifies its own delimiter gestures and sometimes offers a way to change them (see Command Processor Gestures, below).

Delimiter gestures most commonly occur in command lines. When you type a delimiter gesture, CLIM's command processor moves on to read the next field in the command line.

clim:with-delimiter-gestures	Macro
:delimiter-gestures to **clim:accept**	Option
:additional-delimiter-gestures to **clim:accept**	Option
:separator to **clim:subset-completion**	Presentation Type Option
:separator to **clim:subset**	Presentation Type Option
:separator to **clim:subset-sequence**	Presentation Type Option
:separator to **clim:subset-alist**	Presentation Type Option
clim:*delimiter-gestures*	Variable
clim:delimiter-gesture-p	Function

Abort Gestures

When an application reads an abort gesture while looking for input, CLIM signals the **conditions:abort** restart. Aborting is caught by **clim:default-frame-top-level**, which will abort what the application frame is doing and read another command.

The default abort gesture is `#\Abort` in Genera. In Cloe the default abort gesture is `#\Escape` or `#\Esc`. (Note that this cannot be changed on Cloe).

Suspend Gestures

In Cloe, `c-C` is similar to `c-Suspend` in Genera. Its difference is that it causes a break in Lisp execution and you can then abort to top level, examine the stack, or continue execution.

Completion Gestures

Several presentation types, such as **member** and **pathname**, support completion of partial inputs. When accepting input of one of these types, completion gestures and possibilities gestures can be entered. A completion gesture completes the input that has been entered so far; if there is more than one possible completion, CLIM completes it as much as possible. A possibilities gesture causes CLIM to display the possible completions of the input that has been entered so far.

The default completion gesture is `#\Tab`. On Genera, `#\Complete` is also a completion gesture.

The default possibilities gesture is `#\control-?` in Genera, and the `F1` function key for Cloe. You can also click the right-hand button on the pointer to get a menu of possibilities.

On Genera, `#\Help` is also a possibilities gesture.

clim:*completion-gestures*	Variable
clim:*possibilities-gestures*	Variable
clim:*help-gestures*	Variable

Command Processor Gestures

A command dispatcher character introduces a command (rather than a form) in the **clim:command-or-form** presentation type. The default command dispatcher is `Colon` (:).

The default character for both terminating and completing command names is `#\Space`. This is a delimiter gesture while reading a command name.

The default character for terminating command arguments is `#\Space`. This is a delimiter gesture while reading an argument to a command.

Pressing a command previewer gesture while entering a command allows a command to be entered via a dialog instead of the usual command line. This is an activation gesture while reading a command. On Genera, the default command previewer gestures is `#\m-Complete`. On Cloe there is none.

clim:*command-dispatchers*	Variable

Input Editor Commands

Keyboard input to **clim:accept** can be edited until an activation gesture is typed to terminate it. After an activation gesture is entered, if CLIM cannot parse the input, the user must edit and re-activate it. The input editor has a number of Emacs-like keystroke commands, described in the table below. Prefix numeric argu-

ments to input editor commands can be entered using digits and minus sign (-) with control, meta, super, or hyper (as in Genera and Emacs).

The function **clim: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.

Command	*Genera Key*	*Cloe Key*
Forward character	control-F	control-F Right Arrow
Forward word	meta-F	meta-F control-Right Arrow
Forward sexp	c-m-F	c-m-F
Backward character	control-B	control-B Left Arrow
Backward word	meta-B	meta-B control-Left Arrow
Backward sexp	c-m-B	c-m-B
Beginning of line	control-A	control-A Home
End of line	control-E	control-E End
Next line	control-N	control-N Down Arrow
Previous line	control-P	control-P Up Arrow
Beginning of buffer	meta-<	meta-< control-HOME
End of buffer	meta->	meta->> control-END
Delete character	control-D	control-D
Delete word	meta-D	meta-D
Delete sexp	c-m-D	c-m-D
Rubout character	Rubout	Rubout
Rubout word	meta-Rubout	meta-Rubout
Rubout sexp	c-m-Rubout	c-m-Rubout
Kill line	control-K	*none*
Clear all input	Clear Input	Delete
Insert new line	control-O	*none*
Insert parens	control-(	*none*
Transpose characters	control-T	control-T
Transpose words	meta-T	meta-T
Transpose sexps	c-m-T	c-m-T
Upcase word	meta-U	meta-U
Downcase word	meta-L	meta-L

Capitalize word	meta-C	meta-C
Show arglist	c-sh-A	c-sh-A
Show variable value	c-sh-V	c-sh-V
Show doc string	c-sh-D	c-sh-D
Yank from kill ring	control-Y	control-Y
Yank from history	c-m-Y	c-m-Y
Yank next thing	meta-Y	meta-Y
Scroll forward	control-V Scroll	control-V
Scroll backward	meta-V meta-Scroll	meta-V

Dialog Commands

An **clim:accepting-values** dialog supports accelerators for exiting and aborting out of dialogs. The key bindings can be changed using **clim:add-keystroke-to-command-table** and **clim:remove-keystroke-from-command-table** in the usual way.

Command	*Genera Key*	*Cloe Key*
Abort the dialog	Abort	Escape
Exit from the dialog (assuming you are not editing a field)	End	RETURN

Menu Commands

At present there are no special keystroke commands for menus.

Using Views with CLIM Presentation Types

The **clim:present** and **clim:accept** presentation methods can specialize on the *view* 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. For example, the **clim:present** method for the **clim:textual-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 sort of ‘‘radio pushbutton’’ style.

CLIM currently supports ‘‘default’’, menu, and dialog views, in both textual and gadget styles. Some views act as ‘‘indirect’’ views that are decoded into a more specific view; this typically arises for the gadget views.

Operators for Views of CLIM Presentation Types

The following two functions control what view should be used by default on a stream, or for any dialog being managed by a particular frame manager.

clim:stream-default-view *stream*
: Returns the default view for the stream *stream*. You can change the default view for a stream by using **setf** on **clim:stream-default-view**. Calls to **clim:accept** default the **:view** argument from **clim:stream-default-view**.

clim:frame-manager-dialog-view *frame-manager*
: Returns the view object that should be used to control the look-and-feel of **clim:accepting-values** dialogs.

The following classes and constants are all of the predefined ''indirect'' views (that is, views that might be translated into another view depending on the presentation type). Normally, the textual views are not indirected to any other views, and CLIM will just use some sort of textual representation for all of the presentation types that use a textual view. The gadget dialog view is usually indirected to another view, for instance, the **member** type indirects to **clim:+radio-box-view+**.

clim:textual-view
: The class that represents textual views. Textual views are used in most command-line oriented applications.

clim:textual-menu-view
: The class that represents the view that is used inside textual menus.

clim:textual-dialog-view
: The class that represents the view that is used inside textual **clim:accepting-values** dialogs.

clim:+textual-view+
: An instance of the class **clim:textual-view**.

clim:+textual-menu-view+
: An instance of the class **clim:textual-menu-view**. Inside **clim:menu-choose**, the default view for the menu stream may be bound to **clim:+textual-menu-view+**.

clim:+textual-dialog-view+
: An instance of the class **clim:textual-dialog-view**. Inside **clim:accepting-values**, the default view for the dialog stream may be bound to **clim:+textual-dialog-view+**.

clim:gadget-view
: The class that represents gadget views. Gadgets views are used for toolkit-oriented applications.

clim:gadget-menu-view
: The class that represents the view that is used inside toolkit-style menus.

clim:gadget-dialog-view
The class that represents the view that is used inside toolkit-style **clim:accepting-values** dialogs.

clim:+gadget-view+
An instance of the class **clim:gadget-view**.

clim:+gadget-menu-view+
An instance of the class **clim:gadget-menu-view**. Inside **clim:menu-choose**, the default view for the menu stream may be bound to **clim:+gadget-menu-view+**.

clim:+gadget-dialog-view+
An instance of the class **clim:gadget-dialog-view**. Inside **clim:accepting-values**, the default view for the dialog stream may be bound to **clim:+gadget-dialog-view+**.

The following is a table of presentation types and the actual view they map to:

Type	*Gadget*
clim:completion	**clim:+radio-box-view+**
clim:subset-completion	**clim:+check-box-view+**
clim:boolean	**clim:+toggle-button-view+**
real	**clim:+slider-view+**
float	**clim:+slider-view+**
integer	**clim:+slider-view+**
All others	**clim:+text-field-view+**

Functions that Operate on CLIM Presentation Types

These are some general-purpose functions that operate on CLIM presentation types.

clim:describe-presentation-type *presentation-type* &optional *(stream* ***standard-output****) (plural-count* **1***)*
Describes the *presentation-type* on the *stream*.

clim:presentation-typep *object type*
Returns **t** if *object* is of the type specified by *type*, otherwise returns **nil**.

clim:presentation-type-of *object*
Returns the presentation type of the object *object*.

clim:presentation-subtypep *type putative-supertype*
Answers the question ''Is the type specified by *type* a subtype of the type specified by *putative-supertype*?''.

clim:with-presentation-type-decoded *(name-var* &optional *parameters-var options-var) type* &body *body*
The specified variables are bound to the components of the presentation type specifier, the forms in *body* are evaluated, and the values of the last form are returned.

clim:with-presentation-type-options *(type-name type)* &body *body*

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 evaluated in the scope of these variables and the values of the last form are returned.

clim:with-presentation-type-parameters *(type-name type)* &body *body*

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.

Presentation Translators in CLIM

Concept of Presentation Translators in CLIM

CLIM provides a mechanism for translating between types. In other words, within an input context for presentation type A, the translator mechanism allows a programmer to define a translation from presentations of some other type B to objects that are 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 presented object, such as a student. In the absence of a presentation translator, the student is not sensitive because the user must enter a command and cannot enter anything else to this input context.

In the presence of a presentation translator that translates from students to commands, however, the presented student would be sensitive. In one scenario, the presented student is highlighted, and the middle pointer button does ''Show Transcript'' of that student.

A presentation translator defines how to translate from one presentation type to another. In the scenario above, the input context is **clim:command**. A user-defined presentation translator states how to translate from the **student** presentation type to the **clim:command** presentation type.

The concept of translating from an arbitrary presentation type to a command is so useful that CLIM provides a special macro for this purpose, **clim:define-presentation-to-command-translator**. 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.

Presentation-to-command translators make it easier to write applications that give a ''direct manipulation'' feel to the user.

What Controls Sensitivity in CLIM?

A presentation that appears on the screen can be *sensitive*. This means that the presentation can be operated on directly by using the pointer. In other words, the presentation is relevant to the current context. When the user moves the pointer over a sensitive presentation, the presentation is highlighted to indicate that it is sensitive. (In rare cases, the highlighting of some sensitive presentations is turned off.)

Sensitivity is controlled by three factors: the current input context, the location of the pointer, and the chord of modifier keys being pressed.

- Input context type — a presentation type describing 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 — these are control, meta, super, hyper, and shift. These keys expand the space of available gestures beyond what is available from the pointer buttons. Note that some platforms might not provide any way to input all of the modifier keys, but most provide at least control, meta, and shift.

Presentation translators are the link among 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 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. The basic idea is that a presentation translator translates an output presentation into an input presentation. Thus a presentation translator is applicable if the type of the presentation at the pointer ''matches'' *from-presentation-type* and the input context type ''matches'' *to-presentation-type*. (We define what ''match'' means below.) 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.

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 **clim:accept**, or 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 *to-presentation-type* if the body returns only one value.

Applicability of CLIM Presentation Translators

When CLIM is waiting for input (that is, inside a **clim: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.

clim:with-input-context uses **clim:frame-find-innermost-applicable-presentation** (via **clim:highlight-applicable-presentation**) as its ''input wait'' handler, and **clim:frame-input-context-button-press-handler** as its button press ''event handler''.

Given a presentation, an input context established by **clim: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 the section "Command Objects in CLIM".

A translator ''matches'' if all of the following are true. These tests are performed in the order listed.

1. The presentation's type is **clim: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**).
2. The translator's *to-presentation-type* is **clim:presentation-subtypep** of the input context type, ignoring type parameters.
3. 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.
4. If the *from-presentation-type* has parameters, the presentation's object is **clim:presentation-typep** of the translator's *from-presentation-type*.
5. The translator's tester returns a non-**nil** value. If there is no tester, the translator behaves as though the tester always returns **t**.
6. 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 **clim:presentation-typep** of the input context type. In **clim:define-presentation-to-command-translator** and **clim:define-presentation-action** the tester is always taken to be definitive.

The algorithm is somewhat more complicated in the face of nested presentations and nested input contexts. In this case, the applicable presentation is the *smallest* presentation that matches the *innermost* input context (that is, translators matching inner contexts precede translators matching outer contexts, and, in the same input context, inner presentations precede outer presentations).

Sometimes there may be nested presentations that have exactly the same bounding rectangle. In this case, it is not possible for a user to unambiguously point to just one of the nested presentations. Therefore, when CLIM has located the innermost applicable presentation in the innermost input context, it then searches for outer presentations that have exactly the same bounding rectangle, and checks to see if there are any applicable translators for those presentations. If there are multiple applicable translators, CLIM chooses the one having the highest priority.

There can be more than one translator that matches a presentation for the same gesture in a given input context. When this happens, the first translator is chosen, based on the following ordering:

1. Translators with a higher ''high order'' priority precede translators with a lower ''high order'' priority. This allows you to create an ''overriding'' translator that always precedes any other applicable translators.
2. Translators with a more specific *from-presentation-type* precede translators with a less specific *from-presentation-type*.
3. Translators with a higher ''low order'' priority precede translators with a lower ''low order'' priority. This allows you 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.

See the description of the **:priority** option in **clim:define-presentation-translator**.

Input Contexts in CLIM

Roughly speaking, the current *input context* indicates what type of input CLIM is currently asking the user for. These are the ways you can establish an input context in CLIM:

clim:accept

clim:accept-from-string

The command loop of an application

Nested Input Contexts in CLIM

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. It 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 **clim:accept**. The macro **clim: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 se-

quence of pathnames. When reading the argument to the Show File command, the input context contains **pathname** nested inside of **(sequence pathname)**. Acceptable keyboard input is a sequence of pathnames separated by commas. A presentation translator that translates to a **(sequence 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 entry of a SPACE or RETURN to terminate the argument.

When the input context is nested, sensitivity computations consider only the innermost context type that has any applicable presentation translators for the currently pressed chord of modifier keys.

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.

Gestures and Gesture Names in CLIM

A *gesture* in CLIM 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.

A *gesture spec* is a portable way of naming a gesture. For example, the nonportable "character" `#\control-shift-C` has a gesture spec of `(:C :control :shift)`. For convenience, the gesture spec for standard Common Lisp characters (the printing characters, including alphanumerics, ASCII symbols, and `#\Space`), you can use the character itself as the gesture spec.

An *event* is a CLIM object that represents a gesture by the user. (The most important pointer events are those of class **clim:pointer-button-event**.)

A *gesture name* is a symbol that names a gesture or gesture spec. CLIM defines the following gesture names:

:select 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 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.

:delete For translators that delete an object.

:edit For translators that edit an object.

:modify For translators that somehow modify an object.

:menu For translators that pop up a menu.

These correspond to the following *events* in Genera and in Cloe:

Gesture Name	*Genera Gesture*	*Cloe Gesture*
:select	click `Left`	click `Left`
:describe	click `Middle`	click `Middle`
		click `A-Right`
:menu	click `Right`	click `Right`
:delete	click `sh-Middle`	
:edit	click `m-Left`	
:modify	click `c-m-Right`	

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 **clim:define-gesture-name** to define your own gesture names. Avoid the temptation to define pointer gestures named **:left**, **:middle**, and **:right**; doing so can lead you to write less portable applications. If your program use only gesture names, they are more portable than if you to specific pointer buttons and keyboard keys.

Operators for Gestures in CLIM

The following operators can be used to add or remove new gesture names:

clim:add-gesture-name *name type gesture-spec* &key *(:unique* **t***)*

Adds a gesture named *name* (a symbol) to the set of all gesture names. If *:unique* is **t**, an error is signalled if there is already a gesture named *name*.

clim:delete-gesture-name *gesture-name*
Removes the gesture named *gesture-name*.

clim:define-gesture-name *name type gesture-spec* &key *(:unique* **t***)*
Defines a gesture named *name* by calling **clim:add-gesture-name**.

The following operators can be used to examine CLIM event objects or match CLIM event objects against gesture names.

clim:event-sheet *event*
Returns the window on which *event* occurred.

clim:event-modifier-state *event*
Returns the state of the keyboard's shift keys when the event *event* occurred.

clim:pointer-event-button *pointer-button-event*
Returns the button number that was pressed when the pointer button event *pointer-button-event* occurred. The values this can take are **clim:+pointer-left-button+**, **clim:+pointer-middle-button+**, or **clim:+pointer-right-button+**.

clim:pointer-event-x *pointer-event*
Returns the X position of the pointer when the *pointer-event* occurred.

clim:pointer-event-y *pointer-event*
Returns the Y position of the pointer when the *pointer-event* occurred.

clim:keyboard-event-key-name *keyboard-event*
Returns the name of the key that was pressed or released in order to generate the keyboard event.

clim:keyboard-event-character *keyboard-event*
Returns the character corresponding to the key that was pressed or released, if there is a corresponding character.

clim:event-matches-gesture-name-p *event gesture-name* &optional *port*
Returns **t** if the device event *event* ''matches'' the gesture named by *gesture-name*.

clim:modifier-state-matches-gesture-name-p *state gesture-name*
Returns **t** if the modifier state *state* ''matches'' the modifier state of the gesture named by *gesture-name*.

clim:make-modifier-state &rest *modifiers*
Given a set of modifier key names, **clim:make-modifier-state** returns a modifier state corresponding to those keys.

The following constants are the values that can be taken on by **clim:event-modifier-state**. Note that these are bit values that can be combined with ''logical or'' when multiple modifier keys are being held down by the user.

clim:+shift-key+
The modifier state bit that corresponds to the user holding down the shift key on the keyboard.

clim:+control-key+
The modifier state bit that corresponds to the user holding down the control key on the keyboard.

clim:+meta-key+
The modifier state bit that corresponds to the user holding down the meta key on the keyboard.

clim:+super-key+
The modifier state bit that corresponds to the user holding down the super key on the keyboard.

clim:+hyper-key+
The modifier state bit that corresponds to the user holding down the hyper key on the keyboard.

The following constants are the values that can be taken on by **clim:pointer-event-button**.

clim:+pointer-left-button+
The value returned by **clim:pointer-event-button** that corresponds to the user having pressed or released the lefthand button on the pointer.

clim:+pointer-middle-button+
The value returned by **clim:pointer-event-button** that corresponds to the user having pressed or released the middle button on the pointer.

clim:+pointer-right-button+
The value returned by **clim:pointer-event-button** that corresponds to the user having pressed or released the righthand button on the pointer.

CLIM Operators for Defining Presentation Translators

You can write presentation translators that apply to blank areas of the window, that is, areas where there are no presentations. Use **clim: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*.

clim:define-presentation-translator supports the general case, and **clim:define-presentation-to-command-translator** supports a common idiom.

clim:define-presentation-translator *name (from-type to-type command-table* &key *(:gesture* **':select***) :tester :tester-definitive :documentation :pointer-documentation (:menu* **t***) :priority) arglist* &body *body*
Defines a presentation translator named *name* which translates from objects of type *from-type* to objects of type *to-type*.

clim:define-presentation-to-command-translator *name (from-type command-name command-table* &key *(:gesture* **':select***) :tester :documentation :pointer-documentation (:menu* **t***) :priority (:echo* **t***)) arglist* &body *body*

Defines a presentation translator that translates a displayed presentation into a command.

clim:define-presentation-action *name (from-type to-type command-table* &key *(:gesture* **':select***) :tester :documentation :pointer-documentation (:menu* **t***) :priority) arglist* &body *body*
This is similar to **clim:define-presentation-translator**, except that the body of the action is not intended to return a value, but should instead side-effect some sort of application state. Note that actions do not satisfy requests for input (as translators do).

clim:define-drag-and-drop-translator *name (from-type to-type destination-type command-table* &key *(:gesture* **':select***) :tester :documentation (:menu* **t***) :priority :feedback :highlighting :pointer-cursor) arglist* &body *body*
Defines a ''drag and drop'' (or ''direct manipulation'') translator named *name* that translates from objects of type *from-type* to objects of type *to-type* when a ''from presentation'' is ''picked up'', ''dragged'' over, and ''dropped'' on a ''to presentation'' having type *destination-type*.

clim:blank-area
The presentation type that represents all the places in a window where there is no applicable presentation. CLIM provides a single ''null presentation'' (represented by the value of **clim:*null-presentation***) of this type.

clim:*null-presentation*
The ''null'' presentation, which occupies any part of a window where there are no presentations matching the current input context.

Examples of Defining Presentation Translators in CLIM

Defining a Translation from Pathname to Integer

Here is an example that defines a presentation translator to extract the version number, an **integer** object, from a **pathname** presentation. Users have the options of typing in a version number to the input prompt or clicking on a **pathname** presentation that includes a version number.

```
(clim:define-presentation-translator pathname-version
    (pathname integer my-command-table
     :documentation "File version number"
     :gesture :select
     ;; Only works for pathnames with numeric versions
     :tester ((object) (integerp (pathname-version object)))
     :tester-definitive t)
    (object)
 (pathname-version object))

(clim:present #P"KOALA:>KJones>foo.lisp.17" 'pathname)
(clim:accept 'integer)
```

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))
```

Defining a Presentation Translator from the Blank Area

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 **clim:blank-area**. You will often want to use the *x* and *y* arguments to the translator.

For example:

```
(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)
  (list x y))
```

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:

```
(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))
```

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.

clim:find-presentation-translators *from-type to-type command-table*
Returns a list of all the translators associated with *frame*'s current command table that translate from *from-type* to *to-type*, without taking into account any type parameters or testers.

clim:find-applicable-translators *presentation input-context frame window x y* &key *:event :modifier-state :for-menu :fastp*
Returns a list of translators that apply to *presentation* in the input context *input-context*.

clim:presentation-matches-context-type *presentation context-type frame window x y* &key *:event (:modifier-state* **0***)*
Returns **t** if there are any translators that translate from *presentation*'s type to *context-type*.

clim:test-presentation-translator *translator presentation context-type frame window x y* &key *:event (:modifier-state* **0***) :for-menu*
Returns **t** if the translator *translator* applies to the presentation *presentation* in input context type *context-type*.

clim:call-presentation-translator *translator presentation context-type frame event window x y*
Calls the function that implements the body of *translator* on *presentation*'s object, and passes *presentation*, *context-type*, *frame*, *event*, *window*, *x*, and *y* to the body of the translator as well.

clim:document-presentation-translator *translator presentation context-type frame event window x y* &key *(:stream* ***standard-output****) :documentation-type*
Computes the documentation string for *translator*, sending it to *stream*.

clim:call-presentation-menu *presentation input-context frame window x y* &key *(:for-menu* **t***) :label*
Finds all the applicable translators for *presentation* in the input context *input-context*, creates a menu that contains all of the translators, and pops up a menu from which the user can choose a translator.

The following functions are useful for finding an application presentation in an output history.

clim:find-innermost-applicable-presentation *input-context stream x y* &key *(:frame* **clim:*application-frame****) :modifier-state :event*
Given an input context *input-context*, an output recording window stream *window*, and X and Y positions *x* and *y*, this function returns the innermost presentation that matches the innermost input context, using the translator matching algorithm.

clim:throw-highlighted-presentation *presentation input-context button-press-event*
Calls the applicable translator for a given *presentation*, *input-context*, and *button-press-event* (that is, the one corresponding to the user clicking a pointer button while over the presentation), and returns an object and a presentation type.

clim:frame-find-innermost-applicable-presentation *frame input-context stream x y*
Locates and returns the innermost applicable presentation on the window *stream* at the position indicated by *x* and *y*, in the input context *input-context*, on behalf of *frame*. The default method simply calls **clim:find-innermost-applicable-presentation**.

clim:frame-input-context-button-press-handler *frame stream button-press-event*
This function is responsible for handling user pointer gestures on behalf of *frame*. *stream* is the window on which *button-press-event* took place. The default method calls **clim:throw-highlighted-presentation** on the currently applicable presentation.

clim:highlight-applicable-presentation *frame stream input-context* &optional *(prefer-pointer-window* **t***)*
The ''input wait'' handler used by **clim:with-input-context**, responsible for highlighting and unhighlighting presentations.

clim:set-highlighted-presentation *stream presentation* &optional *(prefer-pointer-window* **t***)*
Highlights the presentation *presentation* on *stream*.

clim:unhighlight-highlighted-presentation *stream* &optional *(prefer-pointer-window* **t***)*
Unhighlights any highlighted presentations on *stream*.

Most applications will never need to use any of these functions.

Defining Application Frames in CLIM

Concepts of CLIM Application Frames

Application frames (or simply, *frames*) are the central mechanism in CLIM for presenting an application's user interface. A frame contains the state of the application and a hierarchy of *panes*.

The look and feel of an application frame is managed by a *frame manager*. The frame manager is responsible for creating the concrete, window system dependent gadget that corresponds to each abstract gadget. It is also responsible for the look and feel of menus, dialogs, pointer documentation, and so forth.

Application frames provide support for a standard interaction processing loop, like the Lisp ''read-eval-print'' loop. You are required to write only the code that implements the frame-specific commands and output display functions. A key aspect of this interaction processing 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'' (a command and all of its operands), processing the input (by executing the command), and updating the displayed information as appropriate. CLIM implementations are free to run the display update part of the loop at a lower priority than command execu-

tion. For example, some implementations may choose not to update the display if there is typed-ahead input. Also, command execution and display will not occur simultaneously, so user-defined functions need not cope with multiprocessing.

Note that this definition of the standard interaction loop does not constrain the interaction style to command-line interfaces. The input sentence may be entered via single keystrokes, pointer input, menu selection, or by typing full command lines. CLIM allows the application implementor to choose what subset of approaches will be applicable for each individual command. Furthermore, an application's interaction loop isn't constrained to read only commands; it could read lower-level events in order to implement a completely different interaction style.

For more information about how to use CLIM application frames, see the section "What is an Application?".

Defining CLIM Application Frames

clim:define-application-frame defines CLIM application frames. Application frames are represented by CLOS classes which inherit from **clim:standard-application-frame**. You can specify a name for the application class, the superclasses (if there are any beyond **clim:standard-application-frame**), the slots of the application class, and options.

clim:define-application-frame defines a class with the following characteristics:

- inherits some behavior and slots from the class **clim:standard-application-frame**,
- inherits other behavior and slots from any other *superclasses* which you specify explicitly, and
- has other slots, as explicitly specified by *slots*.

The following options are used by the class **clim:standard-application-frame**:

:panes or **:pane**

:layouts

:top-level

:command-table

:disabled-commands

:command-definer

:menu-bar

Note that **:command-definer** doesn't actually affect a slot, but instead provides an override for the name of the default application command definer macro that automatically generated by **clim:define-application-frame**.

For detailed information about CLIM application frames, see the macro **clim:define-application-frame**.

Panes in CLIM

CLIM panes are similar to the gadgets or widgets of other toolkits. They can be used by application programmers to compose the top-level user interface of their 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 an application written using the toolkits supplied by the underlying window system.

Panes are rectangular objects that are implemented as special sheet classes. An application will typically construct a tree of panes that divide up the screen space allocated to the application frame. The various CLIM pane types can be characterized by whether they have child panes or not: panes that can have other panes as children are called *composite panes*, and those that don't are called *leaf panes*. Composite panes are used to provide a mechanism for spatially organizing (''laying out'') other panes. There are two main kinds of leaf panes: gadgets and ''application'' panes. Leaf panes that implement gadgets have a particular appearance (often defined by the underlying window system toolkit) and react to user input by invoking some kind of callback. ''Application'' panes provide an area of the application's screen real estate that can be used by the application to present application specific information. CLIM provides a number of these *application pane* types that allow the application to use CLIM's graphics and extended stream facilities.

Abstract panes are gadget 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. The purpose of this abstract definition is to allow multiple implementations of the abstract pane, each defining its own specific look and feel. CLIM can then select the appropriate pane implementation based on factors outside the control of the application, 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.

Basic Pane Construction

Applications typically define the hierarchy of panes used in their frames using the **:pane** or **:panes** options of **clim: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 to constructing a pane: **clim:make-pane** of an ''abstract'' pane class name, or **clos: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 specif-

ic class (such as an **clim:hbox-pane** or a **clim:vbox-pane**). In this case, using **clos:make-instance** directly circumvents the abstract pane selection mechanism. However, the **clos: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 **clim:make-pane** instead of **clos: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. **clim: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 **clim:make-pane** in which case the frame manager will generally instantiate that specific class.

See the function **clim:make-pane**.

See the macro **clim:make-clim-stream-pane**.

Using the :panes Option to clim:define-application-frame

The **:panes** option to **clim:define-application-frame** is used to describe the panes used by the application frame. It takes a list of *pane-descriptions*. Each *pane-description* can be one of two possible formats:

- A list consisting of a *pane-name* (which is a symbol), a *pane-type*, and *pane-options*, which are keyword-value pairs. *pane-options* is evaluated at load time.
- A list consisting of a *pane-name* (which is a symbol), followed by an expression that is evaluated to create the pane. See the macro **clim:make-clim-stream-pane** and the function **clim:make-pane**.

The *pane-types* are:

:application
: Application panes are stream panes used for the display of application-generated output. See the class **clim:application-pane**. See the macro **clim:make-clim-application-pane**.

:interactor
: Interactor panes are stream panes that provide a place for the user to do interactive input and output. See the class **clim:interactor-pane**. See the macro **clim:make-clim-interactor-pane**.

:accept-values
: These panes provide for the display of a ''modeless'' **clim:accepting-values** dialog. See the class **clim:accept-values-pane**. See the section "Using an **:accept-values** Pane in a CLIM Application Frame".

:pointer-documentation
These panes provide for pointer documentation. If such a pane is specified, then when the pointer moves over different areas of the frame, this pane displays documentation of the effect of clicking the pointer buttons.

If the host window system has its own way of displaying pointer documentation, this pane may be omitted automatically from the layout.

See the class **clim:pointer-documentation-pane**.

:command-menu
Command menu panes are used to hold a menu of application commands. The default display function is **clim:display-command-menu** which, by default, displays the current command table of the frame. You can display a different command table by supplying the **:command-table** argument to **clim:display-command-menu**.

Many host window systems provide a menu bar, so having panes of type **:command-menu** is not common.

See the class **clim:command-menu-pane**.

:title Title panes are used for displaying the title of the application. The default title is a ''prettied up'' version of the name of the application frame's class.

Many host window systems will automatically display the frame's title in a title bar, so this is only rarely useful.

See the class **clim:title-pane**.

The following *pane-options* are usable by all pane types, unless otherwise noted.

:width, **:height**, **:min-width**, **:min-height**, **:max-width**, and **:max-height**
Provide space requirement specs that specify the sized of the pane. The values the space requirements can take are described in "Using the :LAYOUTS Option to CLIM:DEFINE-APPLICATION-FRAME".

:background and **:foreground** *ink*
Provide initial values for **clim:medium-foreground** and **clim:medium-background** for the pane.

:text-style *text-style*
Specifies a text style to use in the pane. The default depends on the pane type.

:borders Controls whether borders are drawn around CLIM stream panes (**t** or **nil**). The default is **t**. The value may also be a list, in which case the value is used as options to **clim:outlining**.

:spacing Controls whether there is some whitespace between the border and the viewport for a CLIM stream pane (**t** or **nil**). The default is **t**.

The value may also be a list, in which case the value is used as options to **clim:spacing**.

:scroll-bars *scroll-bar-spec*
A *scroll-bar-spec* can be **:both** (the default for **:application** panes), **:horizontal**, **:vertical**, **:none**, or **nil**. The pane will have only those scroll bars which were specified. **:none** means that the pane will support scrolling, but does not have any visible scroll bars. **nil** means that the pane will not support scrolling at all.

:display-after-commands
One of **t**, **nil**, or **:no-clear**. 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 managing the display after commands.

:no-clear behaves the same as **t**, with the following change. If you have not specified **:incremental-redisplay t**, then the pane is normally cleared before the display function is called. However, if you specify **:display-after-commands :no-clear**, then the pane is not cleared before the display function is called.

:display-function *display-spec*
Where *display-spec* is either the name of a function or a list whose first element is the name of a function. The function is to be applied to the application frame, the pane, and the rest of *display-spec* if it was a list when the pane is to be redisplayed.

The function must accept two required arguments (the frame and the pane), plus the two keyword arguments **:max-width** and **:max-height**.

One example of a predefined display function is **clim:display-command-menu**.

:display-string *string*
(for **:title** panes only) The string to display. The default is the frame's pretty name.

:label *string*
A string to be used as a label for the pane, or **nil** (the default).

:incremental-redisplay *boolean*
If **t**, CLIM runs the display function inside of an **clim:updating-output** form. The default is **nil**.

:end-of-line-action, **:end-of-page-action**
Initial values of the corresponding attributes. See the macro **clim:with-end-of-line-action** and see the macro **clim:with-end-of-page-action**.

:initial-cursor-visibility
:off means make the text cursor visible if the window is waiting for input. **:on** means make it visible all the time. **nil** means that the

cursor is never visible. The default is **:off** for **:interactor** and **:accept-values** panes, and **nil** for other panes.

:output-record
Specify this if you want a different output history mechanism than the default (which is **clim:standard-tree-output-history**). For example, a graphic editing program might supply a value of:

```
(make-instance 'clim:r-tree-output-history)
```

Besides **clim:standard-tree-output-history** and **clim:r-tree-output-history**, you can also use **clim:standard-sequence-output-history**.

:draw-p, **:record-p** *boolean*
Specifies the initial state of drawing and output recording.

:default-view *view*
Specifies the view object to use for the stream's default view.

:text-margin *integer*
Text margin to use if **clim:stream-text-margin** isn't set. This defaults to the width of the viewport.

:vertical-spacing *integer*
Amount of extra space between text lines.

:pointer-cursor
Specifies the pointer cursor to use when the pointer is over this pane.

:event-queue
Specifies the event queue to be used by this pane. The default is to share the event queue with the top-level sheet, so that all the panes in the frame use the same event queue.

Using the :LAYOUTS Option to CLIM:DEFINE-APPLICATION-FRAME

A *layout* is an arrangement of panes within the application-frame's top-level window. An application may have many layouts or it may have only one layout that remains constant for the life of the program. If you do not specify any layout, CLIM will construct a default layout for the application.

As the application is running, the current layout may be changed to any of the layouts described in the **:layouts** option of the frame definition. See the generic function **clim:frame-current-layout**.

The **:layouts** option specifies and names the layouts of the application. A layout typically consists of rows, columns, and tables of panes, or more complicated nestings of rows, columns and tables. The value of the **:layouts** option is a list of layout descriptions. Each layout description is a two element list consisting of a symbol, which names the layout, and a corresponding *layout-spec*.

A *layout-spec* is a simply a form consisting of the various layout macros that constructs a pane.

A *size-spec* can be **:fill**, **:compute**, or a real number between zero and one (exclusive) (indicating that the size of the pane is that fraction of the available space), an integer (indicating that the size of the pane is that many device units), or a list whose first element is a real number and whose second element is a ''unit'' (one of **:line**, **:character**, **:mm**, **:point**, or **:pixel**).

This syntax can be expressed as follows:

> **:layouts** (*layout-name layout-panes*)
>
> *layout-name* is a symbol.
>
> *layout-panes* is *layout-panes1* or (*size-spec layout-panes1*).
>
> *layout-panes1* is a *pane-name*, or a *layout-macro-form*, or *layout-code*.
>
> *layout-code* is Lisp code that generates a pane, which may include the name of a named pane.
>
> *size-spec* is a positive real number less than 1, or **:fill**, or **:compute**. A real number (between zero and one, exclusive) is the fraction of the available space to use. **:fill** means that the pane will take as much space as remains when all its sibling panes have been given space. **:compute** means that the pane's display function should be called in order to compute how much space it requires. (Note that the display function is run at frame-creation time, so it must be able to compute the size correctly at that time.)
>
> *size-spec* can also be an integer indicating the size of the pane in device units, or a list whose first element is a real number and whose second element is a ''unit'' (one of **:line**, **:character**, **:mm**, **:point**, or **:pixel**).
>
> *layout-macro-form* is (*layout-macro-name* (*options*) &rest *layout-panes*).
>
> *layout-macro-name* is **clim:vertically**, **clim:horizontally**, **clim:tabling**, **clim:outlining**, **clim:spacing**, or **clim:labelling**.

The following macros and pane classes provide layout for other panes in CLIM.

clim:vertically *(*&rest *options* &key *:spacing* &allow-other-keys*)* &body *contents*
: Lays out one or more child panes vertically, from top to bottom.

clim:vbox-pane
: The layout pane class that arranges its children in a vertical stack.

clim:horizontally *(*&rest *options* &key *:spacing* &allow-other-keys*)* &body *contents*
Lays out one or more child panes horizontally, from left to right.

clim:hbox-pane
The layout pane class that arranges its children in a horizontal row.

clim:tabling *(*&rest *options)* &body *contents*
Lays out its child panes in a two-dimensional table arrangement.

clim:table-pane
The layout pane class that arranges its children in a tabular format.

clim:outlining *(*&rest *options* &key *:thickness* &allow-other-keys*)* &body *contents*
Puts an outlined border of the specified thickness around a single child pane.

clim:outlined-pane
The layout pane class that draws a border around its child pane.

clim:spacing *(*&rest *options* &key *:thickness :background* &allow-other-keys*)* &body *contents*
Reserves some margin space around a single child pane.

clim:spacing-pane
The layout pane class that leaves some empty space around its child pane.

The following macro can be used to label any pane.

clim:labelling *(*&rest *options* &key *:label (:label-alignment* **:bottom***)* &allow-other-keys*)* &body *contents*
Creates a vertical stack consisting of two panes, a label and a single other pane.

The following macro can be used to provide scrolling for a pane.

clim:scrolling *(*&rest *options)* &body *contents*
Creates a composite pane that allows the single child specified by *contents* to be scrolled.

Details of CLIM's Layout Algorithm

CLIM uses a two pass algorithm to lay out a pane hierarchy. In the first pass (called *space composition*), the top level pane is queried to find out how much space it requires. This query may cause the same query to be made, 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 **clim: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 most types of panes, you can indicate the space requirements of the pane at creation time by using the space requirement options (**:width**, **:height**, **:max-width**, **:min-height**, and so on). For example, application 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 how much space they need based on the information they need to display. For example, push button panes automatically calculate their space requirement based on the amount of space required by the push button's label.

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 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 how big they want to be.

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).

The following functions are used in pane layout to compute and allocate space, and to set the size and position of the panes.

clim:make-space-requirement &key *(:width* **0***) (:min-width* **width***) (:max-width* **width***) (:height* **0***) (:min-height* **height***) (:max-height* **height***)*
Constructs and returns a space requirement object having the given components.

clim:space-requirement-components *space-req*
Returns the components of the space requirement *space-req* as six values, the width, minimum width, maximum width, height, minimum height, and maximum height.

clim:space-requirement+ *space-req*
Returns a new space requirement whose components are the sum of each of the components of *sr1* and *sr2*.

clim:compose-space *pane* &key *:width :height*
The value returned by **clim:compose-space** is a space requirement object that represents how much space the pane *pane* requires.

clim:allocate-space *pane width height*
During the space allocation pass, a composite pane will arranges children within the available space and allocates space to them according to their space requirements and its own composition rules by calling **clim:allocate-space** on each of the child panes.

clim:move-sheet *sheet x y*

Moves *sheet* to the new position (*x*,*y*). *x* and *y* are in coordinates relative to *sheet*'s parent.

clim:resize-sheet *sheet width height*

Changes the size of *sheet* to have width *width* and height *height*.

clim:move-and-resize-sheet *sheet x y width height*

Moves *sheet* to the new position (*x*,*y*), and simultaneously changes the size of the sheet to have width *width* and height *height*. *x* and *y* are in coordinates relative to *sheet*'s parent.

Examples of the :panes and :layouts Options to clim:define-application-frame

Here are some examples of how to use the **:panes** and **:layouts** options of **clim:define-application-frame** to describe the appearance of your application.

We begin by showing 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.

```
(clim:define-application-frame graphics-demo
                               ()
    ()
  (:menu-bar nil)
  (:panes
    (commands :command-menu)
    (demo :application)
    (explanation :application :scroll-bars nil))))
```

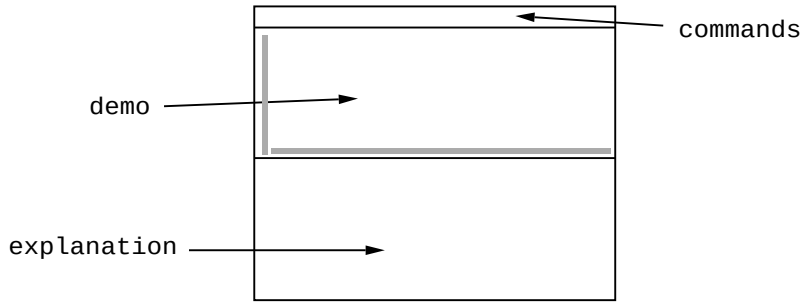


Figure 64. The default layout for the graphic-demo example when no explicit :layout is specified.

Now we add an explicit **:layouts** option to the frame definition from the previous example. 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 necessary to display the command menu.

```
(clim:define-application-frame graphics-demo
                          ()
  ()
 (:menu-bar nil)
 (:panes
   (commands :command-menu)
   (demo :application)
   (explanation :application :scroll-bars nil))
 (:layouts
   (default
     (clim:vertically ()
       (5/6 (clim:horizontally () demo commands))
       (1/6 explanation)))))
```

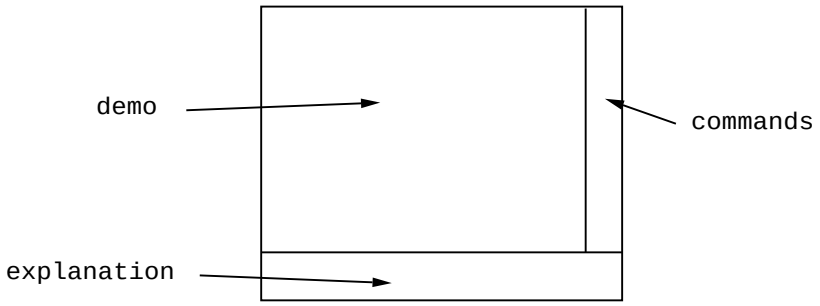


Figure 65. The layout for the graphic-demo example with an explicit :layout.

Finally, here is a stripped-down version of the application frame definition for the CAD demo (in the file SYS:CLIM;REL-2;DEMO;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 he is using. There is a pane named **menu** which 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 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.

The layout named **other** 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.

```
(define-application-frame cad-demo
                          (standard-application-frame output-record)
    ((object-list :initform nil))
  (:menu-bar nil)
  (:pointer-documentation t)
  (:panes
    (title :title :display-string "Mini-CAD")
    (menu :command-menu)
    (design-area :application))
  (:layouts
    (default
      (clim:vertically ()
        title menu design-area))
    (other
      (clim:vertically ()
         title
         (clim:horizontally () menu design-area)))))
```

Using an :accept-values Pane in a CLIM Application Frame

:accept-values panes are used when you want one of the panes of your application to be in the form of an **clim: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 **clim:accept-values-pane** command table.

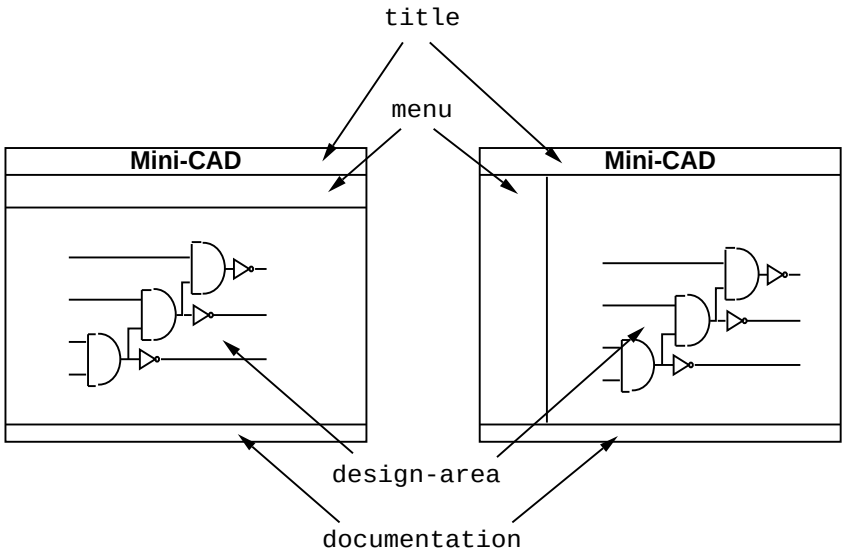


Figure 66. The two layouts of the Mini-CAD demo. Layout **main** is on the left, layout **other** is on the right.

- 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 **clim:accept** just as they would appear inside a **clim:accepting-values** for a dialog. It takes two arguments, the frame and a stream. *my-acceptor-function* doesn't need to call **clim:accepting-values** itself, since that is done automatically.

  See the section "Menus and Dialogs in CLIM", and see the function **clim:accept-values-pane-displayer**.

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 **clim: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 which can not be determined until after the application frame is created. See the macro **clos:defclass** to learn about slot-specifiers.

2. For initializations which 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 **clim:*application-frame*** is not bound to the application since the application is not yet running. You may use **clos:with-slots**, **clos:with-accessors**, or any slot readers or accessors which have been defined.

3. A **:before** or **:around** method for **clim: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. **clim:*application-frame*** will be bound to the application frame.

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 **clim:default-frame-top-level** to achieve the standard behavior for application frames.

Of course, these are only suggestions. Other techniques might be more appropriate for your application. Of those listed, the **:before** or **:around** method on **clim: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 **clim:make-application-frame**. This function provides important initialization arguments specific to application frames which **clos:make-instance** does not. **clim:make-application-frame** passes any keyword value pairs which it does not handle itself on to **clos:make-instance**. Thus, 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 of the commands which were performed while it was executing. This might be useful in the context of a program 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 separate it out into a special class which implements the desired behavior. This class can then be used as a superclass for any application which should keep a log of its actions.

The class has an initialization option **:pathname** which specifies the name of the log file. It has a slot named **transaction-stream** whose value is a stream opened to the log file when the application is running.

```
(defclass transaction-recording-mixin ()
        ((transaction-pathname :type pathname
                               :initarg :pathname
                               :reader transaction-pathname)
         (transaction-stream :accessor transaction-stream)))
```

We use an **:around** method on **clim: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.

```
(defmethod clim:run-frame-top-level :around
           ((frame transaction-recording-mixin))
  (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 **clim:default-frame-top-level** calls **clim:execute-frame-command** to execute a command. Here we add a **:before** method which will log the command.

```
(defmethod clim:execute-frame-command :before
          ((frame transaction-recording-mixin) command)
  (format (transaction-stream frame)
          "~&Command: ~a" command))
```

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 `SYS:CLIM;REL-2;DEMO;PUZZLE.LISP`). Our modifications are shown in *italics*. 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 **clim:standard-application-frame**, which would be included by default when the superclasses argument is empty.

```
(define-application-frame puzzle
      (transaction-recording-mixin standard-application-frame)
    ((puzzle :initform (make-array '(4 4))
             :accessor puzzle-puzzle))
  (:default-initargs :pathname "puzzle-log.text")
  (:panes
    (display :application
             :display-function 'draw-puzzle
             :text-style '(:fix :bold :very-large)
             :incremental-redisplay t
             :text-cursor nil
             :width :compute :height :compute
             :end-of-page-action :allow
             :end-of-line-action :allow))
  (:layouts
    (:default display)))
```

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 evaluating the following:

```
(clim:run-frame-top-level
  (clim:make-application-frame 'puzzle
    :parent (clim:find-port)
    :pathname "my-puzzle-log.text"))
```

Here the **:pathname** initialization argument was used to override the default name for the log file (as was specified by the **:default-initargs** clause in the above application frame definition) and to use the name `"my-puzzle-log.text"` instead.

Accessing Slots and Components of CLIM Application Frames

CLIM application frames are instances of the defined subclass of the **clim:standard-application-frame** class. You explicitly specify accessors for the slots you have specified for storing application-specific state information, and use those accessors as you would for any other CLOS instance. Other CLIM defined components of application frame instances are accessed via documented functions. Such components include frame panes, command tables, top-level window, and layouts.

Running a CLIM Application

This section describes how to use CLIM application frames in Cloe Runtime and in Genera, but the technique is similar on most platforms. For more information on how to run CLIM in these environments,see the section "Using Symbolics CLIM".

Genera

In Genera and in Cloe Developer the recommended way to run CLIM applications is to make them available to the Genera activity selection system. Call **clim:define-genera-application** after **clim:define-application-frame**.

clim:define-genera-application *frame-name* &rest *keys* &key *:pretty-name :select-key :left :top :right :bottom :width :height*
: Makes a CLIM application available to the `Select Activity` command and optionally to the `SELECT` key. This exists only in Genera.

Cloe Runtime

In Cloe Runtime **clim:define-application-frame** automatically does a **cloe:define-program**. When you call **cloe:save-program** with the name of your application frame, it saves a virtual memory image that will run your application when it is started. For test purposes, you can use the **:simulate** argument to **cloe:save-program** to run the program without saving it.

Programmatic Ways to Run CLIM Applications

You can also run a CLIM application using the functions **clim:make-application-frame** and **clim:run-frame-top-level**. First use **clim:find-port** to create a port to pass as the **:parent** argument to **clim:make-application-frame**. Here is a code fragment which illustrates this technique under Genera:

```
(clim:run-frame-top-level
   (clim:make-application-frame 'frame-name
      :parent (clim:find-port :server-path '(:genera))))
```

clim:run-frame-top-level will not return until the application exits.

Note that **clim:*application-frame*** is not bound until **clim:run-frame-top-level** is invoked.

For more information, see the section "Functions for Operating on Windows Directly".

Examples of CLIM Application Frames

These are examples of how to use CLIM application frames. For other examples, see the section "CLIM Tutorial" and see the section "Using the CLIM Demos".

Example of defining a CLIM application frame

Here is an example of an application frame. This frame has three slots, named **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 **clim:define-command-table**) and which inherits from the **clim:accept-values-pane** command table so that the accept-values pane will function properly.

```
(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))))
```

This 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.

```
(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)))
```

This is the display function for the pane named **avv**. It invokes **clim:accept** for each of the application's slots so that the user can alter their values in the **avv** pane.

```
(defmethod display-avv ((frame dingus) stream)
  (with-slots (pathname integer member) frame
    (fresh-line stream)
    (setq pathname (clim:accept 'pathname
                                :prompt "A pathname" :default pathname
                                :stream stream))
    (fresh-line stream)
    (setq integer (clim:accept 'integer
                               :prompt "An integer" :default integer
                               :stream stream))
    (fresh-line stream)
    (setq member (clim:accept '(member :one :two :three)
                              :prompt "One, Two, or Three" :default member
                              :stream stream))
    (fresh-line stream)
    (clim:accept-values-command-button (stream :documentation "You wolf")
        (write-string "Wolf whistle" stream)
      (beep))))
```

This function will start up a new **dingus** application. The argument is a port, such as one returned by **clim:find-port**.

```
(defun run-dingus (port)
  (let ((dingus (clim:make-application-frame 'dingus
                  :parent port :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 dialog pane, **avv**. The new values will automatically be reflected in the **display** pane.

Example of 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.

```
(clim:define-application-frame different-prompts ()
    ((prompt-state ...) ...)
  (:top-level (different-prompts-top-level))
  ...)

(defmethod 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)))
```

Operators for Defining CLIM Application Frames

The following operators are used to define and instantiate CLIM application frames. **clim:define-genera-application** may only be used under Genera.

clim:define-application-frame *name superclasses slots* &rest *options*
: Defines a frame and CLOS class named *name* that inherits from *superclasses* and has state variables specified by *slots*.

clim:make-application-frame *frame-name* &key *:frame-class :pretty-name :parent :left :top :right :bottom :height :width* &allow-other-keys
: Makes an instance of the application frame of type *frame-class*. The keyword arguments are passed as additional arguments to **clos:make-instance**.

clim:find-application-frame *frame-name* &rest *initargs* &key *(:create* **t***) (:activate* **t***) (:own-process* **t***) :port :frame-manager :frame-class* &allow-other-keys
: Calling this function is similar to calling **clim:make-application-frame**, and then calling **clim:run-frame-top-level** on the newly created frame.

clim:define-genera-application *frame-name* &rest *keys* &key *:pretty-name :select-key :left :top :right :bottom :width :height*
: Makes a CLIM application available to the `Select Activity` command and optionally to the `SELECT` key. This exists only in Genera.

CLIM Application Frame Accessors

The following functions may be used to access 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.

clim:*application-frame*
: The current application frame.

clim:with-application-frame *(frame)* &body *body*
: Evaluates *body* with the variable *frame* bound to the current application frame.

clim:frame-name *frame*
Returns the name of *frame*.

clim:frame-pretty-name *frame*
Returns the pretty name of *frame*.

clim:frame-standard-input *frame*
Returns the value that should be used for ***standard-input*** for *frame*.

clim:frame-standard-output *frame*
Returns the value that should be used for ***standard-output*** for *frame*.

clim:frame-error-output *frame*
Returns the value that should be used for ***error-output*** for *frame*.

clim:frame-query-io *frame*
Returns the value that should be used for ***query-io*** for *frame*.

clim:frame-pointer-documentation-output *frame*
Returns the value that should be used for **clim:*pointer-documentation-output*** for *frame*.

clim:frame-current-layout *frame*
Returns the name of current layout for *frame*. You can use **setf** on **clim:frame-current-layout** to change the current layout.

clim:frame-all-layouts *frame*
Returns a list of all of the layout names for *frame*.

clim:frame-current-panes *frame*
Returns a list of all of the named CLIM stream panes that are contained in the current layout for the frame *frame*.

clim:frame-panes *frame*
Returns the single pane acting as the ''root'' pane for the current layout.

clim:get-frame-pane *frame pane-name* &key *(:errorp* **t***)*
Returns the CLIM stream pane named by *pane-name* in *frame*.

clim:find-pane-named *frame pane-name* &optional *errorp*
Returns the CLIM stream pane named by *pane-name* in *frame*.

clim:frame-command-table *frame*
Returns the name of the command table currently being used by the frame *frame*. You can use this function with **setf** to specify the command table to be used.

clim:frame-maintain-presentation-histories *frame*
Returns **t** if the *frame* maintains histories for its presentations, otherwise returns **nil**. The default method on the class **clim:standard-application-frame** returns **t** if and only if the frame has an interactor pane. You can specialize this generic function for your own application frames.

clim:frame-top-level-sheet *frame*
Returns the window that corresponds to the top level window for the frame *frame*.

clim:frame-state *frame*
Returns one of **:enabled**, **:disabled**, **:disowned**, or **:shrunk**, indicating the current state of frame.

Operators on CLIM Application Frames

The following functions are used to start up an application frame, exit from it or destroy 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).

clim:run-frame-top-level *frame* &key &allow-other-keys
Runs the top-level function for *frame*.

clim:default-frame-top-level *frame* &key *:command-parser :command-unparser :partial-command-parser (:prompt* **"Command: "***)*
The default top-level function for application frames, which provides a ''read-eval-print'' loop.

clim:enable-frame *frame*
Enables the application frame *frame* and changes the state of the frame to **:enabled**.

clim:frame-exit *frame*
Exits from the application frame *frame* by signalling a **clim:frame-exit** condition.

clim:raise-frame *frame*
Raises the application frame *frame* so that it is on top of all of the other host windows by calling **clim:raise-sheet** on the frame's top-level sheet.

clim:bury-frame *frame*
Buries the application frame *frame* so that it is underneath all of the other host windows.

clim:destroy-frame *frame*
Disables the application frame *frame*, and then destroys it by deallocating all of its CLIM resources and disowning it from its frame manager.

clim:map-over-frames *function* &key *:port :frame-manager*
Applies *function* to all of the application frames that ''match'' *:port* and *:frame-manager*. *function* is a function of one argument, the frame.

clim:display-command-menu *frame stream* &key *:command-table :max-width :max-height :n-rows :n-columns (:cell-align-x* **':left***) (:cell-align-y* **':top***)*
Displays the menu described by the command table associated with the application frame *frame* onto *stream*. Disabled items in the menu will be ''grayed out''.

clim:accept-values-pane-displayer *frame pane* &key *:displayer :resynchronize-every-pass (:check-overlapping* **t***) :align-prompts :max-height :max-width*
When you use an **:accept-values** pane, the display function must use **clim:accept-values-pane-displayer**.

clim:redisplay-frame-pane *frame pane-name* &key *:force-p*
Causes the pane *pane-name* of *frame* to be redisplayed immediately.

clim:redisplay-frame-panes *frame* &key *force-p*
Causes all of the panes of *frame* to be redisplayed immediately.

clim:frame-replay *frame stream* &optional *region*
Replays all of the output records in *stream*'s output history on behalf of the application frame *frame* that overlap the region *region*.

clim:read-frame-command *frame &key :stream*
clim:read-frame-command reads a command from the user on the stream *:stream*, and returns the command object. *frame* is an application frame. You can specialize this generic function for your own application frames, for example, if you want to have your application be able to read commands using keystroke accelerators.

clim:execute-frame-command *frame command*
clim:execute-frame-command executes the command *command* on behalf of the application frame *frame*. You can specialize this function if you want to change the behavior associated with the execution of commands.

clim:command-enabled *command-name frame* &optional *command-table*
Returns **t** if the command named by *command-name* is presently enabled in *command-table* for the frame *frame*, otherwise returns **nil**. You can use **setf** on **clim:command-enabled** in order to enable or disable a command.

clim:command-menu-enabled *command-table frame*
Returns **t** if the command table *command-table* is presently enabled in the command menu for the frame *frame*, otherwise returns **nil**. This function is like **clim:command-enabled**, except that it operates only on the **:menu** items in a command table's menu for a particular frame.

Using Gadgets in CLIM

CLIM supports the use of gadgets as panes within an application. The following sections describe the basic gadget protocol, and the various gadgets supplied by CLIM.

Basic Gadget Protocol in CLIM

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 more traditional toolkits. For ex-

ample, a push button has an ''activate'' callback function that is invoked by CLIM when the user ''presses'' the button; a scroll bar has a ''value changed'' callback that is invoked by CLIM after the user moves its indicator. (Note that user code will rarely, if ever, call a callback function itself.)

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. The user can control some high-level aspects of the gadgets, but each toolkit implementation will specify the exact ''look and feel'' of their gadgets. Typically, the look and feel will be derived directly from the underlying toolkit.

Every gadget has an id and a client, which are 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 (described below), and other arguments that vary depending on the callback.

For example, the **clim:activate-callback** takes three arguments, a gadget, the client, and the gadget-id. Assuming the programmer has defined an application frame called **button-test** that has a CLIM stream pane in the slot **output-pane**, he could write the following method:

```
(defmethod clim:activate-callback
           ((button clim: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 above as:

```
(defmethod clim:activate-callback
           ((button clim:push-button) (client button-test) (gadget-id (eql 'start)))
  (start-test client))

(defmethod clim:activate-callback
           ((button clim:push-button) (client button-test) (gadget-id (eql 'stop)))
  (stop-test client))
```

```
;; Create the start and stop push buttons
(clim:make-pane 'clim:push-button
  :label "Start"
  :client frame :id 'start)
(clim:make-pane 'clim:push-button
  :label "Stop"
  :client frame :id 'stop)
```

Still another way to distinguish between gadgets is to explicitly specify what function should be called when the callback is invoked. This is specified when the gadget is created by supplying an appropriate initarg. The above example could then be written as follows:

```
;; No callback methods needed, just create the push buttons
(clim:make-pane 'clim:push-button
  :label "Start"
  :client frame
  :activate-callback
    #'(lambda (gadget) (start-test (gadget-client gadget))))
(clim:make-pane 'clim:push-button
  :label "Stop"
  :client frame
  :activate-callback
  #'(lambda (gadget) (stop-test (gadget-client gadget))))
```

The following classes and functions constitute the basic protocol for all of CLIM's gadgets. See the section "Abstract Gadgets in CLIM".

clim:gadget
: The protocol class that corresponds to a gadget.

clim:basic-gadget
: The implementation class on which many CLIM gadgets are built.

clim:gadget-id *gadget*
: Returns the gadget id of the gadget *gadget*.

clim:gadget-client *gadget*
: Returns the client of the gadget *gadget*.

clim:armed-callback *gadget client id*
: The callback that is invoked when the gadget *gadget* is armed.

clim:disarmed-callback *gadget client id*
: The callback that is invoked when the gadget *gadget* is disarmed.

clim:activate-gadget *gadget*
: Causes the gadget to become active, that is, available for input.

clim:deactivate-gadget *gadget*
: Causes the gadget to become inactive, that is, unavailable for input.

clim:gadget-active-p *gadget*
Returns **t** if the gadget is active, that is, available for input. Otherwise, it returns **nil**.

clim:note-gadget-activated *client gadget*
This function is invoked after a gadget is made active.

clim:note-gadget-deactivated *client gadget*
This function is invoked after a gadget is made inactive.

clim:value-gadget
The class used by gadgets that have a value.

clim:gadget-value *gadget*
Returns the value of the gadget *value-gadget*. You can use **setf** on **clim:gadget-value** to change the value of a gadget.

clim:value-changed-callback *gadget client id value*
The callback that is invoked when the value of a gadget is changed, either by the user or programatically. Generally, this function will call another programmer-specified callback function.

clim:drag-callback *gadget client id value*
The callback that is invoked when the value of a slider or scroll bar is changed while the indicator is being dragged. Generally, this function will call another programmer-specified callback function.

clim:action-gadget
The class used by gadgets that perform some kind of action, such as a push button.

clim:activate-callback *gadget client id*
The callback that is invoked when the gadget is activated.

clim:oriented-gadget-mixin
The class that is mixed in to a gadget that has an orientation associated with it, for example, a slider.

clim:gadget-orientation *oriented-gadget*
Returns the orientation of the gadget *oriented-gadget*. Typically, this will be a keyword such as **:horizontal** or **:vertical**.

clim:labelled-gadget-mixin
The class that is mixed in to a gadget that has a label, for example, a push button.

clim:gadget-label *labelled-gadget*
Returns the label of the gadget *labelled-gadget*.

clim:range-gadget-mixin
The class that is mixed in to a gadget that has a range, for example, a slider.

clim:gadget-min-value *range-gadget*
Returns the minimum value of the gadget *range-gadget*.

clim:gadget-max-value *range-gadget*
Returns the maximum value of the gadget *range-gadget*.

Abstract Gadgets in CLIM

Gadgets such as push buttons and sliders in CLIM are called abstract gadgets. This is because the classes, such as **clim:push-button** and **clim:slider**, do not themselves implement gadgets, but rather arrange for the frame manager layer of CLIM to create concrete gadgets that correspond to the 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 **clim:define-application-frame**, the abbreviation for a gadget is the name of the abstract gadget class.

At pane creation time (that is, during **clim: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 should 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.

The following classes and functions comprise CLIM's abstract gadgets. See the section "Basic Gadget Protocol in CLIM".

clim:make-pane *pane-class* &rest *pane-options*
Selects a class that implements the behavior of the abstract pane *pane-class* and constructs a pane of that class.

clim:push-button
The gadget class that provides press-to-activate switch behavior.

clim:toggle-button
The gadget class that provides ''on/off'' switch behavior.

clim:radio-box
A 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''.

clim:check-box
Like a radio box: this gadget constrains one or more toggle buttons. At any one time, zero or more of the buttons managed by the check box may be ''on''.

clim:with-radio-box *(*&rest *options* &key *(:type* **':one-of***)* &allow-other-keys*)* &body *body*
Creates a radio box or a check box whose buttons are created by the forms in *body*.

clim:list-pane
The gadget class that corresponds to a pane whose semantics are similar to a radio box or check box, but whose visual appearance is a list of buttons.

clim:option-pane
: The gadget class that corresponds to a pane whose semantics are identical to a list pane, but whose visual appearance is a single push button which, when pressed, pops up a menu of selections.

clim:scroll-bar
: The gadget class that corresponds to a scroll bar. The usual interface to creating a scroll bar is to use the **clim:scrolling** macro.

clim:slider
: The gadget class that corresponds to a slider.

clim:text-field
: The gadget class that implements a text field. The value of a text field is the text string.

clim:text-editor
: The gadget class corresponds to a multi-line field containing text. The value of a text editor is the text string.

Example of an Application That Uses Gadgets

The following is an example of the frame definition of an application frame that uses several different gadgets.

```
(defclass color-chooser-pane (clim:clim-stream-pane) ())

(defmethod clim:handle-repaint :after ((stream color-chooser-pane) region)
  (declare (ignore region))
  (display-color (clim:pane-frame stream) stream))

(clim:define-application-frame color-chooser ()
    (color
     red blue green
     intensity hue saturation)
  (:menu-bar nil)
  (:panes
    (display (clim:make-clim-stream-pane
               :type 'color-chooser-pane
               :scroll-bars nil
               :display-function 'display-color
               ;; Make sure we don't have a useless cursor blinking away...
               :initial-cursor-visibility nil))
    (exit clim:push-button
          :label "Exit"
          :activate-callback #'(lambda (button)
```

```
                              (clim:frame-exit (clim:pane-frame button))))
      (rgb (with-slots (red green blue) clim:*application-frame*
             (clim:outlining ()
               (clim:horizontally ()
                 (setq red (clim:make-pane clim:'slider
                             :label "Red" :foreground clim:+red+
                             :orientation :vertical
                             :min-value 0.0 :max-value 1.0
                             :show-value-p t :decimal-places 3
                             :client 'color :id 'red))
                 (setq green (clim:make-pane 'clim:slider
                               :label "Green" :foreground clim:+green+
                               :orientation :vertical
                               :min-value 0.0 :max-value 1.0
                               :show-value-p t :decimal-places 3
                               :client 'color :id 'green))
                 (setq blue (make-pane 'clim:slider
                              :label "Blue" :foreground clim:+blue+
                              :orientation :vertical
                              :min-value 0.0 :max-value 1.0
                              :show-value-p t :decimal-places 3
                              :client 'color :id 'blue))))))
      (ihs (with-slots (intensity hue saturation) clim:*application-frame*
             (clim:outlining ()
               (clim:horizontally ()
                 (setq intensity (clim:make-pane 'clim:slider
                                   :label "Intensity"
                                   :orientation :vertical
                                   :min-value 0.0 :max-value (sqrt 3)
                                   :show-value-p t :decimal-places 3
                                   :client 'color :id 'intensity))
                 (setq hue (clim:make-pane 'clim:slider
                             :label "Hue"
                             :orientation :vertical
                             :min-value 0.0 :max-value 1.0
                             :show-value-p t :decimal-places 3
                             :client 'color :id 'hue))
                 (setq saturation (clim:make-pane 'clim:slider
                                    :label "Saturation"
                                    :orientation :vertical
                                    :min-value 0.0 :max-value 1.0
                                    :show-value-p t :decimal-places 3
```

```
                                    :client 'color :id 'saturation)))))))
  (:layouts
    (default
      (clim:horizontally ()
        (clim:outlining ()
          (clim:vertically () display exit))
        rgb ihs))))

(defmethod clim:run-frame-top-level :before ((frame color-chooser) &key)
  (with-slots (color) frame
    (setf color clim:+black+)))

(defmethod color ((frame color-chooser))
  (with-slots (color) frame
    color))

(defmethod (setf color) (new-color (frame color-chooser))
  (with-slots (color) frame
    (setf color new-color)))

(defmethod display-color ((frame color-chooser) stream)
  (clim:with-bounding-rectangle* (left top right bottom)
      (clim:window-viewport stream)
    (clim:with-output-recording-options (stream :record nil)
      (clim:draw-rectangle* stream left top right bottom
                            :filled t :ink (slot-value frame 'color)))))

(defmacro define-rgb-callbacks (color)
  (check-type color (member red green blue))
  (let* ((rgb '(red green blue))
         (new-rgb (substitute 'value color rgb)))
    `(progn
       (defmethod clim:value-changed-callback
                  ((slider clim:slider)
                   (client (eql 'color)) (id (eql ',color)) value)
         (let ((frame (clim:pane-frame slider)))
           (multiple-value-bind (,@rgb) (clim:color-rgb (color frame))
             (declare (ignore ,color))
             (setf (color frame) (clim:make-rgb-color ,@new-rgb)))
```

```
           (update-ihs frame)))
       (defmethod clim:drag-callback
                  ((slider clim:slider)
                   (client (eql 'color)) (id (eql ',color)) value)
         (let ((frame (clim:pane-frame slider)))
           (multiple-value-bind (,@rgb) (clim:color-rgb (color frame))
             (declare (ignore ,color))
             (setf (color frame) (clim:make-rgb-color ,@new-rgb)))
           (update-ihs frame))))))

(define-rgb-callbacks red)
(define-rgb-callbacks green)
(define-rgb-callbacks blue)

(defmethod update-ihs ((frame color-chooser))
  (with-slots (intensity hue saturation) frame
    (multiple-value-bind (ii hh ss) (clim:color-ihs (color frame))
      (setf (clim:gadget-value intensity :invoke-callback nil) ii)
      (setf (clim:gadget-value hue :invoke-callback nil) hh)
      (setf (clim:gadget-value saturation :invoke-callback nil) ss))))

(defmacro define-ihs-callbacks (color)
  (check-type color (member intensity hue saturation))
  (let* ((ihs '(intensity hue saturation))
         (new-ihs (substitute 'value color ihs)))
    `(progn
       (defmethod clim:value-changed-callback
                  ((slider clim:slider)
                   (client (eql 'color)) (id (eql ',color)) value)
         (let ((frame (clim:pane-frame slider)))
           (multiple-value-bind (,@ihs) (clim:color-ihs (color frame))
             (declare (ignore ,color))
             (setf (color frame) (clim:make-ihs-color ,@new-ihs)))
           (update-rgb frame)))
       (defmethod clim:drag-callback
                  ((slider clim:slider)
                   (client (eql 'color)) (id (eql ',color)) value)
         (let ((frame (clim:pane-frame slider)))
           (multiple-value-bind (,@ihs) (clim:color-ihs (color frame))
             (declare (ignore ,color))
             (setf (color frame) (clim:make-ihs-color ,@new-ihs)))
           (update-rgb frame))))))

(define-ihs-callbacks intensity)
(define-ihs-callbacks hue)
(define-ihs-callbacks saturation)
```

```
(defmethod update-rgb ((frame color-chooser))
  (with-slots (red green blue) frame
    (multiple-value-bind (rr gg bb) (clim:color-rgb (color frame))
      (setf (clim:gadget-value red :invoke-callback nil) rr)
      (setf (clim:gadget-value green :invoke-callback nil) gg)
      (setf (clim:gadget-value blue :invoke-callback nil) bb))))

(defmethod clim:value-changed-callback :after
           ((slider clim:slider) (client (eql 'color)) id value)
  (declare (ignore id value))
  (let ((frame (clim:pane-frame slider)))
    (redisplay-frame-pane (pane-frame slider) 'display)))
```

Commands in CLIM

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 **clim:command** and then executes the accepted command. "Command Objects in CLIM" 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 which is desired. See the macro **clim:define-presentation-action** for a discussion about the appropriateness of presentation actions.

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, including direct manipulation (''drag and drop'') interactions. A command can be invoked by clicking on any object displayed by the interface. The particular combination of mouse-buttons and modifier keys (such as shift or 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, and is entirely under the control of the application programmer. 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 serves to mediate between a command input context, a set of commands, and these interaction styles.

Commands may take arguments, which are specified by their presentation types.

For most CLIM applications, **clim:define-application-frame** will automatically create a command table, 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 which might suggest something beyond the simple approach:

- Your application requires more than one command table, for example, if 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, then you can just read "Defining Commands the Easy Way" and skip the remainder of this chapter.

Defining Commands the Easy Way

CLIM provides utilities to make it easy to define commands for most applications. **clim:define-application-frame** will automatically create a command table for your application. This behavior is controlled by the **:command-table** option. It will also define a command defining macro which you will use to define the commands for your application. This is controlled by the **:command-definer** option.

This command definer macro will behave similarly to **clim:define-command**, but will automatically use your application's command table so you needn't specify one.

Here is an example code fragment illustrating the usage of **clim: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. It also defines a macro named **define-editor-command** which the application programmer will use to define commands for the **editor** application and install them in the command table **editor-command-table**.

```
(clim:define-application-frame editor ()
    ()
  (:command-table editor-command-table)
  (:command-definer define-editor-command)
  ...)
```

Note that for this particular example, the **:command-table** and **:command-definer** options need not have been specified, since the names that they specify would be the ones which would be generated by default. These options normally are provided only when you want different names other 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 section "Defining Application Frames in CLIM" and see the macro **clim:define-application-frame** for a description of these options.

Command names and command line names

Every command has a *command name*, which is a symbol. The symbol names the function which 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 separate from the command's actual name. For command line interactions, the end user sees and uses the command line name. For example, the command **com-show-chart** would have a command line name of ''Show Chart''. When defining a command using **clim:define-command** (or the application's command defining macro), you can have a command line name generated automatically.

The automatically generated command line name consists of the command's name with the hyphens replaced by spaces, and the words capitalized; furthermore, if there is a prefix of `"com-"`, the prefix is removed. For example, if the command name is **com-show-file**, the command line name will be `"Show File"`.

The **define-editor-command** macro, which would automatically be generated by the above example fragment, is used to define a command for the **editor** application. **define-editor-command** is used in the same way as **clim:define-command**. However, rather than requiring that the programmer 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** (the same options as for **clim:define-command**), the programmer can provide the command via any number of the above mentioned interaction styles. For example, you could install the command in the **editor** application's menu as well as specify a single keystroke command accelerator for it.

This example defines a command whose command name is **com-save-file**. The **com-save-file** command will appear in the application's command menu, by the name ''Save File'' (which is automatically generated from the command name based on the same method as for command line names). The single keystroke `control-S` will also invoke the command.

```
(define-editor-command (com-save-file :menu t
                                      :keystroke (:s :control))
    ()
  ...)
```

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 **clim:command** and executing the resulting input, any displayed presentation can invoke a command so long as there is a translator defined which translates from the presentation type of the presentation to the presentation type **clim:command**. By this mechanism, the programmer can associate a command with a pointer gesture when applied to a displayed presentation. **clim:define-presentation-to-command-translator** will create such an association.

clim:define-presentation-to-command-translator *name (from-type command-name command-table* &key *(:gesture* **':select***) :tester :documentation :pointer-documentation (:menu* **t***) :priority (:echo* **t***)) arglist* &body *body*
Defines a presentation translator that translates a displayed presentation into a command.

See the section "Making Commands From Presentations" for an example using **clim:define-presentation-to-command-translator**.

Command Objects in CLIM

What is a command?

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 arguments, 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:

clim:command-name *command*
Given a command object *command*, returns the command name.

clim:command-arguments *command*
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 **clim:*unsupplied-argument-marker*** is used in place of any argument which has not yet been specified.

clim:*unsupplied-argument-marker*
The value of **clim:*unsupplied-argument-marker*** serves as a placeholder in a command object for required arguments which have not yet been supplied.

clim:partial-command-p *command*
Returns **t** if the command object *command* is a partial command, otherwise returns **nil**.

One can think of **clim: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.

clim:define-command *name arguments* &body *body*
Defines a command and characteristics of the command, including its name, its arguments, and, as options: the command table in which it should appear, its keystroke accelerator, its command-line name, and whether or not (and how) to add this command to the menu associated with the command table.

CLIM Command Tables

CLIM command tables are represented by instances of the CLOS class **clim: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 associate command names with command line names. Command line names are used in the command line interaction style. They are the textual representation of the command name when presented and accepted.

A command table can describe a menu from which users can choose commands. A command table can support keystroke accelerators for invoking commands.

A command table can have a set of presentation translators and actions, defined by **clim:define-presentation-translator**, **clim:define-presentation-to-command-translator**, and **clim:define-presentation-action**. This allows the pointer to be used to input commands, including command arguments.

We say that a command is *present* in a command table when it has been added to the command table by being associated with some form of interaction. We say that a command is *accessible* in a command table when it is present in the command table or is present in any of the command tables from which the command table inherits.

clim:command-table
The class that represents command tables.

clim:command-table-name *command-table*
Returns the name of the command table *command-table*.

clim:command-table-inherit-from *command-table*
Returns a list of all of the command tables from which *command-table* inherits.

clim:find-command-table *name* &key *(:errorp* **t***)*
Returns the command table named by *name*.

clim:define-command-table *name* &key *:inherit-from :menu :inherit-menu*
Defines a new command table.

clim:make-command-table *name* &key *:inherit-from :menu :inherit-menu (:errorp* **t***)*

Creates a command table named *name* that inherits from *:inherit-from* and has a menu specified by *:menu*.

A command table can inherit from other command tables. This allows larger sets of commands to be built up through the combination of smaller sets. In this way, a tree of command tables can be constructed. During command lookup, if a command is not found in the application's command table, then the command tables from which that command table inherits are searched also. It is only when the entire tree is exhausted that an error is signalled.

CLIM provides several command tables from which it is recommended that your application's command table inherit. See the section "CLIM's Predefined Command Tables" for a description of these command tables.

The macro **clim:do-command-table-inheritance** is provided as a facility for programmers to walk over a command table and the command tables it inherits from in the proper precedence order.

clim:do-command-table-inheritance *(command-table-var command-table)* &body *body*
Successively evaluates *body* with *command-table-var* bound first to the command table *command-table*, and then to all of the command tables from which *command-table* inherits.

These functions are provided for examining and altering the commands in a command table:

clim:add-command-to-command-table *command-name command-table* &key *:name :menu :keystroke (:errorp* **t***)*
Adds the command named by *command-name* to the command table *command-table*.

clim:remove-command-from-command-table *command-name command-table* &key *(:errorp* **t***)*
Removes the command named by *command-name* from the command table *command-table*.

clim:command-present-in-command-table-p *command-name command-table*
Returns **t** if *command-name* is present in *command-table*.

clim:command-accessible-in-command-table-p *command-name command-table*
Returns **t** if *command-name* is accessible in *command-table*.

clim:map-over-command-table-commands *function command-table* &key *(:inherited* **t***)*
Applies *function* to all of the commands accessible in *command-table*.

CLIM's Predefined Command Tables

CLIM provides these command tables:

clim:global-command-table
The ''global'' command table from which all command tables inherit.

clim:user-command-table
A command table reserved for user-defined commands.

clim:accept-values-pane
When you use an **clim:accept-values** pane in a **clim:define-application-frame**, you must inherit from this command table.

It is recommended that an application's command table inherit from **clim:user-command-table**. **clim:user-command-table** inherits from **clim:global-command-table**. If your application uses an **:accept-values** pane, then its command table must inherit from the **clim:accept-values-pane** command table in order for it to work properly.

Conditions Relating to CLIM Command Tables

Command table operations can signal these conditions:

clim:command-table-already-exists
This condition is signalled by **clim:make-command-table** when you try to create a command table that already exists.

clim:command-table-not-found
This condition is signalled by functions such as **clim:find-command-table** when the named command table cannot be found.

clim:command-not-present
A condition that is signalled when the command you are looking for is not present in the command table.

clim:command-not-accessible
A condition that is signalled when the command you are looking for is not accessible in the command table.

clim:command-already-present
A condition that is signalled when a command is already present in the command table.

Styles of Interaction Supported by CLIM

CLIM supports four main styles of interaction:

- Mouse interaction via command menus.
- Mouse interaction via translators, including direct manipulation ("drag and drop") interactions.
- Keyboard interaction using a command-line processor.
- Keyboard interaction using keystroke accelerators.

See the section "Defining Commands the Easy Way" for a simple description of how to use **clim:define-command** to associate a command with any of these interaction styles.

The following subsections describe these interaction styles.

CLIM's 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.

These menu items are typically defined using the **:menu** option to **clim:define-command** (or the application's command defining macro).

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.

clim:display-command-table-menu *command-table stream* &key *:max-width :max-height :n-rows :n-columns :x-spacing :y-spacing (:cell-align-x* **':left***) (:cell-align-y* **':top***) (:initial-spacing* **t***) :row-wise :move-cursor*
Displays the menu for *command-table* on *stream*.

clim:display-command-menu *frame stream* &key *:command-table :max-width :max-height :n-rows :n-columns (:cell-align-x* **':left***) (:cell-align-y* **':top***)*
Displays the menu described by the command table associated with the application frame *frame* onto *stream*. Disabled items in the menu will be ''grayed out''.

clim:menu-choose-command-from-command-table *command-table* &key *(:associated-window* **(clim:frame-top-level-window clim:*application-frame*)***) :text-style :label :cache (:unique-id* **clim:command-table***) (:id-test* **#'eql***) :cache-value (:cache-test* **#'eql***)*
Displays a menu of all of the commands in *command-table*'s menu, and waits for the user to choose one of the commands. The returned value is a command object.

A number of lower level functions for manipulating command menus are also provided:

clim:add-menu-item-to-command-table *command-table string type value* &key *:documentation (:after* **':end***) :keystroke :text-style (:errorp* **t***)*
Adds a command menu item to *command-table*'s menu.

clim:remove-menu-item-from-command-table *command-table string* &key *(:errorp* **t***)*
Removes the item named by *string* from *command-table*'s menu.

clim:map-over-command-table-menu-items *function command-table*
Applies *function* to all of the menu items in *command-table*'s menu.

clim:find-menu-item *menu-name command-table* &key *(:errorp* **t***)*
Given a *menu-name* and a *command-table*, return two values, the command menu item and the command table in which it was found.

clim:command-menu-item-type *item*
Returns the type of the command menu item *item*.

clim:command-menu-item-value *item*
Returns the value of the command menu item *item*. For example, if the type of *item* is **:command**, this will return a command or a command name.

clim:command-menu-item-options *item*
Returns a property list of the options for the command menu item *item*.

clim:command-enabled *command-name frame* &optional *command-table*
Returns **t** if the command named by *command-name* is presently enabled in *command-table* for the frame *frame*, otherwise returns **nil**. You can use **setf** on **clim:command-enabled** in order to enable or disable a command.

clim:command-menu-enabled *command-table frame*
Returns **t** if the command table *command-table* is presently enabled in the command menu for the frame *frame*, otherwise returns **nil**. This function is like **clim:command-enabled**, except that it operates only on the **:menu** items in a command table's menu for a particular frame.

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 pointer gesture (that is, 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.

clim: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 an computer aided design system for electrical circuits. You might have a translator which 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 to enter its resistance.

CLIM also supports a drag and drop interaction style via a special kind of presentation translators that takes into account both a ''source'' object and a ''destination'' object. For example, an interaction that involves dragging a pathname object over a trashcan object might result in a command that causes the specified file to be deleted. You can use **clim:define-drag-and-drop-translator** to define such translators.

For a discussion of the facilities supporting the mouse translator interaction style, see the section "Presentation Types in CLIM".

CLIM's 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 (see the section "Input Editing and Built-in Keystroke Commands in CLIM") and by the prompting and help facilities, which provide a description of the command and the expected arguments. Command entry is also facilitated by the presentation substrate which allows the input of objects matching the input context, both for command names and command arguments.

See the section "Presentation Types in CLIM" for a detailed description.

clim:find-command-from-command-line-name *name command-table* &key *(:errorp* **t***)*

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.

clim:command-line-name-for-command *command-name command-table* &key *(:errorp* **t***)*

Returns the command-line name for *command-name* as it is installed in *command-table*.

clim:map-over-command-table-names *function command-table* &key *(:inherited* **t***)*

Applies *function* to all of the command-line names accessible in *command-table*.

CLIM's Keystroke Interaction Style

Each command table may have a mapping from keystroke accelerators to either command objects or submenus. This mapping is similar to that for menu items as the programmer might provide a single keystroke equivalent to a command menu item.

Since the kinds of characters that can be typed in vary widely from one platform to another, you should 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 **clim:define-command** (or the application's command defining macro).

clim:add-keystroke-to-command-table *command-table keystroke type value* &key *:documentation (:errorp* **t***)*
Adds a keystroke accelerator to the *command-table*.

clim:remove-keystroke-from-command-table *command-table keystroke* &key *(:errorp* **t***)*
Removes the item named by *keystroke* from *command-table*'s accelerator table. *command-table* may be either a command table or a symbol that names a command table.

clim:map-over-command-table-keystrokes *function command-table*
Applies *function* to all of the keystroke accelerators in *command-table*'s accelerator table.

clim:find-keystroke-item *keystroke command-table* &key *:test (:errorp* **t***)*
Given a keystroke accelerator *keystroke* and a *command-table*, returns two values, the command menu item associated with the keystroke and the command table in which it was found.

clim:lookup-keystroke-item *keystroke command-table* &key *:test*
Like **clim:find-keystroke-item**, except that it descends into sub-menus in order to find a keystroke accelerator matching *keystroke*.

clim:lookup-keystroke-command-item *keystroke command-table* &key *:test (:numeric-argument* **1***)*
Like **clim:lookup-keystroke-item**, except that it searches only for enabled commands.

Because of the potential ambiguity between keystroke accelerators and normal typed input, the default CLIM command loop does not handle keyboard accelerators.

In order to use keystroke accelerators, your application will need to specialize the **clim:read-frame-command** generic function. The default method for **clim:read-frame-command** just calls **clim:read-command**. You can specialize it to call **clim:read-command-using-keystrokes** within the context of **clim:with-command-table-keystrokes**:

```
(defmethod clim:read-frame-command ((frame my-application) &key)
  (let ((command-table (clim:find-command-table 'my-command-table)))
    (clim:with-command-table-keystrokes (keystrokes command-table)
      (clim:read-command-using-keystrokes command-table keystrokes))))
```

clim:with-command-table-keystrokes *(keystroke-var command-table)* &body *body*
Binds *keystroke-var* to a list that contains all of the keystroke accelerators in the command table *command-table*, and then evaluates *body* in that context.

clim:read-command-using-keystrokes *command-table keystrokes* &key *(:stream* ***query-io****)* *(:command-parser* **clim:*command-parser****)* *(:command-unparser* **clim:*command-unparser****)* *(:partial-command-parser* **clim:*partial-command-parser****)*
Reads a command from the user via a command line, the pointer, or typing a single keystroke.

If your application also employs the command line interaction style there is the potential for ambiguity as to whether a character is intended as command line input, a keystroke accelerator, or an input editing command (see the section "Input Editing and Built-in Keystroke Commands in CLIM"). For this reason, it is recommended that you choose keystroke accelerators that do not conflict with the standard printed character set (which might be used for command names and the textual representations of arguments) or with the input editor. CLIM will make some attempt to resolve such conflicts if they arise. A keystroke accelerator can only be invoked if there is no other pending command line input. If there is pending input, keystroke accelerators will not be considered and the keystroke will be interpreted as input or as an input editor command. If there is no pending input, the keystroke accelerator behavior will take precedence over that of the input editor.

The way CLIM processes keystroke accelerators is that **clim:stream-read-gesture** checks to see if keystroke is one of the gestures in **clim:*accelerator-gestures***. If it is, CLIM signals a condition of type **clim:accelerator-gesture**. If you need more control over keystroke accelerators than is provided by **clim:read-command-using-keystrokes**, you can use the following:

clim:*accelerator-gestures*
A list of gestures that CLIM will treat as keystroke accelerators when reading commands.

clim:accelerator-gesture
CLIM signals an **clim:accelerator-gesture** condition whenever it reads an accelerator gesture from the user.

clim:accelerator-gesture-event *accelerator-gesture*
Returns the event object that caused the accelerator gesture condition, *accelerator-gesture*, to be signalled.

clim:accelerator-gesture-numeric-argument *accelerator-gesture*
Returns the numeric argument associated with the accelerator gesture condition, *accelerator-gesture*.

For a description of the CLIM command processor, see the section "The CLIM Command Processor".

Command Related Presentation Types

CLIM provides several presentation types pertaining to commands:

clim:command &key *:command-table*
: The presentation type used to represent a CLIM command processor command and its arguments.

clim:command-name &key *:command-table*
: The presentation type used to represent the name of a CLIM command processor command in the command table *:command-table*.

clim:command-or-form &key *:command-table*
: The presentation type used to represent either a Lisp form or a CLIM command processor command and its arguments.

The CLIM Command Processor

This section describes the default behavior of the CLIM command processor.

The command loop of a CLIM application is performed by the application's top-level function (see the section "Defining Application Frames in CLIM"). By default, this is **clim:default-frame-top-level**. After performing some initializations, **clim:default-frame-top-level** enters an infinite loop, reading and executing commands. It invokes the generic function **clim:read-frame-command** to read a command which is then passed to the generic function **clim: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 **clim:default-frame-top-level** with your own top level function.

clim:read-frame-command invokes the command parser by establishing an input context of **clim: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 using the pointer.

clim:read-command *command-table* &key *(:stream* ***query-io***) *(:command-parser* **clim:*command-parser***) *(:command-unparser* **clim:*command-unparser***) *(:partial-command-parser* **clim:*partial-command-parser***) *:use-keystrokes*
: Reads a command. This function is not normally called by programmers.

clim:read-frame-command *frame &key :stream*
: **clim:read-frame-command** reads a command from the user on the stream *:stream*, and returns the command object. *frame* is an application frame. You can specialize this generic function for your own application

frames, for example, if you want to have your application be able to read commands using keystroke accelerators.

clim:execute-frame-command *frame command*

clim:execute-frame-command executes the command *command* on behalf of the application frame *frame*. You can specialize this function if you want to change the behavior associated with the execution of commands.

An application can control which commands are enabled and which are disabled on an individual basis. Use **setf** on **clim:command-enabled** to control this mechanism. The user is not allowed to enter a disabled command via any interaction style.

clim:command-enabled *command-name frame* &optional *command-table*

Returns **t** if the command named by *command-name* is presently enabled in *command-table* for the frame *frame*, otherwise returns **nil**. You can use **setf** on **clim:command-enabled** in order to enable or disable a command.

The special variable **clim:*command-dispatchers*** controls the behavior of the **clim:command-or-form** presentation type.

clim:*command-dispatchers*

This is a list of characters that indicate that CLIM should read a command when CLIM is accepting input of type **clim:command-or-form**.

Menus and Dialogs in CLIM

Concepts of Menus and Dialogs in CLIM

CLIM provides three powerful menu interaction routines for allowing user interfacing through pop-up menus and dialogs, and menus and dialogs embedded in an application window:

- **clim: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 item, see the function **clim:menu-choose**.
- **clim: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.
- **clim:accepting-values** provides the ability to build a dialog. You can specify several items that can be individually selected or modified within the dialog before dismissing it. It differs from the 'Select One' style of **clim:menu-choose** and **clim:menu-choose-from-drawer**.

Operators for Dealing with Menus and Dialogs in CLIM

clim:menu-choose *items* &rest *keys* &key *:associated-window :text-style :foreground :background :default-item :label :scroll-bars :printer :presentation-type :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 :x-position :y-position :pointer-documentation :menu-type*
Displays a menu with the choices in *item-list*. It returns three values: the value of the chosen item, the item itself, and the gesture that selected it. If possible, CLIM will use the menu facilities provided by the host window system when you use **clim:menu-choose**.

clim:menu-choose-from-drawer *menu type drawer* &key *:x-position :y-position :cache :unique-id (:id-test* **#'equal***) (:cache-value* **t***) (:cache-test* **#'eql***) :leave-menu-visible :default-presentation*
The low-level routine used by CLIM for displaying menus.

clim:draw-standard-menu *menu presentation-type items default-item* &key *(:item-printer* **#'clim:print-menu-item***) :max-width :max-height :n-rows :n-columns :x-spacing :y-spacing :row-wise (:cell-align-x* **':left***) (:cell-align-y* **':top***)*
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.

clim:print-menu-item *menu-item* &optional *(stream* ***standard-output****)*
Given a menu item *menu-item*, display it on the stream *stream*.

clim:with-menu *(menu* &optional *associated-window* &rest *options* &key *:label :scroll-bars)* &body *body*
Binds *menu* to a temporary window, exposes the window on the same screen as the *associated-window*, runs the body, and de-exposes the window.

clim:*abort-menus-when-buried*
Indicates whether or not CLIM should abort out of menus when they are ''buried''.

clim:accepting-values *(*&optional *stream* &key *:frame-class :command-table :own-window :exit-boxes :align-prompts :initially-select-query-identifier :modify-initial-query :resynchronize-every-pass (:check-overlapping* **t***) :label :x-position :y-position :width :height :scroll-bars :text-style :foreground :background)* &body *body*
A macro that builds a dialog for user interaction based on calls to **clim:accept** within its body.

clim:accept-values-command-button *((*&optional *stream* &key *:documentation :query-identifier (:cache-value* **t***) (:cache-test* **#'eql***) :view :resynchronize) prompt* &body *body)*
Displays *prompt* on *stream* and creates an area (the ''button'') which, when the pointer is clicked within it, causes *body* to be evaluated. This function can only be used within the **clim:accepting-values** form.

Examples of Menus and Dialogs in CLIM

Example of using clim:accepting-values

This example sets up a dialog in the CLIM window **stream** that displays the current Month, Date, Hour and Minute (as obtained by a call to **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 RETURN. When done, the user selects ''End'' to accept the new values, or ''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~%")
    (conditions: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."
          month day hour minute))
      (abort () (format t "~&Time not set")))))
```

In CLIM, calls to **clim: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 **clim:accept**.

Example of using clim: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))
    (format stream "Enter the time~%")
    (conditions: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"))
          (terpri stream)
          (clim:accept-values-command-button (stream) "Zero seconds"
            (setq second 0)))
        ;; 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:~2,'0D."
          month day hour minute second))
      (abort () (format t "~&Time not set")))))
```

Using :resynchronize-every-pass in clim: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 **clim:accepting-values** you can achieve this effect by using the **:resynchronize-every-pass** argument to **clim:accepting-values** in conjunction with the **:default** argument to **clim:accept**. There are three points to remember:

- The entire body of the **clim: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 **clim:accept** is used, then every time that call to **clim:accept** is run, it will pick up the new value of the default.
- Inside **clim:accepting-values**, **clim: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 'real :default min :prompt "Min"
                                 :stream stream))
    (fresh-line stream)
    (setq max (clim:accept '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.)

Use of the third value from clim:accept in clim:accepting-values

As a second example, consider a dialog that accepts four real numbers that delimit a rectangular region in the plane, only 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

```
Xmax - Xmin = Ymax - Ymin
```

This constraint is a little harder to enforce. Presumably a user would be very disturbed if a value that he or she had just input was changed. So for this example we follow a policy that says if the user changed an X value, then only change Y values to enforce the constraint, and vice versa. When changing values we preserve the center of the interval. (This policy is somewhat arbitrary and only for the purposes of this example.) We use the third returned value from **clim:accept** to control the constraint enforcement.

```
(defun accepting-square (&key (xmin -1.0) (xmax 1.0) (ymin -1.0) (ymax 1.0)
                              (stream *query-io*))
  (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 'real :default xmin :prompt "Xmin"
                           :stream stream))
      (fresh-line stream)
      (multiple-value-setq (xmax ptype xmax-changed)
        (clim:accept 'real :default xmax :prompt "Xmax"
                           :stream stream))
      (fresh-line stream)
      (multiple-value-setq (ymin ptype ymin-changed)
        (clim:accept 'real :default ymin :prompt "Ymin"
                           :stream stream))
      (fresh-line stream)
      (multiple-value-setq (ymax ptype ymax-changed)
        (clim:accept '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 xmax-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 ymax-changed nil)))))
  (values xmin xmax ymin ymax))
```

Example of a dialog that uses gadgets

Try the following example to see a dialog with gadgets in it.

```
(defun gadget-dialog-test (&optional (stream *standard-input*))
  (let ((dest  :file)
        (name  "")
        (copies 0)
        (strip nil))
    (clim:accepting-values (stream :align-prompts t)
      (setq dest  (clim:accept '(member :file :printer :window)
                   :default dest :prompt "Destination type"
                   :stream stream :view clim:+gadget-dialog-view+))
      (setq name  (clim:accept 'string
                   :default name :prompt "Destination name"
                   :stream stream :view clim:+text-field-view+))
      (setq copies (clim:accept 'integer
                    :default copies :prompt "Number of copies"
                    :stream stream :view clim:+slider-view+))
      (setq strip (clim:accept 'boolean
                   :default strip :prompt "Strip text styles"
                   :stream stream :view clim:+gadget-dialog-view+)))
    (values dest name strip)))
```

Examples of using clim:menu-choose

These examples show how to use **clim:menu-choose**.

The simplest use of **clim:menu-choose**. If each item is *not* a list, the entire item will be printed and the entire item is the value to be returned too.

```
(clim:menu-choose '("One" "Two" "Seventeen"))
```

If you want to return a value that is different from what was printed, the simplest method is as below. 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 that **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 "what can be said about this?")))
```

The list of items you pass to **clim:menu-choose** might also serve some other purpose in your application. In that case, it might not be appropriate to put the printed appearance in the first element. You can supply a **:printer** function which 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 -8)))
                 (triangle (clim:draw-polygon*
                             stream '(10 8 0 -10 -10 8))))))
```

The **:items** option of the list form of menu item can be used to describe a set of hierarchical menus.

```
(clim:menu-choose
  '(("Class: Osteichthyes"
     :documentation "Bony fishes"
     :style (nil :italic nil))
    ("Class: Chondrichthyes"
     :documentation "Cartilagenous fishes"
     :style (nil :italic nil)
     :items (("Order: Squaliformes" :documentation "Sharks")
             ("Order: Rajiformes" :documentation "Rays")))
    ("Class: Mammalia"
     :documentation "Mammals"
     :style (nil :italic nil)
     :items (("Order Rodentia"
              :items ("Family Sciuridae"
                      "Family Muridae"
                      "Family Cricetidae"
                      ("..." :value nil)))
             ("Order Carnivora"
              :items ("Family: Felidae"
                      "Family: Canidae"
                      "Family: Ursidae"
                      ("..." :value nil)))
             ("..." :value nil)))
    ("..." :value nil)))
```

Examples of using clim:menu-choose-from-drawer

This example displays in the window ***page-window*** the choices ‘‘One’’ through ‘‘Ten’’ in bold type face. When the user selects one, the string is returned along with the gesture that selected it.

```
(clim:menu-choose-from-drawer *page-window* 'string
  #'(lambda (stream type)
      (clim:with-text-face (:stream bold)
        (dotimes (count 10)
          (clim:present
            (string-capitalize
              (format nil "~R" (1+ count)))
            type :stream stream)
          (terpri stream)))))
```

This example shows how you can use **clim:menu-choose-from-drawer** with **clim:with-menu** to create a temporary menu:

```
(defun choose-compass-direction ()
  (labels ((draw-compass-point (stream ptype symbol x y)
             (clim:with-output-as-presentation (stream symbol 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)
      (clim:menu-choose-from-drawer
        menu 'clim:menu-item #'draw-compass))))
```

Output Recording in CLIM

Concepts of CLIM Output Recording

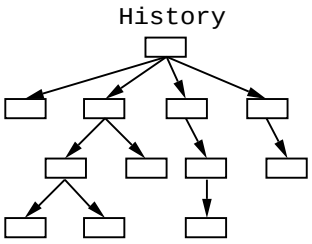
Output recording is a fundamental part of CLIM. It provides the basis for scrolling windows, for formatted output of tables and graphs, for the ability of presentations to retain their semantics, and for incremental redisplay.

The output recording mechanism is enabled by default. Unless you turn it off, all output that occurs on a stream is captured and saved by the output recording mechanism. The output is captured in *output records*. The top-level output record, which contains all the output done on that stream, is called the *history* of the stream.

An output record is:

- an object that contains more output records, or
- a displayed output record (that is, a record that corresponds directly to something drawn on the display device).

Since output records can contain other output records, we can view the organization of output records as a tree structure:



Each rectangle is an output record. The top-level record is an output record called a history. Each output record that is a leaf of the tree is called a displayed output record. The intermediate output records are both output records and children of their immediate superior.

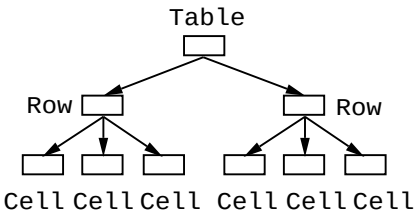
CLIM automatically segments the output into output records. The result of each atomic drawing operation is put into a new output record. Each presentation is put into a new output record. (Strings are treated differently; CLIM concatenates strings into one output record until a newline is encountered, which begins a new output record.)

One use of an output record is to *replay* it; 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, then 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 **clim:formatting-table**, and formats 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 illustrated by the organization of output records of a formatted table. The table itself is stored in an output record; each row has its own output record; each cell has its own output record.



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 *displayed output records*. An output record is an object that stores other output records. Displayed output records are the leaf objects that correspond to something visible on a display device. The following classes and predicates correspond to the objects used in output recording.

clim:output-record
: The protocol class that is used to indicate that an object is an *output record*, that is, a CLIM object that contains other output records.

clim:output-record-p *object*
: Returns **t** if and only *object* is of type **clim:output-record**.

clim:displayed-output-record
: The protocol class that is used to indicate that an object is a *displayed output record*, that is, a CLIM object that represents a visible piece of output on an output device.

clim:displayed-output-record-p *object*
: Returns **t** if and only *object* is of type **clim:displayed-output-record**.

The following functions and macros can be used to create and operate on CLIM output records.

clim:replay *record stream* &optional *region*
: Replays the output record *record* on *stream*.

clim:replay-output-record *record stream* &optional *region x-offset y-offset*
: Replays all of the output captured by the output record *record* on *stream*. If *region* is not **nil**, then *record* is replayed if and only if it overlaps region.

clim:with-output-recording-options *(stream* &key *:draw :record)* &body *body*
: Used to disable output recording and/or drawing on the given *stream*, within the extent of *body*.

clim:with-new-output-record *(stream* &optional *record-type record* &rest *initargs)* &body *body*
: Creates a new output record, installs it in the output history of the *stream*, and then evaluates the *body*.

clim:with-output-to-output-record *(stream* &optional *record-type record* &rest *initargs)* &body *body*
This is similar to **clim:with-new-output-record** except that the new output record is not inserted into the output record hierarchy.

clim:output-record-parent *record*
Returns the output record that it the parent of *record*. If *record* has no parent, **clim:output-record-parent** will return **nil**.

clim:output-record-children *record*
Returns a sequence of output records that are the children of the output record *record*. If *record* has no children, this will return **nil**.

clim:erase-output-record *record stream* &optional *(errorp* **t***)*
Erases the display of the output record *record* from *stream*, and removes the record from *stream*'s output history.

clim:add-output-record *child record*
Adds the output record *child* to the output record *record*.

clim:delete-output-record *child record* &optional *errorp*
Removes the child output record *child* from the output record *record*.

clim:clear-output-record *record*
Removes all of the child output records from the output record *record*.

clim:recompute-extent-for-new-child *record child*
CLIM calls this function whenever a new child is added to an output record. It updates the bounding rectangle of *record* to be large enough to completely contain the new child output record *child*.

clim:recompute-extent-for-changed-child *record child old-left old-top old-right old-bottom*
CLIM calls this function whenever the bounding rectangle of one of the children of a record has been changed. It updates the bounding rectangle of *record* to be large enough to completely contain the new bounding rectangle of the child output record *child*.

clim:tree-recompute-extent *record*
You can use this function whenever the bounding rectangles of a number of children of a record have been changed, such as happens during table and graph formatting. **clim:tree-recompute-extent** computes the bounding rectangle large enough to contain all of the children of *record*, adjusts the bounding rectangle of *record* accordingly, and then calls **clim:recompute-extent-for-changed-child** on *record*.

The following functions can be used to apply a function to all of the children of an output record.

clim:map-over-output-records *function record* &optional *(x-offset* **0***) (y-offset* **0***)* &rest *continuation-args*
Applies *function* to all of the child output records in the output record *record*. Normally, *function* is called with a single argument, an output

record. If *continuation-args* are supplied, they are passed to *function* as well.

clim:map-over-output-records-containing-position *function record x y* &optional *x-offset y-offset* &rest *continuation-args*
Applies *function* to all of the child output records in the output record *record* that overlap the point *(x,y)*. Normally, *function* is called with a single argument, an output record. If *continuation-args* are supplied, they are passed to *function* as well.

clim:map-over-output-records-overlapping-region *function record region* &optional *x-offset y-offset* &rest *continuation-args*
Applies *function* to all of the child output records in the output record *record* that overlap the region *region*. Normally, *function* is called with a single argument, an output record. If *continuation-args* are supplied, they are passed to *function* as well.

The following functions control pointer sensitivity and highlighting for output records.

clim:output-record-refined-position-test *record x y*
CLIM uses **clim:output-record-refined-position-test** to definitively determine that the point *(x,y)* is contained within the output record *record*.

clim:highlight-output-record *record stream state*
CLIM calls this method in order to draw highlighting for the output record *record* on *stream*. *state* is either **:highlight** (meaning to draw the highlighting) or **:unhighlight** (meaning to erase the highlighting).

In the present implementation of CLIM, the coordinates of output records are maintained relative to the coordinates of the output record's parent. Therefore, you must maintain the correct offsets while recursively mapping over output records, as the following example shows.

```
(defun describe-record (record &optional region)
  (labels ((describe (record x-offset y-offset)
             (format t "~&~S: " record)
             (clim:with-bounding-rectangle* (left top right bottom) record
               (incf left x-offset)
               (incf top y-offset)
               (incf right x-offset)
               (incf bottom y-offset)
               (format t "~&  (~D,~D):(~D,~D)" left top right bottom)
               (multiple-value-bind (xoff yoff)
                   (clim:output-record-position record)
                 (clim:map-over-output-records-overlapping-region
                   record region #'describe
                   (- x-offset) (- y-offset)
                   (+ x-offset xoff) (+ y-offset yoff))))))
    (declare (dynamic-extent #'describe))
    (describe record 0 0)))
```

The relevant functions for locating an output record are:

clim:output-record-position *record*
: Returns the X and Y position of *record* as two real numbers. The position is relative to the output record's parent, where (0,0) is the upper-left corner of the parent output record.

clim:output-record-set-position *record x y*
: Changes the position of the output record *record* to the new position *x* and *y*.

clim:convert-from-relative-to-absolute-coordinates *stream output-record*
: Returns the X and Y offsets that map the parent-relative coordinates of an output record to ''absolute'' coordinates.

clim:convert-from-absolute-to-relative-coordinates *stream output-record*
: Returns the X and Y offsets that map the ''absolute'' coordinates of an output record to parent-relative coordinates.

clim:translate-coordinates *x-delta y-delta* &body *coordinate-pairs*
: Translates each of the X and Y coordinate pairs in *coordinate-pairs* by *x-delta* and *y-delta*.

The following functions can be used to operate on output recording streams.

clim:output-recording-stream-p *object*
: Returns **t** if and only if *object* is an output recording stream.

clim:stream-output-history *stream*
: Returns the top level output record for the stream *stream*.

clim:stream-replay *stream* &optional *region*
: Replays all of the output records in *stream*'s output history that overlap the region *region*. If *region* is **nil**, all of the output records are replayed.

clim:stream-drawing-p *stream*

Returns **t** if and only if drawing is enabled on the output recording stream *stream*. You can use **setf** on this to enable or disable drawing on the stream, or you can use the **:draw** option to **clim:with-output-recording-options**.

clim:stream-recording-p *stream*

Returns **t** if and only if output recording is enabled on the output recording stream *stream*. You can use **setf** on this to enable or disable output recording on the stream, or you can use the **:record** option to **clim:with-output-recording-options**.

clim:stream-add-output-record *stream record*

Adds the new output record *record* to *stream*'s current output record (that is, **clim:stream-current-output-record**).

clim:stream-current-output-record *stream*

The current ''open'' output record for the output recording stream *stream*, that is, the one to which **clim:stream-add-output-record** will add a new child record.

clim:copy-textual-output-history *window stream* &optional *region record*

Given a window *window* that supports output recording, this function finds all of the textual output records that overlap the region *region* (or all of the textual output records is *region* is not supplied), and outputs that text to *stream*.

Bounding Rectangles in CLIM

Concepts of Bounding Rectangles

Every bounded region has a derived bounding rectangle, which is a rectangular region whose sides are parallel to the coordinate axes. The bounding rectangle for a region is the smallest rectangle that contains every point in the region, and may contain additional points as well. Unbounded regions do not have a bounding rectangle.

The bounding rectangle for an output record may have a different size depending on the display device on which the output record is drawn. Consider the case of drawing text on different output devices; the font chosen for a particular text style may vary considerably in size from one device to another.

Bounding rectangles can be used for a variety of purposes. For example, repainting of windows is driven from bounding rectangles. **clim:formatting-table** and **clim:format-graph-from-root** run their constraint engines on bounding rectangles. Bounding rectangles are also used internally by CLIM to achieve greater efficiency. For instance, hit detection is done by initially seeing if a point is inside the bounding rectangle.

For output records that establish a new coordinate system (for example, records created by **clim:formatting-row** or **clim:formatting-cell**), the bounding rectangle of that record plays an additional important role: it establishes the coordinate system to which all inferior output records refer. The origin of the coordinate system in which the inferior records reside is the top left corner of the superior's bounding rectangle.

CLIM Operators for Bounding Rectangles

clim:with-bounding-rectangle* *(left top right bottom) region* &body *body*
Binds *left*, *top*, *right*, and *bottom* to the edges of the bounding rectangle of *region*, and then evaluates *body* in that context.

clim:bounding-rectangle* *region*
Returns the bounding rectangle of *region* as four real numbers that specify the left, top, right, and bottom edges of the bounding rectangle.

clim:bounding-rectangle *region*
Returns a new bounding rectangle for *region* as a **clim:standard-bounding-rectangle** object.

clim:make-bounding-rectangle *x1 y1 x2 y2*
Makes an object of class **clim:bounding-rectangle** whose edges are parallel to the coordinate axes. One corner is at *(left,top)* and the opposite corner is at *(right,bottom)*.

clim:standard-bounding-rectangle
The standard instantiable class for bounding rectangles in CLIM.

clim:bounding-rectangle-left *region*
Returns the coordinate of the left edge of the bounding rectangle of *region*.

clim:bounding-rectangle-top *region*
Returns the coordinate of the top edge of the bounding rectangle of *region*.

clim:bounding-rectangle-right *region*
Returns the coordinate of the right edge of the bounding rectangle of *region*.

clim:bounding-rectangle-bottom *region*
Returns the coordinate of the bottom edge of the bounding rectangle of *region*.

clim:bounding-rectangle-min-x *region*
Returns the coordinate of the left edge of the bounding rectangle of *region*.

clim:bounding-rectangle-min-y *region*
Returns the coordinate of the top edge of the bounding rectangle of *region*.

clim:bounding-rectangle-max-x *region*
: Returns the coordinate of the right edge of the bounding rectangle of *region*.

clim:bounding-rectangle-max-y *region*
: Returns the coordinate of the bottom edge of the bounding rectangle of *region*.

clim:bounding-rectangle-position *region*
: Returns the position of the bounding rectangle of *region* as two values, the left and top coordinates of the bounding rectangle.

clim:bounding-rectangle-set-position *region x y*
: Changes the position of the bounding rectangle of *region* to the new position *x* and *y*.

clim:bounding-rectangle-width *region*
: Returns the width of the bounding rectangle of *region*.

clim:bounding-rectangle-height *region*
: Returns the height of the bounding rectangle of *region*.

clim:bounding-rectangle-size *region*
: Returns the size (as two values, width and height) of the bounding rectangle of *region*.

For example, the size of a the output generated by *body* can be determined by calling **clim:bounding-rectangle-size** on the output record:

```
(let ((record (clim:with-output-to-output-record (s) body)))
  (multiple-value-bind (width height)
      (clim:bounding-rectangle-size record)
    (format t "~&Width is ~D, height is ~D" width height)))
```

Formatted Output in CLIM

Formatting Tables in CLIM

Concepts of CLIM Table Formatting

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, as though it has its own drawing surface. You write code which is designed to put ink on a drawing plane. That ink might be text, graphics, or both. CLIM surrounds all the ink with a bounding box (or, more precisely, an axis-aligned rectangle). That bounding box is snipped out of the drawing plane and placed in a

cell of the table. CLIM's table formatter puts whitespace around the ink to make sure that all the cells have the proper size and alignment.

You are responsible only for specifying the contents of the cell. CLIM's table formatter is responsible for figuring out how to lay out the table so that all the cells fit together properly. The table formatter determines the width of each column based on the the widest cell within the column. Similarly, it determines the height of each row based on the the tallest cell within the row.

All the cells in a row all 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 control how CLIM should place the objects within the cell by aligning them both vertically (to the top, bottom, or center of the cell) and horizontally (to the left, right, or center of the cell).

You can specify other constraints that affect the appearance of the table (such as, the spacing between rows or columns, or the width or length of the table).

Formatting Item Lists in CLIM

Where table formatting is a ''two-dimensional'' operation 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, where a single column or row of menu entries would be fine (if the list is small enough). In this case, formatting is done when viewport requirements make it desirable.

These constraints affect the appearance of item lists:

- The number of rows (allowing CLIM to choose the number of columns)
- The number of columns (allowing CLIM to choose the number of rows)
- The maximum height (or width) of the column (letting CLIM determine the number of rows and columns that satisfy that constraint)

CLIM Operators for Table Formatting

This section summarizes the CLIM operators. For more complete documentation of each operator, see the section "Dictionary of CLIM Operators".

These are the general-purpose table formatting operators:

clim:formatting-table *(*&optional *stream* &rest *options* &key *:x-spacing :y-spacing :record-type :multiple-columns :multiple-columns-x-spacing :equalize-column-widths (:move-cursor* **t***))* &body *body*
Establishes a ''table formatting'' context on the *stream*. All output performed within the extent of this macro will be displayed in tabular form. This must be used in conjunction with **clim:formatting-row** or **clim:formatting-column**, and **clim:formatting-cell**.

clim:formatting-row *(*&optional *stream* &key *:record-type)* &body *body*
Establishes a ''row'' context on the *stream*. All output performed on the

stream within the extent of this macro will become the contents of one row of a table. **clim:formatting-row** must be used within the extent of **clim:formatting-table**, and it must be used in conjunction with **clim:formatting-cell**.

clim:formatting-column *(*&optional *stream* &key *:record-type)* &body *body*
Establishes a ''column'' context on the *stream*. All output performed on the stream within the extent of this macro will become the contents of one column of the table. **clim:formatting-column** must be used within the extent of **clim:formatting-table**, and it must be used in conjunction with **clim:formatting-cell**.

clim:formatting-cell *(*&optional *stream* &rest *options* &key *(:align-x* **':left***) (:align-y* **':top***) :min-width :min-height :record-type* &allow-other-keys*)* &body *body*
Establishes a ''cell'' context on the *stream*. All output performed on the stream within the extent of this macro will become the contents of one cell in a table. **clim:formatting-cell** must be used within the extent of **clim:formatting-row**, **clim:formatting-column**, or **clim:formatting-item-list**.

These are the one-dimensional table formatting operators:

clim:formatting-item-list *(*&optional *stream* &key *:record-type :x-spacing :y-spacing :initial-spacing :n-columns :n-rows :max-width :max-height :stream-width :stream-height (:row-wise* **t***) (:move-cursor* **t***))* &body *body*
Use this macro to format the output in a tabular form, when the exact ordering and placement of the cells is not important. **clim:formatting-item-list** must be used with **clim:formatting-cell**.

clim:format-items *items* &key *(:stream* ***standard-output****) :printer :presentation-type :x-spacing :y-spacing :initial-spacing :n-rows :n-columns :max-width :max-height (:row-wise* **t***) :record-type (:cell-align-x* **':left***) (:cell-align-y* **':top***)*

Provides tabular formatting of a list of items. Each item in *items* is formatted as a separate cell within the table.

clim:format-textual-list *sequence printer* &key *(:stream* ***standard-output****) (:separator* **", "***) :conjunction*
Outputs a *sequence* of items as a textual list.

clim:format-items is similar to **clim:formatting-item-list**. Both operators do the same thing, except they accept their input differently:

- **clim:formatting-item-list** accepts its input as a body that calls **clim:formatting-cell** for each item.
- **clim:format-items** accepts its input as a list of items with a specification of how to print them.

In CLIM, menus use the one-dimensional table formatting model.

Examples of CLIM Table Formatting

Formatting a table from a list

The **example1** function formats a simple table whose contents come from a list.

```
(defvar *alphabet* '(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))

(defun example1 (&optional (items *alphabet*)
             &key (stream *standard-output*)
                  (n-columns 6) x-spacing y-spacing)
  (clim:formatting-table (stream :x-spacing x-spacing
                                 :y-spacing y-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
```

The table above shows the result of evaluating **example1** form without providing the **:x-spacing** and **:y-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

```
(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
```

(Note that this example could also be done with **clim:formatting-item-list** or **clim:format-items**, as shown in **example4** below.)

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 day 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.

```
(defvar *days-of-the-week* (vector "Sun" "Mon" "Tue" "Wed" "Thu" "Fri" "Sat"))

(defvar *month-lengths* (vector 31 28 31 30 31 30 31 31 30 31 30 31))
(defun days-in-month (month year)
  (if (= month 2)
      (if (zerop (mod year 4))
          (if (zerop (mod year 400)) 28 29)
          28)
      (svref *month-lengths* (1- month))))

(defun display-calendar (month year &key (stream *standard-output*))
  (let ((days-in-month (days-in-month month year)))
    (multiple-value-bind (nil nil nil nil nil nil 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 :x-spacing " " :y-spacing 2)
        (clim:formatting-row (stream)
          (dotimes (d 7)
            (clim:formatting-cell (stream :align-x :center)
              (clim:with-text-face (stream :italic)
                (write-string (svref *days-of-the-week* (mod d 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)))))))))))
```

Evaluating `(calendar-month 5 90 :stream *my-window*)` shows this table:

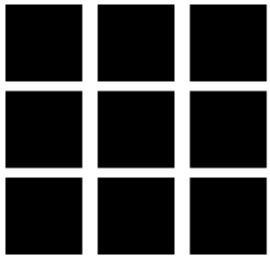
```
Sun Mon Tue Wed Thu Fri Sat
                1   2   3   4   5
  6   7   8   9  10  11  12
 13  14  15  16  17  18  19
 20  21  22  23  24  25  26
 27  28  29  30  31
```

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. Notice that, even though the example passes the same coordinates to **clim:draw-rectangle*** for each cell, the resulting table consists of a number of non-overlapping rectangles.

```
(defun example2 (&key (stream *standard-output*) x-spacing y-spacing)
  (clim:formatting-table (stream :x-spacing x-spacing
                                 :y-spacing y-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-window* :y-spacing 5)` shows this table:

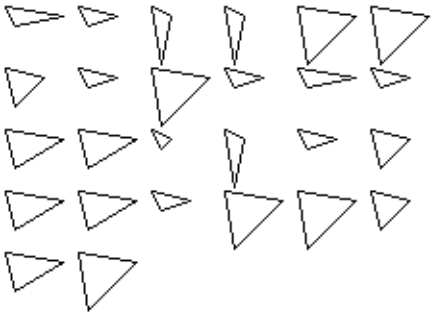


Formatting a table with irregular graphics in the cells

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) x-spacing y-spacing)
  (clim:formatting-table (stream :x-spacing x-spacing
                                 :y-spacing y-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-window*)` shows this table:



Formatting a table of a sequence of items: clim:formatting-item-list

The **example4** function shows how you can use **clim: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 must use **clim:formatting-cell** inside the body of **clim:formatting-item-list** to output each item. You do not use **clim:formatting-column** or **clim: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
                    x-spacing y-spacing max-width max-height)
  (clim:formatting-item-list
    (stream :x-spacing x-spacing :y-spacing y-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:

```
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
```

You can easily add a constraint specifying the number of columns.

Evaluating `(example4 :stream *my-window* :n-columns 8)` 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
```

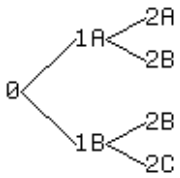
Formatting Graphs in CLIM

Concepts of CLIM Graph Formatting

When you need to format a graph, you specify the nodes to be in the graph, and the scheme for organizing them. CLIM's graph formatter does the layout automatically, obeying any constraints that you supply.

You can format any directed, acyclic graph (DAG). ''Directed'' means that the arcs on the graph have a direction. ''Acyclic'' means that there are no loops (or cycles) in the graph.

Here is an example of such a graph:



To specify the elements and the organization of the graph, you provide to CLIM the following information:

- The root node.
- A ''node printer'', 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 which takes one node and returns its inferior nodes (the nodes to which it points).

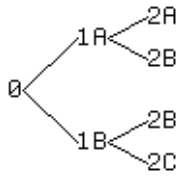
Based on that information, CLIM lays out the graph for you. You can specify a number of options that control the appearance of the graph. For example, you can specify whether you want the graph to grow vertically (downward) or horizontally

(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.

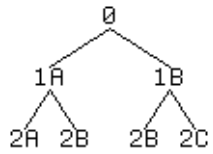
Examples of CLIM Graph Formatting

```
(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:



As shown above, by default, the graph has a horizontal orientation and grows toward the right. We can supply the **:orientation** keyword to control this. Evaluating `(test-graph g1 :stream *my-window* :orientation :vertical)` results in the following graph:



CLIM Operators for Graph Formatting

The following two functions can be used to format a graph.

clim:format-graph-from-roots *root-objects object-printer inferior-producer* &key *(:stream* ***standard-output***) *(:orientation* **':horizontal**) *:center-nodes :cut-*

off-depth :merge-duplicates :graph-type (:duplicate-key **#'identity***) (:duplicate-test* **#'eql***) :arc-drawer :arc-drawing-options :generation-separation :within-generation-separation :maximize-generations (:store-objects* **t***) (:move-cursor* **t***)*

Constructs and displays a directed, acyclic graph. *root-objects* is a sequence of the root elements of the set, from which the graph can be derived. *object-printer* is the function used to display each node of the graph; it takes to arguments, the object to display and the stream. *inferior-producer* is the function that to generates the inferiors from a node object; it takes one argument, the node, and returns a list of inferior nodes.

clim:format-graph-from-root *root-object object-printer inferior-producer* &key *(:stream* ***standard-output***) *(:orientation* **':horizontal***) :center-nodes :cut-off-depth :merge-duplicates :graph-type (:duplicate-key***#'identity***) (:duplicate-test* **#'eql***) :arc-drawer :arc-drawing-options :generation-separation :within-generation-separation :maximize-generations (:store-objects* **t***) (:move-cursor* **t***)*

Like **clim:format-graph-from-roots**, except that *root-object* is a single root object instead of a sequence of roots.

Formatting Text in CLIM

CLIM provides the following two forms for breaking up lengthy output into multiple lines and for indenting output.

clim:filling-output *(*&optional *stream* &rest *keys* &key *(:fill-width* **'(80 :character)***) (:break-characters* **'(#\Space)***) :after-line-break :after-line-break-initially)* &body *body*

Binds *stream* to a stream that inserts line breaks into the output written to it so that the output is no wider then *:fill-width*. The filled output is then written on the stream that is the original value of stream. **clim:filling-output** does not split ''words'' across lines.

clim:indenting-output *(stream indentation* &key *(:move-cursor* **t***))* &body *body*

Binds *stream* to a stream that inserts whitespace at the beginning of each line, and writes the indented output as the stream that is the original value of *stream*.

For example, you might use the following to generate a filled, indented list of the first twenty-one counting numbers:

```
(let ((stream *standard-output*))
  (write-line "Here are the first twenty-one counting numbers:" stream)
  (clim:indenting-output (stream '(2 :character))
    (clim:filling-output (stream :fill-width '(60 :character))
      (dotimes (i 19)
        (format stream "~R, " i))
      (format stream "and ~R." 20))))
```

Bordered Output in CLIM

CLIM provides a mechanism for surrounding arbitrary output with some kind of a border. To specify that a border should be generated, you surround some code that does output with **clim:surrounding-output-with-border**, an advisory macro that describes the type of border to be drawn.

clim:surrounding-output-with-border *(*&optional *stream* &key *(:shape* **:rectangle***) (:move-cursor* **t***))* &body *body*
Binds the local environment in such a way that the output of *body* will be surrounded by a border of the specified *:shape*.

clim:define-border-type *shape arglist* &body *body*
Defines a new kind of border named *shape*. *arglist* will typically be `(stream record left top right bottom)`.

For example, the following produces a piece of output surrounded by a rectangle.

```
(defun bordered-triangle (stream)
  (clim:surrounding-output-with-border (stream :shape :rectangle)
    (clim:draw-polygon* stream '(40 120 50 140 30 140))))
```

The following is the result of evaluating `(bordered-triangle *my-window*)`:

Incremental Redisplay in CLIM

Many applications can benefit greatly by 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. Genera's PEEK application is an example; this window displays the status of processes and other changing system information.

CLIM's output recording mechanism provides the foundation for incremental redisplay. As an application programmer, you need to understand the concepts of output recording before learning how to use the techniques of incremental redisplay.

Concepts of Incremental Redisplay in CLIM

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.

It is ''incremental'' in the sense that CLIM redisplays only the part of the output history visible in the viewport, and only the portions of the output history that have changed, and thus need to be redisplayed.

The way to do redisplay is to call **clim:redisplay** on an output record that was created by **clim:updating-output**. This tells CLIM to start computing that output record over from scratch. CLIM compares the results with the existing output and tries to do minimal redisplay. The **clim:updating-output** form allows you to assist CLIM by informing it that entire branches of the output history are known *not* to have changed. **clim:updating-output** also allows you to communicate the fact that a piece of the output record hierarchy has moved.

clim:redisplay is often quite easy to use, and is 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. However, **clim:redisplay** can be inefficient when you compare it to the best redisplay algorithm that you can implement for any particular special case. For example, a graphical editor whose operations affect only a single object (or a small number of objects) in a drawing might be a poor candidate for **clim:updating-output**. Because such an editor knows exactly what must be redisplayed after an operation, it is probably more efficient to do redisplay ''by hand''. It is often appropriate to use incremental redisplay in order to get your application running, and then implement your own display later if incremental redisplay proves to be too slow.

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.

clim:updating-output *(stream* &rest *args* &key *(:record-type* **''clim:standard-updating-output-record***) :unique-id (:id-test* **'#'eql***) :cache-value (:cache-test* **'#'eql***) :copy-cache-value :parent-cache :output-record :fixed-position :all-new* &allow-other-keys*)* &body *body*

> Informs the incremental redisplay module of the characteristics of the output done by *body* to *stream*. Within **clim:updating-output**, you name a piece of output (with a unique id), and you state how to determine whether the output changes (with a cache value).

clim:redisplay *record stream* &key *(:check-overlapping* **t***)*

> Causes the output of record to be recomputed. CLIM redisplays the changes incrementally, that is, only redisplays those parts of the record that changed.

clim:redisplay-output-record *record stream* &optional *check-overlapping x y parent-x parent-y*
Causes the output of *record* to be recomputed. CLIM redisplays the changes incrementally, that is, only redisplays those parts of the record that changed.

Using clim:updating-output

The primary technique of incremental redisplay is to use **clim:updating-output** to inform CLIM what output has changed, and use **clim:redisplay** to recompute and redisplay that output.

The outermost call to **clim:updating-output** identifies a program fragment that produces incrementally redisplayable output. A nested call to **clim:updating-output** (that is, a call to **clim:updating-output** that occurs during the evaluation of the body of the outermost **clim:updating-output** and 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 **clim:updating-output** evaluates 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 **clim:updating-output** caches its **:unique-id** and **:cache-value** arguments and the portion of the output produced by its body.

clim:redisplay takes an updating output record and evaluates the captured body of **clim:updating-output** over again. When a nested call to **clim:updating-output** is evaluated during redisplay, **clim:updating-output** decides whether the cached output records can be reused or the output needs to be redrawn. This is controlled by the **:cache-value** argument to **clim:updating-output**. If its value matches its previous value, the body would produce output identical to the previous output and thus is unnecessary. In this case the cached output records are reused and **clim:updating-output** does not re-execute its body. If the **:cache-value** does not match, the output needs to be redrawn, so **clim:updating-output** evaluates 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 **clim:updating-output**.

If the **:incremental-redisplay** pane option is used, CLIM supplies the outermost call to **clim:updating-output**, saves the updating output record, and calls **clim:redisplay**. The function specified by the **:display-function** pane option performs only the nested calls to **clim:updating-output**.

If you use incremental redisplay without using the **:incremental-redisplay** pane option, you must perform the outermost call to **clim:updating-output**, save the updating output record, and call **clim:redisplay** yourself.

In order to compare the cache to the output record, two pieces of information are necessary:

- An association between the output being done by the program and a particular cache. This is supplied in the **:unique-id** option to **clim:updating-output**.
- A means of determining whether this particular cache is valid. This is the **:cache-value** option to **clim: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. (In other words, it is 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 **clim: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 also maintain one in their superiors and propagate the modification up when things change. While the simpler approach works, it requires CLIM to do more work than is necessary.

Example of Incremental Redisplay in CLIM

The following function illustrates the standard use of incremental redisplay:

```
(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)
                 (terpri stream))))))
    (sleep 10)
    (setf (nth 2 list) 17)
    (clim:redisplay record stream)))
```

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 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:

```
(setf list (sort list #'(lambda (&rest args)
                          (zerop (random 2)))))
```

Here is a little ''process status'' program that shows how to use incremental redisplay with table formatting. Notice the use of **clim:updating-output** around each row of the table to ''name'' each process, and the use of **clim:updating-output**'s **:cache-value** option in each cell in the rows to indicate when a field has changed.

```
(defun show-processes (&optional (stream *standard-output*))
  (clim:with-end-of-line-action (stream :allow)
    (clim:with-end-of-page-action (stream :allow)
      (let ((record
              (macrolet ((cell (stream item)
                           `(clim:formatting-cell (,stream)
                              (format ,stream "~A" ,item))))
                (clim:updating-output (stream)
                  (clim:formatting-table (stream)
                    (clim:updating-output (stream :unique-id 'headings
                                                  :cache-value 'constant)
                      (clim:with-text-face (stream :italic)
                        (clim:formatting-row (stream)
                          (cell stream "Process")
                          (cell stream "State")
                          (cell stream "Activity"))))
                    (let ((list (copy-list (clim-sys:all-processes))))
                      (dolist (p list)
                        (let ((name (clim-sys:process-name p))
                              (state (clim-sys:process-whostate p))
                              (activity (clim-sys:process-state p)))
                          (clim:updating-output (stream :unique-id p)
                            (clim:formatting-row (stream)
                              (clim:updating-output (stream :cache-value name)
                                (cell stream name))
                              (clim:updating-output (stream :cache-value state)
                                (cell stream state))
                              (clim:updating-output (stream :cache-value activity)
                                (cell stream (string-capitalize activity))))))))))))
        (loop
          (sleep 1)
          (clim:redisplay record stream))))))
```

The following is an example of how to use incremental redisplay with graph formatting. Notice that you do not need to worry about redisplay of the graph's edges; CLIM does this for you.

```
(defun redisplay-graph (stream)
  (macrolet ((make-node (&key name children)
               `(list* ,name ,children))
             (node-name (node)
               `(car ,node))
             (node-children (node)
```

```
            `(cdr ,node)))
  (let* ((3a (make-node :name "3A"))
         (3b (make-node :name "3B"))
         (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)))
         (root (make-node :name "0" :children (list 1a 1b)))
         (graph
           (clim:updating-output (stream :unique-id root)
             (clim:format-graph-from-root
               root
               #'(lambda (node s)
                   (clim:updating-output (s :cache-value node)
                     (write-string (node-name node) s)))
               #'cdr   ;really #'node-children
               :stream stream))))
    (sleep 2)
    (setf (node-children 2a) (list 3a 3b))
    (clim:redisplay graph stream)
    (sleep 2)
    (setf (node-children 2a) nil)
    (clim:redisplay graph stream))))
```

Streams and Windows in CLIM

CLIM performs many of its input and output operations on objects called *streams*. A stream is a special kind of sheet that supports the stream protocols. These protocols are partitioned into two layers: the basic stream protocol and the extended stream protocol.

The basic stream protocol is character-based and compatible with existing Common Lisp programs. (Note that the basic stream protocol is not documented in this user guide).

The standard Common Lisp stream functions work on CLIM streams in all CLIM platforms.

You can use the extended stream protocol to include pointer events and synchronous window-manager communication.

Extended Stream Input in CLIM

CLIM defines an extended input stream protocol. This protocol extends the basic Common Lisp input stream model to allow manipulation of non-character user gestures, such as pointer button presses. It also provides the basis for CLIM's input editor.

Operators for Extended Stream Input

clim:extended-input-stream-p *object*
Returns **t** if the *object* is a CLIM extended input stream, otherwise it returns **nil**.

clim:read-gesture &key *(:stream* ***standard-input****) :timeout :peek-p :input-wait-test :input-wait-handler :pointer-button-press-handler*
Returns the next gesture available in the input stream. Note that **clim:read-gesture** does not echo character input.

clim:stream-input-wait *stream* &key *:timeout :input-wait-test*
Waits until *:timeout* or *:input-wait-test*, if specified. Otherwise the function waits until there is input in the *stream*.

clim:unread-gesture *gesture* &key *(:stream* ***standard-input****)*
Places the specified *gesture* back into *:stream*'s input buffer. The next **clim:read-gesture** request will return the unread gesture. The gesture supplied must be the most recent gesture read from the stream.

Manipulating the Pointer in CLIM

Concepts of Manipulating the Pointer in CLIM

A pointer is an input device that enables pointing at an area of the screen (for example, a mouse, or a a tablet). CLIM offers a set of operators that enable you to manipulate the pointer.

Operators for Manipulating the Pointer in CLIM

These functions are the higher-level functions for doing input via the pointer.

clim:tracking-pointer *(*&optional *stream* &key *:pointer :multiple-window :transformp (:context-type* **t***) :highlight)* &body *clauses*
Provides a general means for running code while following the position of a pointing device, and monitoring for other input events. Programmer-supplied code may be run upon occurrence of events such as motion of the pointer, clicking of a pointer button, or typing something on the keyboard.

clim:drag-output-record *stream output-record* &key *(:repaint* **t***) :multiple-window :erase :feedback (:finish-on-release* **t***)*
Enters an interaction mode in which user moves the pointer, and *output-record* follows the pointer by being dragged on *stream*.

clim:dragging-output *(*&optional *stream* &key *(:repaint* **t***) :multiple-window :finish-on-release)* &body *body*
Evaluates *body* to produce the output, and then invokes **clim:drag-output-record** to drag that output on *stream*.

clim:pointer-place-rubber-band-line* &key *:start-x :start-y (:stream* ***standard-input****) :pointer :multiple-window (:finish-on-release* **t***)*
Prompts for a line via the pointing device specified by *:pointer*. **clim:pointer-place-rubber-band-line*** returns four values, the start-x, start-y, end-x, and end-y of a line.

clim:pointer-input-rectangle* &key *:left :top :right :bottom (:stream* ***standard-input****) :pointer :multiple-window (:finish-on-release* **t***)*
Prompts for a rectangular area via the pointing device specified by *:pointer*. **clim:pointer-input-rectangle*** returns four values, the left, top, right, and bottom edges of a rectangle.

The following are lower level functions for managing the pointer more directly.

clim:stream-pointer-position *stream* &key *:pointer*
This function returns the position (two coordinate values) of the pointer in the stream's drawing plane coordinate system. You can use **clim:stream-set-pointer-position** to set the pointer position.

clim:stream-set-pointer-position *stream x y* &key *:pointer*
This function sets the position (two coordinate values) of the *:pointer* in the *stream*'s drawing plane coordinate system.

clim:port-pointer *port*
Returns the pointer object corresponding to the primary pointing device for the port *port*.

clim:port-modifier-state *basic-port*
Returns the state of the modifier keys for the port *port*.

clim:pointer-button-state *pointer*
Returns the current button state for *pointer*.

clim:pointer-position *pointer*
This function returns the position (as two coordinate values) of the pointer *pointer* in the coordinate system of the sheet that the pointer is currently over.

clim:pointer-set-position *pointer x y*
This function changes the position of the pointer *pointer* to be *(x,y)*.

clim:pointer-native-position *pointer*
This function returns the position (as two coordinate values) of the pointer *pointer* in the coordinate system of the port's graft (that is, its ''root window'').

clim:pointer-set-native-position *pointer x y*
This function changes the position of the pointer *pointer* to be *(x,y)*.

clim:pointer-sheet *pointer*
Returns the sheet over which the pointer *pointer* is currently positioned.

clim:pointer-cursor *pointer*
Returns the current cursor type for *pointer*. You can use **setf** to change it.

The Structure of the CLIM Input Editor

CLIM's input editor provides interactive parsing and prompting by interacting with the rest of CLIM's input facility via *rescanning*. Rescanning is the process of resetting the internal state of the input editor and rereading the user's already-buffered input. CLIM input editing streams encapsulate interactive streams, that is, most stream operations are handled by the encapsulated interactive stream, but some operations are handled directly by the input editing stream itself.

An input editing stream has the following components:

- The encapsulated interactive stream.
- A buffer with a fill pointer, which will be referred to as *FP*. The buffer contains all of the user's input, and *FP* is the length of that input.
- An insertion pointer, which will be referred to as *IP*. The insertion pointer is the point in the buffer at which the ''editing cursor'' is.
- A scan pointer, which will be referred to as *SP*. The scan pointer is the point in the buffer from which CLIM will get the next input gesture object (via **clim:read-gesture**).
- A ''rescan queued'' flag indicating that the programmer (or the input editor itself) requested that a rescan operation should take place before the next gesture is read from the user.
- A ''rescan in progress'' flag that indicates that CLIM is rescanning the user's input, rather than reading freshly supplied gestures from the user.

The overall description of how the input editor works, is that it reads either ''real'' gestures from the user (such as characters from the keyboard or pointer button events), or input editing commands. The input editing commands can modify the state of the input buffer. When such modifications take place, it is necessary to rescan the input buffer, that is, the input editor resets the scan pointer *SP* to its original state and reparses the contents of the input editor buffer before reading any other gestures from the user. While this rescanning operation is taking place, the ''rescan in progress'' flag is set to **t**. The input editor always ensures that *SP* is no greater than *FP*.

The remainder of this section describes the input editor in more detail. If you plan to write complex **clim:accept** methods, you may need to understand the input editor at this level of detail. Otherwise, you may skip the rest of this section.

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 **clim:accept** and **clim:read-token**. When a rescan operation is invoked, it has the effect of throwing to the **rescan** tag in the example above. The loop is terminated when an

activation gesture is seen, and at that point the values produced by *continuation* are returned as values from the input editor.

The most important point is that functions such as **clim:accept**, **clim:read-gesture**, and **clim:unread-gesture** read (or restore) the next gesture object from the buffer at the position pointed to by the scan pointer *SP*. However, insertion and input editing commands take place at the position pointed to by *IP*. The purpose of the rescanning operation is to eventually ensure that all of the user's input (typed characters, pointer button presses, and so forth) have been read by CLIM. During input editing, CLIM displays an editing cursor to remind you of the position of *IP*.

The overall structure of **clim:read-gesture** on an input editing stream is:

```
(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
    ))
```

When a new gesture object is inserted into the input editor buffer, it is inserted at the insertion pointer *IP*. If *IP* is equal to *FP*, this is accomplished by doing a **vector-push-extend**-like operation on the input buffer and *FP*, and then incrementing *IP*. If *IP* is less than *FP*, CLIM first makes room for the new gesture in the input buffer, then inserts the gesture at *IP*, and finally increments both *IP* and *FP*.

When the user requests an input editor motion command, only the insertion pointer *IP* is affected. Motion commands do not need to request a rescan operation.

When the user requests an input editor deletion command, the user input at *IP* is removed, and *IP* and *FP* are modified to reflect the new state of the input buffer. Deletion commands (and other commands that modify the input buffer) arrange for a rescan to occur when they are done modifying the buffer, either by calling **clim:queue-rescan** or **clim:immediate-rescan**.

CLIM sometimes inserts special objects in the input editor buffer, such as ''noise strings'' and ''accept results''. A noise string is used to represent some sort of inline prompt and is never seen as user input; **clim:input-editor-format** and **clim:prompt-for-accept** methods insert noise strings 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); **clim: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.

The following forms are the most useful high-level operators for doing input editing in CLIM:

clim:with-input-editing *(*&optional *stream* &key *:input-sensitizer :initial-contents :class)* &body *body*
Establishes a context in which the user can edit the input he or she types in on the stream *stream*. *body* is then evaluated in this context, and the values returned by *body* are returned as the values of **clim:with-input-editing**.

clim:with-input-editor-typeout *(*&optional *stream* &key *:erase)* &body *body*
If, when you are inside of a call to **clim:with-input-editing**, you want to perform some sort of typeout, it should be done inside **clim:with-input-editor-typeout**.

clim:input-editor-format *input-editing-stream format-string* &rest *format-args*
This function is like **format**, except that it is intended to be called on input editing streams. It arranges to insert ''noise strings'' in the input editor's input buffer.

The following functions are lower-level functions used in writing more sophisticated **clim:accept** methods:

clim:replace-input *stream new-input* &key *:start :end :rescan :buffer-start*
Replaces *stream*'s input buffer with the string *new-input*.

clim:presentation-replace-input *stream object type view* &key *:rescan :buffer-start*
Like **clim:replace-input**, except that the new input to insert into the input buffer is gotten by presenting the object *object* with the presentation type *type* and view *view*.

clim:stream-insertion-pointer *input-editing-stream*
Returns an integer corresponding to the current input position of *input-editing-stream*, that is, the point in the buffer at which the next user input gesture will be inserted.

clim:stream-scan-pointer *input-editing-stream*
Returns the current scan pointer (an integer) for *input-editing-stream*, that is, the point in the buffer at which calls to **clim:accept** have stopped parsing input.

clim:stream-rescanning-p *input-editing-stream*
Returns the state of the input editing stream's ''rescan in progress'' flag, which is **t** if *input-editing-stream* is performing a rescan operation, otherwise it is **nil**. Non-input editing streams always return **nil**.

clim:immediate-rescan *input-editing-stream*
Invokes a rescan operation immediately on *input-editing-stream* by ''throwing'' out to the beginning of the most recent invocation of **clim:with-input-editing**.

clim:queue-rescan *input-editing-stream* &optional *rescan-type*
Indicates that a rescan operation on *input-editing-stream* should take place after the next non-input editing gesture is read.

clim:rescan-if-necessary *input-editing-stream* &optional *inhibit-activation*
Invokes a rescan operation on the input editing stream *input-editing-stream* if **clim: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**.

clim:reset-scan-pointer *input-editing-stream* &optional *sp*
Sets *input-editing-stream*'s scan pointer to *sp* (which defaults to 0) and sets the state of **clim:stream-rescanning-p** to **t**.

clim:erase-input-buffer *input-editing-stream* &optional *start-position*
Erases the part of the display that corresponds to the input editor's buffer starting at the position *start-position*.

clim:redraw-input-buffer *input-editing-stream* &optional *start-position*
Displays the input editor's buffer starting at the position *start-position* on the interactive stream that is encapsulated by the input editing stream *input-editing-stream*.

Extended Stream Output in CLIM

In addition to the basic output stream protocol, CLIM defines an extended output stream protocol. This protocol extends the stream model to allow the manipulation of a text cursor.

Manipulating the Cursor in CLIM

This protocol extends the stream model to allow manipulation of the cursor.

A CLIM stream has a text cursor position, which is the place on the drawing plane where the next piece of text output will be drawn. Common Lisp stream output operations place text at the cursor position and advance the cursor position past the text. Certain CLIM output operations, such as **clim:present** and **clim:formatting-table**, do the same. The CLIM **draw** functions, on the other hand, pay no attention to the text cursor position.

Common Lisp stream input operations that echo, such as **read-line**, as well as **clim:accept**, echo the input at the cursor position, and advance the cursor position.

Operators for Manipulating the Cursor

clim:stream-cursor-position *stream*
Returns two values, the X and Y coordinates of the cursor position on the drawing plane.

clim:stream-set-cursor-position *stream x y*
Moves the cursor position to the specified X and Y coordinates on the drawing plane.

clim:stream-increment-cursor-position *stream dx dy*
Moves the cursor position on stream relatively, adding *dx* to the X coordinate and adding *dy* to the Y coordinate. Either argument *dx* or *dy* can be **nil**, which means not to change that coordinate.

clim:cursor
The protocol class that corresponds to a text cursor.

clim:cursor-position *cursor*
Returns the cursor position of *cursor* as two values (X and Y), relative to the upper left corner of the sheet with which the cursor is associated.

clim:cursor-set-position *cursor x y*
Sets the cursor position of *cursor* to *x* and *y*, which are relative to the upper left corner of the sheet with which the cursor is associated.

clim:cursor-visibility *cursor*
A convenience function that combines the functionality of both **clim:cursor-active** and **clim:cursor-state**.

clim:cursor-sheet *cursor*
Returns the sheet with which *cursor* is associated.

Text Measurement Operations in CLIM

These functions compute the change in the cursor position that would occur if some text were output (that is, without actually doing any output, and without changing the cursor position).

Operators for Text Measurements

clim:text-size *medium string* &key *:text-style :start :end*
Computes how the cursor position would move if the specified string or character were output starting at cursor position (0,0). It does not take into account the value of **clim:stream-text-margin** when computing the size of the output.

clim:stream-character-width *stream character* &optional *text-style*
Returns the horizontal motion of the cursor position that would occur if this *character* were output in this *text-style*. It does not take into account the value of **clim:stream-text-margin** when computing the size of the output.

clim:stream-string-width *stream string* &key *:start :end :text-style*
Computes how the cursor position would move horizontally if the specified *string* were output starting at the left margin. It does not take into account the value of **clim:stream-text-margin** when computing the size of the output.

clim:stream-line-height *stream* &optional *text-style*
Returns what the line height of a line containing text in that *text-style* would be. *text-style* defaults to **(clim:medium-text-style** *stream***)**.

clim:stream-vertical-spacing *stream*
Returns the current inter-line spacing for the stream *stream*.

clim:stream-baseline *stream*
Returns the current text baseline for the stream *stream*.

clim:stream-text-margin *stream*
The X coordinate at which text wraps around (see **clim: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 all the way to the left.

Attracting the User's Attention in CLIM

CLIM supports the following operators for attracting the user's attention:

clim:beep &optional *(stream* ***standard-output****)*
Attracts the user's attention, usually with an audible sound.

clim:notify-user *frame message* &key *:associated-window :title :exit-boxes :text-style :foreground :background :x-position :y-position*
Notifies the user of some event on behalf of the application frame *frame*. *message* is a message string.

Window Stream Operations in CLIM

It is sometimes useful to perform some window management operations directly on a window.

Clearing and Refreshing the Drawing Plane in CLIM

CLIM supports the following operators for clearing and refreshing the drawing plane:

clim:window-clear *window*
Clears the entire drawing plane of *window*, filling it with the background design, and discards the windows output history.

clim:window-erase-viewport *window*
Clears the visible part of the drawing plane of *window*, filling it with the background design.

clim:window-refresh *window*
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.

The Viewport and Scrolling in CLIM

A window 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 stacked 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 (**clim:with-end-of-line-action** and **clim:with-end-of-page-action**, respectively).

Viewport and Scrolling Operators

clim:window-viewport-position *window*
: Returns two values, the X and Y coordinates of the top-left corner of the *window*'s viewport.

clim:window-set-viewport-position *window x y*
: Moves the top-left corner of the window's viewport. This is how you scroll a window.

clim:note-viewport-position-changed *frame pane x y*
: CLIM calls this function whenever a pane gets scrolled, whether it is scrolled programmatically (by **clim:window-set-viewport-position**, for example) or by a user gesture (such as clicking on a scroll bar).

clim:scroll-extent *sheet x y*
: Scrolls the pane *sheet* in its viewport so that the position *(x,y)* of the pane is at the upper-left corner of the viewport.

clim:window-viewport *window*
: If the pane *window* is part of a scroller pane, this returns the region of *window*'s viewport. Otherwise it returns the region of *window* itself.

clim:pane-viewport *pane*
: If the pane *pane* is part of a scroller pane, this returns the viewport pane for *pane*. Otherwise it returns **nil**.

clim:pane-viewport-region *pane*
: If the pane *pane* is part of a scroller pane, this returns the region of *pane*'s viewport. Otherwise it returns **nil**.

clim:stream-end-of-line-action *stream*
: Controls what happens when the cursor position moves horizontally out of the viewport (beyond the text margin). You can use **setf** on this to change the end of line action.

clim:stream-end-of-page-action *stream*
: Controls what happens when the cursor position moves vertically out of the viewport. You can use **setf** on this to change the end of page action.

clim:with-end-of-line-action *(stream action)* &body *body*
: Temporarily changes the end of line action for the duration of evaluation of *body*.

clim:with-end-of-page-action *(stream action)* &body *body*
Temporarily changes the end of page action for the duration of evaluation of *body*.

Functions for Operating on Windows Directly

You can use **clim:open-window-stream** to give you a CLIM window without incorporating it into a frame. First use **clim:find-port** to make a port to pass as the **:port** argument. After calling **clim:open-window-stream**, call **clim:window-expose** to make the resulting window stream visible.

Operators for Creating Ports and Frame Managers

clim:find-port &rest *initargs* &key *(:server-path* **clim:*default-server-path***)* &allow-other-keys
Creates a port, a special object that acts as the ''root'' or ''parent'' of all CLIM windows and application frames. In general, a port corresponds to a connection to a display server. For making color windows under Genera, use the **:screen** keyword in the server path argument to **clim:find-port** and give it the argument **color:color-screen**.

clim:port *object*
Given a CLIM object *object*, **clim:port** returns the port associated with *object*.

clim:destroy-port *port*
Destroys the connection to the display server represented by *port*.

clim:map-over-ports *function*
Applies *function* to all of the current ports.

clim:restart-port *port*
Restarts the event process that manages the port *port*.

clim:find-frame-manager &rest *options* &key *:port (:server-path* **clim:*default-server-path***)* &allow-other-keys
Finds a frame manager that is on the port *:port*, or creates a new one if none exists.

clim:frame-manager *object*
Given a CLIM object *object*, **clim:frame-manager** returns the frame manager associated with *object*.

Operators for CLIM Primitive Layer for Window Streams

clim:open-window-stream &key *:port :frame-manager :left :top :right :bottom :width :height :foreground :background :text-style (:vertical-spacing* **2***) (:end-of-line-action* **:allow***) (:end-of-page-action* **:allow***) :output-record (:draw* **t***) (:record* **t***) (:initial-cursor-visibility* **:off***) :text-margin :default-text-margin :save-under :input-buffer (:scroll-bars* **:vertical***) :borders :label*

A convenient interface for creating a CLIM window outside of an application frame.

clim:window-parent *window*
Returns the window that is the parent (superior) of *window*.

clim:window-children *window*
Returns a list of all of the windows that are children (inferiors) of *window*.

clim:window-label *window*
Returns the label (a string) associated with *window*, or **nil** if there is none.

clim:with-input-focus *(stream)* &body *body*
Temporarily gives the keyboard input focus to the given window (which is most often an interactor pane). By default, a frame will give the input focus to the **clim:frame-query-io** pane.

clim:stream-set-input-focus *stream*
Gives the input focus to *stream*, and returns as a value the stream or sheet that previously had the input focus.

The following functions are most usefully applied to the top level window of a frame. For example,

```
(clim:frame-top-level-sheet clim:*application-frame*)
```

clim:window-expose *window*
Makes the *window* visible on the screen.

clim:window-stack-on-bottom *window*
Puts the *window* underneath all other windows that it overlaps.

clim:window-stack-on-top *window*
Puts the *window* on top of all other windows that it overlaps, so you can see all of it.

clim:window-visibility *stream*
A predicate that returns true if the window is visible.

The following operators can be applied to a window to determine its position and size.

clim:window-inside-edges *window*
Returns four values, the coordinates of the left, top, right, and bottom inside edges of the window *window*.

clim:window-inside-left *window*
Returns the coordinate of the left edge of the window *window*.

clim:window-inside-top *window*
Returns the coordinate of the top edge of the window *window*.

clim:window-inside-right *window*
Returns the coordinate of the right edge of the window *window*.

clim:window-inside-bottom *window*
Returns the coordinate of the bottom edge of the window *window*.

clim:window-inside-size *window*
Returns the inside width and height of *window* as two values.

clim:window-inside-width *window*
Returns the inside width of *window*.

clim:window-inside-height *window*
Returns the inside height of *window*.

Hardcopy Streams in CLIM

CLIM supports hardcopy output through the macro **clim:with-output-to-postscript-stream**:

clim:with-output-to-postscript-stream *(stream-var file-stream* &key *:device-type (:orientation* **:portrait***) :multi-page :scale-to-fit :header-comments (:destination* **:printer***))* &body *body*
Within *body stream-var* is bound to a stream that produces PostScript code. This stream is suitable as a stream or medium argument to any CLIM output utility. A PostScript program describing the output to the *stream-var* stream will be written to *stream*.

clim:new-page *stream*
When called on a PostScript output stream, this causes a PostScript ''newpage'' command to be included in the output at the point **clim:new-page** is called.

Example

This example writes a PostScript program which 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)))))
```

This example uses multi-page mode to draw a graph (by writing a PostScript program to the file *some-pathname*) of the subclasses of the class **clim:bounding-rectangle**.

```
(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:bounding-rectangle)
      #'(lambda (object s)
          (write-string (string (clos:class-name object)) s))
      #'clos:class-direct-superclasses
      :stream stream)))
```

You can then use standard system facilities to print the file, such as Genera's `Hardcopy File` or the Unix `lpr` command.

CLIM's Windowing Substrate

Introduction to CLIM's Windowing Substrate

One of the basic tasks in building user interfaces is allocating screen regions for particular purposes and recursively subdividing these regions into subregions. CLIM's windowing layer defines an extensible framework for constructing, using, and managing such hierarchies. This framework allows uniform treatment of the following things:

- Window objects like those in X Windows.
- Lightweight gadgets typical of toolkit layers, such as Motif or OpenLook.
- Structured graphics like output records and presentations.

From the perspective of most CLIM users, CLIM's windowing layer plays the role of a window system. However, CLIM actually uses the services of a window system platform to provide efficient windowing, input, and output facilities.

The fundamental unit of windowing defined by CLIM is called a *sheet*. A sheet can participate in a *windowing relationship* in which one sheet (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.

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 a 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 top-level 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 descendant of a graft that represents the appropriate host window. The sheet will then appear to be a child of that host window. So, a sheet is put onto a particular screen by making it a child of an appropriate graft and enabling it.

Basic Sheet Protocols

A sheet is the basic abstraction for implementing windows in CLIM. All sheets have the following basic 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 a child 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 parent, if it has a parent.

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.

clim:sheet
: The protocol class that corresponds to a sheet.

clim:sheetp *object*
: Returns **t** if and only if *object* is of type **clim:sheet**, otherwise returns **nil**.

Furthermore, a sheet can participate in a number of protocols. Every sheet must provide or inherit methods for the generic functions that make up these protocols, or delegate some other sheet to handle the methods for it.

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. Depending on the kind of sheet, input 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.

Sheet Geometry Protocols

Every sheet has a region and a coordinate system. A sheet's region refers to its position and extent on the display device, and is represented by some sort of 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.

The following functions can be used to read or change the sheet's region and transformation:

clim:sheet-region *sheet*
Returns a region object that represents the set of points to which *sheet* refers.

clim:sheet-device-region *sheet*
Returns a region object that describes the region that *sheet* occupies on the display device. The coordinates are in the host's native window coordinate system.

clim:sheet-transformation *sheet*
Returns a transformation that converts coordinates in *sheet*'s coordinate system into coordinates in its parent's coordinate system.

clim:sheet-device-transformation *sheet*
Returns a transformation that converts coordinates in *sheet*'s coordinate system into native coordinates on the display device.

clim:note-sheet-region-changed *sheet*
This function is invoked whenever the region of *sheet* is changed.

clim:note-sheet-transformation-changed *sheet*
This function is invoked whenever the transformation of *sheet* is changed.

The following functions are more convenient interfaces used to change the region or location of a sheet:

clim:move-sheet *sheet x y*
Moves *sheet* to the new position (*x*,*y*). *x* and *y* are in coordinates relative to *sheet*'s parent.

clim:resize-sheet *sheet width height*
Changes the size of *sheet* to have width *width* and height *height*.

clim:move-and-resize-sheet *sheet x y width height*
Moves *sheet* to the new position (*x*,*y*), and simultaneously changes the size of the sheet to have width *width* and height *height*. *x* and *y* are in coordinates relative to *sheet*'s parent.

The following functions can be used to convert a position in the coordinate system of one sheet to the coordinate system of a parent or child sheet:

clim:map-sheet-position-to-parent *sheet x y*
Applies *sheet*'s transformation to the point *(x,y)*, returning the coordinates of that point in *sheet*'s parent's coordinate system.

clim:map-sheet-position-to-child *sheet x y*
Applies the inverse of *sheet*'s transformation to the point *(x,y)* (represented in *sheet*'s parent's coordinate system), returning the coordinates of that same point in *sheet*'s coordinate system.

The following functions can be used to map over the sheets that contain a given position or region:

clim:map-over-sheets-containing-position *function sheet x y*
Applies *function* to all of the children of *sheet* containing the position *(x,y)*. *x* and *y* are expressed in *sheet's* coordinate system.

clim:map-over-sheets-overlapping-region *function sheet region*
Applies *function* to all of the children of *sheet* overlapping the region *region*. *region* is expressed in *sheet's* coordinate system.

Sheet Relationship Protocols

Sheets are arranged in a tree-shaped hierarchy. In general, a sheet has one parent (or no parent) and zero or more children. A sheet may have zero or more siblings (that is, other sheets that share the same parent).

The following terms are used to describe the relationships between sheets:

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 possibly 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.

The following generic functions comprise the sheet protocol.

clim:sheet-parent *sheet*
Returns the sheet that is the parent of *sheet*, or **nil** if *sheet* has no parent.

clim:sheet-children *sheet*
Returns a list of all of the sheets that are children of *sheet*.

clim:sheet-adopt-child *sheet child*
Adds the child sheet *child* to the set of children of *sheet*, and makes the *sheet* the child's parent. If *child* already has a parent, CLIM will signal an error.

clim:sheet-disown-child *sheet child &key (:errorp* **t***)*
Removes the child sheet *child* from the set of children of *sheet*, and makes the parent of the child be **nil**.

clim:sheet-enabled-p *sheet*
Returns **t** if *sheet* is enabled by its parent, otherwise returns **nil**.

clim:map-over-sheets *function sheet*
Applies *function* to *sheet*, and then applies *function* to all of the descendants of *sheet*.

Sheet Input Protocols

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, and pointer button and keyboard events. At this level, input is represented as event objects.

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 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 **clim:port-keyboard-input-focus** contains the event client that receives the port's keyboard events.

Event objects and their accessors include:

clim:device-event
The superclass of all other CLIM device events.

clim:event-sheet *event*
Returns the window on which *event* occurred.

clim:event-modifier-state *event*
Returns the state of the keyboard's shift keys when the event *event* occurred.

clim:pointer-motion-event
The class that corresponds to the user moving the pointer.

clim:pointer-enter-event
The class that corresponds to the user moving the pointer into a sheet from another sheet.

clim:pointer-exit-event
The class that corresponds to the user moving the pointer out of a sheet.

clim:pointer-button-press-event
The class that corresponds to the user pressing a button on the pointer.

clim:pointer-button-release-event
The class that corresponds to the user releasing a button on the pointer.

clim:pointer-event-x *pointer-event*
Returns the X position of the pointer when the *pointer-event* occurred.

clim:pointer-event-y *pointer-event*
Returns the Y position of the pointer when the *pointer-event* occurred.

clim:pointer-event-button *pointer-button-event*
Returns the button number that was pressed when the pointer button event *pointer-button-event* occurred. The values this can take are **clim:+pointer-left-button+**, **clim:+pointer-middle-button+**, or **clim:+pointer-right-button+**.

clim:key-press-event
The class that corresponds to pressing a key on the keyboard.

clim:key-release-event
The class that corresponds to releasing a key on the keyboard.

clim:keyboard-event-key-name *keyboard-event*
Returns the name of the key that was pressed or released in order to generate the keyboard event.

The following are the most useful functions in the sheet input protocol. These are what you need to be aware of if you are writing your own classes of gadgets.

clim:handle-event *sheet event*
Handles the event *event* on behalf of *sheet*.

clim:queue-event *sheet event*
Inserts the event *event* into the queue of events for *sheet*.

clim:sheet-event-queue *sheet*
Returns the object that acts as the event queue.

For more information on gadgets, see the section "Using Gadgets in CLIM".

Sheet Output Protocols

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.

Each sheet maintains some output state that 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 the medium associated with the sheet itself, or it could be derived from a parent, or may have some global default, depending on the sheet itself.

The following comprises the basic medium protocol. For more detail on this, see the section "The CLIM Drawing Environment".

clim:medium
The protocol class that corresponds to a medium.

clim:mediump *object*
Returns **t** if and only if *object* is of type **clim:medium**, otherwise returns **nil**.

clim:medium-foreground *medium*
Returns the current foreground design of the *medium*. You can use **setf** on **clim:medium-foreground** to change the foreground design.

clim:medium-background *medium*
Returns the current background design of the *medium*. You can use **setf** on **clim:medium-background** to change the background design.

clim:medium-ink *medium*
Returns the current drawing ink of the *medium*. You can use **setf** on **clim:medium-ink** to change the current ink.

clim:medium-transformation *medium*
Returns the current transformation of the *medium*. You can use **setf** on **clim:medium-transformation** to change the current transformation.

clim:medium-clipping-region *medium*
Returns the current clipping region of the *medium*. You can use **setf** on **clim:medium-clipping-region** to change the clipping region.

clim:medium-line-style *medium*
Returns the current line style of the *medium*. You can use **setf** on **clim:medium-line-style** to change the line style.

clim:medium-text-style *medium*
Returns the current text style of the *medium*. You can use **setf** on **clim:medium-text-style** to change the current text style.

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 stream panes have a medium that is permanently allocated to the window.

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 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 only needs to do output only when the window's shape is changed.

To associate a sheet with a medium, use the macro **clim:with-sheet-medium**. Only sheets that support output may have a medium associated with them.

clim:sheet-medium *sheet*
Returns the medium associated with *sheet*.

clim:with-sheet-medium *(medium sheet)* &body *body*
Within the body, the variable *medium* is bound to *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 as the when the body has been exited.

clim:medium-sheet *medium*
Returns the sheet with which the medium *medium* is associated.

clim:medium-drawable *medium*
Returns the host window system object (or ''drawable'') that is drawn on by the CLIM drawing functions when they are called on *medium*.

Sheet Repainting Protocols

CLIM's repainting 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 cause a repaint to be executed for the newly-exposed region.

The following are the most useful functions in the repainting protocol. These are what you need to be aware of if you are writing your own classes of gadgets.

clim:handle-repaint *sheet region*
Implements repainting for a given sheet class.

clim:queue-repaint *sheet repaint-event*
Inserts the repaint event *repaint-event* into *sheet*'s event queue.

clim:repaint-sheet *sheet region*
Causes *sheet* and all of its descendants that overlap the region *region* to be repainted.

Ports and Mirrored Sheets

A sheet hierarchy must be attached to a display server so as to permit input and output. This is managed by the use of *ports* and *grafts*.

A *port* is a 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 its associated display.

A *graft* is a special sheet that is directly connected to a display server. A graft is the CLIM sheet that represents the root window of the display. CLIM manages grafts invisibly, so you do not need to worry about grafts except to be aware of their existence.

To display a sheet on a display, it must have a graft for 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.

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 machine running Genera will have a Genera window system object stored in one of its slots. Allowing mirrored sheets at any point in the hierarchy enables the adaptive toolkit facilities; for example, in Motif, scroll bars, sliders, push buttons, and so on, are all mirrored.

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.

The following readers are useful when dealing with mirrored sheets:

clim:sheet-mirror *sheet*
: Returns the host window that is used to display *sheet*.

clim:sheet-device-region *sheet*
: Returns a region object that describes the region that *sheet* occupies on the display device. The coordinates are in the host's native window coordinate system.

clim:sheet-device-transformation *sheet*
: Returns a transformation that converts coordinates in *sheet*'s coordinate system into native coordinates on the display device.

A port is described with a server path, which 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 port type-specific.

The following functions are useful in creating and dealing with ports.

clim:find-port &rest *initargs* &key *(:server-path* **clim:*default-server-path***)* &allow-other-keys
Creates a port, a special object that acts as the ''root'' or ''parent'' of all CLIM windows and application frames. In general, a port corresponds to a connection to a display server. For making color windows under Genera, use the **:screen** keyword in the server path argument to **clim:find-port** and give it the argument **color:color-screen**.

clim:port *object*
Given a CLIM object *object*, **clim:port** returns the port associated with *object*.

clim:map-over-ports *function*
Applies *function* to all of the current ports.

clim:port-name *port*
Returns the name of the port as a string.

clim:port-type *port*
Returns the type of the port, that is, the first element of the server path specification.

clim:port-server-path *port*
Returns the server path associated with the port.

clim:restart-port *port*
Restarts the event process that manages the port *port*.

clim:destroy-port *port*
Destroys the connection to the display server represented by *port*.

The clim-sys Package

CLIM provides a number of useful ''system-like'' facilities, including multiprocessing, locks, and resources. The operators for these facilities are all in the **clim-sys** package.

Resources in CLIM

CLIM provides a facility called *resources* that provides for reusing 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.

clim-sys:defresource *name parameters* &key *:constructor :initializer :deinitializer :matcher :initial-copies*
Defines a resource named *name*. *parameters* is a lambda-list giving names and default values (for optional and keyword parameters) of parameters to an object of this type.

clim-sys:using-resource *(variable resource* &rest *parameters)* &body *body*
The forms in *body* are evaluated with *variable* bound to an object allocated from the resource named *name*, using the parameters given by *parameters*.

clim-sys:allocate-resource *name* &rest *parameters*
Allocates an object from the resource named *name*, using the parameters given by *parameters*.

clim-sys:deallocate-resource *name object* &optional *allocation-key*
Returns the object *object* to the resource named *name*.

clim-sys:clear-resource *resource*
Clears the resource named *name*, that is, removes all of the resourced objects from the resource.

clim-sys:map-resource *function resource*
Calls *function* once on each object in the resource named *name*.

Multi-processing in CLIM

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. Using these functions provides a higher degree of portability for your CLIM applications that use multi-processing.

Important note: CLIM currently does not guard against multiple processes doing I/O on sheets, streams, mediums, and so forth. If you have an application that has multiple processes doing I/O onto the same output device, you must manage these processes and any locking issues yourself.See the section "Locks in CLIM".

clim-sys:*multiprocessing-p*
The value of this variable is **t** if the current Lisp environment supports multi-processing, otherwise it is **nil**.

clim-sys:make-process *function* &key *:name*
Creates a process named *name*. The new process will evaluate the function *function*.

clim-sys:processp *object*
Returns **t** if *object* is a process, otherwise returns **nil**.

clim-sys:destroy-process *process*
Terminates the process *process*.

clim-sys:current-process
Returns the currently running process, which will be a process object.

clim-sys:all-processes
Returns a sequence of all of the currently running processes.

clim-sys:process-name *process*
Returns the name of the process *process*.

clim-sys:process-wait *wait-reason predicate*
Causes the current process to wait until *predicate* returns a non-**nil** value. *predicate* is a function of no arguments. *reason* is a ''reason'' for waiting, usually a string.

clim-sys:process-wait-with-timeout *wait-reason timeout predicate*
Causes the current process to wait until either *predicate* returns a non-**nil** value or the number of seconds specified by *timeout* has elapsed. *predicate* is a function of no arguments. *reason* is a ''reason'' for waiting, usually a string.

clim-sys:process-yield
Allows other processes to run. On systems that do not support multiprocessing, this does nothing.

clim-sys:process-interrupt *process function*
Interrupts the process *process* and causes it to evaluate the function *function*.

clim-sys:disable-process *process*
Disables the process *process*, that is, prevents it from becoming runnable until it is enabled again.

clim-sys:enable-process *process*
Allows the process *process* to become runnable again after it has been disabled.

clim-sys:restart-process *process*
Restarts the process *process* by ''unwinding'' it to its initial state, and reinvoking its top-level function.

clim-sys:without-scheduling &body *forms*
Evaluates *body* in a context that is guaranteed to be free from interruption by other processes.

Locks in CLIM

A *lock* is a software construct used for synchronization of two processes. A lock protects some resource or data structure so that only one process at a time can use it. A lock is either held by some process, or is free. When a process tries to seize a lock, it waits until the lock is free, and then it becomes the process holding the lock. When it is finished, it unlocks the lock, allowing some other process to seize it.

CLIM provides a portable interface to the locking primitives provided by most Lisp platforms. Using this interface provides a higher degree of portability for your CLIM applications that use locks.

clim-sys:make-lock &optional *(lock-name* **"a CLIM lock"***)*
Creates a lock whose name is *name*. *name* is a string.

clim-sys:with-lock-held *(place* &optional *state)* &body *forms*
Evaluates *body* while holding the lock named by *place*.

clim-sys:make-recursive-lock &optional *(lock-name* **"a recursive CLIM lock"***)*
Creates a recursive lock whose name is *name*.

clim-sys:with-recursive-lock-held *(place* &optional *state)* &body *forms*
Evaluates *body* while holding the recursive lock named by *place*.

CLIM Dictionary

Dictionary of CLIM Operators

clim:abort-gesture *Condition*

CLIM signals an **clim:abort-gesture** condition whenever it reads an abort gesture from the user. For example, on Genera **clim:read-gesture** will signal this condition if the user presses the ABORT key.

See the macro **clim:catch-abort-gestures**.

clim:abort-gesture-event *abort-gesture* *Generic Function*

Returns the event object that caused the abort gesture condition, *abort-gesture*, to be signalled. The event will usually be a **clim:key-press-event** object.

clim:*abort-gestures* *Variable*

A list of gestures that cause CLIM to abort out of the current input.

When **clim:stream-read-gesture** reads a character, it checks to see if it is one of gestures in **clim:*abort-gestures***. If it is, CLIM signals a condition of type **clim:abort-gesture**.

clim:*abort-menus-when-buried* *Variable*

Indicates whether or not CLIM should abort out of menus when they are ''buried''.

When **clim:*abort-menus-when-buried*** is **t**, **clim:menu-choose** and **clim:menu-choose-from-drawer** return **nil** for all their values, if the menu is aborted by burying it. When **nil**, the menu will await reexposure to become active again.

clim:accelerator-gesture *Condition*

CLIM signals an **clim:accelerator-gesture** condition whenever it reads an accelerator gesture from the user. If you use **clim:read-command-using-keystrokes**, CLIM will handle this condition transparently.

clim:*accelerator-gestures* *Variable*

A list of gestures that CLIM will treat as keystroke accelerators when reading commands.

When **clim:stream-read-gesture** reads a character, it checks to see if it is one of gestures in **clim:*accelerator-gestures***. If it is, CLIM signals a condition of type **clim:accelerator-gesture**.

clim:accelerator-gesture-event *accelerator-gesture* *Generic Function*

Returns the event object that caused the accelerator gesture condition, *accelerator-gesture*, to be signalled. The event will usually be a **clim:key-press-event** object.

clim:accelerator-gesture-numeric-argument *accelerator-gesture* *Generic Function*

Returns the numeric argument associated with the accelerator gesture condition, *accelerator-gesture*. If the user did not supply a numeric argument explicitly, this will return 1.

clim:accept *type* &rest *accept-args* &key *(:stream* ***standard-input***) *(:view* **(clim:stream-default-view stream)**) *:default (:default-type* **type**) *(:history* **type**) *:provide-default (:prompt* **t**) *(:prompt-mode* **':normal**) *(:display-default* **prompt**) *:query-identifier :activation-gestures :additional-activation-gestures :delimiter-gestures :additional-delimiter-gestures :insert-default (:replace-input* **t**) *(:active-p* **t**) *Function*

Requests input of the *type* from the *stream*. **clim:accept** returns two values (or three values when inside of a call to **clim:accepting-values**), the object and its presentation type. **clim:accept** works by prompting, then establishing an input context via **clim:with-input-context**, and then calling the **clim:accept** presentation method for *type* and *:view*.

For more information on using **clim:accept**, see the section "Using CLIM Presentation Types for Input".

Note that **clim:accept** does not insert newlines. If you want to put prompts on separate lines, especially in dialogs, use **terpri** to separate the calls to **clim:accept**.

type A presentation type specifier. *type* may be an presentation type abbreviation. See the section "How to Specify a CLIM Presentation Type". See the section "Predefined Presentation Types in CLIM".

stream

Specifies the input stream, and defaults to ***standard-input***.

:view

An object representing a view. The default is **(clim:stream-default-view** *stream*). For most streams, the default view is the textual view, **clim:+textual-view+**. For dialog streams (that is, within **clim:accepting-values**), the view will typically be either **clim:+textual-dialog-view+** or **clim:+gadget-dialog-view+**.

:default Specifies the object to be used as the default value for this call to **clim:accept**. If this keyword is not supplied and *:provide-default* is **t**, then the default is determined by taking the most recent item from the presentation type history specified by the *:history* argument. If *:default* is supplied and the input provided by the user is empty, then *:default* and *:default-type* are returned as the two values from **clim:accept**.

:default-type
If *:default* is supplied and the input provided by the user is empty, then *:default* and *:default-type* are returned as the two values from **clim:accept**. This defaults to *type*.

:history Specifies which presentation type's history to use for the default value for *:default* and for the input editor's yanking commands. The default is to use the history for *type*. If *:history* is **nil**, no history is used.

:provide-default
Specifies whether or not to provide a default value for this call to **clim:accept** if **:default** is not supplied.

:prompt If *:prompt* is **t**, the prompt is a description of the type. If *:prompt* is **nil**, prompting is suppressed. If it is a string, the string is displayed as the prompt. The default is **t**, which produces "Enter a *type*:" in a top-level **clim:accept** or "(*type*)" in a nested **clim:accept**.

:prompt-mode
Can be **:normal**, the default, or **:raw**, which suppresses putting a colon after the prompt in a top-level call to **clim:accept** and suppresses putting parentheses around the prompt in a nested call to **clim:accept**. In general, you will only use *:prompt-mode* in nested calls to **clim:accept** within a **clim:accept** method.

:display-default
When **t**, displays the default as part of the prompt, if one was supplied. When **nil**, the default is not displayed. *:display-default* defaults to **t** if there was a prompt, otherwise it defaults to **nil**. In general, you will only use *:display-default* in nested calls to **clim:accept** within a **clim:accept** method.

:query-identifier
This option is used to supply a unique identifier for each call to **clim:accept** inside **clim:accepting-values**. Outside of a call to **clim:accepting-values**, *:query-identifier* has no effect. See the section "Menus and Dialogs in CLIM".

:activation-gestures
A list of gestures that overrides the current activation gestures, which terminate input. See the section "Input Editing and Built-in Keystroke Commands in CLIM".

:additional-activation-gestures
: A list of gestures that add to the activation gestures without overriding the current ones. See the section "Input Editing and Built-in Keystroke Commands in CLIM".

:delimiter-gestures
: A list of gestures that overrides the current delimiter gestures, which terminate an individual token but not the entire input sentence. You will generally only use this when you are writing a **clim:accept** method that will read multiple fields. See the section "Input Editing and Built-in Keystroke Commands in CLIM".

:additional-delimiter-gestures
: A list of gestures that add to the delimiter gestures without overriding the current ones. See the section "Input Editing and Built-in Keystroke Commands in CLIM".

:insert-default
: When **t**, inserts the default determined by the values of *:default* and *:provide-default* into the input buffer. The default is **nil**.

:replace-input
: When **t** (the default) and the call to **clim:accept** was satisfied by clicking on something with the pointer, CLIM uses **clim:presentation-replace-input** to insert the input into the input buffer.

:active-p
: When **t** (the default), the call to **clim:accept** will be considered ''active'' for input. When **nil**, **clim:accept** will simply return the values of *:default* and *:default-type* without actually asking the user for any input. You can use this option to have fields in an **clim:accepting-values** dialog that are visible, but not active for input.

clim:accept *type-key parameters options type stream view* &key *:default :default-type* &allow-other-keys *Clim Presentation Method*

This presentation method is responsible for ''parsing'' the representation of *type* for a particular view *view* on the stream *stream*. The **clim:accept** method should return a 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.

:default and *:default-type* are the same as they are for **clim:accept**.

The method must specify **&key**, but need only receive the keyword arguments that it is interested in. The remaining keyword arguments will be ignored automatically since the generic function specifies **&allow-other-keys**.

The **clim:accept** method can specialize on the *view* argument in order to define more than one input view for the data. In particular, the **clim:accept** method specializing on the class **clim:textual-view** must be defined if the programmer wants

to allow the type to be used via the keyboard. CLIM uses the *view* argument to generate gadget fields within **clim:accepting-values**.

clim:accept presentation methods can also call **clim:accept** recursively. Such methods should be careful to call **clim:accept** with **:prompt nil** and **:display-default nil**, unless nested prompting is really desired.

For more information on defining presentation types, see the section "Defining a New Presentation Type in CLIM". CLIM offers a number of other functions that are useful within **clim:accept** methods. See the section "Utilities for **clim:accept** Presentation Methods".

clim:accept-from-string *type string* &key *(:view* **clim:+textual-view+***) :default (:default-type* **type***) :activation-gestures :additional-activation-gestures :delimiter-gestures :additional-delimiter-gestures (:start* **0***) :end* *Function*

Reads the printed representation of an object of type *type* from *string*. This function is similar to **clim:accept**, except that the input is taken from *string*, starting at *:start*, and ending at *:end*. *:view*, *:default*, *:default-type*, *:activation-gestures*, *:additional-activation-gestures*, *:delimiter-gestures*, and *:additional-delimiter-gestures* are used as they are for **clim:accept**. This function is analogous to **read-from-string**.

clim:accept-from-string returns three values: the object, its presentation type, and the index in *string* of the next character after the input.

If *:default* is supplied, then the values of *:default* and *:default-type* are returned when the input string is empty.

clim:accept-present-default *type-key parameters options type stream view default default-supplied-p present-p query-identifier* &key *(:prompt* **t***) (:active-p* **t***)* &allow-other-keys *Clim Presentation Method*

This presentation method is called by **clim:accept**, which (in effect) turns into **clim:present** inside of **clim:accepting-values**. The default method calls **clim:present** or **clim:describe-presentation-type** depending on whether *default-supplied-p* is **t** or **nil**.

type, *stream*, *view*, *default*, *query-identifier*, *:prompt*, and *:active-p* are as for **clim:accept**. *default-supplied-p* will be **t** if and only if the **:default** argument was explicitly supplied to the call to **clim:accept**.

clim:accept-values-command-button *((*&optional *stream* &key *:documentation :query-identifier (:cache-value* **t***) (:cache-test* **#'eql***) :view :resynchronize) prompt* &body *body)* *Function*

Displays *prompt* on *stream* and creates an area (the ''button'') which, when the pointer is clicked within it, causes *body* to be evaluated. This function can only be used within the **clim:accepting-values** form. *stream* defaults to ***standard-input***.

prompt A constant string, a compile-time constant that evaluates to a string, or a form which is used to draw the contents of the ''button''.

:documentation
An object that will be used to produce pointer documentation for the command button. If the object is a string, the string itself will be used as the documentation. Otherwise, it must be a function of one argument, the stream to which the documentation will be written. The default is *prompt*.

:query-identifier
A query identifier that uniquely identifies the command button. This option is the same as it is for **clim:accept** within **clim:accepting-values**.

:view A view object, used to select how the button will appear. Some frame managers will use a **clim:push-button** gadget for the command button.

:cache-value
A value that remains constant if the output produced by *body* does not need to be recomputed. *:cache-value* is used by an internal call to **clim:updating-output**.

:cache-test A function of two arguments that is used for comparing cache values. *:cache-test* is used by an internal call to **clim:updating-output**.

:resynchronize
When this is **t**, the dialog is redisplayed on additional time whenever the command button is clicked on. See the **:resynchronize-every-pass** argument to **clim:accepting-values** for more information.

See the section "Examples of Menus and Dialogs in CLIM".

clim:accept-values-command-parser *command-name command-table stream partial-command* &key *:own-window* *Function*

Reads the remaining arguments of a partially filled-in command on behalf of an application frame's command loop by getting input via a dialog. User programs should not call this function explicitly, but should rather bind **clim:*partial-command-parser*** to it.

clim:command-line-read-remaining-arguments-for-partial-command uses this function when the unsupplied arguments are in the middle of the command line rather than at the very end of the command line.

clim:accept-values-pane *Class*

The pane class that is used to implement modeless **clim:accepting-values** dialog panes. It corresponds to the pane type abbreviation **:accept-values** in the **:panes** clause of **clim:define-application-frame**.

See the section "Using the **:panes** Option to **clim:define-application-frame**" and see the section "Menus and Dialogs in CLIM".

clim:accept-values-pane *Clim Command Table*

When you use an **clim:accept-values** pane as one of the panes in a **clim:define-application-frame**, you must inherit from this command table in order to get the commands that operate on the dialog.

clim:accept-values-pane-displayer *frame pane* &key *:displayer :resynchronize-every-pass (:check-overlapping* **t***) :align-prompts :max-height :max-width* *Function*

When you use an **:accept-values** pane, the display function must use **clim:accept-values-pane-displayer**. See the section "Using an **:accept-values** Pane in a CLIM Application Frame".

:displayer is a function that is the body of an **clim:accepting-values** dialog. It takes two arguments, the frame and a stream. The display function should not call **clim:accepting-values** itself, since that is done by **clim:accept-values-pane-displayer**.

The *:resynchronize-every-pass*, *:check-overlapping*, and *:align-prompts* argument are the same as they are for **clim:accepting-values**.

:max-height and *:max-width* are used to constraint the maximum size of the dialog within the pane. They are typically used only when CLIM is composing layout of the entire application frame.

See the section "Examples of CLIM Application Frames".

clim:accepting-values *(*&optional *stream* &key *:frame-class :command-table :own-window :exit-boxes :align-prompts :initially-select-query-identifier :modify-initial-query :resynchronize-every-pass (:check-overlapping* **t***) :label :x-position :y-position :width :height :scroll-bars :text-style :foreground :background)* &body *body* *Macro*

Builds a dialog for user interaction based on calls to **clim:accept** on the stream *stream* within its body. The user can select the values and change them, or use defaults if they are supplied. The dialog will also contain ''Abort'' and ''End'' choices. If the ''End'' choice is selected then **clim:accepting-values** returns whatever values the body returns. If the ''Abort'' choice is selected, **clim:accepting-values** will invoke the **conditions:abort** restart. Callers of **clim:accepting-values** may want to use **conditions:restart-case** or **conditions:with-simple-restart** in order to locally establish a **conditions:abort** restart.

stream The stream **clim:accepting-values** will use to build up the dialog. When *stream* is **t**, that means ***standard-input***.

body The body of the macro, which contains calls to **clim:accept** that will be intercepted by **clim:accepting-values** and used to build up the dialog.

:frame-class

The type of dialog frame to create. The default is to use CLIM's ''usual'' dialog class (either **clim:accept-values** or **clim:accept-values-own-window**). This option allows you to specialize the **clim:accept-values** class, and to customize some of the behavior of dialogs.

:own-window

When *:own-window* is **t**, the dialog will appear in its own ''popped-up'' window. In this case the initial value of *stream* is a window with which the dialog is associated. This is similar to the **:associated-window** argument to **clim:menu-choose**. Within the *body*, the value of *stream* will be the ''popped-up'' window.

The value of *:own-window* can be **nil**, **t**, or a list of alternating keyword options and values. The accepted options are **:right-margin** and **: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 **:right-margin** are the same as for the **:x-spacing** option to **clim:formatting-table**; the allowed values for **:bottom-margin** are the same as for the **:y-spacing** option to **clim:formatting-table**.

:exit-boxes Allows you to specify what the exit boxes should look like. The default behavior in Genera is as though you specified the following:

```
'((:exit "<End> uses these values")
  (:abort "<Abort> aborts"))
```

See the generic function **clim:display-exit-boxes**.

:align-prompts

When **t**, CLIM formats the dialogs so that all of the prompts are right-aligned in a vertical stack, and all of the input fields are left-aligned in a stack just to the right of the prompts. This can sometimes make for more attractive dialogs. The default is **nil**.

: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 **:query-identifier** option to **clim:accept**; use this tag as the value for the *:initially-select-query-identifier* option, as shown in the following example:

```
(defun simple-dialog ()
  (let (a b c)
    (clim:accepting-values
        (*query-io* :initially-select-query-identifier 'the-tag)
      (setq a (clim:accept 'pathname :prompt "A pathname"))
      (terpri *query-io*)
      (setq b (clim:accept 'integer :prompt "A number"
                           :query-identifier 'the-tag))
      (terpri *query-io*)
      (setq c (clim: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.

:modify-initial-query

When *:initially-select-query-identifier* is supplied and *:modify-initial-query* is **t**, the initially selected field will be selected in a ''modify'' mode. That is, the input buffer will contain the default for the field, and the user can then edit it.

:resynchronize-every-pass

A boolean option specifying whether earlier queries depend on later values; the default is **nil**.

When *:resynchronize-every-pass* is **t**, the contents of the dialog 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 dynamically alter the dialog. The following is a simple example. It initially displays an integer field that disappears if a value other than 1 is entered; in its place a two-field display appears.

```
(defun alter-multiple-accept ()
  (let ((flag 1))
    (clim:accepting-values (*query-io*
                            :resynchronize-every-pass t)
      (setq flag (clim:accept 'integer
                              :default flag :prompt "Number"))
      (when (= flag 1)
        (terpri *query-io*)
        (clim:accept 'string :prompt "String")
        (terpri *query-io*)
        (clim:accept 'pathname :prompt "Pathname")))))
```

As the example shows, to use this option effectively, the controlling variable(s) must be initialized outside the lexical scope of the **clim:accepting-values** macro.

:label Allows you to specify a label in **:own-window t** dialogs. This is just like the *:label* option to **clim:menu-choose**.

:x-position and *:y-position*
Allow you to specify where an **:own-window t** dialog will come up. By default, the dialog will come up ''near'' the current position of the pointer. These are just like the options of the same name to **clim:menu-choose**.

:text-style The default text style to use for the dialog.

:foreground and *:background*
These specify the default foreground and background for **:own-window t** dialogs. These default from the associated window.

:scroll-bars
This can be **nil**, **:none**, **:horizontal**, **:vertical**, or **:both**. This specifies whether or not there are scroll bars for **:own-window t** dialogs. The default is **:vertical**.

Note: you must specify either a unique prompt or a query-identifier for each **clim:accept** form within an **clim:accepting-values** form; otherwise, there will be no way that the dialog can identify which **clim:accept** form is being run.

See the section "Examples of Menus and Dialogs in CLIM".

clim:action-gadget *Class*

The class used by gadgets that perform some kind of action, such as a push button; a subclass of **clim:basic-gadget**.

All subclasses of **clim:action-gadget** must handle the **:activate-callback** initarg, which is used to specify the activate callback of the gadget. The activate callback is **nil** or a function of one argument, the gadget.

clim:activate-callback *gadget client id* *Generic Function*

This callback is invoked when the gadget is activated.

The default method (on **clim:action-gadget**) calls the function specified by the **:activate-callback** initarg with one argument, the gadget.

See the section "Using Gadgets in CLIM".

clim:activate-gadget *gadget* *Generic Function*

Causes the gadget to become active, that is, available for input. The function **clim:note-gadget-activated** is called whenever the gadget is made active.

clim:activation-gesture-p *gesture* *Function*

Returns **t** if *gesture* is a currently active activation gesture.

clim:*activation-gestures* *Variable*

A list containing the gesture names of the currently active activation gestures.

clim:add-command-to-command-table *command-name command-table* &key *:name :menu :keystroke (:errorp* **t***)* *Function*

Adds the command named by *command-name* to the command table *command-table*. *command-table* may be either a command table or a symbol that names a command table. The keyword arguments are:

:name The command-line name for the command, which 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 by calling **clim:command-name-from-symbol**.

For the purposes of command-line name lookup, the character case of *name* is ignored.

:menu A command menu item for the command, which 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 it is **t**, an automatically generated menu name will be used. When it is a cons of the form (*string* . *menu-options*), then *string* is the menu name and *menu-options* consists of keyword-value pairs. The valid keywords are **:after** and **:documentation**, which are interpreted as for **clim:add-menu-item-to-command-table**.

:keystroke The value for *keystroke* is either a standard character, a gesture spec, or **nil**. When it is a character or gesture spec, it is the keystroke accelerator for the command; otherwise the command will not be available via keystroke accelerators.

:errorp If the command is already present in the command table and *:errorp* is **t**, the **clim:command-already-present** condition will be signalled. When the command is already present in the command table and *:errorp* is **nil**, then the old command will first be removed from the command table.

See the section "CLIM Command Tables".

clim:add-gesture-name *name type gesture-spec* &key *(:unique* **t***)* *Function*

Adds a gesture named by the symbol *name* to the set of gesture names. *type* is the type of gesture being created, and must be one of the symbols described below. *gesture-spec* specifies the physical gesture that corresponds to the named gesture; its syntax depends on the value of *type*.

If *:unique* is **t**, an error is signalled if there is already a gesture named *gesture-name*. The default is **nil**.

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 (see below). *modifier-key-names* is a (possibly empty) list of modifier key names in the set **:shift**, **:control**, **:meta**, **:super** or **: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**).

When *type* is **:pointer-button**, *gesture-spec* is a list of the form *(button-name . modifier-key-names)*. *button-name* is the name of a pointer button (**:left**, **:middle**, or **:right**), and *modifier-key-names* is as above.

See the section "Gestures and Gesture Names in CLIM".

clim:add-keystroke-to-command-table *command-table keystroke type value* &key *:documentation (:errorp* **t***)* *Function*

Adds a keystroke accelerator to the *command-table*.

command-table
: Either a command table or a symbol that names a command table.

keystroke
: A gesture spec that specifies the accelerator gesture. For applications that have an interactor pane, this will typically correspond to a non-printing character, such as `control-D`, whose gesture spec is `(:D :control)`. For applications that do not have an interactor pane, *keystroke* can correspond to a standard printing character as well, such as `#\X`.

type
: 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, typing the keystroke invokes the command immediately. Otherwise, the user will be prompted for the remaining required arguments.

value
: Meaning depends on the value of *type*, as described above.

:documentation
: A documentation string, which can be used as documentation for the keystroke accelerator.

:errorp If the command menu item associated with *keystroke* is already present in the command table's accelerator table and *:errorp* is **t**, then the **clim:command-already-present** condition will signalled. When the item is already present in the command table's accelerator table and *:errorp* is **nil**, the old item will first be removed.

The following example creates a keystroke accelerator for the **com-next-frame** on `control-N`.

```
(define-debugger-command (com-next-frame :name t)
    ((nframes 'integer
      :default 1
      :prompt "number of frames"))
 (next-frame :nframes nframes))

(clim:add-keystroke-to-command-table
  'debugger '(:n :control) :command '(com-next-frame 1))
```

See the section "CLIM's Keystroke Interaction Style".

clim:add-menu-item-to-command-table *command-table string type value* &key *:documentation (:after* **':end***) :keystroke :text-style (:errorp* **t***)* *Function*

Adds a command menu item to *command-table*'s menu. The arguments are:

command-table
Either a command table or a symbol that names a command table.

string The name of the command menu item. The character case of *string* is ignored. This is how the item will appear in the menu.

type One of: **:command**, **:function**, **:menu**, or **:divider**. 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 a command menu item invokes the command immediately. Otherwise, the user will be prompted for the remaining required arguments.

When *type* is **:menu**, this item indicates that a sub-menu will be invoked, and so *value* should be another command table or the name of another command table.

When *type* is **:function**, *value* is a function of two arguments (the gesture and the accumulated numeric argument) that is called to generate a command object.

When *type* is **:divider**, some sort of a dividing line is displayed in the menu at that point. If the look-and-feel provided by the underlying window system has no corresponding concept, **:divider** items may be ignored. *value* is ignored. If *string* is a string, it will be

used as the divider. Otherwise, *string* is only useful insofar as other calls to **clim:add-menu-item-to-command-table** may use it when **:after** is supplied.

value Meaning depends on the value of *type*, as described above.

:documentation
A documentation string, which can be used as mouse documentation for the command menu item.

:after States where the command menu item should appear in the menu: either **:start**, **:end**, **nil**, a string, or **:sort**. **:start** means to add the new item to the beginning of the menu. A value of **:end** (the default) or **nil** means to add the new item to the end of the menu. A string naming an existing entry means to add the new item after that entry. If **:after** is **:sort**, then the item is inserted in such a way as to maintain the menu in alphabetical order.

:keystroke If supplied, the command menu item will be added to the command table's keystroke accelerator table. The value of *:keystroke* must be a Common Lisp standard character or a gesture spec. This is exactly equivalent to calling **clim:add-keystroke-to-command-table** with the arguments *command-table*, *keystroke*, *type*, and *value*. When *:keystroke* is supplied and *type* is **:command**, typing the accelerator gesture will invoke the command specified by *value*. When *type* is **:menu**, the command will continue to be read from the sub-menu indicated by *value* in a window system specific manner.

:text-style Allows you to specify the text style for any particular menu item.

:errorp If the item named by *string* is already present in the command table's menu and *:errorp* is **t**, then the **clim:command-already-present** condition will be signalled. 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.

See the section "CLIM's Command Menu Interaction Style".

clim:add-output-record *child record* *Generic Function*

Adds the output record *child* to the output record *record*, sets the parent of *child* to be *record*, and calls **clim:recompute-extent-for-new-child** to inform *record* of the new child.

Any class that is a subclass of **clim:output-record** must implement this method.

See the section "Concepts of CLIM Output Recording".

clim-sys:all-processes *Function*

Returns a sequence of all of the currently running processes.

clim:allocate-pixmap *medium width height* *Function*

Allocates and returns a pixmap object that can be used on any medium that shares the same characteristics as *medium*. (The exact definition of ''shared characteristics'' will vary from host to host.) *medium* can be a medium, a sheet, or a stream.

The resulting pixmap will be at least *width* units wide, *height* units high, and as deep as is necessary to store the information for the medium.

The returned value is the pixmap.

See the section "Pixmaps in CLIM".

clim-sys:allocate-resource *name* &rest *parameters* *Function*

Allocates an object from the resource named *name*, using the parameters given by *parameters*. *name* is a symbol that names a resource. The returned value is the allocated object.

See the section "Resources in CLIM".

clim:allocate-space *pane width height* *Generic Function*

During the space allocation pass, a composite pane arranges its children within the available space and allocates space to them according to their space requirements and its own composition rules by calling **clim:allocate-space** on each of the child panes. *width* and *height* are the width and height of *pane* in device units.

clim:allocate-space is intended to be specialized by most pane classes. For example, the method for the class **clim:vbox-pane** allocates enough space for itself to hold all of its child panes in a vertical stack.

See the section "Details of CLIM's Layout Algorithm".

and &rest *types* *Clim Presentation Type*

The presentation type that is used for multiple inheritance. **and** is usually used in conjunction with **satisfies**. For example,

```
(and integer (satisfies oddp))
```

The elements of *types* can be presentation type abbreviations.

The first type in *types* is in charge of accepting and presenting. The remaining elements of *types* are used for type checking (for example, filtering applicability of presentation translators).

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.

clim:application-frame *Class*

The protocol class that corresponds to a CLIM application frame. If you want to create a new class that obeys the application frame protocol, it must be a subclass of **clim:application-frame**.

clim:*application-frame* *Variable*

The current application frame. The global 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**.

This variable is bound by an **:around** method of **clim:run-frame-top-level** on **clim:application-frame**. You should not rebind it, since CLIM depends on its value.

clim:application-pane *Class*

The pane class that is used to implement ''application'' panes. This is the type of pane created by **clim:make-clim-application-pane**, and corresponds to the pane type abbreviation **:application** in the **:panes** clause of **clim:define-application-frame**. The default method for **clim:frame-standard-output** will return the first pane of this type in a frame.

For **clim:application-pane**, the default for the **:display-time** option is **:command-loop**, and the default for the **:scroll-bars** option is **:both**.

See the section "Using the **:panes** Option to **clim:define-application-frame**".

clim:application-frame-p *object* *Function*

Returns **t** if and only if *object* is of type **clim:application-frame**.

clim:apply-presentation-generic-function *presentation-function-name* &body *arguments* *Macro*

Applies the presentation generic function *presentation-function-name* to arguments *arguments* using **apply**.

The *presentation-function-name* argument is not evaluated. The value of *presentation-function-name* can be any of the presentation generic functions defined by CLIM (**clim:accept**, **clim:present**, **clim:describe-presentation-type**, **clim:presentation-typep**, **clim:presentation-subtypep**, **clim:accept-present-default**, **clim:presentation-type-specifier-p**, **clim:presentation-refined-position-test**, or **clim:highlight-presentation**) or any presentation generic function you have defined yourself.

clim:area *Class*

This is a subclass of **clim:region** that denotes regions that have dimensionality 2 (that is, have area). If you want to create a new class that obeys the area protocol, it must be a subclass of **clim:area**.

Making an **clim:area** object with no area canonicalizes it to **clim:+nowhere+**.

clim:areap *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:area**.

clim:armed-callback *gadget client id* *Generic Function*

This callback is invoked when the gadget *gadget* is armed. The exact definition of arming varies from gadget to gadget, but typically a gadget becomes armed when the pointer is moved into its region.

The default method for **clim:armed-callback** (on **clim:basic-gadget**) calls the function specified by the **:armed-callback** initarg.

See the section "Using Gadgets in CLIM".

clim:+background-ink+ *Constant*

An indirect ink that uses the medium's background design. See the section "Indirect Ink in CLIM".

clim:basic-gadget *Class*

The implementation class on which many CLIM gadgets are built.

clim:beep &optional *(stream* ***standard-output****)* *Function*

Attracts the user's attention, usually with an audible sound.

clim:blank-area *Presentation Type*

The presentation type that represents all the places in a window where there is no applicable presentation. CLIM provides a single ''null presentation'' (represented by the value of **clim:*null-presentation***) of this type.

clim:boolean *Clim Presentation Type*

The presentation type that represents **t** or **nil**. The textual representation is "Yes" and "No", respectively.

clim:bounding-rectangle* *region* *Generic Function*

Returns the bounding rectangle of *region* as four real numbers that specify the left, top, right, and bottom edges of the bounding rectangle. *region* must be a bounded region, such as an output record, a sheet or window, or a geometric object such as a line or an ellipse.

The coordinates of the bounding rectangle of sheets and output records are maintained relative to the parent of the sheet or output record.

See the section "Bounding Rectangles in CLIM".

clim:bounding-rectangle *Class*

The protocol class that corresponds to a bounding rectangle. If you want to create a new class that obeys the bounding rectangle protocol, it must be a subclass of **clim:bounding-rectangle**.

clim:bounding-rectangle *region* *Generic Function*

Returns a new bounding rectangle for *region* as a **clim:standard-bounding-rectangle** object. *region* is as for **clim:bounding-rectangle***.

clim:bounding-rectangle-bottom *region* *Function*

Returns the coordinate of the bottom edge of the bounding rectangle of *region*. *region* is as for **clim:bounding-rectangle***.

clim:bounding-rectangle-height *region* *Function*

Returns the height of the bounding rectangle of *region*. *region* is as for **clim:bounding-rectangle***.

clim:bounding-rectangle-left *region* *Function*

Returns the coordinate of the left edge of the bounding rectangle of *region*. *region* is as for **clim:bounding-rectangle***.

clim:bounding-rectangle-max-x *region* *Generic Function*

Returns the coordinate of the right edge of the bounding rectangle of *region*. In Symbolics CLIM 2.0, this is the same thing as **clim:bounding-rectangle-left**.

clim:bounding-rectangle-max-y *region* *Generic Function*

Returns the coordinate of the bottom edge of the bounding rectangle of *region*. In Symbolics CLIM 2.0, this is the same thing as **clim:bounding-rectangle-bottom**.

clim:bounding-rectangle-min-x *region* *Generic Function*

Returns the coordinate of the left edge of the bounding rectangle of *region*. In Symbolics CLIM 2.0, this is the same thing as **clim:bounding-rectangle-left**.

clim:bounding-rectangle-min-y *region* *Generic Function*

Returns the coordinate of the top edge of the bounding rectangle of *region*. In Symbolics CLIM 2.0, this is the same thing as **clim:bounding-rectangle-top**.

clim:bounding-rectangle-p *object* *Function*

Returns **t** if and only if *object* is of type **clim:bounding-rectangle**.

clim:bounding-rectangle-position *region* *Generic Function*

Returns the position of the bounding rectangle of *region* as two values, the left and top coordinates of the bounding rectangle.

The coordinate system of the position returned by **clim:bounding-rectangle-position** depends on the type of *region*. If *region* is an output record, the position will be relative to the parent output record of *region*. If *region* is a subclass of **clim:region**, the position will be an ''absolute'' position.

clim:bounding-rectangle-right *region* *Function*

Returns the coordinate of the right edge of the bounding rectangle of *region*. *region* is as for **clim:bounding-rectangle***.

clim:bounding-rectangle-set-position *region x y* *Generic Function*

Changes the position of the bounding rectangle of *region* to the new position *x* and *y*. *x* and *y* are the new left and top coordinates of the bounding rectangle.

The coordinate system of *x* and *y* depends on the type of *region*. If *region* is an output record, *x* and *y* should be relative to the parent output record of *region*. If *region* is a subclass of **clim:region**, *x* and *y* should be an ''absolute'' position.

clim:bounding-rectangle-size *region* *Function*

Returns the size (as two values, width and height) of the bounding rectangle of *region*. *region* is as for **clim:bounding-rectangle***.

clim:bounding-rectangle-top *region* *Function*

Returns the coordinate of the top edge of the bounding rectangle of *region*. *region* is as for **clim:bounding-rectangle***.

clim:bounding-rectangle-width *region* *Function*

Returns the width of the bounding rectangle of *region*. *region* is as for **clim:bounding-rectangle***.

clim:bury-frame *frame* *Generic Function*

Buries the application frame *frame* so that it is underneath all of the other host windows. **clim:bury-frame** works by calling **clim:bury-sheet** on the frame's top-level sheet. This does not change the state of the frame.

clim:call-presentation-menu *presentation input-context frame window x y* &key *(:for-menu* **t***) :label* *Function*

Finds all the applicable translators for *presentation* in the input context *input-context*, creates a menu that contains all of the translators, and pops up a menu from which the user can choose a translator. After the translator is chosen, it is called and the values are returned to the appropriate call to **clim:with-input-context**.

frame, window, *x*, and *y* are as for **clim:find-applicable-translators**. *:for-menu*, which defaults to **t**, is used to decide which presentation translators go in the menu (the value of their **:menu** option must match *:for-menu*). *:label* is used as a label for the menu, and defaults to **nil**, meaning the menu will not be labelled.

See the section "Low Level Functions for CLIM Presentation Translators".

For example, CLIM's ''menu'' translator could be defined as follows:

```
(clim:define-presentation-action presentation-menu
    (t nil clim:presentation-menu-command-table
     :documentation "Menu"
     :menu nil                ;this doesn't go into any menu
     :gesture :menu)
    (presentation frame window x y)
  (clim:call-presentation-menu presentation clim:*input-context*
                               frame window x y
                               :for-menu t))
```

clim:call-presentation-translator *translator presentation context-type frame event window x y* *Function*

Calls the function that implements the body of *translator* on *presentation*'s object, and passes *presentation*, *context-type*, *frame*, *event*, *window*, *x*, and *y* to the body of the translator as well.

frame, window, *x*, and *y* are as for **clim:find-applicable-translators**. *context-type* is the presentation type for the context that matched. *event* is the event corresponding to the user's gesture.

The returned values are the same as the values returned by the body of the translator, which should be the translated object and the translated type.

See the section "Low Level Functions for CLIM Presentation Translators".

clim:catch-abort-gestures *(format-string* &rest *format-args)* &body *body* *Macro*

clim:catch-abort-gestures is a convenient macro that you can use in the top level loop of an application (or in any call to **clim:accept**) that establishes a restart for **conditions:abort** and a handler for **clim:abort-gesture**, and then evaluates *body*. *format-string* and *format-args* are used in **conditions:abort** restart.

This macro could be written as follows:

```
(defun handle-abort-gesture (condition)
  (declare (ignore condition))
  (abort))

(defmacro catch-abort-gestures ((format-string &rest format-args) &body body)
  `(conditions:with-simple-restart
       (abort ,format-string ,@format-args)
     (conditions:handler-bind
         ((clim:abort-gesture #'handle-abort-gesture))
       ,@body)))
```

The top level loop of an application could use this as follows:

```
(loop
  (clim:catch-abort-gestures
      ("Return to ~A command level" (clim:frame-pretty-name frame))
    (clim:redisplay-frame-panes frame)
    (when interactor
      (fresh-line *standard-input*)
      (write-string prompt *standard-input*))
    (let ((command (clim:read-frame-command frame :stream command-stream)))
      (when interactor
        (terpri *standard-input*))
      (when command
        (clim:execute-frame-command frame command)))))
```

character *Clim Presentation Type*

The presentation type that represents a Common Lisp character object.

clim:check-box *Class*

A check box is similar to a radio box: it is a special kind of gadget that contains one or more toggle buttons. At any one time, zero or more of the buttons managed by the check box may be ‘‘on’’. The contents of a check box are its buttons, and as such a check box is responsible for laying out the buttons that it contains.

It is a subclass of **clim:value-gadget** and **clim:oriented-gadget-mixin**.

See the section "Using Gadgets in CLIM".

In addition to the initargs for **clim:value-gadget** and the usual pane initargs (**:foreground**, **:background**, **:text-style**, space requirement options, and so forth), the following initargs are supported:

:selection This is used to specify which button, if any, should be initially selected.

:choices This is used to specify all of the buttons that serve as choices.

As the user changes the selections, the newly selected (or deselected) button will have its **clim:value-changed-callback** handler invoked.

Calling **clim:gadget-value** on a check box will return a sequence of the currently selected toggle buttons. The value of the check box can be changed by calling **setf** on **clim:gadget-value**.

A check box might be created as follows, although it is generally more convenient to use **clim:with-radio-box**:

```
(let* ((choices
         (list (clim:make-pane 'clim:toggle-button
                 :label "One" :width 80)
               (clim:make-pane 'clim:toggle-button
                 :label "Two" :width 80)
               (clim:make-pane 'clim:toggle-button
                 :label "Three" :width 80)))
       (current (second choices)))
  (clim:make-pane 'clim:check-box
    :choices choices
    :selection (list current)
    :value-changed-callback 'radio-value-changed-callback))

(defun radio-value-changed-callback (radio-box value)
  (declare (ignore radio-box))
  (format t "~&Radio box toggled to ~S" value))
```

clim:check-box-current-selection *check-box* *Generic Function*

Returns a sequence of the currently selected items in the check box. Each of the selections will be one of the toggle buttons in the check box.

You can use **setf** on this in order to set the current selection for the check box, or you can use **setf** on **clim:gadget-value** of the check box to accomplish the same thing.

clim:check-box-selections *check-box* *Generic Function*

Returns a sequence of all of the selections in the check box. The elements of the sequence will be toggle buttons.

clim:check-box-view *Class*

The class that represents the view corresponding to a check box. This is usually used for a ''some of'' choice, such as a **clim:subset** presentation type.

clim:+check-box-view+ *Constant*

An instance of the class **clim:check-box-view**.

clim:clear-output-record *record* *Generic Function*

Removes all of the child output records from the output record *record*.

Any class that is a subclass of **clim:output-record** must implement this method.

clim-sys:clear-resource *resource* *Function*

Clears the resource named *name*, that is, removes all of the resourced objects from the resource. *name* is a symbol that names a resource.

See the section "Resources in CLIM".

clim:clim-stream-pane *Class*

This class implements a pane that supports the CLIM graphics, extended input and output, and output recording protocols. Most non-gadget application panes are subclasses of this class, including **clim:application-pane** and **clim:interactor-pane**.

See the section "Panes in CLIM".

clim:close *stream* &key *:abort* *Generic Function*

In CLIM, **close** is defined as a generic function. Otherwise, it behaves the same as the normal Common Lisp **close** function.

clim:color *Class*

A color is a completely opaque design that represents the intuitive definition of color (such as white, black, red, or pale yellow). **clim:color** is the class that represents colors in CLIM.

clim:color-ihs *color* *Generic Function*

Returns three values, the *intensity*, *hue*, and *saturation* components of *color*. The first value is a real number between 0 and the square root of 3 (inclusive). The second and third values are real numbers between 0 and 1 (inclusive).

clim:color-rgb *color* *Generic Function*

Returns three values, the *red*, *green*, and *blue* components of *color*. The values are real numbers between 0 and 1 inclusive.

clim:colorp *object* *Function*

Returns **t** if and only if *object* is of type **clim:color**.

clim:command &key *:command-table* *Clim Presentation Type*

The presentation type used to represent a CLIM command processor command and its arguments. *:command-table* can be either a command table or a symbol that names a command table.

If *:command-table* is not supplied, it defaults to the command table for the current application, that is, **(clim:frame-command-table clim:*application-frame*)**.

When you call **clim:accept** on this presentation type, the returned value is a list; the first element is the command name, and the remaining elements are the command arguments. You can use **clim:command-name** and **clim:command-arguments** to access the name and arguments of the command object.

For more information about CLIM command objects, see the section "Command Objects in CLIM".

clim:command-accessible-in-command-table-p *command-name command-table* *Function*

If the command named by *command-name* is not accessible in *command-table*, then this function returns **nil**. Otherwise, it returns the command table in which the command was found. *command-table* may be either a command table or a symbol that names a command table.

A command is *present* in a command table when it has been added to that command table. A command is *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.

clim:command-already-present *Condition*

A condition that is signalled when a command is already present in the command table in functions such as **clim:add-command-to-command-table**.

clim:command-arguments *command* *Function*

Given a command object *command*, returns the command's arguments.

clim:*command-dispatchers* *Variable*

This is a list of characters that indicate that CLIM should read a command when CLIM is accepting input of type **clim:command-or-form**. The default value for this is colon (`#\:`).

clim:command-enabled *command-name frame* &optional *command-table*
Generic Function

Returns **t** if the command named by *command-name* is presently enabled in *command-table* for the frame *frame*, otherwise returns **nil**. If *command-name* is not accessible in *command-table*, **clim:command-enabled** will return **nil**. *command-table* defaults to the current command table for *frame*.

You can use **setf** on **clim:command-enabled** in order to enable or disable a command.

clim:command-line-command-parser *command-table stream* *Function*

Reads a command line on behalf of an application frame's command loop. User programs should not call this function explicitly, but should rather bind **clim:*command-parser*** to it.

This is the function CLIM uses to parse commands in a command-line driven interface.

clim:command-line-command-unparser *command-table stream args-to-go* &rest *keys* &key *:for-context-type (:acceptably* **t***)* &allow-other-keys *Function*

''Unparses'' a command line on behalf of an application frame's command loop. User programs should not call this function explicitly, but should rather bind **clim:*command-unparser*** to it.

clim:command-line-name-for-command *command-name command-table* &key *(:errorp* **t***)* *Function*

Returns the command-line name for *command-name* as it is installed in *command-table*. If the command is not accessible in *command-table* (or the command has no command-line name and *:errorp* is **t**), then the **clim:command-not-accessible** condition is signalled.

If the command does not have a command-line name in the *command-table* and *:errorp* is **:create**, then the returned value will be an automatically created command-line name.

command-table may be either a command table or a symbol that names a command table.

This function is the inverse of **clim:find-command-from-command-line-name**.

clim:command-line-read-remaining-arguments-for-partial-command *partial-command command-table stream start-location* &key *:for-accelerator* *Function*

Reads the remaining arguments of a partial command line on behalf of an application frame's command loop. User programs should not call this function explicitly, but should rather bind **clim:*partial-command-parser*** to it.

clim:command-menu-enabled *command-table frame* *Generic Function*

Returns **t** if the command table *command-table* is presently enabled in the command menu for the frame *frame*, otherwise returns **nil**.

You can use **setf** on **clim:command-menu-enabled** in order to enable or disable a command table in the command menu for *frame*.

This function is like **clim:command-enabled**, except that it operates only on the **:menu** items in a command table's menu for a particular frame.

clim:command-menu-item-options *item* *Function*

Returns a property list of the options for the command menu item *item*. Currently, the only option is **:text-style**, which specifies what text style the item should be displayed in.

clim:command-menu-item-type *item* *Function*

Returns the type of the command menu item *item*. This will be one of **:command**, **:function**, **:menu**, or **:divider**.

clim:command-menu-item-value *item* *Function*

Returns the value of the command menu item *item*. For example, if the type of *item* is **:command**, this will return a command or a command name.

clim:command-menu-pane *Class*

The pane class that is used to implement command menu panes (but *not* menu bars). It corresponds to the pane type abbreviation **:command-menu** in the **:panes** clause of **clim:define-application-frame**. The default display function for panes of this type is **clim:display-command-menu**.

For **clim:command-menu-pane**, the default for the **:display-time** option is **:command-loop**, the default for the **:incremental-redisplay** option is **t**, and the default for the **:scroll-bars** option is **nil**.

Note that many applications will not use any panes of this type, since most frame managers automatically provide a menu bar for the frame.

clim:command-name *command* *Function*

Given a command object *command*, returns the command name.

clim:command-name &key *:command-table* *Clim Presentation Type*

The presentation type used to represent the name of a CLIM command processor command in the command table *:command-table*.

:command-table may be either a command table or a symbol that names a command table. If *:command-table* is not supplied, it defaults to the command table for the current application. The textual representation of a **clim:command-name** object is the command-line name of the command, while the internal representation is the command name.

clim:command-name-from-symbol *symbol* *Function*

Generates a string suitable for use as a command line name from the symbol *symbol*. The string consists the symbol name with the hyphens replaced by spaces, and the words capitalized. If the symbol name is prefixed by `"com-"`, the prefix is removed. For example, if the symbol is **com-show-file**, the resulting string will be `"Show File"`.

clim:command-not-accessible *Condition*

A condition that is signalled when the command you are looking for is not accessible in the command table, for example, **clim:find-command-from-command-line-name**.

clim:command-not-present *Condition*

A condition that is signalled when the command you are looking for is not present in the command table.

clim:command-or-form &key *:command-table* *Clim Presentation Type*

The presentation type used to represent either a Lisp form or a CLIM command processor command and its arguments. In order for the user to indicate that he wishes to enter a command, a command dispatch character must be typed as the first character of the command line.

See the variable **clim:*command-dispatchers***.

:command-table may be either a command table or a symbol that names a command table. If *:command-table* is not supplied, it defaults to the command table for

the current application, that is, **(clim:frame-command-table clim:*application-frame*)**.

clim:*command-parser* *Variable*

The currently active command parsing function.

The default for this is **clim:command-line-command-parser** when there is at least one interactor pane in the application frame, otherwise the default is **clim:menu-command-parser**.

If you want a dialog-driven command processing loop, you can use the parsing function **clim:accept-values-command-parser**.

clim:command-present-in-command-table-p *command-name command-table* *Function*

Returns **t** if *command-name* is present in *command-table*.

A command is *present* in a command table when it has been added to that command table. A command is *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.

clim:command-table *Class*

The class that represents command tables.

clim:command-table-already-exists *Condition*

This condition is signalled by **clim:make-command-table** when you try to create a command table that already exists.

clim:command-table-inherit-from *command-table* *Generic Function*

Returns a list of all of the command tables from which *command-table* inherits. You can **setf** this in order to change the inheritance of *command-table*.

clim:command-table-name *command-table* *Generic Function*

Returns the name of the command table *command-table*.

clim:command-table-not-found *Condition*

This condition is signalled by functions such as **clim:find-command-table** when the named command table cannot be found.

clim:*command-unparser* *Variable*

The currently active command unparsing function.

The default for this is **clim:command-line-command-unparser** when there is at least one interactor pane in the application frame, otherwise there is no command unparser.

clim-sys:current-process *Function*

Returns the currently running process, which will be a process object.

See the section "Multi-processing in CLIM".

clim:complete-from-generator *string generator delimiters* &key *(:action* **:complete***) :predicate* *Function*

Given an input string *string* and a list of delimiter characters *delimiters* that act as partial completion characters, **clim:complete-from-generator** completes against the 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.

:action must be one of **:complete**, **:complete-maximal**, **:complete-limited**, or **:possibilities**. See the function **clim:complete-input**.

:predicate is **nil** or a function of one argument, an object. If the predicate returns **t**, the possibility corresponding to the object is processed, otherwise it is not. You can supply this when you want to prevent some objects from being part of the completion set.

clim:complete-from-generator returns five values, the completed input string, the success value (**t** 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**.

You might use **clim:complete-from-generator** inside the **clim:accept** method for a cardinal number presentation type as follows:

```
(let ((possibilities '(("One" 1) ("Two" 2) ("Three" 3))))
  (flet ((generator (string suggester)
           (declare (ignore string))
           (dolist (possibility possibilities)
             (funcall suggester (first possibility) (second possibility)))))
    (clim:complete-input
      stream
      #'(lambda (string action)
          (clim:complete-from-generator
            string #'generator nil
            :action action)))))
```

clim:complete-from-possibilities *string completions delimiters* &key *(:action* **:complete***) :predicate (:name-key* **#'first***) (:value-key* **#'second***)* *Function*

Given an input string *string* and a list of delimiter characters *delimiters* that act as partial completion characters, **clim:complete-from-possibilities** completes against the possibilities in the sequence (a list or a vector) *completions*.

The completion string is extracted from the possibilities in *completions* by applying *:name-key*. The object is extracted by applying *:value-key*.

:action must be one of **:complete**, **:complete-maximal**, **:complete-limited**, or **:possibilities**. See the function **clim:complete-input**.

:predicate is either **nil** or a function of one argument, an object. If the predicate returns **t**, the possibility corresponding to the object is processed, otherwise it is not. You can supply this when you want to prevent some objects from being part of the completion set.

clim:complete-from-possibilities returns five values, the completed input string, the success value (**t** 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**.

You might use **clim:complete-from-possibilities** inside the **clim:accept** method for a cardinal number presentation type as follows:

```
(let ((possibilities '(("One" 1) ("Two" 2) ("Three" 3))))
  (clim:complete-input
    stream
    #'(lambda (string action)
        (clim:complete-from-possibilities
          string possibilities nil
          :action action))))
```

clim:complete-input *stream function* &key *:partial-completers :allow-any-input :possibility-printer (:help-displays-possibilities* **t***)* *Function*

Reads input from *stream*, completing from a set of possibilities.

function is a function of two arguments which is called to generate the possibilities. Its first argument is a string containing the input so far. Its second argument is the completion mode, one of the following:

- **:complete** — 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.
- **:complete-limited** — Completes the input up to the next partial delimiter.
- **:complete-maximal** — Completes the input as much as possible.

- **:possibilities** — Causes **clim:complete-input** to return a list of the possible completions.

function must return five values:

- *string* — The completed input string.
- *success* — **t** if completion was successful (otherwise **nil**).
- *object* — The accepted object (**nil** if unsuccessful).
- *nmatches* — The number of possible completions of the input.
- *possibilities* — An alist of completions ((*string object*) ...), returned only when the completion mode is **:possibilities**.

clim:complete-input returns three values: *object*, *success*, and *string*.

:partial-completers is a (possibly empty) list of characters that delimit portions of a name that can be completed separately. The default is an empty list. Typical partial completers are spaces and dashes.

If *:allow-any-input* is **t**, **clim:complete-input** will return as soon as the user types an activation gesture, even if the input is not any of the possibilities. The default is **nil**. Use this when you want to complete from a set of existing objects, but want to allow the user to enter a new object.

If *:possibility-printer* is supplied, it must be a function of three arguments: a possibility, a presentation type, and a stream. The function should display 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 types a help gesture (one of the gestures in **clim:*help-gestures***), CLIM will display all the matching possibilities. If **nil**, then CLIM will not display the possibilities unless the user types a possibility gesture (one of the gestures in **clim:*possibilities-gestures***).

clim:completing-from-suggestions *(stream* &rest *options* &key *:partial-completers :allow-any-input :possibility-printer (:help-displays-possibilities* **t***))* &body *body* *Macro*

Reads input from *stream*, completing from a set of possibilities generated by calls to **clim:suggest** in *body*. Returns three values: *object*, *success*, and *string*.

:partial-completers, *:allow-any-input*, *:possibility-printer*, and *:help-displays-possibilities* are as for **clim:complete-input**.

You could use the following in a **clim:accept** method for cardinal numbers.

```
(clim:completing-from-suggestions (stream)
  (map nil
       #'(lambda(x)
           (clim:suggest (car x) (cdr x)))
       '(("One" . 1)
         ("Two" . 2)
         ("Three" . 3))))
```

clim:completion *sequence* &key *:test :value-key* *Clim Presentation Type*

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 **clim:completion** type, including **clim:token-or-type**, **clim:null-or-type**, **member**, **clim:member-sequence**, and **clim:member-alist**.

The presentation type parameters are:

sequence 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. **clim:accept** reads a name and returns a value. **clim:present** is given a value and outputs a name.

:test A function that compares two values for equality. The default is **eql**.

:value-key A function that returns a value given an element of *sequence*. The default is **identity**.

The following presentation type options are available:

:name-key
A function that returns a name, as a string, given an element of *sequence*. The default is a function that behaves as follows:

Argument	*Returned Value*
string	the string
null	the string **"NIL"**
cons	**string** of the **car**
symbol	**string-capitalize** of its name
otherwise	**princ-to-string** of it

:documentation-key
A function that returns **nil** or a descriptive string, given an element of *sequence*. The default always returns **nil**.

:partial-completers
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`.

clim:*completion-gestures* *Variable*

A list of gesture names that cause **clim:complete-input** to complete the input as fully as possible. On most systems, this includes the gesture corresponding to the `#\Tab` character. On Genera, it includes the gesture for `#\Complete` as well.

complex &optional *type* *Clim Presentation Type*

The presentation type that represents a complex number. It is a subtype of **number**.

type is the type to use for the components. It must be a subtype of **real**.

clim:compose-in *design1 design2* *Generic Function*

Composes a design by using the color (or ink) of *design1* and clipping to the inside of *design2*. That is, *design2* specifies the mask to use for changing the shape of the design.

More precisely, at each point in the drawing plane the resulting design specifies a color and an opacity as follows: the color is the ink of *design1*. The opacity is the opacity of *design1*, multiplied by the *stencil opacity* of *design2*.

The *stencil opacity* of a design at a point is defined as the opacity 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. (With this definition, the stencil opacity of a member of class **clim:opacity** is simply its value.)

If *design2* is a solid design, the effect of **clim:compose-in** is to clip *design1* to *design2*. If *design2* is translucent, the effect is a soft matte.

If both arguments are regions, **clim:compose-in** is the same as **clim:region-intersection**.

The result returned by **clim:compose-in** might be freshly constructed or might be an existing object.

See the section "Drawing with Designs in CLIM".

clim:compose-out *design1 design2* *Generic Function*

Composes a design by using the color (or ink) of *design1* and clipping to the outside of *design2*. That is, *design2* specifies the mask to use for changing the shape of the design.

More precisely, at each point in the drawing plane the resulting design specifies a color and an opacity as follows: the color is the ink of *design1*. The opacity is the opacity of *design1*, multiplied by 1 minus the stencil opacity of *design2*.

If *design2* is a solid design, the effect of **clim:compose-out** is to clip *design1* to the complement of d*esign2*. If *design2* is translucent, the effect is a soft matte.

If both arguments are regions, **clim:compose-out** is the same as **clim:region-difference**.

The result returned by **clim:compose-out** might be freshly constructed or might be an existing object.

See the section "Drawing with Designs in CLIM".

clim:compose-over *design1 design2* *Generic Function*

Composes a design that is equivalent to *design1* drawn on top of *design2*. Drawing the resulting design produces the same visual appearance as drawing *design2* and then drawing *design1*, but might be faster and might not allow the intermediate state to be visible on the screen.

If both arguments are regions, **clim:compose-over** is the same as **clim:region-union**.

The result returned by **clim:compose-over** might be freshly constructed or might be an existing object.

See the section "Drawing with Designs in CLIM".

clim:compose-rotation-with-transformation *transform angle* &optional *origin*
Generic Function

Creates a new transformation by composing *transform* with a given rotation, as specified by *angle* and *origin*. *angle* is in radians. If *origin* is supplied it must be a point; if not supplied it defaults to (0,0). The order of composition is that the rotation ''transformation'' is first, followed by *transform*.

This function could have been implemented as follows:

```
(defun compose-rotation-with-transformation
       (transform angle &optional origin)
  (clim:compose-transformations
    transform
    (clim:make-rotation-transformation angle origin)))
```

clim:compose-scaling-with-transformation *transform mx my* &optional *origin*
Generic Function

Creates a new transformation by composing *transform* with a given scaling, as specified by *mx*, *my*, and *origin*. If *origin* is supplied it must be a point; if not supplied it defaults to (0,0). The order of composition is that the scaling ''transformation'' is first, followed by *transform*.

This function could have been implemented as follows:

```
(defun compose-scaling-with-transformation
       (transform mx my &optional origin)
  (clim:compose-transformations
    transform
    (clim:make-scaling-transformation mx my origin)))
```

clim:compose-space *pane* &key *:width :height* *Generic Function*

During the space composition pass, a composite pane queries each of its children to find out how much space they requires by calling **clim:compose-space**. Each child pane answers by returning a **clim:space-requirement** object. The composite then forms its own space requirement by composing the space requirements of its children according to its own rules for laying out its children.

The value returned by **clim:compose-space** is a space requirement object that represents how much space the pane *pane* requires for itself and its children.

:width and *:height* are real numbers that the **clim:compose-space** method for a pane may use as ''recommended'' values for the width and height of the pane. These are used to drive top-down layout, such as occurs when the overall size for a frame is explicitly supplied.

clim:compose-space is intended to be specialized by most pane classes. For example, the method for the class **clim:vbox-pane** requests enough space for itself to hold all of its child panes in a vertical stack.

See the section "Details of CLIM's Layout Algorithm".

clim:compose-transformation-with-rotation *transform angle* &optional *origin* *Generic Function*

Creates a new transformation by composing the rotation given by *angle* and *origin* with *transform*. The order of composition is that *transform* is first, followed by the rotation ''transformation''. The angle is specified in radians. If *origin* is supplied it must be a point; if not supplied it defaults to (0,0).

clim:compose-transformation-with-scaling *transform mx my* &optional *origin* *Generic Function*

Creates a new transformation by composing the scaling given by *mx*, *my*, and *origin* with *transform*. The order of composition is that *transform* is first, followed by the scaling ''transformation''. Multiplies the X-coordinate distance of every point from *origin* by *mx* and the Y-coordinate distance of every point from *origin* by *my*. If *origin* is supplied it must be a point; if not supplied it defaults to (0,0).

clim:compose-transformation-with-translation *transform dx dy* *Generic Function*

Creates a new transformation by composing the translation given by *dx* and *dy* with *transform*. The order of composition is that *transform* is first, followed by the translation ''transformation''. *dx* gives the amount to translate in the X direction, and *dy* gives the amount to translate in the Y direction.

clim:compose-transformations *transform1 transform2* *Generic Function*

Returns a transformation that is the composition of its arguments. Composition is in right-to-left order, that is, the resulting transformation represents the effects of applying *transform2* followed by *transform1*. This is consistent with the order in which **clim:with-translation**, **clim:with-rotation**, and **clim:with-scaling** compose. For example, the following two forms result in the same transformation, presuming that the stream's transformation is the identity transformation:

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

(clim:with-translation (stream dx dy)
  (clim:with-rotation (stream angle)
    (clim:medium-transformation stream)))
```

Any arbitrary transformation can be built up by composing a number of simpler transformations, but that composition is not unique.

clim:compose-translation-with-transformation *transform dx dy* *Generic Function*

Creates a new transformation by composing *transform* with a given translation, as specified by *dx* and *dy*. The order of composition is that the translation ''transformation'' is first, followed by *transform*.

This function could have been implemented as follows:

```
(defun compose-translation-with-transformation
       (transform dx dy)
  (clim:compose-transformations
    transform
    (clim:make-translation-transformation dx dy)))
```

clim:contrasting-dash-patterns-limit *port* *Generic Function*

Returns the number of contrasting dash patterns that the port *port* can generate. On most ports, this number is presently 16.

clim:contrasting-inks-limit *port* *Generic Function*

Returns the number of contrasting inks that the port *port* can generate. On most ports, this number is presently 8.

clim:+control-key+ *Constant*

The modifier state bit that corresponds to the user holding down the control key on the keyboard. See the section "Operators for Gestures in CLIM".

clim:convert-from-absolute-to-relative-coordinates *stream output-record* *Function*

Returns the X and Y offsets that map the ''absolute'' coordinates of an output record to parent-relative coordinates. This is the inverse of **clim:convert-from-relative-to-absolute-coordinates**.

clim:convert-from-relative-to-absolute-coordinates *stream output-record* *Function*

The coordinates of output records in CLIM are maintained so that each record's coordinates are relative to its parent. Many CLIM functions, such as **clim:replay-output-record**, must compute the absolute coordinates of a record in order to work properly. See the section "Output Recording in CLIM".

clim:convert-from-relative-to-absolute-coordinates returns the X and Y offsets that map the parent-relative coordinates of an output record to ''absolute'' coordinates.

Here is an example of using **clim:replay-output-record** in a way that maintains the X and Y offsets correctly:

```
(multiple-value-bind (x-offset y-offset)
    (clim:convert-from-relative-to-absolute-coordinates
      stream (clim:output-record-parent record))
  (clim:replay-output-record record stream region x-offset y-offset))
```

The following function will map over all of the descendants of an output record that overlap a given region:

```
(defun map-over-output-record-tree
       (function record &optional (region clim:+everywhere+))
  (declare (dynamic-extent function))
  (labels ((map-internal (rec x-offset y-offset)
             (multiple-value-bind (xoff yoff)
                 (clim:output-record-position rec)
               (clim:translate-coordinates x-offset y-offset xoff yoff)
               (unless (eql rec record)         ;not the first time
                 (funcall function rec))
               (clim:map-over-output-records-overlapping-region
                 #'map-internal rec region
                 (- x-offset) (- y-offset) xoff yoff))))
    (declare (dynamic-extent #'map-internal))
    (multiple-value-bind (x-offset y-offset)
        (clim:convert-from-relative-to-absolute-coordinates
          nil (output-record-parent record))
      (map-internal record x-offset y-offset))))
```

clim:coordinate *Type Specifier*

The type used by CLIM to represent a coordinate. This will be either **t** or a subtype of **real**.

In Symbolics CLIM, the **clim:coordinate** type is currently the same as **real**, but some implementations might use a more restricted type such as **fixnum**.

clim:coordinate *x* *Function*

Converts the real number *x* to a **user::clim-coordinate** object.

clim:copy-area *medium from-x from-y width height to-x to-y* &optional *(op* **boole-1***)* *Generic Function*

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 sheet or a stream, then *from-x* and *from-y* are transformed by the user transformation.)

op is a boolean operation that controls how the source and destination bits are combined. It defaults to **boole-1**, that is, the source bits are copied unchanged into the destination. Other useful values for *op* are **boole-ior**, **boole-xor**, and **boole-clr**.

See the section "Pixmaps in CLIM".

clim:copy-from-pixmap *pixmap pixmap-x pixmap-y width height medium medium-x medium-y* &optional *(op* **boole-1***)* *Function*

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 sheet or a stream, then *medium-x* and *medium-y* are transformed by the user transformation.)

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

op is a boolean operation that controls how the source and destination bits are combined. It defaults to **boole-1**, that is, the source bits are copied unchanged into the destination.

The returned value is the pixmap.

See the section "Pixmaps in CLIM".

clim:copy-textual-output-history *window stream* &optional *region record* *Function*

Given a window *window* that supports output recording, this function finds all of the textual output records that overlap the region *region* (or all of the textual output records is *region* is not supplied), and outputs that text to *stream*. This can be used when you want to capture all of the text on a window into a disk file for later perusal.

clim:copy-to-pixmap *medium medium-x medium-y width height* &optional *pixmap (pixmap-x* **0***) (pixmap-y* **0***) (op* **boole-1***)* *Function*

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 sheet 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 **clim:allocate-pixmap** that has the appropriate characteristics for *medium*.

op is a boolean operation that controls how the source and destination bits are combined. It defaults to **boole-1**, that is, the source bits are copied unchanged into the destination.

The returned value is the pixmap.

See the section "Pixmaps in CLIM".

clim:cursor *Class*

The protocol class that corresponds to a text cursor. If you want to create a new class that obeys the cursor protocol, it must be a subclass of **clim:cursor**.

clim:cursor-active *cursor* *Generic Function*

An active cursor is one that is being actively maintained by its owning sheet. **clim:cursor-active** returns **t** if the cursor is active.

You can use **setf** on this to change the ''active'' attribute of the cursor.

clim:cursor-focus *cursor* *Generic Function*

Returns the ''focus'' attribute of the cursor. When this returns **t**, the sheet owning the cursor has the input focus.

You can use **setf** on this to change the ''focus'' attribute of the cursor.

clim:cursor-position *cursor* *Generic Function*

Returns the cursor position of *cursor* as two values (X and Y), relative to the upper left corner of the sheet with which the cursor is associated.

See the section "Manipulating the Cursor in CLIM".

clim:cursor-set-position *cursor x y* *Generic Function*

Sets the cursor position of *cursor* to *x* and *y*, which are relative to the upper left corner of the sheet with which the cursor is associated.

See the section "Manipulating the Cursor in CLIM".

clim:cursor-sheet *cursor* *Generic Function*

Returns the sheet with which *cursor* is associated.

clim:cursor-state *cursor* *Generic Function*

Returns **t** if the cursor is active and visible on its associated sheet.

You can use **setf** on this to change the visibility of the cursor.

clim:cursor-visibility *cursor* *Generic Function*

This is a convenience function that combines the functionality of both **clim:cursor-active** and **clim:cursor-state**. The visibility can be either **:on** (meaning that the cursor is both active and visible at its current position), **:off** (meaning that the cursor is active, but not visible), or **nil** (meaning that the cursor is not active).

You can use **setf** on this to change the visibility of the cursor.

clim:cursorp *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:cursor**, otherwise returns **nil**.

clim:deactivate-gadget *gadget* *Generic Function*

Causes the 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. The function **clim:note-gadget-deactivated** is called whenever the gadget is made inactive.

clim:deallocate-pixmap *pixmap* *Function*

Deallocates the pixmap *pixmap*.

See the section "Pixmaps in CLIM".

clim-sys:deallocate-resource *name object* &optional *allocation-key* *Function*

Returns the object *object* to the resource named *name*. *name* is a symbol that names a resource. *object* must be an object that was originally allocated from the same resource.

See the section "Resources in CLIM".

clim:default-describe-presentation-type *description stream plural-count* *Function*

Given a string *description* that describes a presentation type (such as ''integer'') and *plural-count* (either **nil** or an integer), this function pluralizes the string if necessary, prepends an indefinite article if appropriate, and outputs the result onto *stream*.

This function is useful when you are writing your own **clim:describe-presentation-type** method, but want to get most of CLIM's default behavior.

clim:default-frame-top-level *frame* &key *:command-parser :command-unparser :partial-command-parser (:prompt* **"Command: "***)* *Function*

The default top-level function for application frames. This function enables the frame (by calling **clim:enable-frame**), and then enters a ''read-eval-print'' loop that calls **clim:read-frame-command**, then calls **clim:execute-frame-command**, and finally redisplays all of the panes that need to be redisplayed.

clim:default-frame-top-level establishes a simple restart for **conditions:abort**, so that anything that invokes an **conditions:abort** restart will by default throw to the top level command loop of the application frame. (Of course, the programmer can specify a restart-case for the **conditions:abort** restart.)

clim:default-frame-top-level binds several of Lisp's standard stream variables. ***standard-output*** is bound to the value returned by **clim:frame-standard-output**.

standard-input is bound to the value returned by **clim:frame-standard-input**. ***query-io*** is bound to the value returned by **clim:frame-query-io**.

:prompt controls the prompt. You can supply either a string or a function of two arguments (*stream* and *frame*) that outputs the prompt on the stream. The default for *:prompt* is the string `"Command: "`.

To set your own prompt string supply **:prompt** to the **:top-level** option of **clim:define-application-frame**:

```
(clim:define-application-frame different-prompt ()
    (...)
  (:top-level (clim:default-frame-top-level
                :prompt "What next, mate? "))
  ...)
```

If you want the prompt to change as a function of the state of the application, you can supply a function (instead of a string):

```
(defun promptfun (stream frame)
  (with-slots (prompt-state) frame
    (format stream "Prompt ~D: " prompt-state)))

(clim:define-application-frame different-prompts ()
    ((prompt-state ...) ...)
  (:top-level (clim:default-frame-top-level
                :prompt promptfun))
  ...)
```

The frame's top level loop binds **clim:*command-parser***, **clim:*command-unparser***, and **clim:*partial-command-parser*** to the values of *:command-parser*, *:command-unparser*, and *:partial-command-parser*.

If there is an interactor pane in the frame, *:command-parser* defaults to **clim:command-line-command-parser**, *:command-unparser* defaults to **clim:command-line-command-unparser**, and *:partial-command-parser* defaults to **clim:command-line-read-remaining-arguments-for-partial-command**.

If there is no interactor pane, *:command-parser* defaults to **clim:menu-command-parser** and *:partial-command-parser* defaults to **clim:menu-read-remaining-arguments-for-partial-command**; there is no need for an unparser when there is no interactor.

See the section "Examples of CLIM Application Frames".

clim:*default-text-style* *Variable*

This is the ''default default'' text style used by all mediums and streams. That is, unless a default text style can be computed another way (such as by querying the display server), CLIM uses **clim:*default-text-style*** as the default text style for mediums and streams.

When doing any kind of text output, if the text style is not fully specified, it is merged against the medium's default text style using **clim:merge-text-styles**. Therefore, if you change the value of **clim:*default-text-style***, the new value must be a fully specified text style.

clim:define-application-frame *name superclasses slots* &rest *options* *Macro*

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*.

For an overview of how to define CLIM application frames, see the section "Defining CLIM Application Frames".

clim:define-application-frame defines a class with the following characteristics:

- inherits some behavior and slots from the class **clim:standard-application-frame**,
- inherits other behavior and slots from any other *superclasses* which you specify explicitly, and
- has other slots, as explicitly specified by *slots*.

None of the arguments is evaluated. The arguments are:

name A symbol naming the new frame and class.

superclasses
A list of superclasses from which the new class should inherit, as in **clos:defclass**. When *superclasses* is **nil**, it behaves as though a superclass of **clim:standard-application-frame** was supplied. If you do specify superclasses to inherit from, you must arrange to inherit from **clim:standard-application-frame** explicitly.

slots A list of slot specifiers, as in **clos: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 You can use the CLOS **:default-initargs** option to customize the initial values of slots in either your specified *superclasses*, or in the application frame. The following options are also supported:

:panes *pane-descriptions*
Specifies the application's panes. There is no default for this option. The syntax of *pane-descriptions* is given below.

:pane *form*
Another way of specifying the application's panes. There is no default for this option. *form* is a Lisp form that creates the panes of the frame. You may only provide one of **:pane** or **:panes**.

:layouts *layout*

Specifies the layout of the panes. The default layout is to lay the panes out in a vertical stack. The syntax of *layout* is given below.

:top-level *top-level*

Allows you to specify the main loop for your application. By default, the top level loop is **clim:default-frame-top-level** (which is adequate for most applications).

Note: if you use a function other than **clim:default-frame-top-level** as the top level function, you should be sure that it calls **clim:enable-frame**.

top-level is a list whose first element is the name of a function to be called to execute the top level loop. The function should take at least one required argument, the frame. The rest of the list consists of additional keyword arguments to be passed to the function. The default function is **clim:default-frame-top-level**. (You can use the **:prompt** keyword of **clim:default-frame-top-level** to control application frame prompts in the interactor.)

If you supply your own function, it must be prepared to receive the keyword arguments **:command-parser**, **:command-unparser**, **:partial-command-parser**, and **:prompt**.

:command-table *name-and-options*

Allows you to specify a particular command table for the application.

name-and-options is a list consisting of the name of the application's command table followed by some keyword-value pairs. The keywords can be **:inherit-from** or **:menu** (which are as the same as in **clim:define-command-table**). The default is to create a command table with the same name as the application.

:command-definer *value*

When *value* is **nil**, no command-defining macro is defined. When it is **t**, a command-defining macro is defined, whose name is of the form **define-***name***-command**. When it is another symbol, the symbol names the command-defining macro. The default is **t**.

:menu-bar *value*

When *value* is **t** (the default), the frame will automatically be provided with a menu bar. The exact format of the menu bar varies from one window system to another. When *value* is **nil**, the frame will not have or use the menu bar. *value* can also name a command table; in this case, the commands to put in the menu bar are gotten from the named command table.

:disabled-commands *commands*
Allows you to specify a particular set of initially disabled *commands* for the application. The default is **nil**.

:default-initargs *alist*
Provides a set of default initargs for the application frame. This is the same as it is for **clos:defclass**.

Syntax of the :panes option

The **:panes** option keyword is followed by one or more *pane-descriptions*. Each *pane-description* can be one of two possible formats:

- A list consisting of a *pane-name* (which is a symbol), a *pane-type*, and *pane-options*, which are keyword-value pairs. *pane-options* is evaluated at load time.
- A list consisting of a *pane-name* (which is a symbol), followed by an expression that is evaluated to create the pane. See the macro **clim:make-clim-stream-pane**. See the function **clim:make-pane**.

The *pane-types* are:

:application
Application panes are stream panes used for the display of application-generated output. See the class **clim:application-pane**. See the macro **clim:make-clim-application-pane**.

:interactor
Interactor panes are stream panes that provide a place for the user to do interactive input and output. See the class **clim:interactor-pane**. See the macro **clim:make-clim-interactor-pane**.

:accept-values
These panes provide for the display of a ''modeless'' **clim:accepting-values** dialog. See the class **clim:accept-values-pane**. See the section "Using an **:accept-values** Pane in a CLIM Application Frame".

:pointer-documentation
These panes provide for pointer documentation. If such a pane is specified, then when the pointer moves over different areas of the frame, this pane displays documentation of the effect of clicking the pointer buttons.

If the host window system has its own way of displaying pointer documentation, this pane may be omitted automatically from the layout.

See the class **clim:pointer-documentation-pane**.

:command-menu
Command menu panes are used to hold a menu of application commands. The default display function is **clim:display-command-menu**

which, by default, displays the current command table of the frame. You can display a different command table by supplying the **:command-table** argument to **clim:display-command-menu**.

Many host window systems provide a menu bar, so having panes of type **:command-menu** is not common.

See the class **clim:command-menu-pane**.

:title Title panes are used for displaying the title of the application. The default title is a ''prettied up'' version of the name of the application frame's class.

Many host window systems will automatically display the frame's title in a title bar, so this is only rarely useful.

See the class **clim:title-pane**.

In addition to *pane-types*, *pane-options* may be specified as keyword/value pairs. Most *pane-options* can be used by all pane types (exceptions are noted as appropriate). The defaults for the options often vary from one pane type to another.

:width, **:height**, **:min-width**, **:min-height**, **:max-width**, and **:max-height**
Provide space requirement specs that specify the sized of the pane. The values the space requirements can take are described in "Using the :LAYOUTS Option to CLIM:DEFINE-APPLICATION-FRAME".

:background and **:foreground** *ink*
Provide initial values for **clim:medium-foreground** and **clim:medium-background** for the pane.

:text-style *text-style*
Specifies a text style to use in the pane. The default depends on the pane type.

:borders Controls whether borders are drawn around CLIM stream panes (**t** or **nil**). The default is **t**. The value may also be a list, in which case the value is used as options to **clim:outlining**.

:spacing Controls whether there is some whitespace between the border and the viewport for a CLIM stream pane (**t** or **nil**). The default is **t**. The value may also be a list, in which case the value is used as options to **clim:spacing**.

:scroll-bars *scroll-bar-spec*
A *scroll-bar-spec* can be **:both** (the default for **:application** panes), **:horizontal**, **:vertical**, **:none**, or **nil**. The pane will have only those scroll bars which were specified. **:none** means that the pane will support scrolling, but does not have any visible scroll bars. **nil** means that the pane will not support scrolling at all.

:display-after-commands
One of **t**, **nil**, or **:no-clear**. 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 managing the display after commands.

:no-clear behaves the same as **t**, with the following change. If you have not specified **:incremental-redisplay t**, then the pane is normally cleared before the display function is called. However, if you specify **:display-after-commands :no-clear**, then the pane is not cleared before the display function is called.

:display-function *display-spec*
Where *display-spec* is either the name of a function or a list whose first element is the name of a function. The function is to be applied to the application frame, the pane, and the rest of *display-spec* if it was a list when the pane is to be redisplayed.

The function must accept two required arguments (the frame and the pane), plus the two keyword arguments **:max-width** and **:max-height**.

One example of a predefined display function is **clim:display-command-menu**.

:display-string *string*
(for **:title** panes only) The string to display. The default is the frame's pretty name.

:label *string*
A string to be used as a label for the pane, or **nil** (the default).

:incremental-redisplay *boolean*
If **t**, CLIM runs the display function inside of an **clim:updating-output** form. The default is **nil**.

:end-of-line-action, **:end-of-page-action**
Initial values of the corresponding attributes. See the macro **clim:with-end-of-line-action** and see the macro **clim:with-end-of-page-action**.

:initial-cursor-visibility
:off means make the text cursor visible if the window is waiting for input. **:on** means make it visible all the time. **nil** means that the cursor is never visible. The default is **:off** for **:interactor** and **:accept-values** panes, and **nil** for other panes.

:output-record
Specify this if you want a different output history mechanism than the default (which is **clim:standard-tree-output-history**). For example, a graphic editing program might supply a value of:

```
(make-instance 'clim:r-tree-output-history)
```

Besides **clim:standard-tree-output-history** and **clim:r-tree-output-history**, you can also use **clim:standard-sequence-output-history**.

:draw-p, **:record-p** *boolean*
Specifies the initial state of drawing and output recording.

:default-view *view*
Specifies the view object to use for the stream's default view.

:text-margin *integer*
Text margin to use if **clim:stream-text-margin** isn't set. This defaults to the width of the viewport.

:vertical-spacing *integer*
Amount of extra space between text lines.

:pointer-cursor
Specifies the pointer cursor to use when the pointer is over this pane.

:event-queue
Specifies the event queue to be used by this pane. The default is to share the event queue with the top-level sheet, so that all the panes in the frame use the same event queue.

Syntax of the :layouts option

Here is a brief description of the syntax of the **:layouts** option:

:layouts (*layout-name layout-panes*)

layout-name is a symbol.

layout-panes is *layout-panes1* or (*size-spec layout-panes1*).

layout-panes1 is a *pane-name*, or a *layout-macro-form*, or *layout-code*.

layout-code is Lisp code that generates a pane, which may include the name of a named pane.

size-spec is a positive real number less than 1, or **:fill**, or **:compute**. A real number (between zero and one, exclusive) is the fraction of the available space to use. **:fill** means that the pane will take as much space as remains when all its sibling panes have been given space. **:compute** means that the pane's display function should be called in order to compute how much space it requires. (Note that the display function is run at frame-creation time, so it must be able to compute the size correctly at that time.)

size-spec can also be an integer indicating the size of the pane in device units, or a list whose first element is a real number and whose second element is a ''unit'' (one of **:line**, **:character**, **:mm**, **:point**, or **:pixel**).

layout-macro-form is (*layout-macro-name* (*options*) &rest *layout-panes*).

layout-macro-name is **clim:vertically**, **clim:horizontally**, **clim:tabling**, **clim:outlining**, **clim:spacing**, or **clim:labelling**.

For a detailed explanation of the **:layouts** option see the section "Using the :LAYOUTS Option to CLIM:DEFINE-APPLICATION-FRAME".

See the section "Examples of CLIM Application Frames".

clim:define-border-type *shape arglist* &body *body* *Macro*

Defines a new kind of border named *shape* for use by **clim:surrounding-output-with-border**. *arglist* will typically be `(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.

After the border has been drawn, **clim:surrounding-output-with-border** positions the text cursor at the end of the bordered output. Since the size of a border may vary greatly, CLIM needs a hint as to where the text cursor should be placed relative to the bordered output. If the returned value of *body* is a small real number, it is used as the hint that controls the Y offset of the final text cursor.

The predefined border types, **:rectangle**, **:oval**, **:drop-shadow**, and **:underline** are defined using this macro. Here is how the rectangular border type is defined:

```
(define-border-type :rectangle
                    (stream left top right bottom
                     &rest drawing-options &key (filled nil) &allow-other-keys)
  (declare (dynamic-extent drawing-options))
  (let ((offset 2))
    (apply #'draw-rectangle* stream
           (- left offset) (- top offset)
           (+ right offset) (+ bottom offset)
           :filled filled drawing-options))
  ;; Y offset for text cursor is 3
  3)
```

clim:define-command *name arguments* &body *body* *Macro*

Defines a command and its characteristics, including its name, its arguments, and optionally the command table in which it should appear, its keystroke accelerator, its command-line name, and whether or not (and how) to add this command to the menu associated with the command table. For example, the follow defines a com-

mand that has a command-line name, appears in a command menu, and has a keystroke accelerator.

```
(clim:define-command (com-my-favorite-command
                      :name "my fave"
                      :keystroke #\F
                      :menu "favorite"
                      :command-table my-command-table)
   ((arg1 '(or integer string)
          :default "none"
          :display-default t))
 body)
```

clim:define-command is the most basic command-defining form. Usually, the programmer will not use **clim:define-command** directly, but will instead use a **define-***application*-**command** form that is automatically generated by **clim:define-application-frame**. **define-***application*-**command** adds the command to the application's command table. By default, **clim:define-command** does not add the command to any command table.

clim: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 arguments the arguments to the command as specified in the **clim:define-command** form, as required and keyword arguments.

The second function defined by **clim:define-command** is used internally by CLIM and implements the code responsible for parsing and returning the command's arguments.

name Either a command name, or a cons of the command name and a list of keyword-value pairs. The keyword-value pairs in *name* can be:

:command-table *command-table-name*
: Specifies that the command should be added to a command table. *command-table-name* either names a command table to which the command should be added, or is **nil** (the default) to indicate that the command should not be added to any command table. This keyword is only accepted by **clim:define-command**, not by **define-***application*-**command** functions.

:name *string*
: Provides a name to be used as the command-line name for the command for keyboard interactions in the command table specified by the **:command-table** option. *string* is a string to be used; or **nil** (the default) meaning that the command will not be available via command-line interactions; or **t**, which means the command-line name will be generated automatically. See the function **clim:add-command-to-command-table**.

:menu *menu-item*
: Specifies that this command will be 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-item* is a string, then that string will be used as the menu name. If *menu-item* is **t**, then the menu name will be generated automatically. See the function **clim:add-command-to-command-table**. Otherwise, *menu-item* is 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** and **:documentation**, which are interpreted as for **clim:add-menu-item-to-command-table**.

:keystroke *gesture-spec*

Specifies a gesture to be used as a keystroke accelerator in the command table specified by the **:command-table** option. For applications with interactor panes, these gesture should correspond to non-printing characters such as `#\control-D` (whose gesture spec is `(:D :control)`). The default is **nil**, meaning that there is no keystroke accelerator.

The **:name**, **:menu**, and **:keystroke** options are allowed only if the **:command-table** option was supplied explicitly or implicitly, as in **define-***application***-command**.

If the command takes any non-keyword arguments and you have supplied either **:menu** or **:keystroke**, then when you select this command via a command menu or keystroke accelerator, a partial command parser will be invoked in order to read the unsupplied arguments; the defaults will not be filled in automatically. If this behavior is not desired, then you must call **clim:add-menu-item-to-command-table** or **clim:add-keystroke-to-command-table** yourself and fully specify the command. For example, use the following instead of supplying **:keystroke** for the **com-next-frame** command:

```
(define-debugger-command (com-next-frame :name t)
    ((nframes 'integer
      :default 1
      :prompt "number of frames"))
 (next-frame :nframes nframes))

(clim:add-keystroke-to-command-table
  'debugger '(:n :control) :command '(com-next-frame 1))
```

arguments

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 list containing a parameter variable, followed by a presentation type specifier, followed by keyword-value pairs. The keywords can be:

:default *value*
Provides a *value* which is the default that should be used for the argument, as for **clim:accept**.

:default-type *type*
The same as for **clim:accept**. If **:default** is supplied, then the **:default** and the **:default-type** are returned if the input is empty.

:mentioned-default *value*
Provides a *value* which 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 allowed only for keyword arguments. This is most often used for boolean keyword arguments in conjunction with **:default**

```
(delete-after-printer 'boolean
  :default nil :mentioned-default t
  :prompt "yes or no")
```

:display-default *boolean*
The same as for **clim:accept**. When true, displays the default if one was supplied. When **nil**, the default is not displayed.

:prompt *string*
Provides a *string* which is a prompt to print out during command-line parsing, as for **clim:accept**.

:documentation *string*
Provides a documentation string that describes what the argument is. When you type `Help` when entering keyword arguments, this documentation will be displayed.

:when *form*
Provides a *form* that indicates whether this keyword argument is available. The *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 allowed only on keyword arguments, and *form* cannot use the values of other keyword arguments.

:gesture *pointer-gesture-name*
Provides a gesture name that will be used for a translator that translates from the argument to a command. **:gesture** is allowed only when the **:command-table** option was supplied to the command-defining form. The default is that no translator will be created. Explicitly supplying **:gesture nil** creates a translator that will appear only in the translator menus.

pointer-gesture-name can also be a list whose **car** is a pointer gesture name, and whose **cdr** is a list of translator options

that may include **:tester**, **:menu**, **:priority**, **:echo**, **:documentation**, and **:pointer-documentation**.

body Provides the body of the command. It has lexical access to all of the command's arguments. If the body of the command needs access to the application frame, it should use **clim:*application-frame***. The returned values of body are ignored.

clim:define-command arranges 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, and everything else is evaluated at runtime when argument parsing reaches that argument, except that the value for **:when** is evaluated when parsing reaches the keyword arguments, and **:gesture** is not evaluated at all.

See the section "Commands in CLIM".

clim:define-command-table *name* &key *:inherit-from :menu :inherit-menu* *Macro*

Defines a command table whose name is the symbol *name*. The keyword arguments are:

:inherit-from
A list of either command tables or command table names. The new command table inherits from all of the command tables specified by *:inherit-from*. The inheritance is done by union with shadowing. In addition to inheriting from the explicitly specified command tables, every command table defined with **clim:define-command-table** also inherits from CLIM's system command table. (This command table, **clim:global-command-table**, contains such things as the ''menu'' translator that is associated with the right-hand button on pointers.)

:menu Specifies 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 *documentation keystroke*), where *string*, *type*, *value*, *documentation*, and *keystroke* are as for **clim:add-menu-item-to-command-table**.

:inherit-menu
Normally, a menu does not inherit any menu items or keystroke accelerators from its parents. When *:inherit-menu* is **t**, the menu items and keystroke accelerators will be inherited. When it is **:menu**, only the menu items will be inherited. When it is **:keystrokes**, only the keystroke accelerators will be inherited.

If the command table named by *name* already exists, **clim:define-command-table** will modify the existing command table to have the new value for *:inherit-from* and *:menu*, but will otherwise leave the other attributes for the existing table alone.

None of the arguments to **clim:define-command-table** arguments is evaluated.

See the section "CLIM Command Tables".

clim:define-default-presentation-method *presentation-function-name [qualifiers]* specialized-lambda-list* &body *body* *Macro*

This is like **clim:define-presentation-method**, except that it is used to define a default method that will be used if there are no more specific methods.

None of the arguments is evaluated.

clim:define-drag-and-drop-translator *name (from-type to-type destination-type command-table* &key *(:gesture* **':select***) :tester :documentation (:menu* **t***) :priority :feedback :highlighting :pointer-cursor) arglist* &body *body* *Macro*

Defines a ''drag and drop'' (or ''direct manipulation'') translator named *name* that translates from objects of type *from-type* to objects of type *to-type* when a ''from presentation'' is ''picked up'', ''dragged'' over, and ''dropped'' on a ''to presentation'' having type *destination-type*. *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. See the section "Presentation Translators in CLIM".

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.

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 **clim:frame-drag-and-drop-feedback**, whose default method simply draws the bounding rectangle of the object being dragged.

When the ''from presentation'' is dragged over any other presentation that has an applicable 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 **clim:frame-drag-and-drop-highlighting**, whose default method simply draws a box around the object over which the dragged object may be dropped.

The other arguments to **clim:define-drag-and-drop-translator** are the same as for **clim:define-presentation-translator**. See the macro **clim:define-presentation-translator**.

It is possible for there to be more than one drag and drop translator that applies to the same from type, to type, destination type, and gesture. In this case, the exact translator that is chosen for use during the dragging phase is unpredictable. If these translators have different feedback, highlighting, documentation, or pointer documentation, the feedback and highlighting behavior is unpredictable.

For example, suppose you are implementing some sort of desktop interface to a file system editor that has commands such as ''Hardcopy File'', ''Delete File'', and so forth, and you want a drag-and-drop interface. Assuming you have some icons that represent a hardcopy device, a trashcan, and so forth, and presentation types that correspond to those icons, you could do the following:

```
(clim:define-drag-and-drop-translator dm-hardcopy-file
    (pathname command printer fsedit-comtab
     :documentation "Hardcopy this file")
    (object destination-object)
  `(com-hardcopy-file ,object ,destination-object))

(clim:define-drag-and-drop-translator dm-delete-file
    (pathname command trashcan fsedit-comtab
     :documentation "Delete this file")
    (object)
  `(com-delete-file ,object))

(clim:define-drag-and-drop-translator dm-copy-file
    (pathname command folder fsedit-comtab
     :documentation "Copy this file")
    (object destination-object)
  `(com-copy-file ,object
                  ,(make-pathname :name (pathname-name object)
                                  :type (pathname-type object)
                                  :defaults destination-object)))
```

clim:define-genera-application *frame-name* &rest *keys* &key *:pretty-name :select-key :left :top :right :bottom :width :height* *Macro*

Makes a CLIM application available to the `Select Activity` command and optionally to the `SELECT` key. *frame-name* is the symbol used as the *name* argument to **clim:define-application-frame**.

Note: clim:define-genera-application exists only under Genera. Therefore, you should put #+**Genera** in front of any calls to it.

frame-name
: A symbol which is the *name* argument to **clim:define-application-frame**.

:pretty-name
: A string which is the name used as a title and as the activity name. It defaults to a prettified version of *frame-name*.

:select-key A character to be used with the `SELECT` key. It defaults to **nil** which means that no `SELECT` character is assigned.

:left, *:top*, *:right*, *:bottom*
: The coordinates of the frame. *:left* and *:top* default to 0, and *:right* and *:bottom* default to **nil**.

:height, *:width*
: The size of the frame. *:height* and *:width* default to **clim:+fill+**, meaning the frame will fill the entire screen.

None of the arguments is evaluated.

clim:define-gesture-name *name type gesture-spec* &key *(:unique* **t***)* *Macro*

Defines a gesture named *name* by calling **clim:add-gesture-name**.

For more information, see **clim:add-gesture-name**.

None of the arguments is evaluated.

CLIM's standard pointer gestures on a platform with a 3-button mouse (such as Genera) could be defined with the following forms:

```
(clim:define-gesture-name :select   :pointer (:left))
(clim:define-gesture-name :describe :pointer (:middle))
(clim:define-gesture-name :menu     :pointer (:right))
(clim:define-gesture-name :delete   :pointer (:middle :shift))
(clim:define-gesture-name :edit     :pointer (:left :meta))
(clim:define-gesture-name :modify   :pointer (:right :control :meta))
```

clim:define-presentation-action *name (from-type to-type command-table* &key *(:gesture* **':select***) :tester :documentation :pointer-documentation (:menu* **t***) :priority) arglist* &body *body* *Macro*

This is similar to **clim:define-presentation-translator**, except that the body of the action is not intended to return a value, but should instead side-effect some sort of application state. A typical presentation action might scroll a window in an application, or select another translator from a menu.

name The name of the presentation action.

from-type The presentation type of the presentation on a window. Presentation type options are not allowed in *from-type*.

to-type The presentation type of the current input context. Presentation type options are not allowed in *to-type*. When *to-type* is **nil**, this action is applicable in all input contexts.

command-table
This specifies which command table the translators should be stored in. It should be either a command table or the name of a command table. This translator will be applicable only when this command table is one of the command tables from which the current application frame's command table inherits.

The other arguments to **clim:define-presentation-action** are the same as for **clim:define-presentation-translator**. For information on the arguments, see the macro **clim:define-presentation-translator**.

None of the arguments to **clim:define-presentation-action** is evaluated.

Note that an action does not satisfy requests for input as translators do. An ordinary translator satisfies a request for input, but an action is something that happens while waiting for input. After executing an action, the program continues to wait for the same input it was waiting for prior to executing the action.

In general, if you are using **clim:define-presentation-action** to execute any kind of an application command, you should be using **clim:define-presentation-translator** or **clim:define-presentation-to-command-translator** instead.

From time to time, it is appropriate to write application-specific presentation actions. The key test for whether something should be an action is that it makes sense for the action to take place while the user is entering a command sentence and performing the action will not interfere with the input of the command sentence. For example, an application framework might have an action that changes what information is displayed in one of its panes. It makes sense to do this in the middle of entering a command because information displayed in that pane might be used in formulating the arguments to the command. This needn't interfere with the input of the command since a pane can be redisplayed without discarding the pending partial command. It is for these cases that the presentation action mechanism is provided. A simple rule of thumb is that actions may be used to alter how application objects are presented or displayed, but anything having to do with modification of application objects should be embodied in a command, with an appropriate set of translators.

clim:define-presentation-generic-function *generic-function-name presentation-function-name lambda-list* &rest *options* *Macro*

Defines a new presentation named *presentation-function-name* whose methods are named by *generic-function-name*. *lambda-list* and *options* are as for **clos:defgeneric**.

The first few arguments in *lambda-list* are treated specially. The first argument must be either **type-key** or **type-class**. If you wish to be able to access type parameters or options in the method, the next arguments must be either or both of **parameters** and **options**. Finally, a required argument called **type** must also be included in *lambda-list*.

Most user programs will never need to define their own presentation type generic function. CLIM uses it internally. For example, **clim:describe-presentation-type** might be have been defined by the following:

```
(clim:define-presentation-generic-function
  describe-presentation-type-method clim:describe-presentation-type
  (type-key parameters options type stream plural-count))
```

None of the arguments is evaluated.

clim:define-presentation-method *presentation-function-name [qualifiers]* specialized-lambda-list* &body *body* *Macro*

Defines a presentation method for the function named *presentation-function-name* on the presentation type named in *specialized-lambda-list*. The value of *presentation-function-name* can be any of the presentation generic functions defined by CLIM (**clim:accept**, **clim:present**, **clim:describe-presentation-type**, **clim:presentation-typep**, **clim:presentation-subtypep**, **clim:accept-present-default**, **clim:presentation-type-specifier-p**, **clim:presentation-refined-position-test**, or **clim:highlight-presentation**) or any presentation generic function you have defined yourself.

specialized-lambda-list is a CLOS specialized lambda list for the method, and its contents varies depending on what *presentation-function-name* is. *qualifier** is zero or more of the usual CLOS method qualifiers. *body* defines the body of the method.

None of the arguments is evaluated.

For example, the following might be used to implement the **clim:accept** and **clim:present** methods for the **null** type:

```
(clim:define-presentation-method clim:present
    (object (type null) stream (view clim:textual-view) &key)
  (declare (ignore object))
  (write-string "None" stream))

(clim:define-presentation-method clim:accept
    ((type null) stream (view clim:textual-view) &key)
  (let ((token (clim:read-token stream)))
    (unless (string-equal token "None")
      (clim:input-not-of-required-type token type))
    nil))
```

For more information about presentation methods, see the section "Presentation Methods in CLIM".

clim:define-presentation-to-command-translator *name (from-type command-name command-table* &key *(:gesture* **':select***) :tester :documentation :pointer-documentation (:menu* **t***) :priority (:echo* **t***)) arglist* &body *body* *Macro*

Defines a presentation translator that translates a displayed presentation into a command.

This is similar to **clim: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. See the section "Presentation Translators in CLIM".

The *:echo* argument controls whether or not the command should be echoed in the command line when a user invokes this translator. The default for *:echo* is **t**.

The other arguments to **clim:define-presentation-to-command-translator** are the same as for **clim:define-presentation-translator**. For information on the arguments, see the macro **clim:define-presentation-translator**.

The body of the translator should return 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 eventually passed to **clim:execute-frame-command**.

For example, the following translators can be found in the CLIM Lisp Listener:

```
(clim:define-presentation-to-command-translator show-file-translator
    (pathname com-show-file lisp-listener
     :gesture :select
     :pointer-documentation "Show File")
    (object)
  (list object))

(clim:define-presentation-to-command-translator edit-file-translator
    (pathname com-edit-file lisp-listener
     :gesture :edit
     :pointer-documentation "Edit File")
    (object)
  (list object))
```

None of the arguments to **clim:define-presentation-to-command-translator** is evaluated.

clim:define-presentation-translator *name (from-type to-type command-table* &key *(:gesture* **':select***) :tester :tester-definitive :documentation :pointer-documentation (:menu* **t***) :priority) arglist* &body *body* *Macro*

Defines a presentation translator named *name* which 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 presentation type options. *From-type* and *to-type* may also be presentation type abbreviations. *to-type* can also be **nil** , in which case the translator applies in any input context since **nil** is a subtype of all presentation types. See the section "Presentation Translators in CLIM".

None of the arguments to **clim:define-presentation-translator** is evaluated. The arguments are described as follows:

name The name of the presentation translator.

from-type The presentation type of the presentation on a window. Presentation type options are not allowed in *from-type*. When *from-type* is **t**, this translator is applicable to all presentation types.

to-type The presentation type of the returned object. Presentation type options are not allowed in *to-type*. When *to-type* is **nil**, this translator is applicable in all input contexts.

command-table
This specifies which command table the translators should be stored in. It should be either a command table or the name of a command table. This translator will be applicable only when this command table is one of the command tables from which the current application frame's command table inherits.

:gesture A *gesture-name* (see the section "Gestures and Gesture Names in CLIM"). The body of the translator will be run only if the translator is applicable and the pointer event corresponding to the user's gesture matches the gesture name in the translator. For more information, see the section "Applicability of CLIM Presentation Translators". *:gesture* defaults to **:select**.

:gesture t means that any user gestures will match this translator, and **:gesture nil**, means that no user gesture will match this translator. **:gesture nil** is commonly used when the translator should appear only in a menu.

:tester Either a function or a list of the form (*tester-arglist* . *tester-body*), where *tester-arglist* takes the same form as *arglist* (see below), and *tester-body* is the body of the tester. The tester should 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 may be run in order to definitively decide if the translator is applicable (for more information, see the section "Applicability of CLIM Presentation Translators"). If no tester is supplied, CLIM arranges for a tester that always returns **t**.

Use this when you want to restrict the cases when a translator will be applicable.

:tester-definitive
When this is **t** and the tester returns **t**, this translator is definitely applicable. When this is **nil** and the tester returns **t**, this translator might be applicable; in order to find out for sure, the body of the translator is run, and, if it returns an object that matches the input context type (using **clim:presentation-typep**), this translator is applicable.

:documentation
An object that will be used to document the translator. For exam-

ple, in menus: if the object is a string, the string itself will be used as the documentation. Otherwise, it should be either a function or a list of the form (*doc-arglist* . *doc-body*), where *doc-arglist* takes the same form as *arglist*, but includes a *stream* argument as well (see below), and *doc-body* is the body of the documentation function. The body of the documentation function should write the documentation to *stream*. The default is **nil**, meaning that there is no documentation.

:pointer-documentation

Like the **:documentation** option except that *:pointer-documentation* is used in the pointer documentation line. This documentation is usually more succinct than normal documentation. If *:pointer-documentation* is not supplied, it defaults to *:documentation*.

:menu

The value should be **t** or **nil**. The default is **t**, meaning the translator is to be included in the menu popped-up by the **:menu** gesture (click Right on a three-button mouse). Use **:menu t :gesture nil** to make the translator accessible only through the menu. **:menu nil** means that the translator should not appear in the menu.

:priority A non-negative integer that represents the priority of the translator. The default is 0. When there are several translators that match for the same gesture, the one with the highest priority is chosen.

The priority is broken into a ''high order'' part and a ''low order'' part. The high order part of the priority is the ''tens'' place (base 10), and the low order part is the ''ones'' place.

You can create an ''overriding'' translator that always precedes any other applicable translators by supplying a high order priority greater than the high order priority of other translators. You can ''break ties'' between translators that translate from the same type by supplying a low order priority greater than the low order priority of other translators. Since the high order priority always overrides all other applicable translators, you should be careful about supplying priorities that have a high order part.

arglist, *tester-arglist*, *doc-arglist*

An argument list that must be a subset (using **string-equal** to compare symbol names) of the ''canonical'' argument list:

(*object presentation context-type frame event window x y*)

In the body of the translator (or the tester), *object* will be bound to the presentation's object, *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 **clim:*application-frame***), *event* will be bound to the object representing the gesture 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* where the pointer was when the user issued the gesture. The special variable **clim:*input-context*** will be bound to the current input context.

body The body of the translator, and may return one, two, or three values. The first returned value is an object that must be **clim:presentation-typep** of *to-type*. The second value is either **nil** or a presentation type that must be **clim:presentation-subtypep** of *to-type*.

The third returned value is either **nil** or a list of options (as keyword-value pairs) that will be interpreted by **clim:accept**. The only option currently used by **clim:accept** is **:echo**. If *:echo* is **t** (the default), the object returned by the translator will be ''echoed'' by inserting its textual representing into the input buffer. If *:echo* is **nil**, the object will not be echoed.

body is run in the context of the application. The first two values returned by *body* are used, in effect, as the returned values for the call to **clim:accept** that established the matching input context.

clim:define-presentation-type *name parameters* &key *:options :inherit-from :description :history :parameters-are-types* *Macro*

Defines a CLIM presentation type. See the section "Defining a New Presentation Type in CLIM".

name The name of the presentation type. *name* is a symbol or a class object.

parameters
Parameters of the presentation type. These parameters are lexically visible within the **:inherit-from** form and within the methods created with **clim:define-presentation-method**. For example, the parameters are used by **clim:presentation-typep** to refine its tests for type inclusion.

:options A list of option specifiers, which defaults to **nil**. An option specifier is either a symbol or a list (*symbol* &optional *default supplied-p presentation-type accept-options*). The elements *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 the **:inherit-from** form and within most of the methods created with **clim: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.

:inherit-from

A form that evaluates to a presentation type specifier for another type from which the new type inherits. *:inherit-from* can access the parameter variables bound by the *parameters* lambda list and the option variables specified by *options*. If *name* is or names a CLOS 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, it defaults as follows: 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 inherits from **clos:standard-object**.

Note: you cannot use **clim:define-presentation-type** to create a new subclass any of the built-in types, such as **integer** or **symbol**.

:history Specifies what history to use for the presentation type.

nil	(the default) Uses no history.
t	Uses its own history.
type-name	Uses *type-name*'s history.

If you want more flexibility, you can define a **clim:presentation-type-history** presentation method.

:description

A string or **nil**. If **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 **clim:describe-presentation-type** presentation method.

Unsupplied optional or keyword parameters default to * (as they do in **deftype**) if no default is specified in *parameters*. Unsupplied options default to **nil** if no default is specified in *:options*.

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.

clim:define-presentation-type-abbreviation *name parameters expansion* &key *:options* *Macro*

Defines a presentation type that is an abbreviation for the presentation type specifier that is the value of *expansion*. Note that you cannot define any presentation methods on a presentation type abbreviation. If you need to define methods, use **clim:define-presentation-type** instead.

name must be a symbol and must not be the name of a CLOS class. *parameter*s and *:options* are the same as they are for **clim:define-presentation-type**.

The type-specifier produced by evaluating *expansion* can be a real presentation type or another abbreviation.

This example defines a presentation type to read an octal integer:

```
(clim:define-presentation-type-abbreviation octal-integer
                                            (&optional low high)
  `((integer ,low ,high) :base 8 :description "octal integer"))
```

clim-sys:defresource *name parameters* &key *:constructor :initializer :deinitializer :matcher :initial-copies* *Macro*

Defines a resource named *name*; *name* 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 is responsible for creating an object, and 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 that is 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 that is 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**, which is the object to be deinitialized. It is called whenever an object is deallocated back to the resource, but is not called by **clim-sys: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. In addition, 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 should return **t** if there is a match, otherwise it should return **nil**.

:initial-copies is used to specify the number of objects that should be initially put into the resource. It must be an integer or **nil** (which is 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; in this case, the initial copies have the default values of the parameters.

The following example defines a resource of strings that can be used to avoid constantly allocating and garbage collecting strings:

```
(clim-sys:defresource temporary-string
                      (&key (length 100) (adjustable t))
  :constructor
    (make-array length
                :element-type 'character
                :fill-pointer 0
                :adjustable adjustable)
  :matcher
    (and (eq adjustable (adjustable-array-p temporary-string))
         (or (and (not adjustable)
                  (= length (array-dimension temporary-string 0)))
             (<= length (array-dimension temporary-string 0))))
  :initializer (setf (fill-pointer temporary-string) 0))

(defmacro with-temporary-string
          ((var &key (length 100) (adjustable t)) &body body)
  `(clim-sys:using-resource (,var temporary-string
                             :length ,length :adjustable ,adjustable)
     ,@body))
```

See the section "Resources in CLIM".

clim:delete-gesture-name *gesture-name* *Function*

Removes the gesture named *gesture-name*.

clim:delete-output-record *child record* &optional *errorp* *Generic Function*

Removes the child output record *child* from the output record *record*. If *child* is not contained in *record* and *errorp* is **t**, an error is signalled.

Any class that is a subclass of **clim:output-record** must implement this method.

See the section "Concepts of CLIM Output Recording".

clim:delimiter-gesture-p *gesture* *Function*

Returns **t** if *gesture* is a currently active delimiter gesture.

clim:*delimiter-gestures* *Variable*

A list containing the gesture names of the currently active delimiter gestures.

clim:describe-presentation-type *presentation-type* &optional *(stream* ***standard-output****) (plural-count* **1***)* *Function*

Describes the *presentation-type* on the *stream*.

If *stream* is **nil**, a string containing the description is returned. *plural-count* is either **nil** (meaning that 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 *presentation-type* can be a presentation type abbreviation.

clim:describe-presentation-type *type-key parameters options type stream plural-count* *Clim Presentation Method*

This presentation method is responsible for textually describing the type *type*. *stream* will be a stream of some sort, never **nil**. *plural-count* is as for the **clim:describe-presentation-type** function.

For example, CLIM's **complex** number is described with the following methods. It is written as an **:after** method because CLIM provides a default method that does most of the work.

```
(clim:define-presentation-method clim:describe-presentation-type :after
    ((type complex) stream plural-count)
  (declare (ignore type plural-count))
  (unless (eq type '*)
    (format stream " whose components are ")
    (clim:describe-presentation-type type stream t)))
```

clim:design *Class*

A design is an object that represents a way of arranging colors and opacities in the drawing plane. **clim:design** is a generalization of **clim:region** to include color.

clim:destroy-frame *frame* *Generic Function*

Disables the application frame *frame*, and then destroys it by deallocating all of its CLIM resources and disowning it from its frame manager. After the frame has been destroyed, its state will be **:disowned**.

clim:destroy-port *port* *Generic Function*

Destroys the connection to the display server represented by *port*. All of the application frames, frame managers, and sheets associated with *port* will be destroyed. All server resources used by the frames and sheets (such as graphics contexts) are released as part of the shutdown.

clim-sys:destroy-process *process* *Function*

Terminates the process *process*. *process* is a process object, such as is returned by **clim-sys:make-process**.

clim:device-event *Class*

The superclass of all other CLIM device events. This is a subclass of **clim:event**.

clim-sys:disable-process *process* *Function*

Disables the process *process*, that is, prevents it from becoming runnable until it is enabled again.

clim:disarmed-callback *gadget client id* *Generic Function*

This callback is invoked when the gadget *gadget* is disarmed. The exact definition of disarming varies from gadget to gadget, but typically a gadget becomes disarmed when the pointer is moved out of its region.

The default method for **clim:disarmed-callback** (on **clim:basic-gadget**) calls the function specified by the **:disarmed-callback** initarg.

clim:display-command-menu *frame stream* &key *:command-table :max-width :max-height :n-rows :n-columns (:cell-align-x* **':left***) (:cell-align-y* **':top***)* *Function*

Displays the menu described by the command table associated with the application frame *frame* onto *stream*. This is generally used as the display function for application panes of type **:command-menu**.

:command-table is the command table to display; it defaults to *frame*'s current command table. The following options are used to control the appearance of the command menu.

:max-width
: Specifies the maximum width, in device units, of the table display.

:max-height
: Specifies the maximum height, in device units, of the table display.

:n-rows
: Specifies the number of rows of the table. Specifying this overrides *:max-width*.

:n-columns
: Specifies the number of columns of the table. Specifying this overrides *:max-height*.

:cell-align-x
: Specifies the horizontal placement of each of the cells in the command menu. This is like the **:align-x** option to **clim:formatting-cell**.

:cell-align-y
Specifies the horizontal placement of each of the cells in the command menu. This is like the **:align-y** option to **clim:formatting-cell**.

clim:display-command-menu displays the disabled command menu items as well as the enabled ones, but the disabled items will be ''grayed out''.

clim:display-command-table-menu *command-table stream* &key *:max-width :max-height :n-rows :n-columns :x-spacing :y-spacing (:cell-align-x* **':left***) (:cell-align-y* **':top***) (:initial-spacing* **t***) :row-wise :move-cursor* *Function*

Displays the menu for *command-table* on *stream*. The following options are used to control the appearance of the command menu.

:max-width
Specifies the maximum width, in device units, of the table display.

:max-height
Specifies the maximum height, in device units, of the table display.

:n-rows
Specifies the number of rows of the table. Specifying this overrides *:max-width*.

:n-columns
Specifies the number of columns of the table. Specifying this overrides *:max-height*.

:x-spacing
Determines the amount of space inserted between columns of the table; the default is the width of a space character. *:x-spacing* can be specified in one of the following ways:

Integer
A size in the current units to be used for spacing.

String or character
The spacing is the width or height of the string or character in the current text style.

Function
The spacing is the amount of horizontal or vertical space the function would consume when called on the stream.

List of form (*number unit*)
The *unit* is **:point**, **:pixel**, or **:character**.

: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 *:x-spacing* option.

:cell-align-x
Specifies the horizontal placement of each of the cells in the command menu. This is like the **:align-x** option to **clim:formatting-cell**.

:cell-align-y
Specifies the horizontal placement of each of the cells in the command menu. This is like the **:align-y** option to **clim:formatting-cell**.

:initial-spacing
When doing the layout, CLIM tries to evenly space items across the entire width of the stream. When this option is **t**, no whitespace is inserted before the first item on a line.

:row-wise When this is **nil**, if there are multiple columns in the item list, the entries in the item list are arranged in a manner similar to entries in a phone book. Otherwise the entries are arranged in a ''row-wise'' fashion. The default is **t**.

:move-cursor
When **t** (the default), CLIM moves the text cursor to the end (lower right corner) of the output. Otherwise, the cursor is left at the beginning (upper left corner) of the output.

clim:display-exit-boxes *frame stream view* *Generic Function*

Displays the exit boxes for the **clim:accepting-values** frame *frame* on the stream *stream* using the view *view*. The exit boxes specification is not passed in directly, but is a slot in the frame.

CLIM has default methods that either writes a line of text associating the Exit and Abort strings with presentations that either exit or abort from the dialog (in the textual view), or create push buttons that contain the Exit and Abort strings.

You can create your own subclass of the **clim:accepting-values** frame class, and then specialize **clim:display-exit-boxes** for you own kind of exit boxes. In this case, you will need to provide the **:frame-class** option to **clim:accepting-values**. It is often sufficient to simply provide the **:exit-boxes** option to **clim:accepting-values**.

clim:displayed-output-record *Class*

The protocol class that is used to indicate that an object is a *displayed output record*, that is, a CLIM object that represents a visible piece of output on an output device. If you want to create a new class the obeys the displayed output record protocol, it must be a subclass of **clim:displayed-output-record**.

If you think of output records being arranged in a tree, displayed output records are the leaves of the tree. Displayed text and graphics are examples of things that are displayed output records.

See the section "Output Recording in CLIM".

clim:displayed-output-record-p *object* *Function*

Returns **t** if and only *object* is of type **clim:displayed-output-record**.

clim:do-command-table-inheritance *(command-table-var command-table)* &body *body* *Macro*

Successively evaluates *body* with *command-table-var* bound first to the command table *command-table*, and then to all of the command tables from which *command-table* inherits. The recursion follows a depth-first path, considering the ''inheritees'' of the first inheritee before considering the second inheritee. This is the precedence order for command table inheritance.

clim:document-presentation-translator *translator presentation context-type frame event window x y* &key *(:stream* ***standard-output****) :documentation-type* *Function*

Computes the documentation string for *translator*, and displays it on the stream *:stream*. *presentation*, *context-type*, *frame*, *gesture*, *window*, *x*, and *y* are as for **clim:find-applicable-translators**.

:documentation-type should be either **:normal** or **:pointer**. When it is **:normal**, the translator should generate ''normal'' documentation. Otherwise it should generate the pointer documentation, which is usually shorter.

clim:dolist-noting-progress *(var listform name* &optional *stream note-var)* &body *body* *Function*

Binds the dynamic environment such that the progress of a **dolist** special form is noted by a progress bar displayed in the specified stream (usually the pointer documentation pane).

var is a variable bound to each successive element in *listform* on each successive iteration. *listform* is the list. *name* is a string naming the operation being noted; this string is displayed with the progress bar.

note-var is a variable bound to the current note object; the default is **clim:*current-progress-note***.

```
(defun note-element-printing (list)
  (clim:dolist-noting-progress (element list "Printing elements")
    (print element)
    (sleep 1)))
```

clim:dotimes-noting-progress *(var countform name* &optional *stream note-var)* &body *body* *Macro*

Binds the dynamic environment such that the progress of a **dotimes** special form is noted by a progress bar displayed in the specified stream (usually the pointer documentation pane).

var is a variable bound to the count (0, 1, 2, and so on) on each successive iteration. *countform* is the number of iterations. *name* is a string naming the operation being noted; this string is displayed with the progress bar.

note-var is a variable bound to the current note object; the default is **clim:*current-progress-note***.

```
(defun note-square-roots (n)
  (clim:dotimes-noting-progress
      (count n "Calculating square roots")
    (sqrt count)        ;whoopee!
    (sleep 1)))
```

clim:drag-callback *gadget client id value* *Generic Function*

This callback is invoked when the value of a slider or scroll bar is changed while the indicator is being dragged. This is implemented by calling the function specified by the **:drag-callback** initarg with two arguments, the slider (or scroll bar) and the new value. Generally, this function will call another programmer-specified callback function.

clim:drag-output-record *stream output-record* &key *(:repaint* **t***) :multiple-window :erase :feedback (:finish-on-release* **t***)* *Function*

Enters an interaction mode in which user moves the pointer, and *output-record* follows the pointer by being dragged on *stream*.

:repaint Allows you to specify the appearance of windows as the pointer is dragged. If *:repaint* is **t** (the default), displayed contents of windows are not disturbed as *output-record* is dragged over them (that is, those regions of the screen are repainted).

:erase Allows you to identify a function to erase the output record (the default effectively uses **clim:erase-output-record**). *:erase* is a function of two arguments, the output record to erase, and the stream.

:feedback Allows you to identify a feedback function. *:feedback* is a 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**).

Use *:feedback* if you want more complex feedback than is supplied by default, for instance, if you want to draw a ''rubber band'' line as the user moves the mouse. The default for *:feedback* is **nil**.

:multiple-window

When **t**, specifies that the pointer is to be tracked across multiple windows. The default is **nil**.

:finish-on-release

When this is **t** (the default), the body is exited when the user releases the mouse button currently being held down. When this is **nil**, the body is exited when the user clicks a mouse button.

Note that if *:feedback* is supplied, *:erase* is ignored.

clim:dragging-output *(*&optional *stream* &key *(:repaint* **t***) :multiple-window :finish-on-release)* &body *body* *Macro*

Evaluates *body* to produce the output, and then invokes **clim:drag-output-record** to drag that output on *stream*. *stream* defaults to ***standard-input***.

:repaint allows you to control the appearance of windows as the pointer is dragged. If *:repaint* is **t** (the default), displayed contents of windows are not disturbed as *output-record* is dragged over them (that is, those regions of the screen are repainted).

:multiple-window is as for **clim:drag-output-record**.

If *:finish-on-release* is **t** (the default), **clim:dragging-output** is exited when the user releases the mouse button currently being held down. When it is **nil**, **clim:dragging-output** is exited when the user clicks a mouse button.

clim:draw-arrow *medium start-point end-point* &key *:from-head (:to-head* **t***) (:head-length* **10***) (:head-width* **5***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws an arrow on *medium*. The arrow starts at the position specified by *start-point* and ends with the arrowhead at the position specified by *end-point*, two point objects.

This function is the same as **clim:draw-arrow***, except that the positions are specified by points, not by X and Y positions.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-arrow* *medium x1 y1 x2 y2* &key *:from-head (:to-head* **t***) (:head-length* **10***) (:head-width* **5***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws an arrow on *medium*. The arrow starts at the position specified by *(x1,y1)* and ends with the arrowhead at the position specified by *(x2,y2)*.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-circle *medium center radius* &key *:start-angle :end-angle (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws a circle or arc on *medium*. The center of the circle is specified by the point *center*, and the radius is specified by *radius*.

This function is the same as **clim:draw-circle***, except that the center position is expressed as a point instead of X and Y positions. See the function **clim:draw-circle***.

:start-angle and *:end-angle*
: Enable you to draw an arc rather than a complete circle in the same manner as that of the ellipse functions. See the function **clim:draw-ellipse***.

  The defaults for *:start-angle* and *:end-angle* are **nil** (that is, a full circle).

:filled
: Specifies whether the circle should be filled, a boolean value.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-circle* *medium center-x center-y radius* &key *:start-angle :end-angle (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws a circle or arc on *medium*. The center of the circle is specified by *center-x* and *center-y*, and the radius is specified by *radius*.

:start-angle and *:end-angle*
: Enable you to draw an arc rather than a complete circle in the same manner as that of the ellipse functions. See the function **clim:draw-ellipse***.

The defaults for *:start-angle* and *:end-angle* are **nil** (that is, a full circle).

:filled Specifies whether the circle should be filled, a boolean value.

Both **clim:draw-circle*** and **clim:draw-circle** call **clim:medium-draw-ellipse*** to do the actual drawing.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-design *design stream* &key *:ink :clipping-region :transformation :line-style :unit :thickness :joint-shape :cap-shape :dashes :text-style :text-family :text-face :text-size* *Generic Function*

Draws *design* on *stream*. The additional keyword arguments are used in a manner that depends upon the type of the *design*. For example, for designs that are paths (such as lines and unfilled circles), you may include the **:line-style** keyword.

The *design* types are:

area Paints the specified region of the drawing plane with *stream*'s current ink.

path Strokes the path with *stream*'s current ink under control of the line-style.

point The same as **clim:draw-point**.

a color or an opacity
Paints the entire drawing plane (subject to the medium's clipping region).

clim:+nowhere+
This has no effect.

If *design* is a non-uniform design this paints the design positioned at coordinates (x=0, y=0).

clim:draw-design is currently supported for the following designs:

- Designs created by the geometric object constructors (such as **clim:make-line** and **clim:make-ellipse**).
- Designs created by **clim:compose-in**, where the first argument is an ink and the second argument is a design.
- **clim:compose-over** of designs created by **clim:compose-in**.

- Designs returned by **clim:make-design-from-output-record**.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-ellipse *medium point radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws an ellipse or elliptical arc on *medium*. The center of the ellipse is specified by *point*.

This function is the same as **clim:draw-ellipse***, except that the center position is expressed as a point instead of X and Y. See the function **clim:draw-ellipse***.

Two vectors, *radius-1-dx*, *radius-1-dy*, and *radius-2-dx*, *radius-2-dy* specify the bounding parallelogram of the ellipse. Those two vectors must not be collinear in order for the ellipse to be well defined. The special case of an ellipse with its major axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to **0**. For more information about the bounding parallelogram of an ellipse, see the section "Ellipses and Elliptical Arcs in CLIM".

:start-angle and *:end-angle*
: Enable you to draw an arc rather than a complete ellipse. Angles are measured with respect to the positive X-axis. The elliptical arc runs positively from *:start-angle* to *:end-angle*. The angles are measured from the positive X-axis toward the positive Y-axis. In a right-handed coordinate system this direction is counter-clockwise.

  The defaults for *:start-angle* and *:end-angle* are **nil** (that is, a full ellipse). If you supply *:start-angle*, then *:end-angle* defaults to **2pi**. If you supply *:end-angle*, then *:start-angle* defaults to 0.

:filled
: Specifies whether the ellipse should be filled, a boolean value.

In the case of a filled arc, 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 affects the drawing as usual, except that the joint shape has no effect. The dashing of an elliptical arc starts at *:start-angle*.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

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

Draws an ellipse or elliptical arc on *medium*. The center of the ellipse is specified by *center-x* and *center-y*.

This function is the same as **clim:draw-ellipse**, except that the center position is expressed as its X and Y coordinates, instead of as a point. See the function **clim:draw-ellipse**.

Both **clim:draw-ellipse*** and **clim:draw-ellipse** call **clim:medium-draw-ellipse*** to do the actual drawing.

Two vectors, *radius-1-dx*, *radius-1-dy*, and *radius-2-dx*, *radius-2-dy* specify the bounding parallelogram of the ellipse. Those two vectors must not be collinear in order for the ellipse to be well defined. The special case of an ellipse with its major axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to **0**. For more information about the bounding parallelogram of an ellipse, see the section "Ellipses and Elliptical Arcs in CLIM".

:start-angle and *:end-angle*
: Enable you to draw an arc rather than a complete ellipse. Angles are measured with respect to the positive X-axis. The elliptical arc runs positively from *:start-angle* to *:end-angle*. The angles are measured from the positive X-axis toward the positive Y-axis. In a right-handed coordinate system this direction is counter-clockwise.

  The defaults for *:start-angle* and *:end-angle* are **nil** (that is, a full ellipse). If you supply *:start-angle*, then *:end-angle* defaults to **2pi**. If you supply *:end-angle*, then *:start-angle* defaults to 0.

:filled
: Specifies whether the ellipse should be filled, a boolean value.

In the case of a filled arc, 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 affects the drawing as usual, except that the joint shape has no effect. The dashing of an elliptical arc starts at *:start-angle*.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-line *medium point-1 point-2* &key *:line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws a line segment on *medium*. The line starts at the position specified by *point-1* and ends at the position specified by *point-2*, two point objects.

This function is the same as **clim:draw-line***, except that the positions are specified by points, not by X and Y positions.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-line* *medium x1 y1 x2 y2* &key *:line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws a line segment on *medium*. The line starts at the position specified by (*x1*, *y1*), and ends at the position specified by (*x2*, *y2*).

Both **clim:draw-line*** and **clim:draw-line** call **clim:medium-draw-line*** to do the actual drawing.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-lines *medium point-seq* &key *:line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws a set of disconnected line segments onto *medium*. *point-seq* is a sequence of pairs of points. Each point pair specifies the starting and ending point of one line.

This function is semantically equivalent to calling **clim:draw-line** repeatedly, but it can be more convenient and efficient when drawing more than one line segment. See the function **clim:draw-line**.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-lines* *medium coord-seq* &key *:line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws a set of disconnected line segments onto *medium*. *coord-seq* is a sequence of pairs of X and Y positions. Each pair of pairs specifies the starting and ending point of one line.

This function is equivalent to calling **clim:draw-line*** repeatedly, but it can be more convenient and efficient when drawing more than one line segment. See the function **clim:draw-line***.

Both **clim:draw-lines*** and **clim:draw-lines** call **clim:medium-draw-lines*** to do the actual drawing.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-oval *medium point x-radius y-radius* &key *(:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws an oval, that is, a ''race-track'' shape, centered on *point*, a point object. If *x-radius* or *y-radius* is 0, draws a circle with the specified non-zero radius; otherwise, draws the figure that results from drawing a rectangle with dimensions *x-radius* and *y-radius* and then replacing the two short sides with semicircular arc of appropriate size.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-oval* *medium center-x center-y x-radius y-radius* &key *(:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws an oval, that is, a ‘‘race-track’’ shape, centered on (*center-x center-y*): if *x-radius* or *y-radius* is 0, draws a circle with the specified non-zero radius; otherwise, draws the figure that results from drawing a rectangle with dimensions *x-radius* and *y-radius* and then replacing the two short sides with semicircular arc of appropriate size.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-pattern* *stream pattern x y* &key *:clipping-region :transformation*
Function

Draws the pattern *pattern* on *stream* at the position (*x*,*y*). *pattern* is a design created by calling **clim:make-pattern**, for example,

```
(clim:make-pattern #2A((0 0 0 1 1 0 0 0)
                       (0 0 1 1 1 1 0 0)
                       (0 1 1 1 1 1 1 0)
                       (1 1 1 0 0 1 1 1)
                       (1 1 1 0 0 1 1 1)
                       (0 1 1 1 1 1 1 0)
                       (0 0 1 1 1 1 0 0)
                       (0 0 0 1 1 0 0 0))
                   (list clim:+background-ink+
                         clim:+foreground-ink+))
```

You could also make the above pattern translucent by using **clim:+transparent-ink+** instead of **clim:+background-ink+**. In that case, the output underneath the pattern will show through wherever there is a zero value in the pattern.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-pixmap *medium pixmap point* &rest *args* &key *:ink :clipping-region :transformation (:function* **boole-1***)*
Function

Draws the pixmap *pixmap* on *medium* at the position *point*. This function is the same as **clim:draw-pixmap***, except that the position is specified by a point object, not by an X/Y position.

:function is a boolean operation that controls how the source and destination bits are combined.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-pixmap* *medium pixmap x y* &rest *args* &key *:ink :clipping-region :transformation (:function* **boole-1***)* *Function*

Draws the pixmap *pixmap* on *medium* at the position (*x*,*y*). *pixmap* is a pixmap created by using **clim:copy-area** or **clim:with-output-to-pixmap**. Unlike **clim:copy-area**, **clim:draw-pixmap*** will create a ‘‘pixmap output record’’ when called on an output recording stream.

:function is a boolean operation that controls how the source and destination bits are combined.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-point *medium point* &key *:line-style :line-thickness :line-unit :ink :clipping-region :transformation* *Function*

Draws a point on *medium* at the position indicated by *point*.

CLIM uses the medium’s line style to decide how to draw a point on a display device. The *:line-unit* and *:line-thickness* arguments control the size on the display device of the ‘‘blob’’ used to render the point.

The *:line-unit* and *:line-thickness* arguments control the size on the display device of the ‘‘blob’’ used to render the point.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-point* *medium x y* &key *:line-style :line-thickness :line-unit :ink :clipping-region :transformation* *Function*

Draws a point on *medium* at the position indicated by *x* and *y*.

CLIM uses the medium's line style to decide how to draw a point on a display device. The *:line-unit* and *:line-thickness* arguments control the size on the display device of the ''blob'' used to render the point.

Both **clim:draw-point*** and **clim:draw-point** call **clim:medium-draw-point*** to do the actual drawing.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-points *medium point-seq* &key *:line-style :line-thickness :line-unit :ink :clipping-region :transformation* *Function*

Draws a set of points on *medium*. *point-seq* is a sequence of point objects specifying where a point is to be drawn.

This function is equivalent to calling **clim:draw-point** repeatedly, but it can be more convenient and efficient when drawing more than one point.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-points* *medium coord-seq* &key *:line-style :line-thickness :line-unit :ink :clipping-region :transformation* *Function*

Draws a set of points on *medium*. *coord-seq* is a sequence of pairs of X/Y pairs (that is, a sequence of alternating X coordinates and Y coordinates which when taken pairwise specify the points to be drawn).

This function is equivalent to calling **clim:draw-point*** repeatedly, but it can be more convenient and efficient when drawing more than one point.

Both **clim:draw-points*** and **clim:draw-points** call **clim:medium-draw-points*** to do the actual drawing.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-polygon *medium point-seq* &key *(:closed* **t***) (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-joint-shape :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws a polygon, or sequence of connected lines, on *medium*. The keyword arguments control whether the polygon is closed (each segment is connected to two other segments) and filled. *point-seq* is a sequence of points that indicate the start of a new line segment.

This function is the same as **clim:draw-polygon***, except that the segments are specified by points, not X and Y positions.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-polygon* *medium coord-seq* &key *(:closed* **t***) (:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-joint-shape :line-cap-shape :ink :clipping-region :transformation* *Function*

Draws a polygon, or sequence of connected lines, on *medium*. The keyword arguments control whether the polygon is closed (each segment is connected to two other segments) and filled. *coord-seq* is a sequence of alternating X and Y positions that indicate the start of a new line segment.

:filled Specifies whether the polygon should be filled, a boolean value. If **t**, a closed polygon is drawn and filled in. In this case, *:closed* is assumed to be **t**.

:closed When **t**, specifies that a segment is drawn connecting the ending point of the last segment to the starting point of the first segment.

Both **clim:draw-polygon*** and **clim:draw-polygon** call **clim:medium-draw-polygon*** to do the actual drawing.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-rectangle *medium point1 point2* &rest *args* &key *:line-style :line-thickness :line-unit :line-dashes :line-joint-shape :ink :clipping-region :transformation (:filled* **t***)* *Function*

Draws an axis-aligned rectangle on *medium*. The boundaries of the rectangle are specified by the two points *point1* and *point2*.

This function is the same as **clim:draw-rectangle***, except that the positions are specified by points, not by X and Y positions.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-rectangle* *medium x1 y1 x2 y2* &key *(:filled* **t***) :line-style :line-thickness :line-unit :line-dashes :line-joint-shape :ink :clipping-region :transformation* *Function*

Draws an axis-aligned rectangle on *medium*. The boundaries of the rectangle are specified by *x1*, *y1*, *x2*, and *y2*, with *(x1,y1)* at the upper left and *(x2,y2)* at the lower right in the standard +Y-downward coordinate system.

:filled Specifies whether the rectangle should be filled, a boolean value. If **t**, a closed rectangle is drawn and filled in.

Both **clim:draw-rectangle*** and **clim:draw-rectangle** call **clim:medium-draw-rectangle*** to do the actual drawing.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-rectangles *medium point-seq* &rest *args* &key *:line-style :line-thickness :line-unit :line-dashes :line-joint-shape :ink :clipping-region :transformation (:filled* **t***)* *Function*

Draws a set of axis-aligned rectangles on *medium*. *point-seq* is a sequence of pairs of points. Each point specifies the upper left and lower right corner of the rectangle in the standard +Y-downward coordinate system.

This function is equivalent to calling **clim:draw-rectangle** repeatedly, but it can be more convenient and efficient when drawing more than one rectangle. See the function **clim:draw-rectangle**.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-rectangles* *medium coord-seq* &rest *args* &key *:line-style :line-thickness :line-unit :line-dashes :line-joint-shape :ink :clipping-region :transformation (:filled* **t***)* *Function*

Draws a set of axis-aligned rectangles on *medium*. *coord-seq* is a sequence of 4-tuples *x1*, *y1*, *x2*, and *y2*, with *(x1,y1)* at the upper left and *(x2,y2)* at the lower right in the standard +Y-downward coordinate system.

:filled Specifies whether the rectangle should be filled, a boolean value. If **t**, a closed rectangle is drawn and filled in.

This function is equivalent to calling **clim:draw-rectangle*** repeatedly, but it can be more convenient and efficient when drawing more than one rectangle. See the function **clim:draw-rectangle***.

Both **clim:draw-rectangles*** and **clim:draw-rectangles** call **clim:medium-draw-rectangles*** to do the actual drawing.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-standard-menu *menu presentation-type items default-item* &key *(:item-printer* **#'clim:print-menu-item***) :max-width :max-height :n-rows :n-columns :x-spacing :y-spacing :row-wise (:cell-align-x* **':left***) (:cell-align-y* **':top***)* *Function*

clim: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. *menu* is the stream onto which to draw the menu, and *presentation-type* is the presentation type to use for the menu items (usually **clim:menu-item**).

:item-printer is a function of two arguments used to draw each item. The first argument is the menu item, and the second is the stream.

The other arguments are as for **clim:menu-choose**.

clim:draw-text *medium text point* &key *(:start* **0***) :end (:align-x* **:left***) (:align-y* **:baseline***) :towards-point :text-style :text-family :text-face :text-size :ink :clipping-region :transformation* *Function*

Draws *text* onto *medium* starting at the position specified by *point*. *text* can be either a character or a string.

This function is the same as **clim:draw-text***, except that the position is expressed as a point instead of as X and Y coordinate values. See the function **clim:draw-text***.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:draw-text* *medium text x y* &key *(:start* **0***) :end (:align-x* **:left***) (:align-y* **:baseline***) :towards-x :towards-y :text-style :text-family :text-face :text-size :ink :clipping-region :transformation* *Function*

Draws *text* onto *medium* starting at the position specified by *x* and *y*. *text* can be either a character or a string.

The exact definition of ''starting at'' is dependent on *:align-x* and *:align-y*; by default, the first glyph is drawn with its left edge and its baseline at the position specified by *x* and *y*.

:start and *:end*
Delimit a substring of *text* when *text* is a string. *:start* defaults to 0, and *:end* defaults to the length of *text*.

:align-x Specifies the horizontal placement of the text string. Can be one of: **:left** (the default), **:right**, or **:center**.

:left means that the left edge of the first character of the string is at the specified X coordinate. **:right** means that the right edge of the last character of the string is at the specified X coordinate. **:center** means that the string is horizontally centered over the specified X coordinate.

:align-y Specifies the vertical placement of the string. Can be one of: **:baseline** (the default), **:top**, **:bottom**, or **:center**.

:baseline means that the baseline of the string is placed at the specified Y coordinate. **:top** means that the top of the string is at the specified Y coordinate. **:bottom** means that the bottom of the

string is at the specified Y coordinate. **:center** means that the string is vertically centered over the specified Y coordinate.

:towards-x and *:towards-y*
Changes the direction of the baseline from one glyph to the next. Normally, glyphs are drawn from left to right no matter what transformation is in effect. If, however, *:towards-x* is less than *x*, then glyphs will be drawn from right to left. If *:towards-y* is less than *y*, then glyphs will be drawn from bottom to top.

Note that *:towards-x* and *:towards-y* are not presently implemented on all platforms.

Note that the medium's transformation does not affect the text size. It only affects the starting position *(x,y)*, and the ending position (and hence the orientation of the baseline) if *:towards-x* or *:towards-y* is supplied.

Both **clim:draw-text*** and **clim:draw-text** call **clim:medium-draw-text*** to do the actual drawing.

For background information on drawing graphics, see the section "Drawing Graphics in CLIM", and see the section "Using CLIM Drawing Options".

For detailed information about CLIM line style suboptions, see the section "CLIM Line Style Suboptions".

For detailed information about CLIM drawing options, see the section "Set of CLIM Drawing Options".

clim:ellipse *Class*

The protocol class that corresponds to a mathematical ellipse. This is a subclass of **clim:area**. If you want to create a new class that obeys the ellipse protocol, it must be a subclass of **clim:ellipse**.

clim:ellipse-center-point *ellipse* *Generic Function*

Returns the center point of *ellipse*.

clim:ellipse-center-point* *ellipse* *Generic Function*

Returns the center point of *ellipse* as two values representing the coordinate pair.

clim:ellipse-end-angle *ellipse* *Generic Function*

Returns the end angle of *ellipse*. If *elliptical-object* is a full ellipse or closed path then **clim:ellipse-end-angle** will return **nil**; otherwise the value will be a number greater than zero, and less than or equal to **2pi**.

clim:ellipse-radii *ellipse* *Generic Function*

Returns four values corresponding to the two radius vectors of *ellipse*. These values may be canonicalized in some way, and so may not be the same as the values passed to the constructor function.

clim:ellipse-start-angle *ellipse* *Generic Function*

Returns the start angle of *ellipse*. If *elliptical-object* is a full ellipse or closed path then **clim:ellipse-start-angle** will return **nil**; otherwise the value will be a number greater than or equal to zero, and less than **2pi**.

clim:ellipsep *object* *Function*

Returns **t** if and only if *object* is of type **clim:ellipse**.

clim:elliptical-arc *Class*

The protocol class that corresponds to a mathematical elliptical arc. This is a subclass of **clim:path**. If you want to create a new class that obeys the elliptical arc protocol, it must be a subclass of **clim:elliptical-arc**.

clim:elliptical-arc-p *object* *Function*

Returns **t** if and only if *object* is of type **clim:elliptical-arc**.

clim:enable-frame *frame* *Generic Function*

Enables the application frame *frame* and changes the state of the frame to **:enabled**. This involves creating and laying out the panes of the frame (if they have not been created already) and exposing the frame.

Note: If your application frame has its own top level loop (that is, something other than **clim:default-frame-top-level**), it must call **clim:enable-frame** in order to enable the frame.

clim-sys:enable-process *process* *Function*

Allows the process *process* to become runnable again after it has been disabled.

clim:erase-input-buffer *input-editing-stream* &optional *start-position*
Generic Function

Erases the part of the display that corresponds to the input editor's buffer starting at the position *start-position*.

For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:erase-output-record *record stream* &optional *(errorp* **t***)* *Function*

Erases the display of the output record *record* from *stream*, and removes the record from *stream*'s output history. After the record is erased, all of the output records that overlapped it are replayed in order to ensure that the appearance of the rest of the output on *stream* is correct.

If *record* is not in the stream's output history and *errorp* is **t**, CLIM will signal an error.

If *record* is a list of output records rather than a single output record, the replay operation will be delayed until after all of the output records have been removed from the output history. Passing a list of output records to **clim:erase-output-record** can be substantially faster than calling **clim:erase-output-record** multiple times.

clim:even-scaling-transformation-p *transform* *Generic Function*

Returns **t** if *transform* multiplies all X-lengths and Y-lengths by the same magnitude, otherwise returns **nil**. This includes pure reflections through vertical and horizontal lines.

clim:event *Class*

The class that corresponds to any kind of CLIM event. This includes device events (such as keyboard and pointer events), window manager events, and timer events.

clim:event-matches-gesture-name-p *event gesture-name* &optional *port* *Function*

Returns **t** if the device event *event* ''matches'' the gesture named by *gesture-name*.

For pointer button events, the event matches the gesture name when the pointer button from the event matches the name of the pointer button one of the gesture specifications named by *gesture-name*, and the modifier key state from the event matches the names of the modifier keys in that same gesture specification.

For keyboard events, the event matches the gesture name when the key name from the event matches the key name of one of the gesture specifications named by *gesture-name*, and the modifier key state from the event matches the names of the modifier keys in that same gesture specification.

clim:event-modifier-state *event* *Generic Function*

Returns the state of the keyboard's shift keys when the event *event* occurred. The returned value is an integer with 1 bits that correspond to the shift keys that were being held down.

Key	*Mask*
:shift	**clim:+shift-key+**
:control	**clim:+control-key+**
:meta	**clim:+meta-key+**
:super	**clim:+super-key+**
:hyper	**clim:+hyper-key+**

clim:event-sheet *event* *Generic Function*

Returns the window on which the device event *event* occurred.

clim:event-type *event* *Generic Function*

For the event *event*, returns a keyword symbol with the same name as the class name, except stripped of the "`-event`" ending. For example, when you call **clim:event-type** on an object who class is **clim:key-press-event**, the returned value will be **:key-press**.

clim:eventp *object* *Function*

Returns **t** if and only if *object* is an event.

clim:+everywhere+ *Constant*

The region that includes all the points on the infinite drawing plane. This is the opposite of **clim:+nowhere+**.

clim:execute-frame-command *frame command* *Generic Function*

clim:execute-frame-command executes the command *command* on behalf of the application frame *frame*. If you call **clim:execute-frame-command** from a process that is different from the process in which *frame* is running, CLIM will queue the command for later execution in *frame*'s own process.

The default method for **clim:execute-frame-command** simply applies to command name to the command arguments. You can specialize this function if you want to change the behavior associated with the execution of commands.

clim:expand-presentation-type-abbreviation *type* &optional *environment* *Function*

clim:expand-presentation-type-abbreviation is like **clim:expand-presentation-type-abbreviation-1**, except that *type* is repeatedly expanded until all presentation type abbreviations have been expanded.

The optional argument *environment* conveys information about the local macro definitions that might be defined by **macrolet**. It is used the same as it would be for **macroexpand**.

clim:expand-presentation-type-abbreviation-1 *type* &optional *environment* *Function*

If the presentation type specifier *type* is a presentation type abbreviation, or is an **and**, **or**, **sequence**, or **clim:sequence-enumerated** that contains a presentation type abbreviation, then **clim: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.

The optional argument *environment* conveys information about the local macro definitions that might be defined by **macrolet**. It is used the same as it would be for **macroexpand**.

clim:expression *Clim Presentation Type*

The presentation type used to represent any Lisp object. The textual view of this type looks like what the standard **prin1** and **read** functions produce and accept.

This type has one option, **:auto-activate**, which controls whether the expression terminates on a delimiter gestures, or when the Lisp expression ''balances'' (for example, you type enough close parentheses to complete the expression). The default for **:auto-activate** is **nil**, meaning that the user must use an activation gesture to terminate the input.

clim:extended-input-stream-p *object* *Generic Function*

Returns **t** if the *object* is a CLIM extended input stream, otherwise it returns **nil**.

clim:extended-output-stream-p *object* *Generic Function*

Returns **t** if the *object* is a CLIM extended output stream, otherwise it returns **nil**.

clim:+fill+ *Constant*

This constant can be used as a value to any of the min or max size options in **clim:make-space-requirement** or the layout pane macros. It indicates a pane's willingness to adjust an arbitrary amount in the specified direction (width or height).

clim:filling-output *(*&optional *stream* &rest *keys* &key *(:fill-width* **'(80 :character)***) (:break-characters* **'(#\Space)***) :after-line-break :after-line-break-initially)* &body *body* *Function*

Binds *stream* to a stream that inserts line breaks into the output written to it so that the output is no wider then *:fill-width*. The filled output is then written on the stream that is the original value of stream. **clim:filling-output** does not split ''words'' across lines, so it can produce output wider than *:fill-width*.

"Words" are separated by the characters indicated by *: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.

stream The output stream; the default is ***standard-output***.

:fill-width Specifies the width of filled lines. The default is 80 characters. It can be specified in one of the following ways:

integer The width in device units (for example, pixels).

string The spacing is the width of the string.

function The spacing is the amount of space the function would consume when called on the stream.

list The list is of the form (*number unit*), where *unit* is one of

:pixel The width in pixels.

:point The width in printers points.

:character The width of the "usual" character in the stream's current text style.

If CLIM cannot find a break character that would make the output narrow enough, the output might be wider than *:fill-width*

:break-characters
Specifies a list of characters at which to break lines. The default includes only the `#\Space` character.

: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*.

:after-line-break-initially
Boolean option specifying whether the *:after-line-break* text is to be written to *stream* before doing *body*, that is, at the beginning of the first line; the default is **nil**.

Here is an example of using **clim:filling-output**. Notice that this example uses **clim:format-textual-list** to generate the text to fill.

```
(let ((stream *standard-output*)
      (alphabet '(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)))
  (clim:filling-output (stream :fill-width '(30 :character) )
    (clim:format-textual-list alphabet #'princ
                              :stream stream
                              :separator ", " :conjunction "and")
    (write-char #\. stream)))
```

clim:find-applicable-translators *presentation input-context frame window x y* &key *:event :modifier-state :for-menu :fastp* *Function*

Returns a list of translators that apply to *presentation* in the input context *input-context*. Since input contexts can be nested, **clim:find-applicable-translators** iterates over all of contexts in *input-context*. See the section "Applicability of CLIM Presentation Translators".

frame is the application frame. *window*, *x*, and *y* are the window the presentation is on, and the X and Y position of the pointer (respectively).

:event (which defaults to **nil**) is a pointer button event, and may be supplied to further restrict the set of applicable translators to only those whose gesture matches the pointer button event.

:modifier-state (which defaults to the current modifier state for the window) may also be supplied to restrict the set of applicable translators to only those who gesture matches the shift mask. Only one of *:event* or *:modifier-state* may be supplied.

When *:for-menu* is **t** (the default), this returns every possible translator, disregarding *:event* and *:modifier-state*.

When *:fastp* is **t**, this will simply return **t** if there are any translators. *:fastp* defaults to **nil**.

clim:find-application-frame *frame-name* &rest *initargs* &key *(:create* **t***) (:activate* **t***) (:own-process* **t***) :port :frame-manager :frame-class* &allow-other-keys *Function*

Calling this function is similar to calling **clim:make-application-frame**, and then calling **clim:run-frame-top-level** on the newly created frame.

If *:create* is **t**, a new frame will be created if one does not already exist. If *:create* is **:force**, a new frame will be created regardless of whether there is one already.

If *:activate* is **t** (the default), the frame's top level function will be invoked. If *:own-process* is **t** (the default), the frame will run in its own process.

:port and *:frame-manager* can be used to name the parent of the frame. *:frame-class* is as for **clim:make-application-frame**. *initargs* are CLOS initargs that are passed along to **clim:make-application-frame**.

clim:find-command-from-command-line-name *name command-table* &key *(:errorp* **t***)* *Function*

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. If the command is not accessible in *command-table* and *:errorp* is **t**, the **clim:command-not-accessible** condition will be signalled.

name is a command-line name. *command-table* may be either a command table or a symbol that names a command table.

clim:find-command-from-command-line-name ignores character case.

This function is the inverse of **clim:command-line-name-for-command**.

clim:find-command-table *name* &key *(:errorp* **t***)* *Function*

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 **clim:command-table-not-found** condition will be signalled.

clim:find-frame-manager &rest *options* &key *:port (:server-path* **clim:*default-server-path***) &allow-other-keys *Generic Function*

Finds a frame manager that is on the port *:port*, or creates a new one if none exists. If *:port* is not supplied and a new port must be created, *:server-path* may be supplied for use by **clim:find-port**.

options may include other initargs for the frame manager.

On Genera, you may supply the **:gadget-menu-bar** option. When you supply **:gadget-menu-bar t**, the Genera frame manager will use a toolkit-style menu bar. The default Genera frame manager does not use gadget-style menu bars.

clim:find-innermost-applicable-presentation *input-context stream x y* &key *(:frame* **clim:*application-frame***) *:modifier-state :event* *Function*

Given an input context *input-context*, an output recording window stream *stream*, and X and Y positions *x* and *y*, **clim:find-innermost-applicable-presentation** returns the innermost presentation that matches the innermost input context. See the section "Applicability of CLIM Presentation Translators".

:frame is the application frame. *:modifier-state* is a mask that describes what shift keys are held down on the keyboard, and defaults to the window's current modifier state. *:event* is a pointer button event. You may supply only one of *:modifier-state* or *:event*.

clim:find-keystroke-item *keystroke command-table* &key *:test (:errorp* **t***)* *Function*

Given a keystroke accelerator *keystroke* and a *command-table*, returns two values, the command menu item associated with the accelerator and the command table in which it was found. (Since keystroke accelerators are not normally inherited, the second returned value will usually be *command-table*.) *keystroke* is gesture spec, such as `(:C :control :shift)`.

:test specifies a function to use for looking up the items in the command table. It must be a function of two arguments, both of which are events or gesture specs. The default is a function that matches keyboard events against gesture names.

If the keystroke accelerator is not present in any command table and *:errorp* is **t**, then the **clim:command-not-accessible** condition will be signalled. *command-table* may be either a command table or a symbol that names a command table.

clim:find-menu-item *menu-name command-table* &key *(:errorp* **t***)* *Function*

Given a *menu-name* and a *command-table*, return two values, the menu item and the command table in which it was found. (Since menus are not normally inherited, the second returned value will usually be *command-table*.) If the command menu item is not present in any command table and *:errorp* is **t**, then the **clim:command-not-accessible** condition will be signalled. *command-table* may be either a command table or a symbol that names a command table.

clim:find-named-color *name palette* &key *:errorp* *Function*

Finds the color named *name* in the palette *palette*. The palette associated with the current application frame can be found by calling **clim:frame-palette**. The returned value is a CLIM color object.

If the color is not found and *:errorp* is **t**, the **clim:color-not-found** error is signalled. Otherwise if the color is not found, this function returns **nil**.

Palettes are described in more detail in "Predefined Color Names in CLIM".

clim:find-pane-named *frame pane-name* &optional *errorp* *Function*

Returns the pane named by *pane-name* in *frame*. If *pane-name* is a pane object, it is simply returned. This differs from **clim:get-frame-pane** in that the returned value can be any pane type, not just a subclass of **clim:clim-stream-pane**.

If the pane named *pane-name* is not found and *:errorp* is **t**, an error will be signalled. Otherwise if *:errorp* is **nil**, the return value will be **nil**.

clim:find-presentation-translators *from-type to-type command-table* *Function*

Returns a list of all 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* must not be presentation type abbreviations.

clim:find-port &rest *initargs* &key *(:server-path* **clim:*default-server-path***) &allow-other-keys *Function*

Creates a port, a special object that acts as the ''root'' or ''parent'' of all CLIM windows and application frames. In general, a port corresponds to a connection to a display server.

:server-path is a list that specifies the server path. The first element of the list is a keyword naming the type of port to be created (such as **:genera**, **:clx**, or **:cloe**). The rest of the list are keyword-value pairs of options.

In Genera, the only option is **:screen**, which defaults to the main screen. If you are on a Genera Color system, and you give **:screen** the value returned by **color:find-color-screen**, CLIM will create a color port.

Under Cloe, the only supported port type is **:cloe** and there are no options.

If the CLX system (part of the X Remote Screen system) is loaded under Genera, the options are **:host** (a host name or object), and **:display** (a display number) that identify the X Server to be used.

Note: You should call **clim:find-port** only at runtime, not at load time. This function captures information about the screen currently in use.

You may supply a **:allow-loose-text-style-size-mapping** initarg for any type of port to specify whether or not the port will use ''loose'' text style mapping. A port that uses ''loose'' text style mapping picks the closest available font size, rather than picking an exact font size.See the generic function **clim:text-style-mapping**.

clim:+flipping-ink+ *Constant*

A flipping ink that flips **clim:+foreground-ink+** and **clim:+background-ink+**. You can think of this as an ''xor'' on monochrome displays.

float &optional *low high* *Clim Presentation Type*

The presentation type that represents a floating point number between *low* and *high*. This type is a subtype of **clim:real**.

clim:+foreground-ink+ *Constant*

An indirect ink that uses the medium's foreground design.

clim:form *Clim Presentation Type*

The presentation type used to represent a Lisp form. This type is a subtype of **clim:expression**. It has one option, **:auto-activate**, which is treated the same way as the **:auto-activate** option to **clim:expression**.

clim:formatting-cell *(*&optional *stream* &rest *options* &key *(:align-x* **':left***) (:align-y* **':top***) :min-width :min-height :record-type* &allow-other-keys*)* &body *body* *Macro*

Establishes a ''cell'' context on the *stream* (which defaults to ***standard-output***). All output performed on the stream within the extent of this macro will become the contents of one cell in a table. **clim:formatting-cell** must be used within the extent of **clim:formatting-row**, **clim:formatting-column**, or **clim:formatting-item-list**.

A cell can contain text, graphics, or both. The alignment keywords enable you to specify constraints that affect the placement of the contents of the cell. 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.

For background information, examples, and overviews of related functions, see the section "Formatting Tables in CLIM".

stream

The stream to which output should be sent. The default is ***standard-output***.

:align-x

Specifies the horizontal placement of the contents of the cell. Can be one of: **:left** (the default), **:right**, or **:center**.

:left means that the left edge of the cell is at the specified X coordinate. **:right** means that the right edge of the cell is at the specified X coordinate. **:center** means that the cell is horizontally centered over the specified X coordinate.

:align-y

Specifies the vertical placement of the contents of the cell. Can be one of: **:top** (the default), **:bottom**, or **:center**.

:top means that the top of the cell is at the specified Y coordinate. **:bottom** means that the bottom of the cell is at the specified Y coordinate. **:center** means that the cell is vertically centered over the specified Y coordinate.

:min-width

Specifies the minimum width of the cell. The default, **nil**, causes the width of the cell to be only as wide as is necessary to contain the cell's contents.

:min-width can be specified in one of the following ways:

Integer
: A size in the current units to be used for spacing.

String or character
: The spacing is the width or height of the string or character in the current text style.

Function
: The spacing is the amount of horizontal or vertical space the function would consume when called on the stream.

List of form (*number unit*)
: The *unit* is **:point**, **:pixel**, or **:character**.

:min-height

Specifies the minimum height of the cell. The default, **nil**, causes the height of the cell to be only as high as is necessary to contain the cell's contents.

:min-height is specified in the same way as *:min-width*.

:record-type

This option is useful when you have defined a customized record type to replace CLIM's default record type. It specifies the class of the output record to be created.

clim:formatting-column *(*&optional *stream* &key *:record-type)* &body *body* *Macro*

Establishes a ''column'' context on the *stream* (which defaults to ***standard-output***). All output performed on the stream within the extent of this macro will become the contents of one column of the table. **clim:formatting-column** must be used within the extent of **clim:formatting-table**, and it must be used in conjunction with **clim:formatting-cell**.

For background information, examples, and overviews of related functions, see the section "Formatting Tables in CLIM".

stream

The stream to which output should be sent. The default is ***standard-output***.

:record-type

This option is useful when you have defined a customized record type to replace CLIM's default record type. It specifies the class of the output record to be created.

clim:formatting-item-list *(*&optional *stream* &key *:record-type :x-spacing :y-spacing :initial-spacing :n-columns :n-rows :max-width :max-height :stream-width :stream-height (:row-wise* **t***) (:move-cursor* **t***))* &body *body* *Macro*

Establishes a ''menu formatting'' context on the *stream* (which defaults to ***standard-output***). Use this macro to format the output in a tabular form when the exact ordering and placement of the cells is not important. **clim:formatting-item-list** must be used with **clim:formatting-cell**.

This macro expects its *body* to output a sequence of items using **clim:formatting-cell**, which delimits each item. (You do not use **clim:formatting-column** or **clim:formatting-row** within **clim:formatting-item-list**.) If no keyword arguments are supplied, CLIM chooses the number of rows and columns for you. You can specify a constraint such as the number of columns or the number of rows (but not both). Or you can constrain the size of the entire table display, by using *:max-width* or *:max-height* (but not both). If you specify one of these constraints, CLIM will adjust the table accordingly.

For background information, examples, and overviews of related functions, see the section "Formatting Tables in CLIM".

stream

The stream to which output should be sent. The default is ***standard-output***.

:max-width

Specifies the maximum width, in device units, of the table display.

:max-height

Specifies the maximum height, in device units, of the table display.

:n-rows

Specifies the number of rows of the table. Specifying this overrides *:max-width*.

:n-columns

Specifies the number of columns of the table. Specifying this overrides *:max-height*.

:x-spacing

Determines the amount of space inserted between columns of the table; the default is the width of a space character. *:x-spacing* can be specified in one of the following ways:

Integer
: A size in the current units to be used for spacing.

String or character
: The spacing is the width or height of the string or character in the current text style.

Function
: The spacing is the amount of horizontal or vertical space the function would consume when called on the stream.

List of form (*number unit*)
: The *unit* is **:point**, **:pixel**, or **:character**.

: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 *:x-spacing* option.

:initial-spacing

When doing the layout, CLIM tries to evenly space items across the entire width of the stream. When this option is **t**, no whitespace is inserted before the first item on a line.

:row-wise When this is **nil**, if there are multiple columns in the item list, the entries in the item list are arranged in a manner similar to entries in a phone book. Otherwise the entries are arranged in a ''row-wise'' fashion. The default is **t**.

:stream-width

The width of the stream (in device units).

:stream-height

The height of the stream (in device units).

:move-cursor

When **t** (the default), CLIM moves the text cursor to the end (lower right corner) of the output. Otherwise, the cursor is left at the beginning (upper left corner) of the output.

clim:format-items *items* &key *(:stream* ***standard-output****) :printer :presentation-type :x-spacing :y-spacing :initial-spacing :n-rows :n-columns :max-width :max-height (:row-wise* **t***) :record-type (:cell-align-x* **':left***) (:cell-align-y* **':top***)* *Function*

Provides tabular formatting of a list of items. Each item in *items* is formatted as a separate cell within the table. *items* can be a list or a general sequence.

For background information, examples, and overviews of related functions, see the section "Formatting Tables in CLIM".

The options *:stream*, *:x-spacing*, *:y-spacing*, *:initial-spacing*, *:n-rows*, *:n-columns*, *:max-width*, *:max-height*, *:row-wise*, and *:record-type* are the same as for **clim:formatting-item-list**.

Note that you must specify one of *:printer* or *:presentation-type*.

:printer A function that takes two arguments, an item and a stream. It should output the item to the stream. You cannot use this keyword option with *:presentation-type*.

:presentation-type
A presentation type. You cannot use this keyword option with *:printer*.

The items will be printed as if *:printer* were:

```
#'(lambda (item stream)
   (clim:present item presentation-type :stream stream))
```

:cell-align-x
Supplies the **:align-x** option to an implicitly used **clim:formatting-cell**.

:cell-align-y
Supplies the **:align-y** option to an implicitly used **clim:formatting-cell**.

clim:format-graph-from-roots *root-objects object-printer inferior-producer* &key *(:stream* ***standard-output****) (:orientation* **':horizontal***) :center-nodes :cutoff-depth :merge-duplicates :graph-type (:duplicate-key* **#'identity***) (:duplicate-test* **#'eql***) :arc-drawer :arc-drawing-options :generation-separation :within-generation-separation :maximize-generations (:store-objects* **t***) (:move-cursor* **t***)* *Function*

Constructs and displays a directed, acyclic graph. The output from graph formatting takes place in a normalized +Y-downward coordinate system.

root-objects
A sequence of the root elements of the set, from which the graph can be derived.

object-printer
A function of two arguments used to display each node of the graph. The function is passed the object associated with that node and the stream on which to do output.

inferior-producer
A function that knows how to generate the inferiors (children) from a node object. It takes one argument, the node, and returns a list of inferior nodes.

:stream Specifies the output stream; the default is ***standard-output***.

:orientation
Specifies **:vertical** or **:horizontal** orientation for the ''parent node to child node'' direction of the graph display. The default for *:orientation* is **:vertical**.

:graph-type
The type of the graph to be displayed, either **:digraph** (the default when *:merge-duplicates* is **t**) or **:tree** (the default when *:merge-duplicates* is **nil**).

:merge-duplicates
When **t**, duplicate objects in the graph are displayed in the same node in the output. Otherwise, each object is given a unique node.

:center-nodes
When **t**, the display of each node is placed at the center of the space allocated for it. The default, **nil**, causes each node to be placed in the upper left of the space allocated to it.

:cutoff-depth
An integer that specifies how many levels of each branch of the tree should be explored. The default is **nil**, which specifies no cutoff.

:duplicate-key
Specifies the function used to extract the node object attribute used for duplicate comparison. The default is **identity**, that is, the object itself.

:duplicate-test
Specifies the function used to test for duplicates comparison. The default is **eql**, that is, the object itself.

:generation-separation
The amount of space to leave between successive generations of the graph.

:within-generation-separation
The amount of space to leave between nodes in the same generation of the graph.

:arc-drawer
A function (taking 7 required arguments and a rest argument) used to draw the arc between nodes. The default is a function that behave like **clim:draw-line**. The required arguments are: the stream, the ''from'' object, the ''to'' object, the ''from'' X and Y positions,

and the ''to'' X and Y positions. If *:store-objects* is **nil**, the ''from'' and ''to'' objects will be **nil**. The rest argument is a list of drawing options.

:store-objects
When this is **t** (the default), the objects will be stored in the graph and will be available to the arc drawing function. If the objects in your graph have dynamic extent or you require that they be subjected to garbage collection, you should supply **:store-objects nil**.

:move-cursor
When **t** (the default), CLIM moves the text cursor to the end (lower right corner) of the output. Otherwise, the cursor is left at the beginning (upper left corner) of the output.

For background information, examples, and overviews of related functions, see the section "Formatting Graphs in CLIM".

clim:format-graph-from-root *root-object object-printer inferior-producer* &key *(:stream* ***standard-output****) (:orientation* **':horizontal**) *:center-nodes :cutoff-depth :merge-duplicates :graph-type (:duplicate-key***#'identity**) (:duplicate-test* **#'eql**) *:arc-drawer :arc-drawing-options :generation-separation :within-generation-separation :maximize-generations (:store-objects* **t**) *(:move-cursor* **t**) *Function*

Like **clim:format-graph-from-roots**, except that *root-object* is a single root object instead of a sequence of roots. This function is provided only as a convenience.

clim:format-textual-list *sequence printer* &key *(:stream* ***standard-output****) (:separator* **", "**) *:conjunction Function*

Outputs a *sequence* of items as a textual list. For example, the list

```
(1 2 3 4)
```

could be printed as

```
1, 2, 3, and 4
```

The arguments provide control over the appearance of each element of the sequence and over the separators used between each pair of elements. The separator string is output after every element but the last one. The conjunction is output before the last element.

sequence The sequence to output.

printer is a function of two arguments: an element of the sequence and a stream. It is used to output each element of the sequence.

:stream Specifies the output stream. The default is ***standard-output****.

:separator Specifies the characters to use to separate elements of a textual list. The default is `", "` (comma followed by a space).

:conjunction

Specifies a string to use in the position between the last two elements. Typical values are `"and"` and `"or"`. It defaults to **nil**.

clim:formatting-row *(*&optional *stream* &key *:record-type)* &body *body* *Macro*

Establishes a ''row'' context on the *stream* (the default is ***standard-output***). All output performed on the stream within the extent of this macro will become the contents of one row of a table. **clim:formatting-row** must be used within the extent of **clim:formatting-table**, and it must be used in conjunction with **clim:formatting-cell**.

For background information, examples, and overviews of related functions, see the section "Formatting Tables in CLIM".

stream

The stream to which output should be sent. The default is ***standard-output***.

:record-type

This option is useful when you have defined a customized record type to replace CLIM's default record type. It specifies the class of the output record to be created.

clim:formatting-table *(*&optional *stream* &rest *options* &key *:x-spacing :y-spacing :record-type :multiple-columns :multiple-columns-x-spacing :equalize-column-widths (:move-cursor* **t***))* &body *body* *Macro*

Establishes a ''table formatting'' context on the *stream* (the default is ***standard-output***). All output performed within the extent of this macro will be displayed in tabular form. This must be used in conjunction with **clim:formatting-row** or **clim:formatting-column**, and **clim:formatting-cell**.

For background information, examples, and overviews of related functions, see the section "Formatting Tables in CLIM".

stream

The stream to which output should be sent. The default is ***standard-output***.

:x-spacing

Determines the amount of space inserted between columns of the table; the default is the width of a space character. *:x-spacing* can be specified in one of the following ways:

Integer

A size in the current units to be used for spacing.

String or character
: The spacing is the width or height of the string or character in the current text style.

Function
: The spacing is the amount of horizontal or vertical space the function would consume when called on the stream.

List of form (*number unit*)
: The *unit* is **:point**, **:pixel**, or **:character**.

: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 *:x-spacing* option.

:multiple-columns
: Is either **nil**, **t**, or an integer. If it is **t** or an integer, the table rows are broken up into 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
: Controls the spacing between the multiple columns. This option defaults to the value of the *:x-spacing* option. It has the same format as *:x-spacing*.

:equalize-column-widths
: When **t**, CLIM makes all the columns have the same width, which is the width of the widest cell in any column of the table.

:record-type
: This option is useful when you have defined a customized record type to replace CLIM's default record type. It specifies the class of the output record to be created.

:move-cursor
: When **t** (the default), CLIM moves the text cursor to the end (lower right corner) of the output. Otherwise, the cursor is left at the beginning (upper left corner) of the output.

clim:frame-all-layouts *frame* *Generic Function*

Returns a list of all of the layout names for *frame*. These are the names of the layouts that you use when you call **setf** on **clim:frame-current-layout**.

clim:frame-command-table *frame* *Generic Function*

Returns the name of the command table currently being used by the frame *frame*. You can use this function with **setf** to change the command table to be used.

clim:frame-current-layout *frame* *Generic Function*

Returns the name of the current layout for *frame*.

You can use **setf** on **clim:frame-current-layout** to change the current layout.

Note: changing the layout currently causes CLIM to throw out of the application's command loop, all the way back to **clim:run-frame-top-level**. This is done so that CLIM can perform some window management functions, such as rebinding I/O streams that correspond to the windows in the new layout. Therefore, when you call **setf** on **clim:frame-current-layout**, you should only do so *after* you have done everything else in the sequence of operations and the application is prepared to return to command input.

clim:frame-current-panes *frame* *Generic Function*

Returns a list of all of the named CLIM stream panes that are contained in the current layout for the frame *frame*. The elements of the list will be CLIM pane objects.

clim:frame-error-output *frame* *Generic Function*

Returns the value that should be used for ***error-output*** for *frame*.

The default method (defined on **clim:application-frame**) uses the first pane of type **:application** in the current layout.

It can be useful to specialize **clim:frame-error-output** on your own frame class when you want to use a different pane for ***error-output***.

clim:default-frame-top-level calls this to determine what to bind ***error-output*** to.

clim:frame-exit *frame* *Generic Function*

Exits from the application frame *frame* by signalling a **clim:frame-exit** condition. If you call **clim:frame-exit** from a process that is different from the process in which *frame* is running, CLIM will queue the request for later execution in *frame*'s own process.

clim:frame-exit *Class*

The condition signalled by **clim:frame-exit**.

clim:frame-exit-frame *frame-exit* *Generic Function*

Returns the frame being exited from. *frame-exit* is a **clim:frame-exit** object.

clim:frame-find-innermost-applicable-presentation *frame input-context stream x y*
Generic Function

Locates and returns the innermost applicable presentation on the window *stream* at the pointer position indicated by *x* and *y*, in the input context *input-context*, on behalf of the application frame *frame*.

You can specialize this generic function for your own application frames. The default method calls **clim:find-innermost-applicable-presentation**.

clim:frame-input-context-button-press-handler *frame stream button-press-event*
Generic Function

This function is responsible for handling user pointer gestures on behalf of *frame*. *stream* is the window on which *button-press-event* took place.

The default method on **clim:standard-application-frame** calls **clim:frame-find-innermost-applicable-presentation** to find the innermost applicable presentation, and then calls **clim:throw-highlighted-presentation** to execute the translator that corresponds to the user's gesture.

clim:frame-maintain-presentation-histories *frame* *Generic Function*

Returns **t** if the *frame* maintains histories for its presentations, otherwise returns **nil**. The default method on the class **clim:standard-application-frame** returns **t** if and only if the frame has an interactor pane.

You can specialize this generic function for your own application frames.

clim:frame-manager *Class*

The protocol class that corresponds to a frame manager. If you want to create a new class that obeys the frame manager protocol, it must be a subclass of **clim:frame-manager**.

clim:frame-manager *object* *Function*

Given a CLIM object *object*, **clim:frame-manager** returns the frame manager associated with *object*. If *object* is not presently ''owned'' by any frame manager, **clim:frame-manager** will return **nil**.

You can call **clim:frame-manager** on sheets and frames.

clim:frame-manager-dialog-view *frame-manager* *Generic Function*

Returns the view object that should be used to control the look-and-feel of **clim:accepting-values** dialogs. Typically, this will be either **clim:+textual-dialog-view+** or **clim:+gadget-dialog-view+**.

You can use **setf** to change the default dialog view.

clim:frame-manager-p *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:frame-manager**, otherwise returns **nil**.

clim:frame-manager-palette *frame-manager* *Generic Function*

Returns the palette that will be used, by default, by all the frames managed by *frame-manager*, if those frame's don't have a palette of their own. You can use **setf** on this to change the frame manager's palette.

A palette is an object that contains mappings from color names (which are strings or symbols) to CLIM color objects. See the section "Predefined Color Names in CLIM".

The frame manager's palette defaults to the palette for its port, **clim:port-default-palette**.

clim:frame-name *frame* *Generic Function*

Returns the name of the application *frame*. The name is a symbol. You can change the name of an application frame by using **setf** on **clim:frame-name**.

clim:frame-palette *frame* *Generic Function*

Returns the palette associated with the application frame *frame*. The palette for a frame defaults to the default palette for the frame's frame manager.

A palette is an object that contains mappings from color names (which are strings or symbols) to CLIM color objects. See the section "Predefined Color Names in CLIM".

The frame's palette defaults to the palette for its frame manager, **clim:frame-manager-palette**.

clim:frame-panes *frame* *Generic Function*

Returns the single pane acting as the ''root'' pane for the current layout.

The pane returned by this function will typically be some sort of layout pane, such as a **clim:vbox-pane** that holds the menu bar and the rest of the panes, or an **clim:outlined-pane** that serves as the border around an application frame.

clim:frame-pointer-documentation-output *frame* *Generic Function*

Returns the value that should be used for **clim:*pointer-documentation-output*** for *frame*.

The default method (defined on **clim:application-frame**) uses the first pane of type **:pointer-documentation** in the current layout.

clim:default-frame-top-level calls this to determine what to bind **clim:*pointer-documentation-output*** to.

clim:frame-pretty-name *frame* *Generic Function*

Returns the pretty name of the application *frame*. The pretty name is a string. You can change the pretty name of an application frame by using **setf** on **clim:frame-pretty-name**.

clim:frame-replay *frame stream* &optional *region* *Generic Function*

Replays all of the output records in *stream*'s output history on behalf of the application frame *frame* that overlap the region *region*. If *region* is **nil**, all of the output records in the stream's viewport are replayed.

You can specialize this generic function for your own application frames. The default method for this calls **clim:stream-replay**.

clim:frame-query-io *frame* *Generic Function*

Returns the value that should be used for ***query-io*** for *frame*.

The default method (defined on **clim:standard-application-frame**) first tries to use the value returned by **clim:frame-standard-input**, and if it is **nil**, it uses the value returned by **clim:frame-standard-output**.

clim:default-frame-top-level calls this to determine what to bind ***query-io*** to.

clim:frame-standard-input *frame* *Generic Function*

Returns the value that should be used for ***standard-input*** for *frame*.

The default method (defined on **clim:standard-application-frame**) uses the first pane of type **clim:interactor-pane**. If there are no interactor panes, the value returned by **clim:frame-standard-output** is used.

It is often useful to specialize **clim:frame-standard-input** on your own frame class when you want to use a different pane for ***standard-input***.

clim:default-frame-top-level calls this to determine what to bind ***standard-input*** to.

clim:frame-standard-output *frame* *Generic Function*

Returns the value that should be used for ***standard-output*** for *frame*.

The default method (defined on **clim:standard-application-frame**) uses the first pane of type **clim:application-pane** in the current layout.

It is often useful to specialize **clim:frame-standard-output** on your own frame class when you want to use a different pane for ***standard-output***.

clim:default-frame-top-level calls this to determine what to bind ***standard-output*** to.

clim:frame-state *frame* *Generic Function*

Returns one of **:enabled**, **:disabled**, **:disowned**, or **:shrunk**, indicating the current state of frame.

:enabled means the frame currently enabled and visible on some port; this is the state that a frame is in when it is actively running. **clim:enable-frame** causes a frame to enter the **:enabled** state.

:disabled means that the frame is owned by a frame manager, but is not visible anywhere; this is the state that a frame is in when it has been exited but not destroyed. **clim:disable-frame** causes a frame to enter the **:disabled** state. When a frame is initially created, it starts in the **:disabled** state.

:disowned means that no frame manager owns the frame; this is the state a frame is in when it has been destroyed. **clim:destroy-frame** causes a frame to enter the **:disowned** state.

:shrunk means that the frame has been iconified.

clim:frame-top-level-sheet *frame* *Generic Function*

Returns the window that corresponds to the top level window for the frame *frame*. This is the window that has as its children all of the panes of the frame.

By default, all of the panes of an application frame share their event queue with the event queue of the frame's top level sheet.

clim:funcall-presentation-generic-function *presentation-function-name* &body *arguments* *Macro*

Funcalls the presentation generic function *presentation-function-name* with arguments *arguments* using **funcall**.

The *presentation-function-name* argument is not evaluated. The value of *presentation-function-name* can be any of the presentation generic functions defined by CLIM (**clim:accept**, **clim:present**, **clim:describe-presentation-type**, **clim:presentation-typep**, **clim:presentation-subtypep**, **clim:accept-present-default**, **clim:presentation-type-specifier-p**, **clim:presentation-refined-position-test**, or **clim:highlight-presentation**) or any presentation generic function you have defined yourself.

clim:gadget *Class*

The protocol class that corresponds to a gadget.

See the section "Using Gadgets in CLIM".

All subclasses of **clim:gadget** must handle the four initargs **:id**, **:client**, **:armed-callback**, and **:disarmed-callback**, which are used to specify, respectively, the gadget id, client, armed callback, and disarmed callback of the gadget.

The armed callback and disarmed callback are either **nil** or a function that takes a single argument, the gadget that was armed (or disarmed).

clim:gadget-active-p *gadget* *Generic Function*

Returns **t** if the gadget is active, that is, available for input. Otherwise, it returns **nil**.

clim:gadget-client *gadget* *Generic Function*

Returns the client of the gadget *gadget*. The client is usually an application frame, but it could be another gadget (for example, in the case of a push button that is contained in a radio box).

You can use **setf** on **clim:gadget-client** to change the gadget's client.

clim:gadget-dialog-view *Class*

The class that represents the view that is used inside toolkit-style **clim:accepting-values** dialogs.

The gadget dialog view is one example of an indirect view. When you use this view when calling **clim:accepting-values**, CLIM decodes the view into a more specific view based on the presentation type. These more specific views include **clim:+radio-box-view+**, **clim:+check-box-view+**, **clim:+toggle-button-view+**, **clim:+slider-view+**, **clim:+text-field-view+**, **clim:+text-editor-view+**, **clim:+list-pane-view+**, and **clim:+option-pane-view+**.

The following is a table of presentation types and the actual view they map to:

Type	*Gadget*
clim:completion	**clim:+radio-box-view+**
clim:subset-completion	**clim:+check-box-view+**
clim:boolean	**clim:+toggle-button-view+**
real	**clim:+slider-view+**
float	**clim:+slider-view+**
integer	**clim:+slider-view+**
All others	**clim:+text-field-view+**

Try the following example in a CLIM Lisp Listener.

```
(defun gadget-dialog-test (&optional (stream *standard-input*))
  (let ((dest  :file)
        (name  "")
        (strip nil))
    (clim:accepting-values (stream :align-prompts t)
      (setq dest  (clim:accept '(member :file :printer :window)
                    :default dest :prompt "Destination type"
                    :stream stream :view clim:+gadget-dialog-view+))
      (setq name  (clim:accept 'string
                    :default name :prompt "Destination name"
                    :stream stream :view clim:+text-field-view+))
      (setq strip (clim:accept 'boolean
                    :default strip :prompt "Strip text styles"
                    :stream stream :view clim:+gadget-dialog-view+)))
    (values dest name strip)))
```

clim:+gadget-dialog-view+ *Constant*

An instance of the class **clim:gadget-dialog-view**. Inside **clim:accepting-values**, the default view for the dialog stream may be bound to **clim:+gadget-dialog-view+**.

clim:gadget-id *gadget* *Generic Function*

Returns the gadget id of the gadget *gadget*. The id is typically a simple Lisp object that uniquely identifies the gadget.

You can use **setf** on **clim:gadget-id** to change the id of the gadget.

clim:gadget-label *labelled-gadget* *Generic Function*

Returns the label of the gadget *labelled-gadget*. The label must be a string or a pixmap.

You can use **setf** to change the label of a gadget, but this may result in invoking the layout protocol on the gadget and its ancestor sheets (that is, the entire application frame may be re-layed out).

clim:gadget-max-value *range-gadget* *Generic Function*

Returns the maximum value of the gadget *range-gadget*, such as a slider. It will be a real number.

You can use **setf** to change the maximum value of the gadget.

clim:gadget-min-value *range-gadget* *Generic Function*

Returns the minimum value of the gadget *range-gadget*, such as a slider. It will be a real number.

You can use **setf** to change the minimum value of the gadget.

clim:gadget-menu-view *Class*

The class that represents the view that is used inside toolkit-style menus.

clim:+gadget-menu-view+ *Constant*

An instance of the class **clim:gadget-menu-view**. Inside **clim:menu-choose**, the default view for the menu stream may be bound to **clim:+gadget-menu-view+**.

clim:gadget-orientation *oriented-gadget* *Generic Function*

Returns the orientation of the gadget *oriented-gadget*. Typically, this will be a keyword such as **:horizontal** or **:vertical**.

clim:gadget-value *gadget* *Generic Function*

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 for each individual gadget specifies how to interpret the value.)

You can use **setf** on **clim:gadget-value** to change the value of a gadget. When the value is changed, the value change callback will be called if you also specify **:invoke-callback t**.

For example, the following fragment is from a color chooser that has two sets of linked sliders, one of which selects the RGB components of a color and the other of which selects the IHS components. When a slider from one set is changed, the other slider set is updated. Updating the other slider set should not cause any callbacks to be invoked.

```
(defmethod update-ihs ((frame color-chooser))
  (with-slots (intensity hue saturation) frame
    (multiple-value-bind (ii hh ss) (clim:color-ihs (color frame))
      (setf (clim:gadget-value intensity :invoke-callback nil) ii)
      (setf (clim:gadget-value hue :invoke-callback nil) hh)
      (setf (clim:gadget-value saturation :invoke-callback nil) ss))))
```

clim:gadget-view *Class*

The class that represents gadget views. Gadgets views are used for toolkit-oriented applications.

clim:+gadget-view+ *Constant*

An instance of the class **clim:gadget-view**.

clim:gadgetp *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:gadget**.

clim:get-frame-pane *frame pane-name* &key *(:errorp* **t***)* *Function*

Returns the CLIM stream pane (that is, a pane that is a subclass of **clim:clim-stream-pane**) named by *pane-name* in *frame*. If *pane-name* is a pane object, it is simply returned.

If the pane named *pane-name* is not found and *:errorp* is **t**, an error will be signalled. Otherwise if *:errorp* is **nil**, the return value will be **nil**.

clim:global-command-table *Clim Command Table*

The ''global'' command table from which all command tables inherit.

clim:handle-event *sheet event* *Generic Function*

Handles the event *event* on behalf of *sheet*.

If you implement your own gadget classes, you will probably write one or more **clim:handle-event** methods that manage such things as pointer button presses, pointer motion into the gadget, and so on. For example, if you want to highlight a sheet in response to an event that informs it that the pointer has entered its territory, you might write a method on your sheet class and **clim:pointer-enter-event** that highlights the sheet, and another method on **clim:pointer-exit-event** that unhighlights the sheet.

This function is called by CLIM's event dispatcher, so you will not generally need to call this function yourself.

See the section "Using Gadgets in CLIM". See the section "Sheet Input Protocols".

clim:handle-repaint *sheet region* *Generic Function*

Implements repainting for a given sheet class. *sheet* is the sheet to repaint and *region* is the region to repaint.

If you implement your own gadget classes, you will probably write a **clim:handle-repaint** method that draws the gadget. This function is called by CLIM's event dispatcher, so you will not generally need to call this function yourself.

See the section "Using Gadgets in CLIM". See the section "Sheet Input Protocols".

clim:hbox-pane *Class*

The layout pane class that arranges its children in a horizontal row. **clim:horizontally** generates a pane of this type.

In addition to the usual sheet initargs (the space requirement initargs, **:foreground**, **:background**, and **:text-style**), this class supports two other initargs:

:spacing An integer that specifies the amount of space to leave between each of the child panes, in device units.

:contents A list of panes that will be the child panes of the box pane.

clim:*help-gestures* *Variable*

A list of gesture names that cause **clim:accept** and **clim:complete-input** to display a help message, and, for some presentation types, the list of possibilities. On Genera, this includes the gesture corresponding to the `#\Help` character.

clim:highlight-applicable-presentation *frame stream input-context* &optional *(prefer-pointer-window* **t***)* *Function*

This is the ''input wait'' handler used by **clim:with-input-context**. It is responsible for locating the innermost applicable presentation, unhighlighting presentations that are not applicable, and highlighting the presentation that is applicable, if there is one.

frame is the application frame, *stream* is the output recording stream, and *input-context* is the input context.

When *prefer-pointer-window* is **t** (the default), CLIM will use the window under the pointer instead of *stream*. This is the usual behavior.

clim:highlight-output-record *record stream state* *Generic Function*

CLIM calls this method in order to draw highlighting for the output record *record* on *stream*. *state* is either **:highlight** (meaning to draw the highlighting) or **:unhighlight** (meaning to erase the highlighting).

The default method (on CLIM's basic output record class) simply draws a rectangle around the bounding rectangle of *record*. Some displayed output records provide their own methods, for example, output records for ellipses implement a method that draws a slightly larger ellipse around the ellipse being highlighted.

The following example shows how you might implement this method for an output record that records a filled-in circle:

```
(defmethod clim:highlight-output-record
           ((record circle-output-record) stream state)
  (declare (ignore state))
  (multiple-value-bind (xoff yoff)
      (clim:convert-from-relative-to-absolute-coordinates
        stream (clim:output-record-parent record))
    (with-slots (center-x center-y radius) record
      (clim:with-output-recording-options (stream :record nil)
        (clim:draw-circle*
           stream (+ center-x xoff) (+ center-y yoff) (+ radius 2)
           :filled nil :ink clim:+flipping-ink+)))))
```

clim:highlight-presentation *type-key parameters options type record stream state*
Clim Presentation Method

This method is responsible for drawing a highlighting box for the presentation *record* on the stream *stream*. *state* will be either **:highlight** or **:unhighlight**, meaning that the highlighting box should either be drawn or erased.

It can be useful to define a **clim:highlight-presentation** method when you want to have special highlighting behavior, such as ‘‘inverse video’’, for a presentation type.

Here is an example from an application that displays the floor plan of a suite of offices. In one pane, there is a list of the names of the people in the offices, and in another pane there is the floor plan itself. When a user waves the mouse over either pane, the relevant office and names are highlighted in both panes.

```
(clim:define-presentation-type room ())

(clim:define-presentation-method clim:highlight-presentation
    ((type room) record stream state)
  (declare (ignore state))
  (let ((room (clim:presentation-object record)))
    (clim:draw-polygon* stream (room-coords room)
                        :ink clim:+flipping-ink+)
    (when (room-associated-people room)
      (let ((stream (clim:get-frame-pane clim:*application-frame* 'people)))
        (clim:with-output-recording-options (stream :record nil)
          (dolist (person (room-associated-people room))
            (let ((person-presentation (person-directory-presentation person)))
              (when person-presentation
                (clim:with-bounding-rectangle* (rl rt rr rb)
                    person-presentation
                  (clim:draw-rectangle* stream rl rt rr rb
                                        :ink clim:+flipping-ink+)))))))))))

(clim:define-presentation-type person ())
```

```
(clim:define-presentation-method clim:highlight-presentation
    ((type person) record stream state)
  (declare (ignore state))
  (clim:with-bounding-rectangle* (rl rt rr rb) record
    (clim:draw-rectangle* stream rl rt rr rb
                          :ink clim:+flipping-ink+))
  (let ((person (clim:presentation-object record)))
    (when (person-office person)
      (let ((stream (clim:get-frame-pane clim:*application-frame* 'map))
            (office (person-office person)))
        (clim:with-output-recording-options (stream :record nil)
          (clim:draw-polygon* stream (room-coords office)
                              :ink clim:+flipping-ink+))))))
```

clim:horizontally *(*&rest *options* &key *:spacing* &allow-other-keys*)* &body *contents*
Macro

The **clim:horizontally** macro lays out one or more child panes horizontally, from left to right. The **clim:horizontally** macro serves as the usual interface for creating an **clim:hbox-pane**.

:spacing is an integer that specifies how much space should be left between each child pane, in device units. *options* may include other pane initargs, such as space requirement options, **:foreground**, **:background**, **:text-style**, and so forth.

contents is one or more forms that produce the child panes. Each form in *contents* is of the form:

- A pane. The pane is inserted at this point and its space requirements are used to compute the size.
- A number. The specified number of device units should be allocated at this point.
- The symbol **clim:+fill+**. This means that an arbitrary amount of space can be absorbed at this point in the layout.
- 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 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:

```
(clim:horizontally ()
  (1/3 (clim:make-pane 'label-button-pane))
  (2/3 (clim:make-pane 'label-button-pane)))
```

would create a horizontal row of two ''label button'' panes. The first pane takes one-third of the space, and the second takes two-thirds of the space.

See the section "Using the :LAYOUTS Option to CLIM:DEFINE-APPLICATION-FRAME".

clim:+hyper-key+ *Constant*

The modifier state bit that corresponds to the user holding down the hyper key on the keyboard. See the section "Operators for Gestures in CLIM".

clim:+identity-transformation+ *Constant*

An instance of a transformation that is guaranteed to be an identity transformation, that is, the transformation that ''does nothing''.

clim:identity-transformation-p *transform* *Generic Function*

Returns **t** if *transform* is equal (in the sense of **clim:transformation-equal**) to the identity transformation, otherwise returns **nil**.

clim:immediate-rescan *input-editing-stream* *Generic Function*

Invokes a rescan operation immediately on *input-editing-stream* by ''throwing'' out to the beginning of the most recent invocation of **clim:with-input-editing**.

For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:indenting-output *(stream indentation* &key *(:move-cursor* **t**)) &body *body* *Function*

Binds *stream* to a stream that inserts whitespace at the beginning of each line, and writes the indented output to the stream that is the original value of *stream*.

stream The output stream.

indentation

What gets inserted at the beginning of each line output to the stream. Four possibilities exist:

integer The width in device units (for example, pixels).

string The spacing is the width of the string.

function The spacing is the amount of space the function would consume when called on the stream.

list The list is of the form (*number unit*), where *unit* is one of

:pixel	The width in pixels.
:point	The width in printers points.
:character	The width of the ''usual'' character in the stream's current text style.

:move-cursor
When **t** (the default), CLIM moves the text cursor to the end (lower right corner) of the output. Otherwise, the cursor is left at the beginning (upper left corner) of the output.

You should begin the body with `(terpri` *stream*`)` (or the equivalent) to position the stream to the initial indentation.

Note: if you use **clim:indenting-output** in conjunction with **clim:filling-output**, you should put the call to **clim:indenting-output** *outside* of the call to **clim:filling-output**. This is necessary because **clim:filling-output** does not track what sort of output is being done inside it, except for pure textual output.

clim:*input-context* *Variable*

The current input context, which describes the presentation type(s) currently being input by CLIM. **clim:*input-context*** gets bound by calls to **clim:with-input-context**.

clim:*input-context* is a list, each element of which is a list consisting of a presentation type and a tag. The tag corresponds to a point in the control structure of CLIM at which that input context for the presentation type was established. The first element of **clim:*input-context*** is the most recently establish context, and the last element is the oldest context.

The value of **clim:*input-context*** and all of the sublists within **clim:*input-context*** have dynamic extent.

clim:input-context-type *context-entry* *Function*

Given one element from **clim:*input-context***, *context-entry*, this returns the presentation type of the context entry.

clim:input-editor-format *input-editing-stream format-string* &rest *format-args* *Function*

This function is like **format**, except that it is intended to be called on input editing streams. It arranges to insert ''noise strings'' in the input editor's input buffer. You can use this to display in-line prompts in **clim:accept** methods.

For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:input-not-of-required-type *Condition*

This condition is signalled when CLIM gets input that does not satisfy the specified type while inside of **clim:accept**. It is built on **conditions:parse-error**.

clim:input-not-of-required-type *object type* *Function*

Reports that input does not satisfy the specified type. *object* is a parsed object or an unparsed token (a string). *type* is a presentation type specifier. This function does not return.

clim:input-stream-p *stream* *Generic Function*

Returns **t** if the object passed in is a CLIM basic input stream.

clim:input-editing-stream-p *object* *Generic Function*

Returns **t** if the *object* is a CLIM input editing stream, that is, the sort of stream created by **clim:with-input-editing**.

integer &optional *low high* *Clim Presentation Type*

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**.

clim-lisp:interactive-stream-p *stream* *Generic Function*

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 Common Lisp, except that in CLIM it is a generic function.

CLIM's input editor is intended to work only on interactive streams.

clim:interactor-pane *Class*

The pane class that is used to implement ''interactor'' panes. This is the type of pane created by **clim:make-clim-interactor-pane**, and corresponds to the pane type abbreviation **:interactor** in the **:panes** clause of **clim:define-application-frame**. The default method for **clim:frame-standard-input** will return the first pane of this type in a frame.

For **clim:interactor-pane**, the default for the **:display-time** option is **nil**, and the default for the **:scroll-bars** option is **:vertical**.

See the section "Using the **:panes** Option to **clim:define-application-frame**".

clim:invert-transformation *transform* *Generic Function*

Returns a transformation that is the inverse of *transform*. The result of composing a transformation with its inverse is the identity transformation.

If *transform* is singular, **clim:invert-transformation** signals the **clim:singular-transformation** condition, with a named restart that is invoked with a transformation and makes **clim: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.

With finite-precision arithmetic there are several 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.) **clim:invert-transformation** signals the **clim:singular-transformation** error for all of these cases.

clim:invertible-transformation-p *transform* *Generic Function*

Returns **t** if *transform* has an inverse, otherwise returns **nil**.

clim:invoke-with-drawing-options *stream continuation* &rest *options* *Generic Function*

The functional equivalent of **clim:with-drawing-options**. This binds the *stream*'s drawing options to the new drawing options specified by *options*. *options* may include any of the drawing options allowed in **clim:with-drawing-options**. When the drawing options are in effect, *continuation* is called. *continuation* is a function of zero arguments.

The returned value is the value returned by *continuation*.

See the macro **clim:with-drawing-options**.

clim:invoke-with-new-output-record *stream continuation record-type constructor* &rest *initargs* &key *:parent* &allow-other-keys *Generic Function*

The functional equivalent of **clim:with-new-output-record**. This creates a new output record of type *record-type* using the supplied *initargs*. If *constructor* is not **nil**, it is a constructor function that creates the new output record.

When the record has been created, *continuation* is called in order to create the contents of the record. *continuation* is a function of one argument, the stream.

The returned value is the value returned by *continuation*.

See the macro **clim:with-new-output-record**.

clim:invoke-with-output-recording-options *stream continuation record draw* *Generic Function*

The functional equivalent of **clim:with-output-recording-options**. This binds recording and drawing state of the stream *stream* to the new state *record* and *draw*, and then calls the *continuation*. *continuation* is a function of zero arguments.

The returned value is the value returned by *continuation*.

See the macro **clim:with-output-recording-options**.

clim:invoke-with-text-style *medium style continuation original-stream*
Generic Function

The functional equivalent of **clim:with-text-style**. This binds the current text style of *medium* to the new text style *style*, and then calls the *continuation*. *continuation* is a function of one argument, the stream.

The returned value is the value returned by *continuation*.

See the macro **clim:with-text-style**.

clim:key-press-event *Class*

The class that corresponds to pressing a key on the keyboard. This is a subclass of **clim:keyboard-event**.

clim:key-release-event *Class*

The class that corresponds to releasing a key on the keyboard. This is a subclass of **clim:keyboard-event**.

clim:keyboard-event *Class*

The class that corresponds to a keyboard event. This is a subclass of **clim:device-event**.

clim:keyboard-event-character *keyboard-event* *Generic Function*

Returns the character corresponding to the key that was pressed or released, if there is a corresponding character in the Common Lisp character set. If there is no corresponding Common Lisp character, this will return **nil**.

clim:keyboard-event-key-name *keyboard-event* *Generic Function*

Returns the name of the key that was pressed or released in order to generate the keyboard event. The name of the key will be a symbol.

keyword *Clim Presentation Type*

The presentation type that represents a symbol in the **keyword** package. It is a subtype of **symbol**.

clim:labelled-gadget-mixin *Class*

The class that is mixed into a gadget that has a label, for example, a push button. The label may be a string, a pattern, or a pixmap.

All subclasses of **clim:labelled-gadget-mixin** must handle the initargs **:label**, **:alignment**, and **:text-style**, which are used to specify the label, the label's alignment within the gadget, and the label's text style.

clim:labelling *(*&rest *options* &key *:label (:label-alignment* **:bottom***)* &allow-other-keys*)* &body *contents* *Macro*

Creates a vertical stack consisting of two panes. One pane contains the specified label, which is a string, a pattern, or a pixmap. The other pane is specified by *contents*.

The **clim:labelling** macro is the usual way of creating a pane of type **clim:label-pane**.

options may include other pane initargs, such as space requirement options, **:foreground**, **:background**, **:text-style**, and so forth.

clim:line *Class*

The protocol class that corresponds to a mathematical line-segment, that is, a polyline with only a single segment. This is a subclass of **clim:polyline**. If you want to create a new class that obeys the line protocol, it must be a subclass of **clim:line**.

clim:linep *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:line**.

clim:line-end-point *line* *Generic Function*

Returns the ending point of *line*.

clim:line-end-point* *line* *Generic Function*

Returns the ending point of *line* as two values representing the coordinate pair.

clim:line-start-point *line* *Generic Function*

Returns the starting point of *line*.

clim:line-start-point* *line* *Generic Function*

Returns the starting point of *line* as two values representing the coordinate pair.

clim:line-style *Class*

The class that represents line styles.

clim:line-style-cap-shape *line-style* *Generic Function*

Returns the cap shape component of a line style object. This will be one of **:butt**, **:square**, **:round**, or **:no-end-point**. See the section "CLIM Line Style Suboptions".

clim:line-style-dashes *line-style* *Generic Function*

Returns the dashes component of a line style object. This will be **nil** to indicate a solid line, **t** to indicate a dashed line whose dash pattern is unspecified, or will be a sequence specifying some sort of a dash pattern. See the section "CLIM Line Style Suboptions".

clim:line-style-joint-shape *line-style* *Generic Function*

Returns the joint shape component of a line style object. This will be one of **:miter**, **:bevel**, **:round**, or **:none**. See the section "CLIM Line Style Suboptions".

clim:line-style-p *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:line-style**.

clim:line-style-thickness *line-style* *Generic Function*

Returns the thickness component of a line style object, which is an integer. See the section "CLIM Line Style Suboptions".

clim:line-style-unit *line-style* *Generic Function*

Returns the unit component of a line style object, which will be one of **:normal** or **:point**. See the section "CLIM Line Style Suboptions".

clim:list-pane *Class*

The **clim:list-pane** gadget class corresponds to a list pane, that is, a pane whose semantics are similar to a radio box or check box, but whose visual appearance is a list of buttons. It is a subclass of **clim:value-gadget**.

See the section "Using Gadgets in CLIM".

In addition to the initargs for **clim:value-gadget** and the usual pane initargs (**:foreground**, **:background**, **:text-style**, space requirement options, and so forth), the following initargs are supported:

:mode Either **:one-of** or **:some-of**. When it is **:one-of**, the list pane acts like a radio box, that is, only a single item can be selected. Otherwise, the list pane acts like a check box, in that zero or more items can be selected. The default is **:one-of**.

:items A list of items.

:name-key
A function of one argument that generate the name of an item from the item. The default is **princ-to-string**.

:value-key
A function of one argument that generate the value of an item from the item. The default is **identity**.

:test A function of two arguments that compares two items. The default is **eql**.

Calling **clim:gadget-value** on a list pane will return the single selected item when the mode is **:one-of**, or a sequence of selected items when the mode is **:some-of**.

The **clim:value-changed-callback** is invoked when the select item (or items) is changed.

Here are some examples of list panes:

```
(clim:make-pane 'clim:list-pane
  :value "Symbolics"
  :test 'string=
  :value-changed-callback 'list-pane-changed-callback
  :items '("Franz" "Lucid" "Harlequin" "Symbolics"))

(clim:make-pane 'clim:list-pane
  :value '("Lisp" "C++")
  :mode :some-of
  :value-changed-callback 'list-pane-changed-callback
  :items '("Lisp" "Fortran" "C" "C++" "Cobol" "Ada"))

(defun list-pane-changed-callback (tf value)
  (format t "~&List pane ~A changed to ~S"  tf value))
```

clim:list-pane-view *Class*

The class that represents the view corresponding to a list pane. List panes are another way of representing ‘‘one of’’ or ‘‘some of’’ fields.

clim:+list-pane-view+ *Constant*

An instance of the class **clim:list-pane-view**.

clim:lookup-keystroke-command-item *keystroke command-table* &key *:test (:numeric-argument* **1***)* *Function*

This is like **clim:lookup-keystroke-item**, except that it searches only for enabled commands. If it cannot find an accelerator associated with an enabled command, **clim:lookup-keystroke-command-item** returns **nil**.

:test is a function of two arguments used to compare the keystroke to the gestures in the command table. It defaults to **clim:event-matches-gesture-name-p**.

:numeric-argument is the accumulated numeric argument. It will be substituted for any occurrences of **clim:*numeric-argument-marker*** in the resulting command.

clim:lookup-keystroke-item *keystroke command-table* &key *:test* *Function*

This is like **clim:find-keystroke-item**, except that it descends into sub-menus in order to find a keystroke accelerator matching *keystroke*. If it cannot find any such accelerator, **clim:lookup-keystroke-item** returns **nil**.

:test is a function of two arguments used to compare the keystroke to the gestures in the command table. It defaults to **clim:event-matches-gesture-name-p**.

clim:make-3-point-transformation *point-1 point-2 point-3 point-1-image point-2-image point-3-image* *Function*

Makes a transformation that takes *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.)

It is an error for *point-1*, *point-2*, and *point-3* to be collinear; if they are collinear, the **clim:transformation-underspecified** condition is signalled. If *point-1-image*, *point-2-image*, and *point-3-image* are collinear, the resulting transformation will be singular, but this is not an error.

clim:make-3-point-transformation* *x1 y1 x2 y2 x3 y3 x1-image y1-image x2-image y2-image x3-image y3-image* *Function*

Makes a transformation that takes (*x1*, *y1*) into (*x1-image*, *y1-image*), (*x2*, *y2*) into (*x2-image*, *y2-image*) and (*x3*, *y3*) into (*x3-image*, *y3-image*). (Three non-collinear points and their images under the transformation are enough to specify any affine transformation.)

It is an error for (*x1*, *y1*), (*x2*, *y2*) and (*x3*, *y3*) to be collinear; if they are collinear, the **clim:transformation-underspecified** condition is signalled. 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.

clim:make-application-frame *frame-name* &key *:frame-class :pretty-name :parent :left :top :right :bottom :height :width* &allow-other-keys *Function*

Makes an instance of the application frame of type *:frame-class*. In addition to the keyword arguments listed, you can also supply initialization arguments for *:frame-class*. The keyword arguments not handled by **clim:make-application-frame** are passed as initargs to **clos:make-instance**.

frame-name
A symbol. This will be the same as one of the *name* arguments to **clim:define-application-frame**.

:frame-class
The class to instantiate, defaults to *frame-name*. For special purposes you can supply a subclass of *frame-name*.

:pretty-name
A string that is used as a title. It defaults to a ''prettified'' version of *frame-name*.

:parent The ''parent'' of the application. This can be either a port or a frame manager. If it is not supplied, it defaults to the current port. You should provide this argument when you want to create an application frame on some display server other than the current one, or with a non-default frame manager. See the function **clim:find-port** and the function **clim:find-frame-manager**.

:left, *:top*, *:right*, *:bottom*
The coordinates of the left, top, right, and bottom edges of the frame, in device units. *:left* and *:top* default to 0, and *:right* and *:bottom* default to **nil**.

:width, *:height*
The size of the frame in device units. *:width* and *:height* default so that the frame will fill the entire screen.

clim:make-bounding-rectangle *x1 y1 x2 y2* *Function*

Makes an object of class **clim:standard-bounding-rectangle** whose edges are parallel to the coordinate axes. One corner is at *(x1,y1)* and the opposite corner is at *(x2,y2)*.

The representation of rectangles in CLIM is chosen to be efficient. CLIM represents rectangles by storing the coordinates of two opposing corners of the rectangle. Because this representation is not sufficient to represent the result of arbitrary transformations of arbitrary rectangles, CLIM is allowed to return a polygon as the result of such a transformation. (The most general class of transformations that is guaranteed to always turn a rectangle object into another rectangle object is the class of transformations that satisfy **clim:rectilinear-transformation-p**.)

clim:make-ihs-color *intensity hue saturation* *Function*

Creates a color object. *intensity* is a real number between 0 and the square root of 3 inclusive. *hue* and *saturation* are real numbers between 0 and 1 inclusive.

clim:make-rgb-color *red green blue* *Function*

Creates a color object. *red*, *green*, and *blue* are real numbers between 0 and 1 inclusive that specify the values of the corresponding color components.

clim:make-clim-application-pane &rest *options* *Macro*

Like **clim:make-clim-stream-pane**, except that the type is forced to be **clim:application-pane**.

clim:make-clim-interactor-pane &rest *options* *Macro*

Like **clim:make-clim-stream-pane**, except that the type is forced to be **clim:interactor-pane**.

clim:make-clim-stream-pane &rest *options* &key *(:type* **'clim:clim-stream-pane***) :label (:label-alignment* **:bottom***) (:scroll-bars* **':vertical***) (:borders* **t***) :spacing :display-after-commands* &allow-other-keys *Macro*

Creates a pane of type *:type*, which defaults to **clim:clim-stream-pane**.

If *:label* is supplied, it is a string used to label the pane. *:label-alignment* is either **:bottom** or **:top**, and specifies whether to place the label at the bottom or top of the window.

: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. If *:scroll-bars* is **:none**, the pane will be scrollable but will not have scroll bars. If *:scroll-bars* is **nil**, the pane will not be scrollable.

If *:borders* is **t**, the pane will have a border drawn around it.

:display-after-commands may be **t** (meaning that the pane should be redisplayed after each command is executed), **nil** (meaning that the pane should not be redisplayed except by an explicit call to **clim:redisplay-frame-pane**), or **:no-clear** (which is like **t**, except that the pane is not cleared before the display function is called).

Other options may include all of the valid CLIM application pane options, including **:incremental-redisplay**, the space requirement initargs, **:foreground**, **:background**, **:text-style**, and so forth.

clim:make-command-table *name* &key *:inherit-from :menu :inherit-menu (:errorp* **t***)* *Function*

Creates a command table named *name* that inherits from *:inherit-from* and has a menu specified by *:menu*. *:inherit-from*, *:menu*, and *:inherit-menu* are as for **clim:define-command-table**. If the command table already exists and *:error-p* is **t**, then a **clim:command-table-already-exists** condition will be signalled.

clim:make-contrasting-dash-patterns *n* &optional *k* *Function*

Makes a simple vector of *n* dash patterns with recognizably different appearances.

If *k* (an integer between 0 and *n*-1) is supplied, **clim:make-contrasting-dash-patterns** returns the *k*'th dash pattern.

If the implementation does not have *n* different contrasting dash patterns, **clim:make-contrasting-dash-patterns** signals an error. This will not happen unless *n* is greater than sixteen.

clim:make-contrasting-inks *n* &optional *k* *Function*

Makes a simple vector of *n* inks with different appearances.

If *k* (an integer between 0 and *n*-1) is supplied, **clim:make-contrasting-inks** returns the *k*'th design.

If the implementation does not have *n* different contrasting inks, **clim:make-contrasting-inks** signals an error. This will not happen unless *n* is greater than eight.

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

clim:make-design-from-output-record *record* *Function*

Makes a design that replays the output record *record* when the design is drawn by **clim:draw-design**.

Presently, only output records whose displayed representation consists only of graphics (such as the output records created by **clim:draw-line*** and **clim:draw-ellipse***) can be turned into designs by **clim:make-design-from-output-record**.

You can use **clim:transform-region** on the result of **clim:make-design-from-output-record** in order to apply a transformation to it.

clim:make-ellipse *center-point radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle* *Function*

Makes an object of class **clim:standard-ellipse**. The center of the ellipse is *center-point*.

This function is the same as **clim:make-ellipse*** except that the location of the center of the ellipse is specified as a point rather than as X and Y coordinates.

clim:make-ellipse* *center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle* *Function*

Makes an object of class **clim:standard-ellipse**. The center of the ellipse is (*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 above. It is an error for those two vectors to be collinear (in order for the ellipse to be well-defined). 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 *: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 **2pi**; if neither is supplied then the region is a full ellipse and the angles are meaningless.

See the section "Ellipses and Elliptical Arcs in CLIM".

clim:make-elliptical-arc *center-point radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle* *Function*

Makes an object of class **clim:standard-elliptical-arc**. The center of the ellipse is *center-point*.

This function is the same as **clim:make-elliptical-arc*** except that the location of the center of the arc is specified as a point rather than as X and Y coordinates.

clim:make-elliptical-arc* *center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key *:start-angle :end-angle* *Function*

Makes an object of class **clim:standard-elliptical-arc**. The center of the ellipse is (*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 above. It is an error for those two vectors to be collinear (in order for the ellipse to be well-defined). The special case of an elliptical arc with its axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to 0.

The arc is swept from *: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 **2pi**; if neither is supplied then the region is a closed elliptical path and the angles are meaningless.

See the section "Ellipses and Elliptical Arcs in CLIM".

clim:make-flipping-ink *design1 design2* *Function*

Returns a design that interchanges occurrences of two designs. Drawing this design over a background changes the color in the background that would have been drawn by *design1* at that point into the color that would have been drawn by *design2* at that point, and vice versa.

In the present implementation, both designs must be colors. On the basic CLX port, only **clim:+flipping-ink+** is supported at present.

clim:make-gray-color *luminosity* *Function*

Creates a color object whose ''brightness'' is *luminosity*. *luminosity* is a real number between 0 and 1 inclusive. 0 means black and 1 means white.

clim:make-line *start-point end-point* *Function*

Makes an object of class **clim:standard-line** that connects *start-point* to *end-point*.

clim:make-line* *start-x start-y end-x end-y* *Function*

Makes an object of class **clim:standard-line** that connects (*start-x*, *start-y*) to (*end-x, end-y*).

clim:make-line-style &key *(:unit* **:normal***) (:thickness* **1***) :dashes (:joint-shape* **:miter***) (:cap-shape* **:butt***)* *Function*

Creates a line style object with the supplied characteristics.

For information about the keyword arguments, see the section "CLIM Line Style Suboptions".

These line style keywords correspond to the CLIM line style suboptions that begin with ''**:line-**'' (for example, **:unit** corresponds to the line style suboption **:line-unit**).

clim-sys:make-lock &optional *(lock-name* **"a CLIM lock"***)* *Function*

Creates a lock whose name is *name*. *name* is a string.

On systems that do not support locking, this will return a new list of one element, **nil**.

See the section "Locks in CLIM".

clim:make-modifier-state &rest *modifiers* *Function*

Given a set of modifier key names, **clim:make-modifier-state** returns a modifier state corresponding to those keys. The returned value is a fixnum.

The modifier key names may be zero or more of **:shift**, **:control**, **:meta**, **:super**, and **:hyper**. See the section "Gestures and Gesture Names in CLIM".

clim:make-opacity *value* *Function*

Creates a member of class **clim:opacity** whose opacity is *value*, which is a real number in the range from 0 to 1 (inclusive), where 0 is fully transparent and 1 is fully opaque.

clim:make-pane *pane-class* &rest *pane-options* *Function*

Selects a class that implements the behavior of the abstract pane *pane-class* and constructs a pane of that class. **clim:make-pane** must be used either within the dynamic scope of a call to **clim:with-look-and-feel-realization**, which is automatically established within the **:pane**, **:panes**, and **:layouts** options of a **clim:define-application-frame**.

clim:make-pane passes *pane-options* to the pane constructor function to be used as initargs for the pane.

See the section "Panes in CLIM". See the section "Using Gadgets in CLIM".

clim:make-pattern *array designs* *Function*

Creates a pattern design that has `(array-dimension 2d-array 0)` cells in the vertical direction and `(array-dimension 2d-array 1)` cells in the horizontal direction.

array must be a two-dimensional array of non-negative integers, each of which is less than the length of *designs*. *designs* must be a sequence of designs. The design in cell (i,j) of the resulting pattern is the *n*th element of *designs*, if *n* is the value of `(aref array i j)`. For example, *array* can be a bit-array and *designs* can be a list of two designs, the design drawn for 0 and the one drawn for 1.

Each cell of a pattern can be regarded as a hole that allows the design in it to show through. Each cell might have a different design in it. The portion of the design that shows through a hole is the portion on the part of the drawing plane where the hole is located. In other words, incorporating a design 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 designs 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.

Each cell of a pattern occupies a 1 by 1 square. You can use **clim:transform-region** to scale the pattern to a different cell size and shape, or to rotate the pattern so that the rectangular cells become diamond-shaped. Applying a coordinate transformation to a pattern does not affect the designs that make up the pattern.

It only changes the position, size, and shape of the cells' holes, allowing different portions of the designs in the cells to show through. Consequently, applying **clim:make-rectangular-tile** to a pattern of nonuniform designs can produce a different appearance in each tile. The pattern cells' holes are tiled, but the designs in the cells are not tiled and a different portion of each of those designs shows through in each tile.

If *array* or *designs* is modified after calling **clim:make-pattern**, the consequences are unspecified.

In the present implementation, patterned designs are not fully supported as a foreground or background, and the only patterned designs supported as the **:ink** drawing option are tilings of patterns of colors. In Cloe there is an additional restriction that the X offset and Y offset of the tiling must be 8.

The following creates a pattern that consists of a little arrow, which could be used as a command-line prompt.

```
(defvar *prompt-arrow*
        (clim:make-pattern
          #2A((0 0 0 0 0 0 0 0 0 0 0 0)
              (0 0 0 0 0 1 0 0 0 0 0 0)
              (0 0 0 0 0 1 1 0 0 0 0 0)
              (0 1 1 1 1 1 1 1 0 0 0 0)
              (0 1 1 1 1 1 1 1 1 0 0 0)
              (0 0 0 0 0 0 0 1 1 1 0 0)
              (0 0 0 0 0 0 0 0 1 1 1 0)
              (0 0 0 0 0 0 0 1 1 1 0 0)
              (0 1 1 1 1 1 1 1 1 0 0 0)
              (0 1 1 1 1 1 1 1 0 0 0 0)
              (0 0 0 0 0 1 1 0 0 0 0 0)
              (0 0 0 0 0 1 0 0 0 0 0 0))
          (list clim:+background-ink+ clim:+foreground-ink+)))
```

See the section "Drawing with Designs in CLIM".

clim:make-pattern-from-bitmap-file *pathname* &rest *args* &key *(:type* **:x11***) :designs :format* &allow-other-keys *Function*

Reads the bitmap file specified by *pathname* and creates a CLIM pattern object from it. The only file type currently supported is **:x11**, in the two formats **:bitmap** and **:pixmap**.

If the bitmap file did not supply colors for use in the image, you must also supply a sequence of designs. This has the same format as for **clim:make-pattern**. For a bitmap, you should probably supply the following as the value for the *:designs* argument:

```
(list clim:+background-ink+ clim:+foreground-ink+)
```

If you use **clim:+transparent-ink+** instead of **clim:+background-ink+**, the resulting pattern will be translucent instead of opaque.

The directory SYS:X11;INCLUDE;BITMAPS; (or its equivalent) usually has a number of X11 bitmap files in it. You might want try try calling **clim:make-pattern-from-bitmap-file** on some of the files in this directory.

clim:make-point *x y* *Function*

Creates and returns a point object whose coordinates are *x* and *y*. The point object is an instance of **clim:standard-point**.

clim:make-polygon *point-seq* *Function*

Makes an object of class **clim:standard-polygon** consisting of the area contained in the boundary that is specified by the segments connecting each of the points in *point-seq*.

clim:make-polygon* *coord-seq* *Function*

Makes an object of class **clim:standard-polygon** consisting of the area contained in the boundary that is specified by the segments connecting each of the points represented by the coordinate pairs in *coord-seq*.

clim:make-polyline *point-seq* &key *:closed* *Function*

Makes an object of class **clim:standard-polyline** consisting of the segments connecting each of the points in *point-seq*.

If *:closed* is **t**, the segment connecting the first point and the last point is included in the polyline.

clim:make-polyline* *coord-seq* &key *:closed* *Function*

Makes an object of class **clim:standard-polyline** consisting of the segments connecting each of the points represented by the coordinate pairs in *coord-seq*.

If *:closed* is **t**, the segment connecting the first point and the last point is included in the polyline.

clim:make-presentation-type-specifier *type-name-and-parameters* &rest *options* *Function*

Given a presentation type name and its parameters *type-name-and-parameters* and some presentation type options, make a new presentation type specifier that includes all of the type parameters and options. This is useful for assembling a presentation type specifier with options equal to their default values omitted. This is useful for **clim:define-presentation-type-abbreviation**, but not for the **:inherit-from** clause of **clim:define-presentation-type**.

For example,

```
(clim:make-presentation-type-specifier '(integer 1 10) :base 10)
  ==> (integer 1 10)

(clim:make-presentation-type-specifier '(integer 1 10) :base 8)
  ==> ((integer 1 10) :base 8)
```

clim-sys:make-process *function* &key *:name* *Function*

Creates a process named *name*. The new process will evaluate the function *function* (that is, *function* will be its top-level function). *name* is a string, and *function* is a function of no arguments.

On systems that do not support multi-processing, **clim-sys:make-process** will signal an error.

The exact representation of a process object varies from one platform to another. **clim-sys:processp** will return **t** for any process object returned by **clim-sys:make-process**.

See the section "Multi-processing in CLIM".

clim:make-rectangle *min-point max-point* *Function*

Makes an object of class **clim:standard-rectangle** whose edges are parallel to the coordinate axes. One corner is at *min-point* and the opposite corner is at *max-point*.

clim:make-rectangle* *min-x min-y max-x max-y* *Function*

Makes an object of class **clim:standard-rectangle** whose edges are parallel to the coordinate axes. One corner is at *(min-x, min-y)* and the opposite corner is at *(max-x,max-y)*.

The representation of rectangles in CLIM is chosen to be efficient. CLIM represents rectangles by storing the coordinates of two opposing corners of the rectangle. Because this representation is not sufficient to represent the result of arbitrary transformations of arbitrary rectangles, CLIM is allowed to return a polygon as the result of such a transformation. (The most general class of transformations that is guaranteed to always turn a rectangle object into another rectangle object is the class of transformations that satisfy **clim:rectilinear-transformation-p**.)

clim:make-rectangular-tile *design width height* *Function*

Creates a design that tiles the specified rectangular portion of *design* across the entire drawing plane. The resulting design repeats with a period of *width* horizontally and *height* vertically. The portion of the argument *design* that appears in each tile is a rectangle whose top-left corner is at (0,0) and whose bottom-right corner is at (*width*,*height*).

The repetition of *design* is accomplished by applying a coordinate transformation to shift *design* into position for each tile, and then extracting an *width* by *height* portion of that design.

Applying a coordinate transformation to a rectangular tile does not change the portion of the argument *design* that appears in each tile. It can change the period, phase, and orientation of the repeated pattern of tiles.

For example, the following creates several ‘‘stipple’’ tilings that can used as an ink to draw filled-in graphics.

```
(defun make-stipple (height width patterns)
  (assert (= height (length patterns)) (height patterns)
          "Height should be same as number of patterns supplied")
  (check-type width (integer 0))
  (clim:make-rectangular-tile
    (clim:make-pattern (make-stipple-array height width patterns)
                       (vector clim:+background-ink+ clim:+foreground-ink+))
    width height))

(defun make-stipple-array (height width patterns)
  (let ((array (make-array (list height width) :element-type 'bit)))
    (let ((h -1))
      (dolist (pattern patterns)
        (incf h)
        (let ((w width))
          (dotimes (pos width)
            (decf w)
            (setf (aref array h w) (ldb (byte 1 pos) pattern))))))
    array))

(defvar *tiles-stipple*
        (make-stipple 8 8 '(#b10000000
                            #b10000000
                            #b01000001
                            #b00111110
                            #b00001000
                            #b00001000
                            #b00010100
                            #b11100011)))
```

```
(defvar *hearts-stipple*
        (make-stipple 8 8 '(#b01101100
                            #b10010010
                            #b10010010
                            #b01000100
                            #b00101000
                            #b00010000
                            #b00000000
                            #b00000000)))

(defvar *parquet-stipple*
        (make-stipple 8 8 '(#b10000000
                            #b11000001
                            #b00100010
                            #b00011100
                            #b00001000
                            #b00010000
                            #b00100000
                            #b01000000)))
```

See the section "Drawing with Designs in CLIM".

clim-sys:make-recursive-lock &optional *(lock-name* **"a recursive CLIM lock"***)*
Function

Creates a recursive lock whose name is *name*. *name* is a string. A recursive lock differs from an ordinary lock in that a process that already holds the recursive lock can call **clim-sys:with-recursive-lock-held** on the same lock without blocking.

On systems that do not support locking, this will return a new list of one element, **nil**.

See the section "Locks in CLIM".

clim:make-reflection-transformation *point-1 point-2* *Function*

Makes a transformation that reflects every point through the line passing through the points *point-1* and *point-2*.

clim:make-reflection-transformation* *x1 y1 x2 y2* *Function*

Makes a transformation that reflects every point through the line passing through the points (*x1*, *y1*) and (*x2*, *y2*).

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’’.

clim:make-rotation-transformation *angle* &optional *origin* *Function*

Makes a transformation that rotates all points clockwise by *angle* around the point *origin*. The angle is specified in radians. If *origin* is supplied it must be a point; if not supplied it defaults to (0,0).

clim:make-rotation-transformation* *angle origin-x origin-y* *Function*

Makes a transformation that rotates all points clockwise by *angle* around the point, (*origin-x, origin-y*). The angle is specified in radians.

A rotation is a transformation that preserves length and angles of all geometric entities. Rotations also preserve one point and the distance of all entities from that point.

clim:make-scaling-transformation *mx my* &optional *origin* *Function*

Makes a transformation that multiplies the X-coordinate distance of every point from *origin* by *mx* and the Y-coordinate distance of every point from *origin* by *my*. If *origin* is supplied it must be a point; if not supplied it defaults to (0,0).

clim:make-scaling-transformation* *mx my origin-x origin-y* *Function*

Makes a transformation that multiplies the X-coordinate distance of every point from *origin-x* by *mx* and the Y-coordinate distance of every point from *origin-y* by *my*.

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.

clim:make-space-requirement &key *(:width* **0***) (:min-width* **width***) (:max-width* **width***) (:height* **0***) (:min-height* **height***) (:max-height* **height***)* *Function*

Constructs and returns a space requirement object having the given components. The space requirement object will generally be used to indicate the preferred sizes for a pane.

:width specifies the preferred width of the pane, and *:height* specifies the preferred height.

:min-width specifies that minimum size a pane can take, and *:min-height* specifies the minimum height.

:max-width specifies the maximum width a pane can take, and *:max-height* specifies the maximum height. If the maximum width or height is **clim:+fill+**, the pane may stretch to fill the available space in that direction.

clim:make-stencil *array* *Function*

Creates a pattern design that has **(array-dimension** *array* **0)** cells vertically and **(array-dimension** *array* **1)** cells horizontally. *array* must be a two-dimensional array of real numbers between 0 and 1. The design in cell *(i,j)* of the resulting pattern is the value of the following:

```
(clim:make-opacity (aref array i j))
```

The stencil opacity of the result at a given point in the drawing plane depends on which cell that point falls in. If the point is in cell *(i,j)*, the stencil opacity is `(aref array i j)`. The stencil opacity is 0 outside the region defined by the array.

Each cell of a pattern occupies a 1 x 1 square. The entity protocol can be used to scale the pattern to a different cell size and shape, or to rotate the pattern so that the rectangular cells become diamond-shaped.

If *array* is modified after calling **clim:make-stencil**, the consequences are unspecified.

clim:make-text-style *family face size* *Function*

Creates a text style object with the supplied characteristics. Generally, there is no need to call **clim:make-text-style**; you should use **clim:parse-text-style** or **clim:merge-text-styles** instead.

family	One of **:fix**, **:serif**, **:sans-serif**, or **nil**.
face	One of **:roman**, **:bold**, **:italic**, **(:bold :italic)**, or **nil**.
size	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**.

For example,

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

=> **Here is a text-style example.**

A text style object is called *fully specified* if each of its components has a non-**nil** value, and the size component is not a relative size (that is, is neither **:smaller** nor **:larger**).

You can use **clim:text-style-family**, **clim:text-style-face**, and **clim:text-style-size** to extract components from a text style object.

See the section "Text Styles in CLIM".

clim:make-transformation *mxx mxy myx myy tx ty* *Function*

Makes a general transformation whose effect is,

$$x' = m_{xx}\, x + m_{xy}\, y + t_x$$

$$y' = m_{yx}\, x + m_{yy}\, y + t_y$$

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.

clim:make-translation-transformation *delta-x delta-y* *Function*

Makes a transformation that translates all points by *delta-x* in the X direction and *delta-y* in the Y direction.

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

clim:map-over-command-table-commands *function command-table* &key *(:inherited* **t***)* *Function*

Applies *function* to all of the commands accessible in *command-table*. *function* is a function that takes a single argument, the command name.

If *:inherited* is **nil** instead of **t**, this applies *function* only to those commands present in *command-table*, that is, it does not map over any inherited command tables.

clim:map-over-command-table-keystrokes *function command-table* *Function*

Applies *function* to all of the keystroke accelerators in *command-table*'s accelerator table. *function* is a function of three arguments, the menu name (which will be **nil** if there is none), the keystroke accelerator, and the menu item.

clim:map-over-command-table-keystrokes does not descend into sub-menus. If you require this behavior, you should examine the type of the menu item to see if it is **:menu**.

clim:map-over-command-table-menu-items *function command-table* *Function*

Applies *function* to all of the menu items in *command-table*'s menu. *function* is a function of three arguments, the menu name, the keystroke accelerator (which will be **nil** if there is none), and the menu item. The menu items are mapped in the order specified by **clim:add-menu-item-to-command-table**.

clim:map-over-command-table-menu-items does not descend into sub-menus. If you require this behavior, you should examine the type of the menu item to see if it is **:menu** and make the recursive call from *function*.

clim:map-over-command-table-names *function command-table* &key *(:inherited* **t***)* *Function*

Applies *function* to all of the command-line names accessible in *command-table*. *function* is a function of two arguments, the command-line name and the command name.

If *:inherited* is **nil** instead of **t**, this applies *function* only to those command-line names present in *command-table*, that is, it does not map over any inherited command tables.

clim:map-over-frames *function* &key *:port :frame-manager* *Generic Function*

Applies *function* to all of the application frames that ''match'' *:port* and *:frame-manager*. If *:frame-manager* is supplied, only those frames that use that frame manager match. If *:port* is supplied, only those frames that use that port match. If neither *:port* nor *:frame-manager* is supplied, **clim:map-over-frames** will call *function* on all of the existing application frames.

function is a function of one argument, the frame.

clim:map-over-output-records *function record* &optional *(x-offset* **0***) (y-offset* **0***)* &rest *continuation-args* *Function*

Applies *function* to all of the child output records in the output record *record*. Normally, *function* is called with a single argument, an output record. If *continuation-args* are supplied, they are passed to *function* as well.

x-offset and *y-offset* are output record offsets that are necessitated by CLIM's representation of output records. In a later release of CLIM, the representation of output records may change in such a way that the *x-offset* and *y-offset* arguments are removed.

clim:map-over-output-records-containing-position *function record x y* &optional *x-offset y-offset* &rest *continuation-args* *Generic Function*

Applies *function* to all of the child output records in the output record *record* that overlap the point (*x*,*y*). Normally, *function* is called with a single argument, an output record. If *continuation-args* are supplied, they are passed to *function* as well.

x-offset and *y-offset* are output record offsets that are necessitated by CLIM's representation of output records. In a later release of CLIM, the representation of output records may change in such a way that the *x-offset* and *y-offset* arguments are removed.

When **clim:map-over-output-records-containing-position** maps over the children in the record, it does so in such a way that, when it maps over overlapping children, the bottommost (least recently inserted) child is hit last. This is because this function is used for things like locating the presentation under the pointer, where the topmost child should be the one that is found.

Any class that is a subclass of **clim:output-record** must implement this method.

See the section "Output Recording in CLIM".

clim:map-over-output-records-overlapping-region *function record region* &optional *x-offset y-offset* &rest *continuation-args* *Generic Function*

Applies *function* to all of the child output records in the output record *record* that overlap the region *region*. Normally, *function* is called with a single argument, an output record. If *continuation-args* are supplied, they are passed to *function* as well.

x-offset and *y-offset* are output record offsets that are necessitated by CLIM's representation of output records. In a later release of CLIM, the representation of output records may change in such a way that the *x-offset* and *y-offset* arguments are removed.

When **clim:map-over-output-records-overlapping-region** maps over the children in the record, it does so in such a way that, when it maps over overlapping children, the topmost (most recently inserted) child is hit last. This is because this function is used for things such as replaying, where the most recently drawn thing must come out on top (that is, must be drawn last).

Any class that is a subclass of **clim:output-record** must implement this method.

See the section "Output Recording in CLIM".

clim:map-over-polygon-coordinates *function polygon* *Generic Function*

Applies *function* to all of the coordinates of the vertices of *polygon*. The *function* takes two arguments, the X and Y coordinates.

clim:map-over-polygon-segments *function polygon* *Generic Function*

Applies *function* to the line segments that compose *polygon*. The *function* takes four arguments, the X and Y coordinates of the start of the line segment, and the X and Y coordinates of the end of the line segment.

When **clim:map-over-polygon-segments** is called on a closed polyline, it will call *function* on the line segment that connects the last point back to the first point.

clim:map-over-ports *function* *Generic Function*

Applies *function* to all of the current ports. *function* is a function of one argument.

clim:map-over-region-set-regions *function region* &key *:normalize* *Generic Function*

Calls *function* on each region in *region*. This is often more efficient than calling **clim:region-set-regions**. *region* can be either a **clim:region-set** or any member of **clim:region**, in which case *function* is called once on *region* itself.

:normalize can be either **:x-banding** or **:y-banding**, and is interpreted as it is for **clim:region-set-regions**.

clim:map-over-sheets *function sheet* *Generic Function*

Applies *function* to *sheet*, and then applies *function* to all of the descendants (the children, the children's children, and so forth) of *sheet*. *function* is a function of one argument, the sheet.

clim:map-over-sheets-containing-position *function sheet x y* *Generic Function*

Applies *function* to all of the children of *sheet* containing the position *(x,y)*. *x* and *y* are expressed in *sheet's* coordinate system.

function is a function of one argument, the sheet.

clim:map-over-sheets-overlapping-region *function sheet region* *Generic Function*

Applies *function* to all of the children of *sheet* overlapping the region *region*. *region* is expressed in *sheet's* coordinate system.

function is a function of one argument, the sheet.

clim-sys:map-resource *function resource* *Function*

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*.

See the section "Resources in CLIM".

clim:map-sheet-position-to-parent *sheet x y* *Function*

Applies *sheet*'s transformation to the point *(x,y)*, returning the coordinates of that point in *sheet*'s parent's coordinate system.

See the section "Sheet Geometry Protocols".

clim:map-sheet-position-to-child *sheet x y* *Function*

Applies the inverse of *sheet*'s transformation to the point *(x,y)* (represented in *sheet*'s parent's coordinate system), returning the coordinates of that same point in *sheet*'s coordinate system.

See the section "Sheet Geometry Protocols".

clim:medium *Class*

The protocol class that corresponds to a medium. If you want to create a new class that obeys the medium protocol, it must be a subclass of **clim:medium**.

clim:medium-background *medium* *Generic Function*

Returns the current background design of the *medium*. You can use **setf** on **clim:medium-background** to change the background design. You must not set the background ink to an indirect ink.

For background information and related operations, see the section "Components of CLIM Mediums".

clim:medium-clipping-region *medium* *Generic Function*

Returns the current clipping region of the *medium*. You can use **setf** on **clim:medium-clipping-region** to change the clipping region.

In the current implementation of CLIM, the clipping region must be either a rectangle or a rectangle set.

For background information and related operations, see the section "Components of CLIM Mediums".

clim:medium-default-text-style *medium* *Generic Function*

The default text style for *medium*. **clim:medium-default-text-style** must be a fully specified text style, unlike **clim:medium-text-style** which can have null components. Any text styles that are not fully specified by the time they are used for rendering are merged against **clim:medium-default-text-style** using **clim:merge-text-styles**.

You can use **setf** on **clim:medium-default-text-style** to change the default text style, but the text style must be a fully specified text style.

See the section "Text Styles in CLIM".

clim:medium-draw-ellipse* *medium center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy start-angle end-angle filled* *Generic Function*

Draws an ellipse on the medium *medium*. The center of the ellipse is at (*x,y*), and the radii are specified by the two vectors (*radius-1-dx,radius-1-dy*) and (*radius-2-dx,radius-2-dy*). The center point and radii are transformed by the medium's current transformation.

start-angle and *end-angle* are real numbers that specify an arc rather than a complete ellipse. The medium transformation must be applied to the angles as well.

If *filled* is **t**, the ellipse is filled, otherwise it is not.

The ink, clipping region, and line style are gotten from the medium. See the section "The CLIM Drawing Environment".

clim:medium-draw-line* *medium x1 y1 x2 y2* *Generic Function*

Draws a line on the medium *medium*. The line is drawn from (*x1,y1*) to (*x2,y2*), with the start and end positions transformed by the medium's current transformation.

The ink, clipping region, and line style are gotten from the medium. See the section "The CLIM Drawing Environment".

clim:medium-draw-lines* *medium position-seq* *Generic Function*

Draws a set of disconnected lines on the medium *medium*. *position-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if the length of *position-seq* is not evenly divisible by 4. The coordinates in *position-seq* are transformed by the medium's current transformation.

The ink, clipping region, and line style are gotten from the medium. See the section "The CLIM Drawing Environment".

clim:medium-draw-point* *medium x y* *Generic Function*

Draws a point on the medium *medium*. The point is drawn at (*x*,*y*), transformed by the medium's current transformation.

The ink, clipping region, and line style are gotten from the medium. See the section "The CLIM Drawing Environment".

clim:medium-draw-points* *medium position-seq* *Generic Function*

Draws a set of points on the medium *medium*. *position-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *position-seq* does not contain an even number of elements. The coordinates in *position-seq* are transformed by the medium's current transformation.

The ink, clipping region, and line style are gotten from the medium. See the section "The CLIM Drawing Environment".

clim:medium-draw-polygon* *medium position-seq closed filled* *Generic Function*

Draws a polygon or polyline on the medium *medium*. *position-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *position-seq* does not contain an even number of elements. The coordinates in *position-seq* are transformed by the medium's current transformation.

If *filled* is **t**, the polygon is filled, otherwise it is not. If *closed* is **t**, the coordinates in *position-seq* are considered to define a closed polygon, otherwise the polygon will not be closed.

The ink, clipping region, and line style are gotten from the medium. See the section "The CLIM Drawing Environment".

clim:medium-draw-rectangle* *medium x1 y1 x2 y2 filled* *Generic Function*

Draws a rectangle on the medium *medium*. The corners of the rectangle are at (*x1*,*y1*) and (*x2*,*y2*), with the corner positions transformed by the medium's current transformation. If *filled* is **t**, the rectangle is filled, otherwise it is not.

The ink, clipping region, and line style are gotten from the medium. See the section "The CLIM Drawing Environment".

clim:medium-draw-rectangles* *medium position-seq filled* *Generic Function*

Draws a set of rectangles on the medium *medium*. *position-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if the length of *position-seq* is not evenly divisible by 4. The coordinates in *position-seq* are transformed by the medium's current transformation. If *filled* is **t**, the rectangle is filled, otherwise it is not.

The ink, clipping region, and line style are gotten from the medium. See the section "The CLIM Drawing Environment".

clim:medium-draw-text* *medium string-or-char x y start end align-x align-y towards-x towards-y transform-glyphs* *Generic Function*

Draws a character or a string on the medium *medium*. The text is drawn starting at (*x,y*), and towards (*toward-x,toward-y*); these positions are transformed by the medium's current transformation.

The ink, clipping region, and text style are gotten from the medium. See the section "The CLIM Drawing Environment".

clim:medium-drawable *medium* *Generic Function*

Returns the host window system object (or ''drawable'') that is drawn on by the CLIM drawing functions when they are called on *medium*. If *medium* is not associated with any sheet, or the sheet with which it is associated is not ''mirrored'' on a display server, this function will return **nil**.

When some part of a program must be able to draw on a window with maximum speed and portability is less important than performance, you can use **clim:medium-drawable** to get the host window system object and then draw directly on that using the host window system's drawing functions. See the macro **clim:with-medium-state-cached**.

clim:medium-foreground *medium* *Generic Function*

Returns the current foreground design of the *medium*. You can use **setf** on **clim:medium-foreground** to change the foreground design. You must not set the foreground ink to an indirect ink.

For background information and related operations, see the section "Components of CLIM Mediums".

clim:medium-ink *medium* *Generic Function*

Returns the current drawing ink of the *medium*. You can use **setf** on **clim:medium-ink** to change the current ink.

For background information and related operations, see the section "Components of CLIM Mediums".

clim:medium-line-style *medium* *Generic Function*

Returns the current line style of the *medium*. You can use **setf** on **clim:medium-line-style** to change the line style.

For background information and related operations, see the section "Components of CLIM Mediums".

clim:medium-merged-text-style *medium* *Generic Function*

Returns the fully merged text style that will be used when drawing text on *medium*. It returns the result of

```
(clim:merge-text-styles (clim:medium-text-style medium)
                        (clim:medium-default-text-style medium))
```

That is, the components of the current text style that are not **nil** will replace the defaults from medium's default text style. Unlike the preceding **clim:medium-text-style** and **clim:medium-default-text-style**, **clim:medium-merged-text-style** is read-only.

clim:medium-sheet *medium* *Generic Function*

Returns the sheet with which the medium *medium* is associated. See the section "Sheet Output Protocols".

clim:medium-text-style *medium* *Generic Function*

Returns the current text style of the *medium*. You can use **setf** on **clim:medium-text-style** to change the current text style.

For background information and related operations, see the section "Components of CLIM Mediums".

clim:medium-transformation *medium* *Generic Function*

Returns the current transformation of the *medium*. You can use **setf** on **clim:medium-transformation** to change the current transformation.

For background information and related operations, see the section "Components of CLIM Mediums".

clim:mediump *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:medium**, otherwise returns **nil**.

member &rest *elements* *Clim Presentation Type Abbreviation*

The presentation type that specifies one of *elements*. The options (**:name-key**, **:value-key**, and **:partial-completers**) are the same as for **clim:completion**.

clim:member-sequence *sequence* &key *:test* *Clim Presentation Type Abbreviation*

Like **member**, except that the set of possibilities is the sequence *sequence*. The parameter *:test* and the options (**:name-key**, **:value-key**, and **:partial-completers**) are the same as for **clim:completion**.

clim:member-alist *alist* &key *:test* *Clim Presentation Type Abbreviation*

Like **member**, except that the set of possibilities is the alist *alist*. Each element of *alist* is either an atom (as in **clim: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 containing one or more of the following properties:

  :value — the value
  :documentation — a descriptive string

The *:test* parameter and the options are the same as for **clim:completion** except that the **:value-key** and **:documentation-key** options default to functions that support the specified alist format.

clim:menu-choose *items* &rest *keys* &key *:associated-window :text-style :foreground :background :default-item :label :scroll-bars :printer :presentation-type :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 :x-position :y-position :pointer-documentation :menu-type* *Function*

Displays a menu with the choices in *items*. It returns three values: the value of the chosen item, the item itself, and the event corresponding to the gesture that the user used to select it. *items* can be a list or a general sequence.

If possible, **clim:menu-choose** will use the menu facilities provided by the host window system. It is generally possible for CLIM to use the native menu facilities when you do not supply any of the arguments that require the menu to be formatted in any special way.

When enabled by **clim:*abort-menus-when-buried***, this function returns **nil** for all values if the menu is aborted by burying it.

items A sequence of menu items. Each menu item has a visual representation derived from a display object, an internal representation which 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 described below.

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 — Specifies the value object.

:style — Specifies the text style used to **princ** the display object when neither the *:presentation-type* nor the *:printer* option is specified.

:items — Specifies an item list for a sub-menu used if this item is selected.

:documentation — Associates some documentation with the menu item. When *:pointer-documentation* is not **nil**, this documentation will be used as pointer documentation for the item.

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 **clim: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 write on. The *:printer* function should output some text or graphics at the stream's cursor position, but need not call **clim:present**. If neither *:presentation-type* nor *:printer* is supplied, the visual representation is produced by **princ** of the display object. 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
The CLIM window with which the menu is associated. This defaults to the top-level sheet of the current application frame. You should only rarely need to supply this argument.

:text-style A text style that defines how the menu items are presented.

:foreground and *:background*
These specify the default foreground and background for the menu. These default from the associated window.

:scroll-bars
This can be **nil**, **:none**, **:horizontal**, **:vertical**, or **:both**. The default is **:vertical**.

:label The string that the menu title will be set to.

:printer The function used to print the menu items in the menu. The function must take two arguments, the menu item and the stream to print it to. The default is a function that displays an ordinary menu item.

:presentation-type
Specifies the presentation type of the menu items. The default is **clim:menu-item**.

:max-width
Specifies the maximum width of the table display (in device units). (Can be overridden by *:n-rows*.)

:max-height
Specifies the maximum height of the table display (in device units). (Can be overridden by *:n-columns*.)

:n-rows Specifies the number of rows in the menu.

:n-columns
Specifies the number of columns in the menu.

:x-spacing
Determines the amount of space inserted between columns of the table; the default is the width of a space character. *:x-spacing* can be specified in one of the following ways:

Integer
A size in the current units to be used for spacing.

String or character
The spacing is the width or height of the string or character in the current text style.

Function
The spacing is the amount of horizontal or vertical space the function would consume when called on the stream.

List of form (*number unit*)
The *unit* is **:point**, **:pixel**, or **:character**.

: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 *:x-spacing* option.

:cell-align-x
Specifies the horizontal placement of the contents of the cell. Can be one of: **:left**, **:right**, or **:center**. The default is **:left**.

:cell-align-y
Specifies the vertical placement of the contents of the cell. Can be one of: **:top**, **:bottom**, or **:center**. The default is **:top**.

:pointer-documentation
Either **nil** (the default), meaning the no pointer documentation should be computed, or a stream on which pointer documentation should be displayed.

:x-position and *:y-position*
These can be supplied to position the menu. If the are not supplied, the menu will be positioned near the pointer.

:default-item
The menu item over which the mouse will appear.

:cache Indicates whether CLIM should cache this menu for later use. If **t**, then *:unique-id* and *:id-test* serve to uniquely identify this menu. Caching menus can speed up later uses of the same menu. The default is **nil**.

:unique-id If *:cache* is non-**nil**, this is used to identify the menu. It defaults to the *item*s, but can be set to a more efficient tag.

:id-test The function that compares *unique-ids*. Defaults to **equal**.

:cache-value
If *:cache* is non-**nil**, this is the value that is compared to see if the cached menu is still valid. It defaults to *item*s, but you may be able to supply a more efficient cache value than that.

:cache-test The function that compares *cache-values*. Defaults to **equal**.

See the section "Examples of Menus and Dialogs in CLIM".

clim:menu-choose-from-drawer *menu type drawer* &key *:x-position :y-position :cache :unique-id (:id-test* **#'equal**) *(:cache-value* **t**) *(:cache-test* **#'eql**) *:leave-menu-visible :default-presentation* *Function*

A lower-level routine for displaying menus that allows you much more flexibility in the menu layout. Unlike **clim:menu-choose**, which automatically creates and lays

out the menu, **clim:menu-choose-from-drawer** takes a programmer-provided window and drawing function. Then it draws the menu items into that window using the drawing function. The drawing function gets called with arguments (*stream type*). It can use *type* for its own purposes, the usual being to use it in a call to **clim:present**.

clim:menu-choose-from-drawer returns two values; the object the user clicked on, and the gesture.

You can create a temporary window for drawing their menu using **clim:with-menu**.

When enabled by **clim:*abort-menus-when-buried***, this function returns **nil** for all values if the menu is aborted by burying it.

menu The CLIM window to use for the menu.

type The presentation type of the mouse-sensitive items in the menu. This is the input context that will be established once the menu is displayed. For users who don't need to define their own types, a useful presentation-type is **clim:menu-item**.

drawer A function that takes arguments (*stream type*) that draws the contents of the menu.

:x-position The requested left edge of the menu (if supplied).

:y-position
The requested top edge of the menu (if supplied).

:leave-menu-visible
If non-**nil**, the window will not be deexposed once the selection has been made. The default is **nil**, meaning that the window will be deexposed once the selcetion has been made.

:default-presentation
Identifies 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 **clim:menu-choose**.

See the section "Examples of Menus and Dialogs in CLIM".

clim:menu-choose-command-from-command-table *command-table* &key *(:associated-window* **(clim:frame-top-level-window clim:*application-frame*)***)** *:text-style :label :cache (:unique-id* **clim:command-table***) (:id-test* **#'eql***) :cache-value (:cache-test* **#'eql***)* *Function*

Displays a menu of all of the commands in *command-table*'s menu, and waits for the user to choose one of the commands. The returned value is a command object. **clim:menu-choose-command-from-command-table** can invoke itself recursively if there are sub-menus.

:associated-window, *:text-style*, *:label*, *:cache*, *:unique-id*, *:id-test*, *:cache-value*, and *:cache-test* are as for **clim:menu-choose**.

clim:menu-command-parser *command-table stream* &key *:timeout* *Function*

Reads a command on behalf of an application frame's command loop by soliciting input via command menus. User programs should not call this function explicitly, but should rather bind **clim:*command-parser*** to it.

This is the function CLIM uses to parse commands in a menu driven interface.

clim:menu-read-remaining-arguments-for-partial-command *partial-command command-table stream start-location* &key *:for-accelerator* *Function*

Reads the remaining arguments of a partially filled-in command on behalf of an application frame's command loop by getting input via the mouse. User programs should not call this function explicitly, but should rather bind **clim:*partial-command-parser*** to it.

clim:merge-text-styles *style1 style2* *Generic Function*

Merges *style1* against the defaults provided by *style2*. That is, any **nil** components in *style1* are filled in from *style2*.

If the size component of *style1* is a relative size, the resulting size will be the size component of *style2* as modified by the relative size.

If the face component of *style1* is **:bold** and the face component of *style2* is **:italic** (or vice-versa), the resulting face will be **(:bold :italic)**.

Here are some examples of text style merging.

```
(clim:merge-text-styles '(:fix :bold 12) '(nil :roman nil))
=> #<STANDARD-TEXT-STYLE :FIX.:BOLD.12 1116707341>
(clim:merge-text-styles '(:fix :bold nil) '(nil :roman 10))
=> #<STANDARD-TEXT-STYLE :FIX.:BOLD.10 1116707675>
(clim:merge-text-styles '(:fix :bold 12) '(nil :italic 10))
=> #<STANDARD-TEXT-STYLE :FIX.(:BOLD :ITALIC).12 1116707454>
```

See the section "Text Styles in CLIM".

clim:+meta-key+ *Constant*

The modifier state bit that corresponds to the user holding down the meta key on the keyboard. See the section "Operators for Gestures in CLIM".

clim:modifier-state-matches-gesture-name-p *state gesture-name* *Function*

Returns **t** if the modifier state *state* ''matches'' the modifier state of the gesture named by *gesture-name*. *state* is an integer such as is returned by **clim:make-modifier-state**.

clim:move-sheet *sheet x y* *Generic Function*

Moves *sheet* to the new position (*x*,*y*). *x* and *y* are in coordinates relative to *sheet*'s parent.

clim:move-sheet works by modifying the sheet's transformation, and could be implemented as follows:

```
(defmethod clim:move-sheet
           ((sheet clim:basic-sheet) x y)
  (let ((transform (clim:sheet-transformation sheet)))
    (multiple-value-bind (old-x old-y)
        (clim:transform-position transform 0 0)
      (setf (clim:sheet-transformation sheet)
            (clim:compose-translation-with-transformation
              transform (- x old-x) (- y old-y))))))
```

You should not generally use this function to move a sheet, because it does not interact directly with the frame layout protocol. That is, moving a sheet with **clim:move-sheet** may appear not to have any effect until you invoke the entire layout protocol. If you need to move a top-level sheet, use **clim:position-sheet-carefully**.

clim:move-and-resize-sheet *sheet x y width height* *Generic Function*

Moves *sheet* to the new position (*x*,*y*), and simultaneously changes the size of the sheet to have width *width* and height *height*. *x* and *y* are in coordinates relative to *sheet*'s parent.

clim:move-and-resize-sheet could be implemented as follows:

```
(defmethod clim:move-and-resize-sheet
           ((sheet clim:basic-sheet) x y width height)
  (clim:move-sheet sheet x y)
  (clim:resize-sheet sheet width height))
```

You should not generally use this function to move or resize a sheet, because it does not interact directly with the frame layout protocol. That is, moving and resizing a sheet with **clim:move-and-resize-sheet** may appear not to have any effect until you invoke the entire layout protocol. If you need to move and resize a top-level sheet, use **clim:position-sheet-carefully** and **clim:size-frame-from-contents**.

clim-sys:*multiprocessing-p* *Variable*

The value of this variable is **t** if the current Lisp environment supports multiprocessing, otherwise it is **nil**.

clim:new-page *stream* *Generic Function*

When called on a PostScript output stream, this causes a PostScript ''newpage'' command to be included in the output at the point **clim:new-page** is called.

The exact effect of **clim:new-page** is undefined if you specified either **:multi-page t** or **:scale-to-fit t** to **clim:with-output-to-postscript-stream**.

See the section "Hardcopy Streams in CLIM".

clim:note-gadget-activated *client gadget* *Generic Function*

This function is invoked after a gadget is made active. You can specialize this generic function on the client or gadget in order to modify the behavior when the gadget is activated.

clim:note-gadget-deactivated *client gadget* *Generic Function*

This function is invoked after a gadget is made inactive. You can specialize this generic function on the client or gadget in order to modify the behavior when the gadget is deactivated.

clim:note-progress *numerator* &optional *(denominator* **1***) (note* **clim:*current-progress-note***)** *Function*

Notes the progress of an operation by updating the progress bar. This function is only used in the body of the **clim:noting-progress** macro. The progress bar is updated by fractional amounts between 0 and 1.

numerator is the numerator of the fraction by which to update the bar. *denominator* is the denominator of the fraction by which to update the bar; the default is 1.

note-var is a variable bound to the current note object; the default is **clim:*current-progress-note***.

```
(clim:noting-progress ("Working Away By Tenths")
  (loop for i from 0.1 to 1.0 by 0.1 do
    (tv:note-progress i)
    (sleep 1)))
```

clim:note-sheet-region-changed *sheet* *Generic Function*

This function is invoked whenever the region of *sheet* is changed. See the section "Sheet Geometry Protocols".

clim:note-sheet-transformation-changed *sheet* *Generic Function*

This function is invoked whenever the transformation of *sheet* is changed. See the section "Sheet Geometry Protocols".

clim:note-viewport-position-changed *frame pane x y* *Generic Function*

CLIM calls this function whenever a pane gets scrolled, whether it is scrolled programmatically (by **clim:window-set-viewport-position**, for example) or by a user gesture (such as clicking on a scroll bar).

pane is that pane that was scrolled, *frame* is the frame in which the pane resides, and *x* and *y* are the new viewport position.

You can specialized this function when you want to extend the scrolling behavior of a pane. This might be used to ''link'' to panes together, so that when one pane is scrolled, the other pane gets scrolled as well. If you want to implement completely new scrolling behavior for a sheet, you should specialize **clim:scroll-extent** instead.

clim:notify-user *frame message* &key *:associated-window :title :exit-boxes :text-style :foreground :background :x-position :y-position* *Generic Function*

Notifies the user of some event on behalf of the application frame *frame*. *message* is a message string. *:associated-window* is the window with which the notification is associated. *:title* is a string used to label the notification. *:exit-boxes* is as for **clim:accepting-values**.

:text-style, *:foreground*, and *:background* are the text style, foreground ink, and background ink to use in the notification window. *:x-position* and *:y-position* can be supplied to position the notification window.

On Genera, this pops up a small dialog box containing the message. On other platforms, this may use an alert box or the equivalent.

clim:noting-progress *(stream name* &optional *note-var)* &body *body* *Macro*

Binds the dynamic environment such that the progress of an operation performed within the body of the macro is noted by a progress bar displayed in the specified stream (such as the pointer documentation pane). The function **clim:note-progress** does the updating of the progress bar.

name is a string naming the operation being noted; this string is displayed with the progress bar.

note-var is a variable bound to the current note object; the default is **clim:*current-progress-note***.

```
(clim:noting-progress ("Working Away By Tenths")
  (loop for i from 0.1 to 1.0 by 0.1 do
    (tv:note-progress i)
    (sleep 1)))
```

clim:+nowhere+ *Constant*

The empty region (the opposite of **clim:+everywhere+**).

null *Clim Presentation Type*

The presentation type that represents ‘‘nothing’’. The single object associated with this type is **nil**, and its printed representation is "None".

clim:null-or-type *type* *Clim Presentation Type Abbreviation*

A compound type that is used to select **nil**, whose printed representation is the special token "None", or an object of type *type*.

type can be a presentation type abbreviation.

clim:*null-presentation* *Constant*

The ‘‘null’’ presentation, which occupies any part of a window where there are no presentations matching the current input context. The presentation type of this object is **clim:blank-area**.

number *Clim Presentation Type*

The presentation type that represents a general number. It is the supertype of all the number types.

clim:*numeric-argument-marker* *Variable*

If you are building a command object that has numeric arguments that have not yet been supplied (for example, a numeric argument gotten from a keystroke accelerator), CLIM uses the value of **clim:*numeric-argument-marker*** as a placeholder for those arguments. The command processor will fill in the places so marked by inserting the value of the input editor's accumulated numeric argument.

The following might come from a debugger. When the user types `control-5 control-N`, the debugger will move ‘‘down’’ five frames in the stack.

```
(define-debugger-command (com-next-frame :name t)
    (&key (n-frames '((integer) :base 10)
                    :default 1
                    :documentation "Move this many frames")
          (detailed 'boolean
                    :default nil :mentioned-default t
                    :documentation "Show locals and disassembled code"))
  "Show the next frame in the stack"
  (cond ((and (plusp n-frames)
              (bottom-frame-p (current-frame clim:*application-frame*)))
         (format t "~&You are already at the bottom of the stack."))
        (t
         (show-frame (nth-frame n-frames) :detailed detailed))))
```

```
(clim:add-keystroke-to-command-table
  'debugger '(:n :control) :command
  `(com-next-frame :n-frames ,clim:*numeric-argument-marker*))
```

clim:opacity *Class*

A member of the class **clim:opacity** is a completely colorless design that is typically used as the second argument to **clim:compose-in** to adjust the opacity of another design.

clim:opacity-value *opacity* *Generic Function*

Returns the *value* of *opacity*, which is a real number in the range from 0 to 1 (inclusive).

clim:opacityp *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:opacity**.

clim:open-stream-p *stream* *Generic Function*

In CLIM, **open-stream-p** is defined as a generic function. Otherwise, it behaves the same as the normal Common Lisp **open-stream-p** function.

clim:open-window-stream &key *:port :frame-manager :left :top :right :bottom :width :height :foreground :background :text-style (:vertical-spacing* **2***) (:end-of-line-action* **:allow***) (:end-of-page-action* **:allow***) :output-record (:draw* **t***) (:record* **t***) (:initial-cursor-visibility* **:off***) :text-margin :default-text-margin :save-under :input-buffer (:scroll-bars* **:vertical***) :borders :label* *Function*

clim:open-window-stream is a convenient interface for creating a CLIM window outside of an application frame. This function is not used often. Instead, you will usually use windows that are created by an application frame or by the menu and dialog functions.

Note that some of these keyword arguments are also available as pane options in **clim:define-application-frame**.

:port, *:frame-manager*
: The port or frame manager on which to create the window. You must supply one of these two arguments.

:left, *:top*, *:right*, *:bottom*, *:width*, *:height*
: Position and shape of the window within its parent, in device units. The default is locate the window at (0,0) and to fill the entire parent.

:foreground, *:background*, *:text-style*, *:draw*, *:record*, *:end-of-line-action*, *:end-of-page-action*, *:text-margin*
Initial values for the corresponding stream attributes.

:default-text-margin Text margin to use if **clim:stream-text-margin** isn't set. This defaults to the width of the viewport.

:borders Controls whether borders are drawn around the window (**t** or **nil**). The default is **t**.

:initial-cursor-visibility
:off (the default) means make the cursor visible if the window is waiting for input. **:on** means make it visible all the time. **nil** means the cursor is never visible.

:label **nil** or a string label for the window. The default is **nil** for no label.

:output-record Specify this if you want a different output history mechanism than the default.

:scroll-bars Indicates whether the new window should have scroll bars. One of **nil**, **:none**, **:vertical**, **:horizontal**, or **:both**. The default is **:both**.

:vertical-spacing Amount of extra space between text lines, in device units.

:input-buffer The event queue to use for this window.

The remaining keyword arguments are internal and should not be used.

You can use **clim:position-sheet-carefully** and **clim:size-frame-from-contents** to set the size and position of windows created using **clim:open-window-stream**.

clim:option-pane *Class*

The **clim:option-pane** gadget class corresponds to an option pane, that is, a pane whose semantics are identical to a list pane, but whose visual appearance is a single push button which, when pressed, pops up a menu of selections. It is a subclass of **clim:value-gadget**.

See the section "Using Gadgets in CLIM".

In addition to the initargs for **clim:value-gadget** and the usual pane initargs (**:foreground**, **:background**, **:text-style**, space requirement options, and so forth), the following initargs are supported:

:mode Either **:one-of** or **:some-of**. When it is **:one-of**, the option pane acts like a radio box, that is, only a single item can be selected. Otherwise, the option pane acts like a check box, in that zero or more items can be selected. The default is **:one-of**.

:items A list of items.

:name-key
A function of one argument that generate the name of an item from the item. The default is **princ-to-string**.

:value-key
A function of one argument that generate the value of an item from the item. The default is **identity**.

:test A function of two arguments that compares two items. The default is **eql**.

Calling **clim:gadget-value** on an option pane will return the single selected item when the mode is **:one-of**, or a sequence of selected items when the mode is **:some-of**.

The **clim:value-changed-callback** is invoked when the select item (or items) is changed.

Here are some examples of option panes:

```
(clim:make-pane 'clim:option-pane
  :label "Select a vendor"
  :value "Symbolics"
  :test 'string=
  :value-changed-callback 'option-pane-changed-callback
  :items '("Franz" "Lucid" "Harlequin" "Symbolics"))

(clim:make-pane 'clim:option-pane
  :label "Select some languages"
  :value '("Lisp" "C++")
  :mode :some-of
  :value-changed-callback 'option-pane-changed-callback
  :items '("Lisp" "Fortran" "C" "C++" "Cobol" "Ada"))

(defun option-pane-changed-callback (tf value)
  (format t "~&Option menu ~A changed to ~S"  tf value))
```

clim:option-pane-view *Class*

The class that represents the view corresponding to an option pane. Options panes are another way of representing ‘‘one of’’ or ‘‘some of’’ fields.

clim:+option-pane-view+ *Constant*

An instance of the class **clim:option-pane-view**.

or &rest *types* *Clim Presentation Type*

The presentation type that is used to specify one of several types, for example,

```
(or (member :all :none) integer)
```

clim:accept returns one of the possible types as its second value, not the original **or** presentation type specifier.

The elements of *types* can be presentation type abbreviations.

The **clim:accept** method for the **or** type works by iteratively calling **clim:accept** on each of the presentation types in *types*. It establishes a condition handler for **user::parse-error**, calls **clim:accept**, and returns the result if no condition is signalled. If a **user::parse-error** condition is signalled, CLIM calls the **clim:accept** method for the next type. If all of the calls to **clim:accept** fail, the **clim:accept** method for **or** signals a **user::parse-error**.

clim:oriented-gadget-mixin *Class*

The class that is mixed in to a gadget that has an orientation associated with it, for example, a slider.

All subclasses of **clim:oriented-gadget-mixin** must handle the **:orientation** initarg, which is used to specify the orientation of the gadget.

clim:outlining *(*&rest *options* &key *:thickness* &allow-other-keys*)* &body *contents* *Macro*

The **clim:outlining** layout macro puts an outlined border of the specified thickness around a single child pane. *:thickness* is the thickness in device units. *contents* is a form that produces a single pane.

The **clim:outlining** macro is the usual way of creating a pane of type **clim:outlined-pane**.

options may include other pane initargs, such as space requirement options, **:foreground**, **:background**, **:text-style**, and so forth.

clim:outlined-pane *Class*

The layout pane class that draws a border around its child pane. **clim:outlining** generates a pane of this type.

In addition to the usual sheet initargs (the space requirement initargs, **:foreground** and **:background**), this class supports two other initargs:

:thickness
: An integer that specifies the thickness of the border, in device units.

:contents The pane that will be the child.

clim:output-record *Class*

The protocol class that is used to indicate that an object is an *output record*, that is, a CLIM object that contains other output records. If you want to create a new class the obeys the output record protocol, it must be a subclass of **clim:output-record**.

If you think of output records being arranged in a tree, output records are the non-leaf nodes of the tree.

See the section "Output Recording in CLIM".

clim:output-record-count *record* *Generic Function*

Returns the number of output records that are children of *record*.

clim:output-record-children *record* *Function*

Returns a sequence of output records that are the children of the output record *record*. If *record* has no children, this will return **nil**.

For some classes of output records, this function may create a new sequence holding the child output records. Because of this, you should use **clim:map-over-output-records** instead of this, if it is possible to do so.

See the section "Output Recording in CLIM".

clim:output-record-p *object* *Function*

Returns **t** if and only *object* is of type **clim:output-record**.

clim:output-record-parent *record* *Generic Function*

Returns the output record that it the parent of *record*. If *record* has no parent, **clim:output-record-parent** will return **nil**.

See the section "Output Recording in CLIM".

clim:output-record-position *record* *Generic Function*

Returns the X and Y position of *record* as two real 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 output record's parent, where (0,0) is the upper-left corner of the parent output record.

See the section "Output Recording in CLIM". See the function **clim:convert-from-relative-to-absolute-coordinates**.

clim:output-record-refined-position-test *record x y* *Generic Function*

CLIM uses **clim:output-record-refined-position-test** to definitively determine that the point *(x,y)* is contained within the output record *record*. Once the position *(x,y)* has been determined to lie within the bounding rectangle of the record, CLIM calls **clim:output-record-refined-position-test**.

You can define methods for this generic function for your own output record classes in order to provide a more precise definition of a hit. For example, output records for ellipses implement a method that detects whether the pointer cursor is within the ellipse.

The following example shows how you might implement this method for an output record that records a filled-in circle:

```
(defmethod clim:output-record-refined-position-test
           ((record circle-output-record) x y)
  (with-slots (center-x center-y radius) record
    (point-inside-circle-p (- x center-x) (- y center-y) radius)))
```

clim:output-record-set-position *record x y* *Generic Function*

Changes the position of the output record *record* to the new position *x* and *y*. *x* and *y* are the new left and top coordinates of the record. The position is relative to the output record's parent, where (0,0) is the upper-left corner of the parent output record.

See the section "Output Recording in CLIM". See the function **clim:convert-from-absolute-to-relative-coordinates**.

clim:output-recording-stream-p *object* *Function*

Returns **t** if and only if *object* is an output recording stream.

clim:output-stream-p *stream* *Generic Function*

Returns **t** if *stream* is a CLIM output stream, otherwise returns **nil**.

clim:palette-color-p *palette* *Generic Function*

Returns **t** if the palette supports color, otherwise returns **nil**.

You can determine whether or not a particular stream or medium supports color with the following function:

```
(defun color-stream-p (stream)
  (clim:palette-color-p
    (clim:port-default-palette (clim:port stream))))
```

clim:pane *Class*

The protocol class that corresponds to any kind of CLIM pane (that is, a sheet that participates in the layout protocol within an application frame).

clim:pane-frame *pane* *Generic Function*

Returns the application frame in which *pane* is one of the panes.

clim:pane-name *pane* *Generic Function*

Returns the name of the pane *pane* if it is a named pane, otherwise returns **nil**.

clim:pane-viewport *pane* *Generic Function*

If the pane *pane* is part of a scroller pane, this returns the viewport pane for *pane*. Otherwise it returns **nil**.

clim:pane-viewport-region *pane* *Generic Function*

If the pane *pane* is part of a scroller pane, this returns the region of *pane*'s viewport. Otherwise it returns **nil**.

The region is usually an instance of **clim:standard-bounding-rectangle**.

clim:panep *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:pane**.

conditions:parse-error *Condition*

This is the superclass of both **clim:simple-parse-error** and **clim:input-not-of-required-type**.

clim:parse-text-style *text-style* *Generic Function*

If *text-style* is either a text style object or a device font, **clim:parse-text-style** returns *text-style*. Otherwise, *text-style* must be a list of three elements, the text style family, face, and size. In this case, *text-style* is parsed and a text style object is returned.

```
(clim:parse-text-style '(:fix :roman :normal))
 => #<CLIM:STANDARD-TEXT-STYLE :FIX.:ROMAN.:NORMAL 20153772131>

(clim:parse-text-style '(:serif :bold-italic :normal))
 => #<CLIM:STANDARD-TEXT-STYLE :SERIF.(:BOLD :ITALIC).:NORMAL 401035455>
```

See the section "Text Styles in CLIM".

clim:partial-command-p *command* *Function*

Returns **t** if the command object *command* is a partial command, otherwise returns **nil**.

clim:*partial-command-parser* *Variable*

The currently active partial command parsing function.

The default for this is **clim:command-line-read-remaining-arguments-for-partial-command** when there is at least one interactor pane in the application frame, otherwise the default is **clim:menu-read-remaining-arguments-for-partial-command**.

clim:path *Class*

This is a subclass of **clim:region** that denotes regions that have dimensionality 1. If you want to create a new class that obeys the path protocol, it must be a subclass of **clim:path**.

Making a **clim:path** object with no length canonicalizes it to **clim:+nowhere+**. When paths are used to construct an area by specifying its outline, they need to have a direction associated with them.

clim:pathp *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:path**.

clim-lisp:pathname *Clim Presentation Type*

The presentation type that represents a pathname.

The options are **:default-type** (which defaults to **nil**), **:default-version** (which defaults to **:newest**), and **:merge-default** (which defaults to **t**). If **:merge-default** is **nil**, **clim:accept** returns the exact pathname that was entered, otherwise **clim:accept** merges against the default provided to **clim:accept** and **:default-type** and **:default-version**, using **merge-pathnames**. If no default is specified, it defaults to ***default-pathname-defaults***.

clim:pattern-height *pattern* *Generic Function*

Returns the height of the pattern *pattern*.

clim:pattern-width *pattern* *Generic Function*

Returns the width of the pattern *pattern*.

clim:pixmap-depth *pixmap* *Generic Function*

Returns the depth of the pixmap *pixmap*.

clim:pixmap-height *pixmap* *Generic Function*

Returns the height of the pixmap *pixmap*.

clim:pixmap-width *pixmap* *Generic Function*

Returns the width of the pixmap *pixmap*.

clim:point *Class*

The protocol class that corresponds to a mathematical point. If you want to create a new class that obeys the point protocol, it must be a subclass of **clim:point**.

clim:point-position *point* *Generic Function*

Returns two values, the X and Y coordinates of *point*.

clim:point-x *point* *Generic Function*

Returns the X coordinate of *point*.

clim:point-y *point* *Generic Function*

Returns the Y coordinate of *point*.

clim:pointer *Class*

The protocol class that corresponds to a pointing device, such as a mouse. If you want to create a new class that obeys the pointer protocol, it must be a subclass of **clim:pointer**.

clim:pointer-button-event *Class*

The class that corresponds to some sort of CLIM pointer button event, such as a button press or release. This is a subclass of **clim:pointer-event**.

clim:pointer-button-press-event *Class*

The class that corresponds to the user pressing a button on the pointer. This is a subclass of **clim:pointer-button-event**.

clim:pointer-button-release-event *Class*

The class that corresponds to the user releasing a button on the pointer. This is a subclass of **clim:pointer-button-event**.

Note that CLIM will usually not pass button release events to the stream layer, except in specific circumstances, such as inside of **clim:tracking-pointer**. That is, functions like **read-char** and **clim:read-gesture** will not generally ever seen pointer button release events.

clim:pointer-button-state *pointer* *Generic Function*

Returns the current button state for *pointer*. This is a bit mask that can be checked against the values of **clim:+pointer-left-button+**, **clim:+pointer-middle-button+**, and **clim:+pointer-right-button+**.

clim:pointer-cursor *pointer* *Generic Function*

Returns the current cursor type for *pointer*.

You can use **setf** on **clim:pointer-cursor** to temporarily change the cursor type. Note that some window managers can change the pointer cursor without notifying the application, so it is generally better to change the pointer cursor by setting the **clim:sheet-pointer-cursor** attribute of the sheets for which you want the cursor to change.

The valid cursor types are **:default**, **:vertical-scroll**, **:scroll-up**, **:scroll-down**, **:horizontal-scroll**, **:scroll-left**, **:scroll-right**, **:busy**, **:upper-left**, **:upper-right**, **:lower-left**, **:lower-right**, **:vertical-thumb**, **:horizontal-thumb**, **:button**, **:prompt**, **:move**, **:position**, and **:i-beam**.

For example, you can temporarily change the pointer cursor to indicate that the current program is busy using the following code:

```
(defmacro with-busy-cursor ((&optional (frame 'clim:*application-frame*))
                            &body body)
  (let ((pointer '#:pointer)
        (old-cursor '#:old-cursor))
    `(let* ((,pointer (clim:port-pointer (port ,frame)))
            (,old-cursor (clim:pointer-cursor ,pointer)))
       (setf (clim:pointer-cursor ,pointer) :busy)
       (unwind-protect
           ,@body
         (setf (clim:pointer-cursor ,pointer) ,old-cursor)))))
```

clim:pointer-documentation-pane *Class*

The pane class that is used to implement the pointer documentation pane. It corresponds to the pane type abbreviation **:pointer-documentation** in the **:panes** clause of **clim:define-application-frame**.

For **clim:pointer-documentation-pane**, the default for the **:display-time** option is **nil**, and the default for the **:scroll-bars** option is **nil**.

clim:pointer-documentation-view *Class*

The view that is used for pointer documentation.

clim:*pointer-documentation-output* *Variable*

Either **nil** or a stream to which pointer documentation should be written. If **nil**, no pointer documentation is computed.

clim:+pointer-documentation-view+ *Variable*

An instance of the class **clim:pointer-documentation-view**.

clim:pointer-event *Class*

The class that corresponds to some sort of CLIM pointer event. This is a subclass of **clim:device-event**.

clim:pointer-event-button *pointer-button-event* *Generic Function*

Returns the button number that was pressed when the pointer button event *pointer-button-event* occurred. The returned value is an integer with 1 bits that correspond to the button that was pressed.

Button	*Mask*
Left	**clim:+pointer-left-button+**
Middle	**clim:+pointer-middle-button+**
Right	**clim:+pointer-right-button+**

clim:pointer-event-x *pointer-event* *Generic Function*

Returns the X position of the pointer when the pointer event *pointer-event* occurred.

clim:pointer-event-y *pointer-event* *Generic Function*

Returns the Y position of the pointer when the pointer event *pointer-event* occurred.

clim:pointer-enter-event *Class*

The class that corresponds to the user moving the pointer into a sheet from another sheet. This is a subclass of **clim:pointer-event**.

clim:pointer-exit-event *Class*

The class that corresponds to the user moving the pointer out of a sheet. This is a subclass of **clim:pointer-event**.

clim:pointer-input-rectangle* &key *:left :top :right :bottom (:stream* ***standard-input****) :pointer :multiple-window (:finish-on-release* **t***)* *Function*

You can use this function to prompt for and input the corners of a rectangle on the stream *:stream* (which defaults to ***standard-input****). *:pointer*, *:multiple-window*, and *:finish-on-release* are as for **clim:drag-output-record**.

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 place at *(:right,:bottom)*. Otherwise, the upper left corner of the rectangle is selected by pressing a button on the pointer.

clim:pointer-input-rectangle* returns four values, the left, top, right, and bottom corners of the rectangle.

clim:+pointer-left-button+ *Constant*

The value returned by **clim:pointer-event-button** that corresponds to the user having pressed or released the lefthand button on the pointer.

clim:+pointer-middle-button+ *Constant*

The value returned by **clim:pointer-event-button** that corresponds to the user having pressed or released the middle button on the pointer.

clim:pointer-motion-event *Class*

The class that corresponds to the user moving the pointer. This is a subclass of **clim:pointer-event**.

clim:pointer-native-position *pointer* *Generic Function*

This function returns the position (as two coordinate values) of the pointer *pointer* in the coordinate system of the port's graft (that is, its ''root window'').

You can use **clim:pointer-set-position** to set the pointer's native position.

clim:pointer-place-rubber-band-line* &key *:start-x :start-y (:stream* ***standard-input****) :pointer :multiple-window (:finish-on-release* **t***)* *Function*

You can use this function to prompt for and input the end points of a line on the stream *stream* (which defaults to ***standard-input***). *:pointer*, *:multiple-window*, and *:finish-on-release* are as for **clim:drag-output-record**.

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.

clim:pointer-place-rubber-band-line* returns four values, the start X and Y positions, and the end X and Y positions.

clim:pointer-position *pointer* *Generic Function*

This function returns the position (two coordinate values) of the pointer *pointer* in the coordinate system of the sheet that the pointer is currently over.

You can use **clim:pointer-set-position** to set the pointer's position.

clim:+pointer-right-button+ *Constant*

The value returned by **clim:pointer-event-button** that corresponds to the user having pressed or released the righthand button on the pointer.

clim:pointer-set-native-position *pointer x y* *Generic Function*

This function changes the position of the pointer *pointer* to be *(x,y)*. *x* and *y* are in the coordinate system of the port's graft (that is, its ''root window'').

clim:pointer-set-position *pointer x y* *Generic Function*

This function changes the position of the pointer *pointer* to be *(x,y)*. *x* and *y* are in the coordinate system of the sheet that the pointer is currently over.

clim:pointer-sheet *pointer* *Generic Function*

Returns the sheet over which the pointer *pointer* is currently positioned.

clim:pointerp *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:pointer**, otherwise returns **nil**.

clim:pointp *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:point**.

clim:polygon *Class*

The protocol class that corresponds to a mathematical polygon. This is a subclass of **clim:area**. If you want to create a new class that obeys the polygon protocol, it must be a subclass of **clim:polygon**.

clim:polygon-points *polygon* *Generic Function*

Returns a sequence of points that specify the segments in *polygon*.

clim:polygonp *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:polygon**.

clim:polyline *Class*

The protocol class that corresponds to a polyline. This is a subclass of **clim:path**. If you want to create a new class that obeys the polyline protocol, it must be a subclass of **clim:polyline**.

clim:polyline-closed *polyline* *Generic Function*

Returns **t** if *polyline* is closed, otherwise returns **nil**.

clim:polylinep *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:polyline**.

clim:port *Class*

The protocol class that corresponds to a port. If you want to create a new class that obeys the port protocol, it must be a subclass of **clim:port**.

clim:port *object* *Generic Function*

Given a CLIM object *object*, **clim:port** returns the port associated with *object*. If *object* is not presently ''owned'' by any port, **clim:port** will return **nil**.

You can call **clim:port** on sheets, mediums, frames, frame managers, pointers, cursors, and pixmaps.

clim:port-default-palette *port* *Generic Function*

Returns the palette associated with the port *port*.

A palette is an object that contains mappings from color names (which are strings or symbols) to CLIM color objects. See the section "Predefined Color Names in CLIM".

clim:port-modifier-state *basic-port* *Generic Function*

Returns the state of the modifier keys for the port *port*. This is a bit mask that can be checked against the values of **clim:+shift-key+**, **clim:+control-key+**, **clim:+meta-key+**, **clim:+super-key+**, and **clim:+hyper-key+**.

clim:port-name *port* *Generic Function*

Returns the name of the port as a string whose syntax varies from port to port. For example, a Genera port might have a name of `"Summer:Main Screen"`.

clim:port-pointer *port* *Generic Function*

Returns the pointer object corresponding to the primary pointing device for the port *port*.

The pointer is of type **clim:pointer**.

clim:port-server-path *port* *Generic Function*

Returns the server path associated with the port. For example, a Genera port might return the following:

```
(:genera :screen #<MAIN-SCREEN Main Screen 20006600000 exposed>)
```

clim:port-type *port* *Generic Function*

Returns the type of the port, that is, the first element of the server path specification (**:genera**, **:clx**, and so on).

clim:portp *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:port**, otherwise returns **nil**.

clim:position-sheet-carefully *sheet x y* *Function*

Moves the sheet *sheet* to the position specified by *x* and *y*, taking care not to move the sheet outside of its parent (for example, off the screen).

This function is intended to work only on top level sheets. If you call it on a sheet that is not a top level sheet, the results are unpredictable. For example, if you want to position an application frame, do the following:

```
(clim:position-sheet-carefully
   (clim:frame-top-level-sheet frame) x y))
```

clim:position-sheet-near-pointer *sheet* &optional *x y* *Function*

Moves the sheet *sheet* to a position near the current pointer position. If *x* and *y* are supplied, these override the pointer position. **clim:position-sheet-near-pointer** takes care not to move the sheet outside of its parent (for example, off the screen).

This function is intended to work only on top level sheets. If you call it on a sheet that is not a top level sheet, the results are unpredictable.

clim:*possibilities-gestures* *Variable*

A list of gesture names that cause **clim:complete-input** to display a help message and the list of possibilities. On most systems, this includes the gesture corresponding to the `#\Control-?` character.

clim:present *object* &optional *(presentation-type* **(clim:presentation-type-of object)***)* &key *(:stream* ***standard-output****) (:view* **(clim:stream-default-view stream)***) :modifier :acceptably (:for-context-type* **presentation-type***) :single-box :allow-sensitive-inferiors :sensitive (:record-type* **'clim:standard-presentation***)* *Function*

Creates a presentation on the stream of the specified object, using the given type and view to determine visual appearance. The manner in which the object is displayed depends on the presentation type of the object; the display is done by the type's **clim:present** method for the given *view*.

For background information, see the section "Presentation Types in CLIM".

object The object to be presented.

presentation-type
A presentation type specifier, which may be a presentation type abbreviation. This defaults to the most specific type that CLIM can determine, based on the Lisp data type of *object*.

stream
The stream to which output should be sent. The default is ***standard-output****.

:view
An object representing a view. The default is **(clim:stream-default-view** *stream***)**. For most streams, the default view is the textual view, **clim:+textual-view+**. For dialog streams (that is, within **clim:accepting-values**), the view will typically be either **clim:+textual-dialog-view+** or **clim:+gadget-dialog-view+**.

:modifier
Specifies a function of one argument (the new value) that can be called in order to store a new value for *object* after the user edits the presentation. The default is **nil**.

:acceptably
Defaults to **nil**, which requests the **clim:present** method to produce

output designed to be read by the user. If **t**, this option requests the **clim:present** method to produce output that can be parsed by the **clim:accept** method. This option makes no difference for most presentation types.

:for-context-type

A presentation type indicating an input context. The present method can look at this to determine if the object should be presented differently. For example, the **clim:present** method for the **clim:command** presentation type uses this in order to determine whether or not to display a colon (":") before commands in **clim:command-or-form** contexts. *:for-context-type* defaults to *presentation-type*.

:single-box

Controls how CLIM determines whether the pointer is pointing at this presentation and controls how this presentation is highlighted when it is sensitive.

The possible values are:

t — If the pointer's position is inside the bounding rectangle of this presentation, it is considered to be pointing at this presentation. This presentation is highlighted by highlighting its bounding rectangle.

nil — If the pointer is pointing at a visible piece of output (text or graphics) drawn as part of the visual representation of this presentation, it is considered to be pointing at this presentation. This presentation is highlighted by highlighting every visible piece of output that is drawn as part of its visual representation. This is the default.

:position — Like **t** for determining whether the pointer is pointing at this presentation, like **nil** for highlighting.

:highlighting — Like **nil** for determining whether the pointer is pointing at this presentation, like **t** for highlighting.

Supplying **:single-box :highlighting** is useful when the default behavior produces an ugly appearance (for example, a very jagged highlighting box).

Supplying **:single-box :position** is useful when the visual representation of a presentation consists of one or more small graphical objects with a lot of space between them. In this case the default behavior offers only small targets that the user might find difficult to position the pointer over.

:allow-sensitive-inferiors

When *:allow-sensitive-inferiors* is **nil**, it indicates that nested calls to **clim:present** or **clim:with-output-as-presentation** inside this one should not generate presentations. The default is **t**.

:sensitive

If *:sensitive* is **nil**, no presentation is produced. The default is **t**.

:record-type

This option is useful when you have defined a customized record type to replace CLIM's default record type. It specifies the class of the output record to be created.

clim:present *type-key parameters options object type stream view* &key *:acceptably :for-context-type* *Clim Presentation Method*

This presentation method is responsible for displaying the representation of *object* having type *type* for a particular view *view*. The method's caller takes care of creating the presentation, so the method need only display the content of the presentation.

The method must specify **&key**, but need only receive the keyword arguments that it is interested in. The remaining keyword arguments will be ignored automatically since the generic function specifies **&allow-other-keys**.

The **clim: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; for example, you should define a **clim:present** method specializing on the class **clim:textual-view** if you want to allow the type to be displayed textually.

Note that CLIM stores the presentation type for its own use, so you should not modify it once you have handed it to CLIM.

For more information on defining presentation types, see the section "Defining a New Presentation Type in CLIM".

clim:present-to-string *object* &optional *(presentation-type* **(clim:presentation-type-of object)***)* &key *(:view* **clim:+textual-view+***) :acceptably (:for-context-type* **presentation-type***) :string :index* *Function*

Presents an object into a string in such a way that it can subsequently be accepted as input by **clim:accept-from-string**. **clim:present-to-string** is the same as **clim:present** within **with-output-to-string**.

object, *presentation-type*, *:view*, *:acceptably*, and *:for-context-type* are as for **clim:present**.

For background information, see the section "Presentation Types in CLIM".

clim:presentation *Class*

The protocol class that corresponds to a presentation. If you want to create a new class that obeys the presentation protocol, it must be a subclass of **clim:presentation**.

clim:presentationp *object* *Function*

Returns **t** if and only if *object* is of type **clim:presentation**.

clim:presentation-matches-context-type *presentation context-type frame window x y* &key *:event (:modifier-state* **0***)* *Function*

Returns a non-**nil** value if there are any translators that translate from *presentation*'s type to *context-type*. (There is no *from-type* argument because it is derived from *presentation*.) *frame*, *window*, *x*, *y*, *:event*, and *:modifier-state* are as for **clim:find-applicable-translators**.

If there are no applicable translators, **clim:presentation-matches-context-type** will return **nil**.

clim:presentation-object *presentation* *Generic Function*

Returns the application object represented by the presentation *presentation*. You can use **setf** on **clim:presentation-object** to change the object associated with the presentation.

Any class that is a subclass of **clim:presentation** must implement this method.

clim:presentation-refined-position-test *type-key parameters options type record x y* *Clim Presentation Method*

CLIM uses this method to definitively answer hit detection queries for a presentation, that is, determining whether or not the point *(x,y)* is contained within the output record *record*. Its contract is exactly the same as for **clim:output-record-refined-position-test**, except that it is intended to specialize on the presentation type *type*.

It can be useful to define a **clim:presentation-refined-position-test** method when the displayed output records that represent the presentation do not themselves implement the desired hit detection behavior. In practice, this comes up only rarely, since using the **:single-box** option to **clim:present** and **clim:with-output-as-presentation** will often produce the desired behavior.

clim:presentation-replace-input *stream object type view* &key *:rescan :buffer-start* *Generic Function*

This is like **clim:replace-input**, except that the new input to insert into the input buffer is gotten by presenting the object *object* with the presentation type *type* and view *view*.

:rescan and *:buffer-start* are the same as for **clim:replace-input**.

For example, the following **clim:accept** method reads a token followed by a ''system'' or a pathname, but if the user clicks on either a ''system'' or a pathname, it inserts that object into the input buffer and returns:

```
(clim:define-presentation-method clim:accept
                                 ((type library) stream (view clim:textual-view)
                                  &key default)
  (clim:with-input-context ('(or system pathname)) (object type)
       (let ((system (clim:accept '(clim:token-or-type (:private) system)
                                  :stream stream :view view
                                  :prompt nil :display-default nil
                                  :default default
                                  :additional-delimiter-gestures '(#\space)))
             file)
         (let ((char (clim:read-gesture :stream stream)))
           (unless (eql char #\space)
             (clim:unread-gesture char :stream stream))
           (when (eql system ':private)
             (setq file (clim:accept 'pathname
                                     :stream stream :view view
                                     :prompt "library pathname"
                                     :display-default t)))
           (if (eql system ':private) file system)))
     (t (clim:presentation-replace-input stream object type view)
        (values object type))))
```

See the section "Utilities for **clim:accept** Presentation Methods". See the section "The Structure of the CLIM Input Editor".

clim:presentation-subtypep *type putative-supertype* *Function*

Answers the question ''Is the type specified by *type* a subtype of the type specified by *putative-supertype*?''. Neither *type* nor *putative-subtype* may be presentation type abbreviations.

This function is analogous to **subtypep**.

clim:presentation-subtypep returns two values, *subtypep* and *known-p*. *subtypep* can be **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 **t**, or that the answer cannot be determined if *known-p* is **nil**).

See the clim presentation method **clim:presentation-subtypep** for a detailed description of how this works.

clim:presentation-subtypep *type-key type putative-supertype*
Clim Presentation Method

This presentation method is called when the **clim:presentation-subtypep** function requires type-specific knowledge.

The **clim:presentation-subtypep** function walks the type lattice to determine that *type* is a subtype of *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 **clim:presentation-subtypep** method for that type is called in order to resolve the question by looking at the type parameters (that is, if the **clim:presentation-subtypep** method is called, *type* and *putative-supertype* are guaranteed to be the same type, differing only in their parameters).

Unlike all other presentation methods, **clim: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.

clim:presentation-subtypep returns two values, *subtypep* and *known-p*. *subtypep* can be **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 **t**, or that the answer cannot be determined if *known-p* is **nil**).

Since **clim:presentation-subtypep** takes two arguments that are presentation types, the parameters are not lexically available as variables in the body of a presentation method. Use **clim:with-presentation-type-parameters** if you want to access the parameters of the presentation types.

Note: You must define a **clim:presentation-subtypep** method if the presentation type has parameters.

For example, the CLIM presentation type **complex** has a type parameter that indicates what numeric type the real and imaginary components of the number must be. Its **clim:presentation-subtypep** method could be written as follows:

```
(clim:define-presentation-method clim:presentation-subtypep
                                 ((type complex) putative-supertype)
  (let ((type1
          (clim:with-presentation-type-parameters
            (complex type) type))
        (type2
          (clim:with-presentation-type-parameters
            (complex putative-supertype) type)))
    (cond ((eq type2 '*)
           (values t t))
          ((eq type1 '*)
           (values nil t))
          (t
           (clim:presentation-subtypep type1 type2)))))
```

Returns the presentation type of the presentation *presentation*. You can use **setf** on **clim:presentation-type** to change the presentation type associated with the presentation.

Any class that is a subclass of **clim:presentation** must implement this method.

clim:presentation-type-of *object* *Function*

Returns the presentation type of the object *object*. If the type cannot be readily computed, this may return **t** or **clim:expression**.

This function is analogous to **type-of**.

clim:presentation-type-specifier-p *type-key parameters options type*
Clim Presentation Method

This presentation method is responsible for checking the validity of the parameters and options. The default method returns **t**.

clim:presentation-typep *object type* *Function*

Returns **t** if *object* is of the type specified by *type*, otherwise returns **nil**. *type* may not be a presentation type abbreviation.

This function is analogous to **typep**.

clim:presentation-typep *type-key parameters object type* *Clim Presentation Method*

This presentation method is called when the **clim:presentation-typep** function requires type-specific knowledge. If the type name in *type* is or names a CLOS class, the method is called only if *object* is a member of the class and *type* contains parameters, and 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.

Note: You must define a **clim:presentation-typep** method if the presentation type does not have the same name as a CLOS class.

For example, the CLIM presentation type **complex** has a type parameter that indicates what numeric type the real and imaginary components of the number must be. Its **clim:presentation-typep** method could be written as follows:

```
(clim:define-presentation-method clim:presentation-typep
                                 (object (type complex))
  (declare (ignore type))
  (or (eq type '*)
      (and (clim:presentation-typep (realpart object) type)
           (clim:presentation-typep (imagpart object) type))))
```

clim:print-menu-item *menu-item* &optional *(stream* ***standard-output****)* *Function*

Given a menu item *menu-item*, display it on the stream *stream*. This is the function that **clim:menu-choose** uses to display menu items if no printer is supplied.

clim-sys:process-name *process* *Function*

Returns the name of the process *process*.

clim-sys:processp *object* *Function*

Returns **t** if *object* is a process, otherwise returns **nil**.

See the section "Multi-processing in CLIM".

clim-sys:process-interrupt *process function* *Function*

Interrupts the process *process* and causes it to evaluate the function *function*.

On systems that do not support multi-processing, this is equivalent to simply calling *function*.

clim-sys:process-wait *wait-reason predicate* *Function*

Causes the current process to wait until *predicate* returns a non-**nil** value. *predicate* is a function of no arguments. *reason* is a ''reason'' for waiting, usually a string.

On systems that do not support multi-processing, **clim-sys:process-wait** will simply loop until *predicate* returns a non-**nil** value.

See the section "Multi-processing in CLIM".

clim-sys:process-wait-with-timeout *wait-reason timeout predicate* *Function*

Causes the current process to wait until either *predicate* returns a non-**nil** value or the number of seconds specified by *timeout* has elapsed. *predicate* is a function of no arguments. *reason* is a ''reason'' for waiting, usually a string.

On systems that do not support multi-processing, **clim-sys:process-wait-with-timeout** will simply loop until *predicate* returns a non-**nil** value, or the timeout has elapsed.

See the section "Multi-processing in CLIM".

clim-sys:process-yield *Function*

Allows other processes to run. On systems that do not support multi-processing, this does nothing.

See the section "Multi-processing in CLIM".

clim:push-button *Class*

The **clim:push-button** gadget class provides press-to-activate switch behavior. It is a subclass of **clim:action-gadget** and **clim:labelled-gadget-mixin**.

See the section "Using Gadgets in CLIM".

In addition to the initargs for **clim:action-gadget** and the usual pane initargs (**:foreground**, **:background**, **:text-style**, space requirement options, and so forth), the following initargs are supported:

:label A string or pixmap that is used to label the button.

:show-as-default-p
When **t**, the push button will be drawn with a heavy border, which indicates that this button is the ''default operation''. The default is **nil**.

:pattern If supplied, this is a pattern (created by **clim:make-pattern**) that specifies that shape of the button. The default pattern for CLIM's ''native'' gadgets is a rectangular button that is just big enough to enclose the label.

:external-label
If supplied, this is a string that will be used as a label that is drawn *outside* of the push button instead of inside of the button.

clim:armed-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), **clim:activate-callback** will be invoked. Finally, **clim:disarmed-callback** will be invoked after **clim:activate-callback**, or when the pointer is moved outside of the button.

A push button might be created as follows:

```
(clim:make-pane 'clim:push-button
  :label "Button"
  :activate-callback 'push-button-callback)

(defun push-button-callback (button)
  (format t "~&Button ~A pushed" (clim:gadget-label button)))
```

clim:queue-event *sheet event* *Generic Function*

CLIM's event processor calls this function to insert the event *event* into the queue of events for *sheet*.

CLIM always handles some events immediately, such as window repaint events. In this case, instead of inserting the event into the event queue, the **clim:queue-event** method will call **clim:handle-event** directly.

Some sheets in CLIM also handle certain other events immediately. For example, scroll bars in CLIM respond to pointer events even when user applications are not waiting for input. Such sheets specialize the **clim:queue-event** method to call **clim:handle-event** directly.

clim:queue-repaint *sheet repaint-event* *Generic Function*

Inserts the repaint event *repaint-event* into *sheet*'s event queue. A program that reads events out of the queue will be expected to call **clim:handle-repaint** for the sheet using the repaint region gotten from the event.

clim:queue-rescan *input-editing-stream* &optional *rescan-type* *Generic Function*

Indicates that a rescan operation on *input-editing-stream* should take place after the next non-input editing gesture is read. This works by setting the ''rescan queued'' flag to **t**. Use this function when you are writing new ''destructive'' input editing commands.

For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:radio-box *Class*

A radio box is a special kind of gadget that contains one or more toggle buttons. At any one time, only one of the buttons managed by the radio box may be ''on''. The contents of a radio box are its buttons, and as such a radio box is responsible for laying out the buttons that it contains.

It is a subclass of **clim:value-gadget** and **clim:oriented-gadget-mixin**.

See the section "Using Gadgets in CLIM".

In addition to the initargs for **clim:value-gadget** and the usual pane initargs (**:foreground**, **:background**, **:text-style**, space requirement options, and so forth), the following initargs are supported:

:selection This is used to specify which button, if any, should be initially selected.

:choices This is used to specify all of the buttons that serve as choices.

As the current selection changes, the previously selected button and the newly selected button both have their **clim:value-changed-callback** handlers invoked.

Calling **clim:gadget-value** on a radio box will return the currently selected toggle button. The value of the radio box can be changed by calling **setf** on **clim:gadget-value**.

A radio box might be created as follows, although it is generally more convenient to use **clim:with-radio-box**:

```
(let* ((choices
         (list (clim:make-pane 'clim:toggle-button
                 :label "One" :width 80)
               (clim:make-pane 'clim:toggle-button
                 :label "Two" :width 80)
               (clim:make-pane 'clim:toggle-button
                 :label "Three" :width 80)))
       (current (second choices)))
  (clim:make-pane 'clim:radio-box
    :choices choices
    :selection current
    :value-changed-callback 'radio-value-changed-callback))

(defun radio-value-changed-callback (radio-box value)
  (declare (ignore radio-box))
  (format t "~&Radio box toggled to ~S" value))
```

clim:radio-box-current-selection *radio-box* *Generic Function*

Returns the current selection for the radio box. The current selection will be one of the toggle buttons in the radio box.

You can use **setf** on this in order to set the current selection for the radio box, or you can use **setf** on **clim:gadget-value** of the radio box to accomplish the same thing.

clim:radio-box-selections *radio-box* *Generic Function*

Returns a sequence of all of the selections in the radio box. The elements of the sequence will be toggle buttons.

clim:radio-box-view *Class*

The class that represents the view corresponding to a radio box. This is usually used for a ‘‘one of’’ choice, as in a **member** presentation type.

clim:+radio-box-view+ *Constant*

An instance of the class **clim:radio-box-view**.

clim:raise-frame *frame* *Generic Function*

Raises the application frame *frame* so that it is on top of all of the other host windows by calling **clim:raise-sheet** on the frame’s top-level sheet. This does not change the state of the frame, it simply makes it visible on the screen.

clim:range-gadget-mixin *Class*

The class that is mixed in to a gadget that has a range, for example, a slider.

All subclasses of **clim:range-gadget-mixin** must handle the two initargs **:min-value** and **:max-value**, which are used to specify the minimum and maximum value of the gadget.

ratio &optional *low high* *Clim Presentation Type*

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**.

rational &optional *low high* *Clim Presentation Type*

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. It is a subtype of **real**.

clim:read-command *command-table* &key *(:stream* ***query-io***) *(:command-parser* **clim:*command-parser***) *(:command-unparser* **clim:*command-unparser***) *(:partial-command-parser* **clim:*partial-command-parser***) *:use-keystrokes* *Function*

Reads a command from the user via command lines or the pointer. This function is not normally called by programmers.

command-table
: Specifies the command table from which commands should be read.

:stream The stream from which to read the command.

:use-keystrokes
: The default for this is **nil**. If it is **t**, **clim:read-command** calls **clim:read-command-using-keystrokes** to read the command. The keystroke accelerators are those generated by **clim:with-command-table-keystrokes**.

:command-parser
: A function of two arguments, *command-table* and *stream*. This function should read a command from the user and return a command object.

:command-unparser
: A function of three arguments, *command-table*, *stream*, and *command-to-unparse*. The function should print a textual description of the command and the set of arguments supplied on *stream*.

:partial-command-parser
: A function of four arguments, *command-table*, *stream*, *partial-command*, and *start-position*. A partial command is a command

structure with **clim:*unsupplied-argument-marker*** in place of any argument that remains to be filled in. The function should read the remaining arguments in any way it sees fit and should return a command object. *start-position* is the original input-editor scan position of *stream* if *stream* is an interactive stream.

You should not normally supply the *:command-parser*, *:command-unparser*, or *:partial-command-parser* arguments. CLIM will arrange to choose the correct default values for these.

clim:read-command-using-keystrokes *command-table keystrokes* &key *(:stream* ***query-io****) (:command-parser* **clim:*command-parser***) (:command-unparser* **clim:*command-unparser***) (:partial-command-parser* **clim:*partial-command-parser***)* *Function*

Reads a command from the user via a command line, the pointer, or typing a single keystroke. It returns either a command object, or a key press event if the user typed a keystroke that is in *keystrokes* but does not have a command associated with it in the command table.

keystrokes is a list of gesture specs. The other arguments are as for **clim:read-command**.

See also **clim:with-command-table-keystrokes**.

clim:read-frame-command *frame &key :stream* *Generic Function*

clim: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 **clim:read-frame-command** calls **clim:read-command** on *frame*'s current command table. You can specialize this generic function for your own application frames, for example, if you want to have your application be able to read commands using keystroke accelerators.

clim:read-gesture &key *(:stream* ***standard-input****) :timeout :peek-p :input-wait-test :input-wait-handler :pointer-button-press-handler* *Function*

Returns the next gesture available in the input stream (either a character or a pointer button event). Note that **clim:read-gesture** does not echo its input; CLIM's input editor does this when reading input inside of a call to **clim:with-input-editing**.

When a user types any sort of abort gesture (such as `#\Abort` on Genera), the **clim:abort-gesture** condition is signalled.

:timeout Specifies the number of seconds that **clim:read-gesture** will wait for input to become available. If no input is available, **clim:read-gesture** will return the two values, **nil** and **:timeout**. The default is that there is no timeout.

:peek-p If **t**, specifies that the gesture returned will be left in the stream's input buffer. The default is **nil**.

:input-wait-test
The value of this argument is a function. The function will be invoked with one argument, the stream. This argument will be passed on to **clim:stream-input-wait**.

:input-wait-handler
The value of this argument is a function. The function will be invoked with one argument, the stream, when the invocation of **clim:stream-input-wait** returns, but no input gesture is available. This option can be used in conjunction with *:input-wait-test* to handle conditions other than user keystroke gestures.

:pointer-button-press-handler
The value of this is a function of one argument, a pointer button event. This function is invoked when the user clicks the pointer.

You will rarely, if ever, need to supply *:input-wait-test*, *:input-wait-handler*, or *:pointer-button-press-handler*. CLIM uses these arguments to connect presentation types to streams.

clim:read-token *stream* &key *:timeout :input-wait-handler :pointer-button-press-handler :click-only* *Function*

Reads characters from *stream* until it encounters an activation gesture, a delimiter gesture, or a pointer gesture. All printing standard characters are acceptable (CLtL p. 336, CLtLII p. 512). **clim:read-token** returns the accumulated string that was delimited by an activation or delimiter gesture, leaving the delimiter unread, that is, still in the stream's input buffer.

:timeout Specifies the number of seconds that **clim:read-token** will wait for input to become available. If no input is available, **clim:read-token** will return the two values, **nil** and **:timeout**. The default is that there is no timeout.

:click-only If true, only pointer gestures are expected and anything else will result in a beep. The default is **nil**.

:input-wait-handler Passed along to **clim:read-gesture** . The default is a function that supports highlighting for **clim:with-input-context**.

:pointer-button-press-handler
Passed along to **clim:read-gesture**. The default is a function that supports presentation translators for **clim:with-input-context**.

You will rarely, if ever, need to supply *:input-wait-handler* or *:pointer-button-press-handler*. CLIM uses these arguments to connect presentation types to streams.

future-common-lisp:real &optional *low high* *Clim Presentation Type*

The presentation type that represents either a ratio, an integer, or a floating point number between *low* and *high*. *low* and *high* can be inclusive or exclusive, as in Common Lisp type specifiers.

Options to this type are **:base** and **:radix**, which are the same as for the **integer** type. This type is a subtype of **number**.

clim:rectangle *Class*

The protocol class that corresponds to an axis-aligned mathematical rectangle, that is, rectangular polygons whose sides are parallel to the coordinate axes. This is a subclass of **clim:polygon**. If you want to create a new class that obeys the rectangle protocol, it must be a subclass of **clim:rectangle**.

clim:rectangle-edges* *rectangle* *Generic Function*

Returns the coordinate of the minimum X and Y and maximum X and Y of *rectangle* as four values.

clim:rectangle-height *rectangle* *Function*

Returns the height of *rectangle*. The height is the difference between the maximum Y and the minimum Y.

clim:rectangle-max-point *rectangle* *Generic Function*

Returns the maximum point of *rectangle*. (The position of a rectangle is specified by its minimum point).

clim:rectangle-max-x *rectangle* *Function*

Returns the coordinate of the maximum X of *rectangle*.

clim:rectangle-max-y *rectangle* *Function*

Returns the coordinate of the maximum Y of *rectangle*.

clim:rectangle-min-point *rectangle* *Generic Function*

Returns the minimum point of *rectangle*. The position of a rectangle is specified by its minimum point.

clim:rectangle-min-x *rectangle* *Function*

Returns the coordinate of the minimum X of *rectangle*.

clim:rectangle-min-y *rectangle* *Function*

Returns the coordinate of the minimum Y of *rectangle*.

clim:rectangle-size *rectangle* *Function*

Returns two values, the width and the height of *rectangle*.

clim:rectangle-width *rectangle* *Function*

Returns the width of *rectangle*. The width of a rectangle is the difference between the maximum X and the minimum X.

clim:rectanglep *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:rectangle**.

clim:rectilinear-transformation-p *transform* *Generic Function*

Returns **t** if *transform* will always transform any axis-aligned rectangle into another axis-aligned rectangle, otherwise 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.

clim:recompute-extent-for-changed-child *record child old-left old-top old-right old-bottom* *Generic Function*

CLIM calls this function whenever the bounding rectangle of one of the children of a record has been changed. It updates the bounding rectangle of *record* to be large enough to completely contain the new bounding rectangle of the child output record *child*. CLIM notifies all of the ancestors of *record* by recursively calling **clim:recompute-extent-for-changed-child**.

CLIM provides an **:after** method on **clim:delete-output-record** that calls **clim:recompute-extent-for-changed-child** to inform the parent of the record that a change has taken place, so you will rarely need to call this yourself.

See the section "Concepts of CLIM Output Recording".

clim:recompute-extent-for-new-child *record child* *Generic Function*

CLIM calls this function whenever a new child is added to an output record. It updates the bounding rectangle of *record* to be large enough to completely contain the new child output record *child*. CLIM notifies the parent of *record*, and all its ancestors, of the change in size by calling **clim:recompute-extent-for-changed-child** on all of *record*'s ancestors.

CLIM provides and **:after** method on **clim:add-output-record** that calls **clim:recompute-extent-for-new-child**, so you will rarely need to call it yourself.

See the section "Concepts of CLIM Output Recording".

clim:redisplay *record stream* &key *(:check-overlapping* **t***)* *Function*

Causes the output of *record* to be recomputed by calling **clim:redisplay-output-record** on *record*. CLIM redisplays the changes incrementally, that is, only redisplays those parts of the record that changed. *record* must be an output record created by a previous call to **clim:updating-output**, and may be any part of the output history of *stream*.

The *:check-overlapping* argument insures that **clim:redisplay** will correctly redisplay output records that overlap at the same level in the output record hierarchy. It defaults to **t**. If you have output that you know does not not have any such overlapping output records, you can pass in **nil**; this will speed up incremental redisplay, but at the risk of failing to draw some records due to overlap.

See the section "Example of Incremental Redisplay in CLIM".

clim:redisplay-frame-pane *frame pane-name* &key *:force-p* *Generic Function*

Causes the pane *pane-name* of *frame* to be redisplayed immediately. CLIM either calls the pane's display function, or uses **clim:redisplay** if the pane is using incremental redisplay.

If *:force-p* is **t**, then the pane is forcibly redisplayed even if it is an incrementally redisplayed pane that would not otherwise require redisplay.

clim:redisplay-frame-panes *frame* &key *force-p* *Generic Function*

Causes all of the panes of *frame* to be redisplayed immediately. If *:force-p* is **t**, then the panes are forcibly redisplayed even if they are incrementally redisplayed panes that would not otherwise require redisplay.

clim:redisplay-output-record *record stream* &optional *check-overlapping x y parent-x parent-y* *Generic Function*

Causes the output of *record* to be recomputed. CLIM redisplays the changes incrementally, that is, only redisplays those parts of the record that changed. *record* must be an output record created by a previous call to **clim:updating-output**, and may be any part of the output history of *stream*.

The 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 the parent output record of record, before the record is redisplayed. The default values for *x* and *y* are the starting position of the output record.

parent-x and *parent-y* can be supplied to say: do the output as if the superior started at positions *parent-x* and *parent-y* (which are in absolute coordinates). The default values for *parent-x* and *parent-y* are the absolute coordinate of the output record's parent.

The *check-overlapping* argument insures that **clim:redisplay** checks for overlapping records. It defaults to **t**. If you make it **nil** it speeds up redisplay, at the risk of failing to draw some records due to overlap. If you are sure that no sibling records overlap, you can use this argument to optimize redisplay.

You can specialize this generic function for your own classes of output records, but you should not generally call this function yourself.

clim:redraw-input-buffer *input-editing-stream* &optional *start-position*
Generic Function

Displays the input editor's buffer starting at the position *start-position* on the interactive stream that is encapsulated by the input editing stream *input-editing-stream*.

You will rarely need to call this function yourself, since the input editor will almost always do this for you. For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:reflection-transformation-p *transform* *Generic Function*

Returns **t** if *transform* inverts the ''handedness'' of the coordinate system, otherwise 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.

clim:region *Class*

The protocol class that corresponds to a closed set of points. If you want to create a new class that obeys the region protocol, it must be a subclass of **clim:region**.

clim:regionp *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:region**.

clim:region-contains-position-p *region x y* *Generic Function*

Returns **t** if the point *(x,y)* is contained in *region*, otherwise returns **nil**. Since regions in CLIM are closed, this will return **t** if *(x,y)* is on the region's boundary. This is a special case of **clim:region-contains-region-p**.

clim:region-contains-region-p *region1 region2* *Generic Function*

Returns **t** if all points in *region2* are members of *region1*, otherwise returns **nil**.

clim:region-difference *region1 region2* *Generic Function*

Returns a region that contains all points in *region1* that are not in *region2* (plus additional boundary points to make the result closed). The result of **clim:region-difference** has the same dimensionality as *region1*, or is **clim:+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.

clim:region-equal *region1 region2* *Generic Function*

Returns **t** if *region1* and *region2* contain exactly the same set of points, otherwise returns **nil**.

clim:region-intersection *region1 region2* *Generic Function*

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

clim:region-intersects-region-p *region1 region2* *Generic Function*

Returns **nil** if **clim:region-intersection** of the two regions would be **clim:+nowhere+**, otherwise returns **t**.

clim:region-set *Class*

The class that represents region sets; a subclass of *region*.

clim:region-set-function *region* *Generic Function*

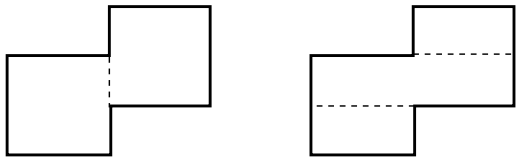
Returns a symbol representing the operation that relates the regions in *region*. This will be one of the Common Lisp symbols **union**, **intersection**, or **set-difference**. For the case of region sets that are composed entirely of rectangular regions, CLIM canonicalizes the set so that the symbol will always be **union**. If *region* is a region that is not a region-set, the result is always **union**.

clim:region-set-regions *region* &key *:normalize* *Generic Function*

Returns a sequence of the regions in *region*. *region* can be either a **clim:region-set** or any member of **clim:region**, in which case the result is simply a sequence of

one element: *region*. For the case of region sets that are unions of rectangular regions, CLIM canonicalizes the set so that the rectangles returned by **clim:region-set-regions** are guaranteed not to overlap.

If *:normalize* is supplied, it may be either **:x-banding** or **:y-banding**. If it is **:x-banding** and all the regions in *region* are 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.



`x-banding` `y-banding`

Normalizing a region set that is not composed entirely of rectangles using X- or Y-banding causes CLIM to signal the **clim:region-set-not-rectangular** error.

clim:region-union *region1 region2* *Generic Function*

Returns a region that contains all points that are in either *region1* or *region2* (possibly with some points removed to satisfy the dimensionality rule).

The result of **clim: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.

clim:remove-command-from-command-table *command-name command-table* &key *(:errorp* **t***)* *Function*

Removes the command named by *command-name* from the command table *command-table*. *command-table* may be either a command table or a symbol that names a command table.

If the command is not present in the command table and *:errorp* is **t**, the **clim:command-not-present** condition will be signalled.

clim:remove-keystroke-from-command-table *command-table keystroke* &key *(:errorp* **t***)* *Function*

Removes the item named by *keystroke* from *command-table*'s accelerator table. *command-table* may be either a command table or a symbol that names a command table. *keystroke* is gesture spec, such as `(:C :control :shift)`.

If the command menu item associated with *keystroke* is not present in the command table's menu and *:errorp* is **t**, then the **clim:command-not-present** condition will be signalled.

clim:remove-menu-item-from-command-table *command-table string* &key *(:errorp* **t***)* *Function*

Removes the item named by *string* from *command-table*'s menu. *command-table* may be either a command table or a symbol that names a command table.

If the command menu item is not present in the command table's menu and *:errorp* is **t**, then the **clim:command-not-present** condition will be signalled.

This function ignores the character case of the command menu item's name when searching through the command table's menu.

clim:repaint-sheet *sheet region* *Generic Function*

Causes *sheet* and all of its descendants that overlap the region *region* to be repainted. **clim:handle-repaint** is called to repaint each affected sheet.

You can call this function when you want to redraw the whole hierarchy of panes that start at *sheet*. If you want to repaint only *sheet*, use the function **clim:handle-repaint** instead.

clim:replace-input *stream new-input* &key *:start :end :rescan :buffer-start* *Generic Function*

Replaces *stream*'s input buffer with the string *new-input*. *:start* and *:end* specify what part of *new-input* will be inserted into the buffer, and default to 0 and the end of the string, respectively.

:buffer-start specifies where *new-input* should be inserted, and defaults to the current position in the input line. If *rescan* is **t**, a rescan operation will be queued; the default is **nil**. Usually, you should use the default values for *:buffer-start* and *:rescan*, since the input editor automatically arranges for the correct behavior to occur under those circumstances.

You can use this in an **clim:accept** method that needs to replace some of the user's input by something else. For example, **clim:complete-input** uses it to replace partial input with the completed input.

The returned value is the position in the input buffer.

See the section "Utilities for **clim:accept** Presentation Methods". See the section "The Structure of the CLIM Input Editor".

clim:replay *record stream* &optional *region* *Function*

Replays all of the output captured by the output record *record* on *stream* by calling **clim:replay-output-record**. If *region* is not **nil**, then *record* is replayed if and only if it overlaps region. *region* defaults to the intersection of *stream*'s viewport and *record*'s bounding rectangle.

Changing the transformation of the stream during replaying has no effect on what is output by **clim:replay**.

clim:replay-output-record *record stream* &optional *region x-offset y-offset*
Generic Function

Replays all of the output captured by the output record *record* on *stream*. If *region* is not **nil**, then *record* is replayed if and only if it overlaps region.

x-offset and *y-offset* are output record offsets that are necessitated by CLIM's representation of output records. In a later release of CLIM, the representation of output records may change in such a way that the *x-offset* and *y-offset* arguments are removed.

Changing the transformation of the stream during replaying has no effect on what is output by **clim:replay-output-record**.

You can specialize this generic function for your own classes of output records, but you should not generally call this function yourself. Any class that is a subclass of **clim:displayed-output-record** must implement this method.

Here is an example of using **clim:replay-output-record** in a way that supplies the X and Y offsets correctly:

```
(multiple-value-bind (x-offset y-offset)
   (clim:convert-from-relative-to-absolute-coordinates
    stream (clim:output-record-parent record))
 (clim:replay-output-record record stream region x-offset y-offset))
```

clim:rescan-if-necessary *input-editing-stream* &optional *inhibit-activation*
Generic Function

Invokes a rescan operation on the input editing stream *input-editing-stream* if **clim: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**.

If *inhibit-activation* is **nil**, the input line will not be activated even if there is an activation gesture in it.

You will rarely need to call this function yourself. Most programs should use **clim:queue-rescan** or **clim:immediate-rescan** instead. For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:reset-scan-pointer *input-editing-stream* &optional *sp* *Generic Function*

Sets *input-editing-stream*'s scan pointer to *sp* (which defaults to 0) and sets the state of **clim:stream-rescanning-p** to **t**.

You will rarely need to call this function yourself. For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:resize-sheet *sheet width height* *Generic Function*

Changes the size of *sheet* to have width *width* and height *height*.

clim:resize-sheet works by modifying the sheet's region, and could be implemented as follows:

```
(defmethod clim:resize-sheet
           ((sheet clim:basic-sheet) width height)
  (setf (clim:sheet-region sheet)
        (clim:make-bounding-rectangle 0 0 width height)))
```

You should not generally use this function to resize a sheet, because it does not interact directly with the frame layout protocol. That is, resizing a sheet with **clim:resize-sheet** may appear not to have any effect until you invoke the entire layout protocol. If you need to resize a top-level sheet, use **clim:size-frame-from-contents**.

clim:restart-port *port* *Generic Function*

Restarts the event process that handles the port *port*.

Occasionally the event process that handles a port may hang for some reason, although this is rather uncommon. Symptoms that this has happened include lack of pointer highlighting, and no echoing of typein. In the event that this happens, you can call **clim:restart-port** to restart the event process.

clim-sys:restart-process *process* *Function*

Restarts the process *process* by ''unwinding'' it to its initial state, and reinvoking its top-level function.

clim:rigid-transformation-p *transform* *Generic Function*

Returns **t** if *transform* 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.

clim:r-tree-output-history *Class*

A standard class provided by CLIM for use as a top-level output history. This is a subclass of both **clim:r-tree-output-record** and **clim:stream-output-history-mixin**.

You should consider using this as the output history for any pane that will have lots of overlapping output records, such as a complex graphical editor. You can create a pane using this history class as follows:

```
(clim:make-clim-application-pane
  :output-record (make-instance 'clim:r-tree-output-history))
```

clim:r-tree-output-record *Class*

CLIM provides this output record class to store sequences of output records that tend to overlap a great deal. The ordering of the tree is based on all four corners of the bounding rectangles of the records contained in it.

The insertion and retrieval complexity of this class is O(log n), but the overhead is fairly high. However, this output record class maintains temporal ordering (that is, stacking) very well.

clim:run-frame-top-level *frame* &key &allow-other-keys *Generic Function*

Runs the top-level function for *frame*. The default method merely runs the top-level function of *frame* as specified by the **:top-level** option of **clim:define-application-frame**. If no **:top-level** was specified, **clim:default-frame-top-level** is used.

The returned values are the values returned by the top level function of *frame*.

clim:application-frame provides an **:around** method which binds **clim:*application-frame*** to *frame*.

clim:scaling-transformation-p *transform* *Generic Function*

Returns **t** if *transform* multiplies all X-lengths by one magnitude and all Y-lengths by another magnitude, otherwise returns **nil**. This category includes even scalings as a subset.

clim:scroll-bar *Class*

The **clim:scroll-bar** gadget class corresponds to a scroll bar. It is a subclass of **clim:value-gadget**, **clim:oriented-gadget-mixin**, and **clim:range-gadget-mixin**.

In addition to the initargs for **clim:value-gadget** and the usual pane initargs (**:foreground**, **:background**, space requirement options, and so forth), the following initargs are supported:

:orientation
: Either **:vertical** or **:horizontal**.

:drag-callback
: Specifies the callback to be invoked when the scroll bar indicator is dragged.

:scroll-to-bottom-callback
: Specifies the callback to be invoked when the scroll bar will scroll to the bottom of the viewport.

:scroll-to-top-callback
: Specifies the callback to be invoked when the scroll bar will scroll to the top of the viewport.

:scroll-down-line-callback
Specifies the callback to be invoked when the scroll bar will scroll down one line.

:scroll-up-line-callback
Specifies the callback to be invoked when the scroll bar will scroll up one line.

:scroll-down-page-callback
Specifies the callback to be invoked when the scroll bar will scroll down one page.

:scroll-up-page-callback
Specifies the callback to be invoked when the scroll bar will scroll up one page.

The **clim:value-changed-callback** is invoked only after the indicator is released after dragging it.

Calling **clim:gadget-value** on a scroll bar will return a real number within the specified range of the scroll bar, usually between 0 and 1 (inclusive).

clim:scroll-extent *sheet x y* *Generic Function*

If the pane *sheet* is part of a scroller pane, this scrolls the pane in its viewport so that the position *(x,y)* of *sheet* is at the upper-left corner of the viewport. Otherwise, it does nothing.

clim:scroll-extent, and all other functions that change the position of the viewport, also call **clim:note-viewport-position-changed** to notify the sheet that it has been scrolled.

You can specialize **clim:scroll-extent** when you want to implement new scrolling behavior for a sheet. If you want to simply do something in addition to the normal scrolling behavior, you should specialize **clim:note-viewport-position-changed** instead.

clim:scrolling *(*&rest *options)* &body *contents* *Macro*

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; **:horizontal**, which indicates that only a horizontal scroll bar should be created; or **:none**, which indicates that the pane is scrollable but will have no visible scroll bars.

The pane created by the **clim:scrolling** macro is a composite pane that includes a ''scroller pane''. The ''scroller pane'' has as children the scroll bars and a ''viewport pane''. The viewport of a pane is the portion of the window's drawing plane that is currently visible to the user. The viewport has as its child the specified contents.

clim:select-file *frame* &key *:default :associated-window :title :exit-boxes :text-style :foreground :background :x-position :y-position* *Generic Function*

Pops up a file selection dialog in order to input a pathname from the user. The default pathname is specified by *:default*. The returned value is the pathname.

:associated-window is the window with which the notification is associated. *:title* is a string used to label the notification. *:exit-boxes* is as for **clim:accepting-values**.

:text-style, *:foreground*, and *:background* are the text style, foreground ink, and background ink to use in the notification window. *:x-position* and *:y-position* can be supplied to position the notification window.

sequence *type* *Clim Presentation Type*

The presentation type that represents a sequence of elements of type *type*. The printed representation of a **sequence** type is the elements separated by the separator character. It is unspecified whether **clim:accept** returns a list or a vector. You can specify the following options:

:separator The character that separates members of the set of possibilities in the printed representation when there is more than one. The default is comma (#\,).

:echo-space If this is **t**, then CLIM will insert a space automatically after the separator, otherwise it will not. The default is **t**.

type can be a presentation type abbreviation.

clim:sequence-enumerated &rest *types* *Clim Presentation Type*

clim:sequence-enumerated is like **sequence**, except that the type of each element in the sequence is individually specified. It is unspecified whether **clim:accept** returns a list or a vector. You can specify the following options:

:separator The character that separates members of the set of possibilities in the printed representation when there is more than one. The default is comma (#\,).

:echo-space If this is **t**, then CLIM will insert a space automatically after the separator, otherwise it will not. The default is **t**.

The elements of *types* can be presentation type abbreviations.

clim:set-highlighted-presentation *stream presentation* &optional *(prefer-pointer-window* **t***)* *Function*

Highlights the presentation *presentation* on *stream*. If *presentation* is **nil**, any highlighted presentations are unhighlighted.

If *prefer-pointer-window* is **t** (the default), this sets the highlighted presentation for the window that is located under the pointer. Otherwise it sets the highlighted presentation for the window *stream*.

clim:sheet *Class*

The protocol class that corresponds to a sheet. If you want to create a new class that obeys the sheet protocol, it must be a subclass of **clim:sheet**.

clim:sheet-adopt-child *sheet child* *Generic Function*

Adds the child sheet *child* to the set of children of *sheet*, and makes the *sheet* the child's parent. If *child* already has a parent, CLIM will signal an error.

Some sheet classes support only a single child. For such sheets, attempting to adopt more than a single child will cause CLIM to signal an error.

See the section "Sheet Relationship Protocols".

clim:sheet-children *sheet* *Generic Function*

Returns a list of all of the sheets that are children of *sheet*.

clim:sheet-children may cons a new list each time it is called. It may be more efficient to use **clim:map-over-sheets** instead of **clim:sheet-children**.

See the section "Sheet Relationship Protocols".

clim:sheet-device-region *sheet* *Generic Function*

Returns a region object that describes the region that *sheet* occupies on the display device. The coordinates are in the host's native window coordinate system.

The device region is usually an instance of **clim:standard-bounding-rectangle**.

Note that the region object returned by this function is volatile, so you must not depend on the components of the object remaining constant.

clim:sheet-device-transformation *sheet* *Generic Function*

Returns a transformation that converts coordinates in *sheet*'s coordinate system into native coordinates on the display device.

Note that the transformation object returned by this function is volatile, so you must not depend on the components of the object remaining constant.

clim:sheet-disown-child *sheet child &key (:errorp* **t***)* *Generic Function*

Removes the child sheet *child* from the set of children of *sheet*, and makes the parent of the child be **nil**. If *child* is not actually a child of *sheet* and *:errorp* is **t**, then an error will be signalled.

See the section "Sheet Relationship Protocols".

clim:sheet-enabled-p *sheet* *Generic Function*

Returns **t** if *sheet* is enabled by its parent, otherwise returns **nil**.

You can used **setf** on this in order to enable or disable a sheet.

In order for a sheet to be enabled and ''viewable'', all of a sheet's ancestors must be enabled and ''viewable'' as well.

clim:sheet-event-queue *sheet* *Generic Function*

Any sheet that can process events will have an event queue from which the events are gotten. **clim:sheet-event-queue** returns the object that acts as the event queue.

Typically, you will only use this function in order to cause one or more sheets to share the same event queue. For example, you can cause a window created with **clim:open-window-stream** to share the event queue of another frame as follows:

```
(setf (clim:sheet-event-queue window)
      (clim:sheet-event-queue
        (clim:frame-top-level-sheet frame)))
```

See the section "Sheet Input Protocols".

clim:sheet-medium *sheet* *Generic Function*

Returns the medium associated with *sheet*. If *sheet* does not have a medium allocated to it, **clim:sheet-medium** returns **nil**.

This function will signal an error if *sheet* does not support doing output.

See the section "Sheet Output Protocols".

clim:sheet-mirror *sheet* *Generic Function*

Returns the host window that is used to display *sheet*.

Note that this will nearly always return the same result as calling **clim:medium-drawable** on **clim:sheet-medium** of *sheet*.

clim:sheet-parent *sheet* *Generic Function*

Returns the sheet that is the parent of *sheet*, or **nil** if *sheet* has no parent.

clim:sheet-pointer-cursor *sheet* *Generic Function*

Returns the type of cursor that will be used for the pointer when the pointer is over *sheet*.

The valid cursor types are **:default**, **:vertical-scroll**, **:scroll-up**, **:scroll-down**, **:horizontal-scroll**, **:scroll-left**, **:scroll-right**, **:busy**, **:upper-left**, **:upper-right**, **:lower-left**, **:lower-right**, **:vertical-thumb**, **:horizontal-thumb**, **:button**, **:prompt**, **:move**, **:position**, and **:i-beam**.

You can use **setf** on **clim:sheet-pointer-cursor** to change the cursor type.

clim:sheet-region *sheet* *Generic Function*

Returns a region object that represents the set of points to which *sheet* refers. The region is usually a bounding rectangle object. Its coordinates are in the sheet's coordinate system (that is, the upper left corner of the region will typically be at (0,0)).

The region is usually an instance of **clim:standard-bounding-rectangle**.

You can use **setf** on this to change the sheet's region, although it is generally preferable to use **clim:resize-sheet** instead. When you change a sheet's region, CLIM will call **clim:note-sheet-region-changed** *sheet* to notify the sheet of the change.

See the section "Sheet Geometry Protocols".

clim:sheet-transformation *sheet* *Generic Function*

Returns a transformation that converts coordinates in *sheet*'s coordinate system into coordinates in its parent's coordinate system.

You can use **setf** on this to change the sheet's transformation, although it is generally preferable to use **clim:move-sheet** instead. When you change a sheet's region, CLIM will call **clim:note-sheet-transformation-changed** *sheet* to notify the sheet of the change.

See the section "Sheet Geometry Protocols".

clim:sheetp *object* *Generic Function*

Returns **t** if and only if *object* is of type **clim:sheet**, otherwise returns **nil**.

clim:+shift-key+ *Constant*

The modifier state bit that corresponds to the user holding down the shift key on the keyboard.

clim:simple-parse-error *Condition*

This condition is signalled when CLIM does not know how to parse some sort of user input while inside of **clim:accept**. It is built on **conditions:parse-error**.

clim:simple-parse-error *format-string* &rest *format-arguments* *Function*

Signals an error of type **clim:simple-parse-error**. This can be called while parsing an input token, for example, by a method on **clim:accept**. This function does not return.

clim:singular-transformation *Condition*

The condition that is signalled when you try to invert a singular (non-invertible) transformation.

clim:size-frame-from-contents *stream* &key *:width :height (:right-margin* **10***) (:bottom-margin* **10***) (:size-setter* **#'clim:window-set-inside-size***)* *Function*

This function resizes the frame that contains the pane *stream* to have the size *:width* and *:height*. If *:width* and *:height* are not supplied, they are computed by taking the size of the output contained within *stream*. The width and height are incremented by *:right-margin* and *:bottom-margin*.

This function is intended to be called on streams that are the only pane in their parent frame. For example, CLIM uses this function to properly size pop-up menus and own-window dialogs.

clim:slider *Class*

The **clim:slider** gadget class corresponds to a slider. It is a subclass of **clim:value-gadget**, **clim:oriented-gadget-mixin**, **clim:range-gadget-mixin**, and **clim:labelled-gadget-mixin**.

See the section "Using Gadgets in CLIM".

In addition to the initargs for **clim:value-gadget** and the usual pane initargs (**:foreground**, **:background**, **:text-style**, space requirement options, and so forth), the following initargs are supported:

:orientation
: Specifies the orientation of the slider, either **:horizontal** or **:vertical**. The default is **:horizontal**.

:drag-callback
: Specifies the drag callback for the slider.

:show-value-p
: Whether the slider should show its current value. The default is **nil**.

:decimal-places
: An integer that specifies the number of decimal places that should be shown if the current value is being shown. The default is to show all significant digits.

:min-label
: A string to use to label the minimum end of the slider. By default, there is no min label.

:max-label
: A string to use to label the maximum end of the slider. By default, there is no max label.

:range-label-text-style
: The text style to use for the min and max labels.

:number-of-tick-marks
: The number of tick marks to draw on the slider. By default, there are no tick marks on the slider.

:number-of-quanta
: Either **nil** (the default) or an integer. If an integer, specifies the number of ''quanta'' in the slider. In this case the slider is not continuous, and can only assume a value that falls on one of the ''quanta''. For example, a slider for real numbers will not be quantized, whereas a slider for integers will be quantized.

The **clim:drag-callback** callback is invoked when the value of the slider is changed while the indicator is being dragged. This is implemented by calling the function specified by the **:drag-callback** initarg with two arguments, the slider and the new value.

The **clim:value-changed-callback** is invoked only after the indicator is released after dragging it.

Calling **clim:gadget-value** on a slider will return a real number within the specified range of the slider.

Here are some examples of sliders:

```
(clim:make-pane 'clim:slider
  :label "A slider"
  :value-changed-callback 'slider-changed-callback
  :drag-callback 'slider-dragged-callback)

(clim:make-pane 'clim:slider
  :label "A slider with tick marks and range labels"
  :number-of-tick-marks 20
  :min-label "0" :max-label "20"
  :value-changed-callback 'slider-changed-callback
  :drag-callback 'slider-dragged-callback)
```

```
(clim:make-pane 'clim:slider
  :label "A vertical slider with visible value"
  :orientation :vertical
  :show-value-p t)

(clim:make-pane 'clim:slider
  :label "A very hairy quantized slider"
  :orientation :vertical
  :number-of-tick-marks 20
  :number-of-quanta 20
  :show-value-p t
  :min-value 0 :max-value 20
  :min-label "Min" :max-label "Max"
  :value-changed-callback 'slider-changed-callback
  :drag-callback 'slider-dragged-callback)

(defun slider-changed-callback (slider value)
  (format t "~&Slider ~A changed to ~S" (clim:gadget-label slider) value))

(defun slider-dragged-callback (slider value)
  (format t "~&Slider ~A dragged to ~S" (clim:gadget-label slider) value))
```

clim:slider-view *Class*

The class that represents the view corresponding to a slider. This is usually used for fields that represent a continuous range of values (such as a bounded range of real numbers).

clim:+slider-view+ *Constant*

An instance of the class **clim:slider-view**.

clim:space-requirement *Class*

The object that specifies a pane's desired width and height, as well as the amount it is willing to shrink or grow along its width and height.

clim:space-requirement+ *space-req* *Function*

Returns a new space requirement whose components are the sum of each of the components of *sr1* and *sr2*.

clim:space-requirement-components *space-req* *Function*

Returns the components of the space requirement *space-req* as six values, the width, minimum width, maximum width, height, minimum height, and maximum height.

clim:spacing *(*&rest *options* &key *:thickness :background* &allow-other-keys*)* &body *contents* *Macro*

The **clim:spacing** reserves some margin space around a single child pane. *:thickness* specifies the amount of space in device units, and *:background* specifies the ink to be used as the pane's background (that is, the color of the margin space). *contents* is a form that produces a single pane.

The **clim:spacing** macro is the usual way of creating a pane of type **clim:spacing-pane**.

options may include other pane initargs, such as space requirement options, **:foreground**, **:background**, **:text-style**, and so forth.

clim:spacing-pane *Class*

The layout pane class that leaves some empty space around its child pane. **clim:spacing** generates a pane of this type.

In addition to the usual sheet initargs (the space requirement initargs, **:foreground** and **:background**), this class supports two other initargs:

:thickness
An integer that specifies the amount of space to leave around the child pane, in device units.

:contents The pane that will be the child.

clim:*standard-activation-gestures* *Variable*

A list of gesture names that cause the current input to be activated. On most systems, this includes the gestures corresponding to the `#\Newline` (or `#\Return`) characters. On Genera, it includes the gesture for the `#\End` character as well.

clim:standard-application-frame *Class*

This is the class on which all standard CLIM application frames are based.

Typically when you make a new class of application frame, it should inherit from **clim:standard-application-frame**. If you do not explicitly supply any superclasses in a **clim:define-application-frame** form, CLIM will arrange for the new frame class to inherit from **clim:standard-application-frame**.

clim:standard-bounding-rectangle *Class*

The standard instantiable class for bounding rectangles in CLIM. All of CLIM's output record classes are built on **clim:standard-bounding-rectangle**.

clim:standard-ellipse *Class*

The standard class CLIM uses to implement an ellipse. This is a subclass of **clim:ellipse**. This is the class that **clim:make-ellipse** and **clim:make-ellipse*** instantiate.

clim:standard-elliptical-arc *Class*

The standard class CLIM uses to implement an elliptical arc. This is a subclass of **clim:elliptical-arc**. This is the class that **clim:make-elliptical-arc** and **clim:make-elliptical-arc*** instantiate.

clim:standard-line *Class*

The standard class CLIM uses to implement lines. This is a subclass of **clim:line**. This is the class that **clim:make-line** and **clim:make-line*** instantiate.

clim:standard-point *Class*

The standard class CLIM uses to implement points. This is the class that **clim:make-point** instantiates.

clim:standard-polygon *Class*

The standard class CLIM uses to implement polygons. This is a subclass of **clim:polygon**.

This is the class that **clim:make-polygon** and **clim:make-polygon*** instantiate.

clim:standard-polyline *Class*

The standard class CLIM uses to implement polylines. This is a subclass of **clim:polyline**. This is the class that **clim:make-polyline** and **clim:make-polyline*** instantiate.

clim:standard-presentation *Class*

The standard class used by CLIM to represent presentations. By default, **clim:present** and **clim:with-output-as-presentation** create presentations using this class.

clim:standard-rectangle *Class*

The standard class CLIM uses to implement rectangles. This is a subclass of **clim:rectangle**. This is the class that **clim:make-rectangle** and **clim:make-rectangle*** instantiate.

clim:standard-sequence-output-history *Class*

A standard class provided by CLIM for use as a top-level output history. This is a subclass of both **clim:standard-sequence-output-record** and **clim:stream-output-history-mixin**.

clim:standard-sequence-output-record *Class*

The standard class provided by CLIM to store a relatively short sequence of output records; a subclass of **clim:output-record**.

The insertion and retrieval complexity of this class is O(n). Most of the formatted output facilities (such as **clim:formatting-table**) create output records that are a subclass of **clim:standard-sequence-output-record**.

clim:standard-tree-output-history *Class*

The class used by CLIM as the default top-level output history. This is a subclass of both **clim:standard-tree-output-record** and **clim:stream-output-history-mixin**.

clim:standard-tree-output-record *Class*

The standard class provided by CLIM to store longer sequences of output records. The child records of a tree output record area maintained in a sorted order, based on the lexicographic ordering on the X and Y coordinates of the children.

The insertion and retrieval complexity of this class is O(log n), but it is highly optimized for doing textual-style output where most new output comes at the end (lower right) of the record.

clim:stream-add-output-record *stream record* *Generic Function*

Adds the new output record *record* to *stream*'s current output record (that is, **clim:stream-current-output-record**). This also takes care of other bookkeeping, such as adjusting *stream*'s scroll bars if the stream supports scrolling.

clim:stream-baseline *stream* *Generic Function*

Returns the current text baseline for the stream *stream*.

clim:stream-character-width *stream character* &optional *text-style* *Generic Function*

Returns the horizontal motion of the cursor position that would occur if this *character* were output onto *stream* in the text style *text-style*. **clim:stream-character-width** does not take into account the value of **clim:stream-text-margin** when computing the size of the output.

text-style defaults to the *stream*'s current text style. The answer depends on the current cursor position when the *character* is `#\Tab` or `#\Newline`.

clim-lisp:stream-clear-input *stream* *Generic Function*

Clears all pending input from *stream*'s input buffer.

In CLIM, **clear-input** is implemented by call **clim-lisp:stream-clear-input**.

clim:stream-current-output-record *stream* *Generic Function*

The current ''open'' output record for the output recording stream *stream*, that is, the one to which **clim:stream-add-output-record** will add a new child record. Initially, this is the same as **clim:stream-output-history**. As applications created nested output records, this acts as a stack of open output records.

clim:stream-cursor-position *stream* *Generic Function*

Returns two values, the X and Y coordinates of the cursor position on the drawing plane. You can use **clim:stream-set-cursor-position** or **clim:stream-increment-cursor-position** to change the cursor position.

clim:stream-default-view *stream* *Generic Function*

Returns the default view for the stream *stream*. You can change the default view for a stream by using **setf** on **clim:stream-default-view**. Calls to **clim:accept** default the **:view** argument from **clim:stream-default-view**.

Many CLIM streams will have the textual view, **clim:+textual-view+**, as their default view. Inside of **clim:menu-choose**, the default view will be **clim:+textual-menu-view+**. Inside of **clim:accepting-values**, the default view will be either **clim:+textual-dialog-view+** or **clim:+gadget-dialog-view+**.

clim:stream-drawing-p *stream* *Generic Function*

Returns **t** if and only if drawing is enabled on the output recording stream *stream*. You can use **setf** on this to enable or disable drawing on the stream, or you can use the **:draw** option to **clim:with-output-recording-options**.

clim:stream-element-type *stream* *Generic Function*

In CLIM, **stream-element-type** is defined as a generic function. Otherwise, it behaves the same as the normal Common Lisp **stream-element-type** function.

clim:stream-end-of-line-action *stream* *Generic Function*

Controls what happens when the cursor position moves horizontally out of the viewport (beyond the text margin). You can use **setf** on this to change the end of line action.

The possible values for **clim:stream-end-of-line-action** are:

: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 output.

:allow Ignore the text margin and just keep doing output.

You can use **clim:with-end-of-line-action** to temporarily change the end-of-line action.

clim:stream-end-of-page-action *stream* *Generic Function*

Controls what happens when the cursor position moves vertically out of the viewport. You can use **setf** on this to change the end of page action.

The possible values for **clim:stream-end-of-page-action** are:

: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 just keep doing output.

:wrap Wrap the text around (that is, go back to the top of the viewport). This is not currently implemented.

You can use **clim:with-end-of-page-action** to temporarily change the end-of-page action.

clim-lisp:stream-finish-output *stream* *Generic Function*

Some streams are implemented in an asynchronous, or buffered, manner. **clim-lisp:stream-finish-output** attempts to ensure that all output sent to *stream* has reached its destination, and only then returns **nil**.

In CLIM, **finish-output** is implemented by calling **clim-lisp:stream-finish-output**.

clim-lisp:stream-force-output *stream* *Generic Function*

Some streams are implemented in an asynchronous, or buffered, manner. **clim-lisp:stream-force-output** initiates the emptying of any internal buffers, and returns **nil** without waiting for completion or acknowledgment.

In CLIM, **force-output** is implemented by calling **clim-lisp:stream-force-output**.

clim-lisp:stream-fresh-line *stream* *Generic Function*

Outputs a newline only if *stream* is not already at the start of a line. If for any reason this cannot be determined, then a newline is output anyway. This guarantees that the stream will be on a fresh line while consuming as little vertical space as possible.

In CLIM, **fresh-line** is implemented by calling **clim-lisp:stream-fresh-line**.

clim:stream-increment-cursor-position *stream dx dy* *Generic Function*

Moves the cursor position on stream relatively, adding *dx* to the X coordinate and adding *dy* to the Y coordinate. Either argument *dx* or *dy* can be **nil**, which means not to change that coordinate.

clim:stream-input-wait *stream* &key *:timeout :input-wait-test* *Generic Function*

Waits until *:timeout* has expired or *:input-wait-test* returns a non-**nil** value. Otherwise the function waits until there is input in the *stream*.

:timeout Specifies the number of seconds that **clim:stream-input-wait** will wait for input to become available. If no input is available when the timeout expires, **clim:stream-input-wait** will return the two values **nil** and **:timeout**. If **nil** (the default), it will wait indefinitely.

:input-wait-test The value of this argument is a function. The function will be invoked with one argument, the stream. If the function returns **nil**, **clim:stream-input-wait** will continue to wait for user input. If it returns **t**, **clim:stream-input-wait** will return the two values **nil** and **:input-wait-test**.

clim:stream-insertion-pointer *input-editing-stream* *Generic Function*

Returns an integer corresponding to the current input position of *input-editing-stream*, 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 **(fill-pointer (clim:stream-input-buffer** *stream*)). The insertion pointer is used as the location of the editing cursor.

You can use **setf** on **clim:stream-insertion-pointer** to change the insertion pointer.

For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:stream-line-height *stream* &optional *text-style* *Generic Function*

Returns what the line height of a line containing text in that *text-style* would be. *text-style* defaults to **(clim:medium-text-style** *stream*).

clim-lisp:stream-listen *stream* *Generic Function*

Returns **t** if there is input available on *stream*, **nil** if not.

In CLIM, **listen** is implemented by calling **clim-lisp:stream-listen**.

clim:stream-output-history *stream* *Generic Function*

Returns the top level output record for the stream *stream*.

clim-lisp:stream-peek-char *stream* *Generic Function*

Returns the next character available in the input *stream*. The character is not removed from the input buffer. Thus, the same character will be returned by a subsequent call to **clim-lisp:stream-read-char**.

In CLIM, **peek-char** is implemented by calling **clim-lisp:stream-peek-char**.

clim:stream-pointer-position *stream* &key *:pointer* *Generic Function*

This function returns the position (two coordinate values) of the *:pointer* in the *stream'*s drawing plane coordinate system. If *:pointer* is not supplied, the default pointer for *stream*'s port is used.

You can use **clim:stream-set-pointer-position** to set the pointer position.

Manipulating the Pointer in CLIM

Concepts of Manipulating the Pointer in CLIM

A pointer is an input device that enables pointing at an area of the screen (for example, a mouse, or a a tablet). CLIM offers a set of operators that enable you to manipulate the pointer.

Operators for Manipulating the Pointer in CLIM

These functions are the higher-level functions for doing input via the pointer.

clim:tracking-pointer *(*&optional *stream* &key *:pointer :multiple-window :transformp (:context-type* **t***) :highlight)* &body *clauses*

Provides a general means for running code while following the position of a pointing device, and monitoring for other input events. Programmer-supplied code may be run upon occurrence of events such as motion of the pointer, clicking of a pointer button, or typing something on the keyboard.

clim:drag-output-record *stream output-record* &key *(:repaint* **t***) :multiple-window :erase :feedback (:finish-on-release* **t***)*
Enters an interaction mode in which user moves the pointer, and *output-record* follows the pointer by being dragged on *stream*.

clim:dragging-output *(*&optional *stream* &key *(:repaint* **t***) :multiple-window :finish-on-release)* &body *body*
Evaluates *body* to produce the output, and then invokes **clim:drag-output-record** to drag that output on *stream*.

clim:pointer-place-rubber-band-line* &key *:start-x :start-y (:stream* ***standard-input****) :pointer :multiple-window (:finish-on-release* **t***)*
Prompts for a line via the pointing device specified by *:pointer*. **clim:pointer-place-rubber-band-line*** returns four values, the start-x, start-y, end-x, and end-y of a line.

clim:pointer-input-rectangle* &key *:left :top :right :bottom (:stream* ***standard-input****) :pointer :multiple-window (:finish-on-release* **t***)*
Prompts for a rectangular area via the pointing device specified by *:pointer*. **clim:pointer-input-rectangle*** returns four values, the left, top, right, and bottom edges of a rectangle.

The following are lower level functions for managing the pointer more directly.

See the generic function **clim:stream-pointer-position**.

clim:stream-set-pointer-position *stream x y* &key *:pointer*
This function sets the position (two coordinate values) of the *:pointer* in the *stream*'s drawing plane coordinate system.

clim:port-pointer *port*
Returns the pointer object corresponding to the primary pointing device for the port *port*.

clim:port-modifier-state *basic-port*
Returns the state of the modifier keys for the port *port*.

clim:pointer-button-state *pointer*
Returns the current button state for *pointer*.

clim:pointer-position *pointer*
This function returns the position (as two coordinate values) of the pointer *pointer* in the coordinate system of the sheet that the pointer is currently over.

clim:pointer-set-position *pointer x y*
This function changes the position of the pointer *pointer* to be *(x,y)*.

clim:pointer-native-position *pointer*
This function returns the position (as two coordinate values) of the pointer *pointer* in the coordinate system of the port's graft (that is, its ''root window'').

clim:pointer-set-native-position *pointer x y*

This function changes the position of the pointer *pointer* to be *(x,y)*.

clim:pointer-sheet *pointer*

Returns the sheet over which the pointer *pointer* is currently positioned.

clim:pointer-cursor *pointer*

Returns the current cursor type for *pointer*. You can use **setf** to change it.

clim:stream-recording-p *stream* *Generic Function*

Returns **t** if and only if output recording is enabled on the output recording stream *stream*. You can use **setf** on this to enable or disable output recording on the stream, or you can use the **:record** option to **clim:with-output-recording-options**.

clim-lisp:stream-read-char *stream* *Generic Function*

Returns the next character available in the input *stream*. If no character is available, this function will wait until one becomes available.

In CLIM, **read-char** is implemented by calling **clim-lisp:stream-read-char**. Note that, unlike **read-char**, **clim-lisp:stream-read-char** does not echo the character it reads; CLIM's input editor does this when reading input inside of a call to **clim:with-input-editing**.

clim-lisp:stream-read-char-no-hang *stream* *Generic Function*

Like **clim-lisp:stream-read-char** except that if no character is available the function returns **nil**.

In CLIM, **read-char-no-hang** is implemented by calling **clim-lisp:stream-read-char-no-hang**.

clim:stream-read-gesture *stream* &key *:timeout :peek-p :input-wait-test :input-wait-handler :pointer-button-press-handler* *Generic Function*

Returns the next gesture available in the input stream. The arguments are as for **clim:read-gesture**. Note that **clim:stream-read-gesture** does not echo its input; CLIM's input editor does this when reading input inside of a call to **clim:with-input-editing**.

This function invokes **clim:stream-input-wait** on the stream and processes the subsequent input, if any. CLIM's input editor is implemented by specializing **clim:stream-read-gesture** to process input editing commands.

clim:read-gesture is implemented by calling **clim:stream-read-gesture**. This function is a generalization of **clim-lisp:stream-read-char** to extended input streams.

clim-lisp:stream-read-line *stream* *Generic Function*

Returns a string containing a line of text, delimited by the `#\Newline` character.

In CLIM, **read-line** is implemented by calling **clim-lisp:stream-read-line**.

clim:stream-replay *stream* &optional *region* *Generic Function*

Replays all of the output records in *stream*'s output history that overlap the region *region*. If *region* is **nil**, all of the output records are replayed.

clim:stream-rescanning-p *input-editing-stream* *Generic Function*

Returns the state of the input editing stream's ''rescan in progress'' flag, which is **t** if *input-editing-stream* is performing a rescan operation, otherwise it is **nil**. Non-input editing streams always return **nil**.

For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:stream-scan-pointer *input-editing-stream* *Generic Function*

Returns the current scan pointer (an integer) for *input-editing-stream*, that is, the point in the buffer at which calls to **clim:accept** have stopped parsing input. The scan pointer will always be less than or equal to **clim:stream-insertion-pointer** of *input-editing-stream*.

The next call to **clim:read-gesture** on *input-editing-stream* will return the gesture at the scan pointer.

You can use **setf** on **clim:stream-scan-pointer** to change the scan pointer.

For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:stream-set-cursor-position *stream x y* *Generic Function*

Moves the cursor position to the specified X and Y coordinates on the drawing plane.

clim:stream-set-input-focus *stream* *Generic Function*

Gives the input focus to *stream*, and returns as a value the stream or sheet that previously had the input focus.

clim:stream-set-pointer-position *stream x y* &key *:pointer* *Generic Function*

This function sets the position (two coordinate values) of the *pointer* in the *stream*'s drawing plane coordinate system (if possible). If not possible, the function

leaves the pointer where it was. If *:pointer* is not supplied, the pointer for stream's port is used.

Be careful when you use this function. It is often best to avoid creating user interfaces where the pointer jumps around unexpectedly.

Manipulating the Pointer in CLIM

Concepts of Manipulating the Pointer in CLIM

A pointer is an input device that enables pointing at an area of the screen (for example, a mouse, or a a tablet). CLIM offers a set of operators that enable you to manipulate the pointer.

Operators for Manipulating the Pointer in CLIM

These functions are the higher-level functions for doing input via the pointer.

clim:tracking-pointer *(*&optional *stream* &key *:pointer :multiple-window :transformp (:context-type* **t***) :highlight)* &body *clauses*
Provides a general means for running code while following the position of a pointing device, and monitoring for other input events. Programmer-supplied code may be run upon occurrence of events such as motion of the pointer, clicking of a pointer button, or typing something on the keyboard.

clim:drag-output-record *stream output-record* &key *(:repaint* **t***) :multiple-window :erase :feedback (:finish-on-release* **t***)*
Enters an interaction mode in which user moves the pointer, and *output-record* follows the pointer by being dragged on *stream*.

clim:dragging-output *(*&optional *stream* &key *(:repaint* **t***) :multiple-window :finish-on-release)* &body *body*
Evaluates *body* to produce the output, and then invokes **clim:drag-output-record** to drag that output on *stream*.

clim:pointer-place-rubber-band-line* &key *:start-x :start-y (:stream* ***standard-input****) :pointer :multiple-window (:finish-on-release* **t***)*
Prompts for a line via the pointing device specified by *:pointer*. **clim:pointer-place-rubber-band-line*** returns four values, the start-x, start-y, end-x, and end-y of a line.

clim:pointer-input-rectangle* &key *:left :top :right :bottom (:stream* ***standard-input****) :pointer :multiple-window (:finish-on-release* **t***)*
Prompts for a rectangular area via the pointing device specified by *:pointer*. **clim:pointer-input-rectangle*** returns four values, the left, top, right, and bottom edges of a rectangle.

The following are lower level functions for managing the pointer more directly.

clim:stream-pointer-position *stream* &key *:pointer*
This function returns the position (two coordinate values) of the pointer in the stream's drawing plane coordinate system. You can use **clim:stream-set-pointer-position** to set the pointer position.

See the generic function **clim:stream-set-pointer-position**.

clim:port-pointer *port*
Returns the pointer object corresponding to the primary pointing device for the port *port*.

clim:port-modifier-state *basic-port*
Returns the state of the modifier keys for the port *port*.

clim:pointer-button-state *pointer*
Returns the current button state for *pointer*.

clim:pointer-position *pointer*
This function returns the position (as two coordinate values) of the pointer *pointer* in the coordinate system of the sheet that the pointer is currently over.

clim:pointer-set-position *pointer x y*
This function changes the position of the pointer *pointer* to be *(x,y)*.

clim:pointer-native-position *pointer*
This function returns the position (as two coordinate values) of the pointer *pointer* in the coordinate system of the port's graft (that is, its ''root window'').

clim:pointer-set-native-position *pointer x y*
This function changes the position of the pointer *pointer* to be *(x,y)*.

clim:pointer-sheet *pointer*
Returns the sheet over which the pointer *pointer* is currently positioned.

clim:pointer-cursor *pointer*
Returns the current cursor type for *pointer*. You can use **setf** to change it.

clim:stream-string-width *stream string* &key *:start :end :text-style* *Generic Function*

Computes how the cursor position would move horizontally if the specified *string* were output starting at the left margin. **clim:stream-string-width** does not take into account the value of **clim:stream-text-margin** when computing the size of the output.

The first value is the X coordinate the cursor position would move to. The second 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`.)

:start and *:end* default to 0 and the length of the string, respectively.

:text-style defaults to the stream's current text style.

Note that **clim:stream-string-width** is a low level function. Unless you are writing your own ''formatting engine'', if you find you need to use it very often, you may be working at too low a level of abstraction.

clim-lisp:stream-terpri *stream* *Generic Function*

Outputs a newline to *stream*, and returns **nil**. It is identical in effect to:

```
(write-char #\Newline stream)
```

In CLIM, **terpri** is implemented by calling **clim-lisp:stream-terpri**.

clim:stream-text-cursor *stream* *Generic Function*

Returns the text cursor for the stream *stream*.

The text cursor is of type **clim:cursor**.

clim:stream-text-margin *stream* *Generic Function*

The X coordinate at which text wraps around (see **clim: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** on **clim:stream-text-margin**. If a value of **nil** is specified, the width of the viewport will be used. If the width of the viewport is later changed, the text-margin will change too.

clim-lisp:stream-unread-char *stream character* *Generic Function*

Places the specified *character* back into *stream*'s input buffer. The next **read-char** request will return the unread character. The character supplied must be the most recent character read from the stream.

In CLIM, **unread-char** is implemented by calling **clim-lisp:stream-unread-char**.

clim:stream-unread-gesture *stream gesture* *Generic Function*

Places the specified *gesture* back into *stream*'s input buffer. The next **clim:stream-read-gesture** request will return the unread gesture. The gesture supplied must be the most recent gesture read from the stream.

clim:read-gesture is implemented by calling **clim:stream-read-gesture**. This function is a generalization of **clim-lisp:stream-unread-char** to extended input streams.

clim:stream-vertical-spacing *stream* *Generic Function*

Returns the current inter-line spacing for the stream *stream*.

clim-lisp:stream-write-char *stream char* *Generic Function*

Writes *char* to *stream* and returns *char* as its value.

In CLIM, **write-char** is implemented by calling **clim-lisp:stream-write-char**.

clim-lisp:stream-write-string *stream string* &optional *start end* *Generic Function*

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.

In CLIM, **write-string** is implemented by calling **clim-lisp:stream-write-string**.

string &optional *length* *Clim Presentation Type*

The presentation type that represents a string. If *length* is specified, the string must have exactly that many characters.

clim:subset &rest *elements* *Clim Presentation Type Abbreviation*

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 the separator character. The options (**:name-key**, **:value-key**, **:partial-completers**, **:separator**, and **:echo-space**) are the same as for **clim:subset-completion**.

clim:subset-completion *sequence* &key *:test :value-key* *Clim Presentation Type*

The presentation 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 **clim:completion** with the following additional options:

:separator The character that separates members of the set of possibilities in the printed representation when there is more than one. The default is comma.

:echo-space (**t** or **nil**) Whether to insert a space automatically after the separator. The default is **t**.

The other subset types (**clim:subset**, **clim:subset-sequence**, and **clim:subset-alist**) are implemented in terms of the **clim:subset-completion** type.

clim:subset-sequence *sequence* &key *:test* *Clim Presentation Type Abbreviation*

Like **clim:subset**, except that the set of possibilities is the sequence *sequence*. The parameter *:test* and the options (**:name-key**, **:value-key**, **:partial-completers**, **:separator**, and **:echo-space**) are the same as for **clim:subset-completion**.

clim:subset-alist *alist* &key *:test* *Clim Presentation Type Abbreviation*

Like **clim:subset**, except that the set of possibilities is the alist *alist*. The parameter *:test* and the options (**:name-key**, **:value-key**, **:partial-completers**, **:separator**, and **:echo-space**) are the same as for **clim:subset-completion**. The parameter *alist* has the same format as **clim:member-alist**.

clim:substitute-numeric-argument-marker *command numeric-arg* *Function*

Given a command object *command*, this substitutes the value of *numeric-arg* for all occurrences of the value of **clim:*numeric-argument-marker*** in the command, and returns a command object with those substitutions.

clim:suggest *name* &rest *objects* *Function*

Specifies one possibility for **clim:completing-from-suggestions**. *completion* is a string, the printed representation. *object* is the internal representation.

This function has lexical scope and is defined only inside the body of **clim:completing-from-suggestions**.

clim:+super-key+ *Constant*

The modifier state bit that corresponds to the user holding down the super key on the keyboard. See the section "Operators for Gestures in CLIM".

clim:surrounding-output-with-border *(*&optional *stream* &key *(:shape* **:rectangle***)* *(:move-cursor* **t***))* &body *body* *Macro*

Binds the local environment in such a way that the output of *body* will be surrounded by a border of the specified *:shape*.

:shape The shape of the border to draw. The default is **:rectangle**. Other valid shapes are **:oval**, **:drop-shadow**, and **:underline**.

:move-cursor

When **t** (the default), CLIM moves the text cursor to the end (lower right corner) of the output. Otherwise, the cursor is left at the beginning (upper left corner) of the output.

symbol *Clim Presentation Type*

The presentation type that represents a symbol.

t *Clim Presentation Type*

The supertype of all other presentation types.

Note that the **clim:accept** method for this type allows input only via the pointer; if the user types anything on the keyboard, the **clim:accept** method just beeps.

clim:table-pane *Class*

The layout pane class that arranges its children in a tabular format. **clim:tabling** generates a pane of this type.

clim:tabling *(*&rest *options)* &body *contents* *Macro*

The **clim:tabling** macro lays out its child panes in a two-dimensional table arrangement. Each of the table is specified by an extra level of list in *contents*. For example,

```
(clim:tabling ()
  (list
    (clim:make-pane 'label :text "Red")
    (clim:make-pane 'label :text "Green")
    (clim:make-pane 'label :text "Blue"))
  (list
    (clim:make-pane 'label :text "Intensity")
    (clim:make-pane 'label :text "Hue")
    (clim:make-pane 'label :text "Saturation")))
```

The **clim:tabling** macro serves as the usual interface for creating a **clim:table-pane**.

clim:test-presentation-translator *translator presentation context-type frame window x y* &key *:event (:modifier-state* **0***) :for-menu* *Function*

Returns **t** if the translator *translator* applies to the presentation *presentation* in input context type *context-type*. (There is no *from-type* argument because it is derived from *presentation*.)

frame is the application frame. *window*, *x*, and *y* are the window the presentation is on, and the X and Y position of the pointer (respectively).

:event and *:modifier-state* are, respectively, a pointer button event and a modifier state. These are compared against the translator's gesture-name. *:event* defaults to **nil**, and *:modifier-state* defaults to 0, meaning that no shift 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.

If the translator is not applicable, **clim:test-presentation-translator** will return **nil**.

clim:test-presentation-translator is responsible for matching type parameters and calling the translator's tester. Under some circumstances, **clim:test-presentation-translator** may also call the body of the translator to ensure that its value matches *to-type*.

clim:text-editor *Class*

The **clim:text-editor** gadget class corresponds to a multi-line field containing text, a subclass of **clim:text-field**.

The value of a text editor is the text string.

See the section "Using Gadgets in CLIM".

In addition to the usual pane initargs (**:foreground**, **:background**, **:text-style**, space requirement options, and so forth), the following initargs are supported:

:editable-p
When **nil**, the text field cannot be modified. When **t** (the default), the text field can be modified.

:ncolumns
An integer specifying the width of the text editor in characters.

:nlines An integer specifying the height of the text editor in lines.

```
(clim:make-pane 'clim:text-editor
  :value "Isn't Lisp the greatest?"
  :value-changed-callback 'text-field-changed
  :ncolumns 40 :nlines 5)

(defun text-field-changed (tf value)
  (format t "~&Text field ~A changed to ~S" tf value))
```

clim:text-editor-view *Class*

The class that represents the view corresponding to a text editor pane (that is, a multi-line text editing field).

clim:+text-editor-view+ *Constant*

An instance of the class **clim:text-editor-view**.

clim:text-field *Class*

The gadget class that implements a text field. This is a subclass of both **clim:value-gadget** and **clim:action-gadget**.

The value of a text field is the text string.

See the section "Using Gadgets in CLIM".

In addition to the usual pane initargs (**:foreground**, **:background**, **:text-style**, space requirement options, and so forth), the following initargs are supported:

:editable-p
: When **nil**, the text field cannot be modified. When **t** (the default), the text field can be modified.

You might create a text field as follows:

```
(clim:make-pane 'clim:text-field
  :value "Name of your product here"
  :value-changed-callback 'text-field-changed
  :width 400)

(defun text-field-changed (tf value)
  (format t "~&Text field ~A changed to ~S" tf value))
```

clim:text-field-view *Class*

The class that represents the view corresponding to a text editor pane (that is, a single line text editing field).

clim:+text-field-view+ *Constant*

An instance of the class **clim:text-field-view**.

clim:text-size *medium string* &key *:text-style :start :end* *Generic Function*

Computes how the cursor position would move if the specified *string* or character were output to *medium* starting at cursor position (0,0). **clim:text-size** does not take into account the value of **clim:stream-text-margin** when computing the size of the output.

clim:text-size returns five values;

- 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 for each `#\Newline` character in the string
- The string's baseline.

:text-style defaults to `(clim:medium-text-style medium)`.

You can use *:start* and *:end* to specify a substring of *string*.

Note that **clim:text-size** is a low level function. Unless you are writing your own ''formatting engine'', if you find you need to use it very often, you may be working at too low a level of abstraction.

clim:text-style *Class*

The class that represents text styles. CLIM text styles have family, face, and size components. Each of these components has a corresponding reader accessor that can be used to extract a particular component from a text style.

See the section "Text Styles in CLIM".

clim:text-style-ascent *text-style medium* *Generic Function*

The ascent (a real number) of *text-style* as it would be rendered on *medium*.

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.

clim:text-style-components *text-style medium* *Generic Function*

Returns the components of *text-style* as three values (family, face, and size).

clim:text-style-descent *text-style medium* *Generic Function*

The descent (a real number) of *text-style* as it would be rendered on *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'').

clim:text-style-face *text-style* *Generic Function*

Returns the face component of the *text-style*.

clim:text-style-family *text-style* *Generic Function*

Returns the family component of the *text-style*.

clim:text-style-fixed-width-p *text-style medium* *Generic Function*

Returns **t** if *text-style* will map to a fixed-width font on *medium*, otherwise returns **nil**.

clim:text-style-height *text-style medium* *Generic Function*

Returns the height (a real number) of the ''usual character'' in *text-style* on *medium*.

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

clim:text-style-mapping *port style* &optional *character-set* *Generic Function*

Returns the font object that will be used if characters in *character-set* in the text style *style* are drawn on any medium on the port *port*. *character-set* defaults to the standard character set.

If the port is using exact text style mapping, CLIM will choose a font whose size exactly matches the size specified in the text style. Otherwise if the port is using loose text style mappings, CLIM will choose the font whose size is closest to the desired size.

If no mapping exists, CLIM will signal an error.

clim:text-style-mapping-exists-p *port style* &optional *character-set exact-size-required* *Generic Function*

Returns **t** if there is a font associated with the text style *style* on the port *port*, otherwise returns **nil**. *character-set* defaults to the standard character set. If *exact-size-required* is **t**, only fonts whose size is the exact size specified in the text style will be considered to match.

clim:text-style-p *object* *Generic Function*

Returns **t** if and only if *object* is a CLIM text style, otherwise returns **nil**.

clim:text-style-size *text-style* *Generic Function*

Returns the size component of the *text-style*.

clim:text-style-width *text-style medium* *Generic Function*

Returns the width (a real number) of the ''usual character'' in *text-style* on *medium*.

clim:textual-dialog-view *Class*

The class that represents the view that is used inside textual **clim:accepting-values** dialogs.

clim:+textual-dialog-view+ *Constant*

An instance of the class **clim:textual-dialog-view**. Inside **clim:accepting-values**, the default view for the dialog stream may be bound to **clim:+textual-dialog-view+**.

clim:textual-menu-view *Class*

The class that represents the view that is used inside textual menus.

clim:+textual-menu-view+ *Constant*

An instance of the class **clim:textual-menu-view**. Inside **clim:menu-choose**, the default view for the menu stream may be bound to **clim:+textual-menu-view+**.

clim:textual-view *Class*

The class that represents textual views. Textual views are used in most command-line oriented applications.

clim:+textual-view+ *Constant*

An instance of the class **clim:textual-view**.

clim:throw-highlighted-presentation *presentation input-context button-press-event* *Function*

Calls the applicable translator for the *presentation*, *input-context*, and *button-press-event* (that is, the one corresponding to the user clicking a pointer button while over the presentation). This function returns an object and a presentation type. These values are returned from the translator to the call to **clim:with-input-context** that establish the matching input context.

When you call **clim:throw-highlighted-presentation** yourself, you should be careful to ensure that **clim:*application-frame*** is bound to the relevant application frame

clim:title-pane *Class*

The pane class that is used to implement a title pane. It corresponds to the pane type abbreviation **:title** in the **:panes** clause of **clim:define-application-frame**. The default display function for panes of this type is **clim:display-title**.

For **clim:title-pane**, the default for the **:display-time** option is **t**, and the default for the **:scroll-bars** option is **nil**.

clim:toggle-button *Class*

The **clim:toggle-button** gadget class provides ‘‘on/off’’ switch behavior. It is a subclass of **clim:value-gadget** and **clim:labelled-gadget-mixin**. This gadget typically

appears as a box that is optionally highlighted with a check-mark. If the check-mark is present, the gadget's value is **t**, otherwise it is **nil**.

See the section "Using Gadgets in CLIM".

In addition to the initargs for **clim:value-gadget** and the usual pane initargs (**:foreground**, **:background**, **:text-style**, space requirement options, and so forth), the following initargs are supported:

:label A string or pixmap that is used to label the button.

:indicator-type
This is used to initialize the indicator type property for the gadget, and must 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.

clim:armed-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), **clim:value-changed-callback** will be invoked. Finally, **clim:disarmed-callback** will be invoked after **clim:value-changed-callback**, or when the pointer is moved outside of the toggle button.

Calling **clim:gadget-value** on a toggle button will return **t** if the button is selected, otherwise it will return **nil**. The value of the toggle button can be changed by calling **setf** on **clim:gadget-value**.

A toggle button might be created as follows:

```
(clim:make-pane 'clim:toggle-button
  :label "Toggle" :width 80
  :value-changed-callback 'toggle-button-callback)

(defun toggle-button-callback (button value)
  (format t "~&Button ~A toggled to ~S" (clim:gadget-label button) value))
```

clim:toggle-button-view *Class*

The class that represents the view corresponding to a toggle button. This is usually used for boolean (yes or no) values.

clim:+toggle-button-view+ *Constant*

An instance of the class **clim:toggle-button-view**.

clim:token-or-type *tokens type* *Clim Presentation Type Abbreviation*

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 *alist* parameter to **clim:member-alist**; typically it is a list of keyword symbols.

type can be a presentation type abbreviation.

For example, the following is a common way of using **clim:token-or-type**:

```
(clim:accept '(clim:token-or-type (:all :none) integer)
             :prompt "How many?")
```

clim:tracking-pointer *(*&optional *stream* &key *:pointer :multiple-window :transformp (:context-type* **t***) :highlight)* &body *clauses* *Macro*

Provides a general means for running code while following the position of a pointing device on the stream *stream*, and monitoring for other input events. *stream* defaults to ***standard-input***.

Programmer-supplied code may be run upon 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 over a presentation
- Keyboard event (typing a character)

The keyword arguments to **clim:tracking-pointer** are:

:pointer Specifies a pointer to track. It defaults to **(clim:port-pointer (clim:port** *stream*)). Unless there is more than one pointing device available, it is unlikely that this option will be useful.

:multiple-window
When **t**, specifies that the pointer is to be tracked across multiple windows. The default is **nil**.

Note that when *:multiple-window* is **t**, the **clim:tracking-pointer** clauses will be invoked on many different types of panes besides just CLIM stream panes, including scroll bars, borders, and so forth. Your program must filter out any panes it is not interested in.

:transformp
When **t**, specifies that coordinates supplied to the *:pointer-motion* clause are to be expressed in the user coordinate system. The default is **nil**.

:context-type
Specifies the type of presentations that will be ‘‘visible’’ to the

tracking code. It defaults to **t**, meaning that all presentations are visible.

:highlight Specifies whether or not CLIM should highlight presentations. It defaults to **t** when there are any presentation clauses, meaning that presentations that match *:context-type* should be highlighted. If there are no presentation clauses, it defaults to **nil**.

The body of **clim:tracking-pointer** consists of *clauses*. Each *clause* in *clauses* is of the form (*clause-keyword arglist* &body *clause-body*) and defines a local function to be run upon occurrence of each type of event. The possible *clause-keywords*, their *arglists*, and their uses are:

:pointer-motion (*window x y*)

Defines a clause that runs 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** above for a description of the coordinate system in which *x* and *y* is expressed.)

When both **:presentation** and **:pointer-motion** clauses are provided, only one of them will be run for a given motion event. The **:presentation** clause will run if it is applicable, otherwise the **:pointer-motion** clause will run.

:pointer-button-press (*event x y*)

Defines a clause that runs whenever a pointer button is pressed. In the clause, *event* is bound to the event object. (The window and the coordinates of the pointer are part of *event*.)

When both **:presentation-button-press** and **:pointer-button-press** clauses are provided, only one of them will be run for a given button press event. The **:presentation-button-press** clause will run if it is applicable, otherwise the **:pointer-button-press** clause will run.

x and *y* are the transformed X and Y positions of the pointer. These will be different from **clim:pointer-event-x** and **clim:pointer-event-y** if *stream* is using a non-identity transformation.

:pointer-button-release (*event x y*)

Defines a clause that runs whenever the pointer button is released. In the clause, *event* is bound to the event object. (The window and the coordinates of the pointer are part of *event*.)

When both **:presentation-button-release** and **:pointer-button-release** clauses are provided, only one of them will be run for a given button release event. The **:presentation-button-release** clause will run if it is applicable, otherwise the **:pointer-button-release** clause will run.

x and *y* are the transformed X and Y positions of the pointer. These will be different from **clim:pointer-event-x** and **clim:pointer-event-y** if *stream* is using a non-identity transformation.

:presentation (*presentation window x y*)
Defines a clause that runs whenever the pointer moves over a presentation of the desired type. (See the keyword argument **:context-type** above 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** above for a description of the coordinate system in which *x* and *y* is expressed.)

:presentation-button-press (*presentation event x y*)
Defines a clause that runs whenever the pointer button is pressed while the pointer is over a presentation of the desired type. (See the keyword argument **:context-type** above for a description of how to specify the desired type.) In the clause, *presentation* is bound to the presentation, and *event* to the event object. (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 **clim:pointer-event-x** and **clim:pointer-event-y** if *stream* is using a non-identity transformation.

:presentation-button-release (*presentation event x y*)
Defines a clause that runs whenever the pointer button is released while the pointer is over a presentation of the desired type. (See the keyword argument **:context-type** above for a description of how to specify the desired type.) In the clause, *presentation* is bound to the presentation, and *event* to the event object. (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 **clim:pointer-event-x** and **clim:pointer-event-y** if *stream* is using a non-identity transformation.

:keyboard (*event*)
Defines a clause that runs whenever a key is typed on the keyboard. In the clause, *event* is bound to the keyboard event typed. If the event corresponds to a standard printing character, *event* may be a character object.

A simple version of **clim:pointer-place-rubber-band-line*** could be implemented in the following manner using **clim:tracking-pointer**.

```
(defun pointer-place-rubber-band-line* (&optional (stream *standard-input*))
  (let (start-x start-y end-x end-y)
    (flet ((finish (event finish &optional press)
             (let ((x (clim:pointer-event-x event))
                   (y (clim:pointer-event-y event))
                   (window (clim:event-sheet event)))
               (when (eq window stream)
                 (cond (start-x
                        (clim:with-output-recording-options
                            (window :draw t :record nil)
                          (clim:draw-line* window start-x start-y end-x end-y
                                           :ink clim:+flipping-ink+))
                        (clim:draw-line* window start-x start-y end-x end-y)
                        (when finish
                          (return-from pointer-place-rubber-band-line*
                            (values start-x start-y end-x end-y))))
                       (press (setq start-x x start-y y)))))))
      (declare (dynamic-extent #'finish))
      (clim:tracking-pointer (stream)
        (:pointer-motion (window x y)
         (when (and start-x (eq window stream))
           (clim:with-output-recording-options (window :draw t :record nil)
             (when end-x
               (clim:draw-line* window start-x start-y end-x end-y
                                :ink clim:+flipping-ink+))
             (setq end-x x end-y y)
             (clim:draw-line* window start-x start-y end-x end-y
                              :ink clim:+flipping-ink+))))
        (:pointer-button-press (event)
         (finish event nil t))
        (:pointer-button-release (event)
         (finish event t))))))
```

clim:transform-distance *transform dx dy* *Generic Function*

Applies *transform* to the distance represented by *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.

clim:transform-position *transform x y* *Generic Function*

Applies *transform* to the point whose coordinates are *x* and *y*, and returns two values, the transformed X-coordinate and the transformed Y-coordinate.

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

clim:transform-rectangle* *transform x1 y1 x2 y2* *Generic Function*

Applies the transformation *transform* to the rectangle specified by the four coordinate arguments, which are real numbers. One corner of the rectangle is at *(x1,y1)* and the opposite corner is at *(x2,y2)*. If *transform* is not a rectilinear transformation, this will signal an error.

This 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*.

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

clim:transform-region *transformation region* *Generic Function*

Applies *transformation* to *region*, and returns a new transformed region.

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, it does not side-effect the *region* argument.

clim:transformation *Class*

The protocol class for all transformations. There are one or more subclasses of **clim:transformation** with implementation-dependent names that implement transformations. If you want to create a new class that obeys the transformation protocol, it must be a subclass of **clim:transformation**.

clim:transformation-equal *transform1 transform2* *Generic Function*

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

clim:transformation-underspecified *Condition*

The condition that is signalled when you try to make a 3-point transformation from three collinear points.

clim:transformationp *object* *Function*

Returns **t** if an only if *object* is a CLIM transformation.

clim:translate-coordinates *x-delta y-delta* &body *coordinate-pairs* *Function*

Translates each of the X and Y coordinate pairs in *coordinate-pairs* by *x-delta* and *y-delta*. *x-delta* and *y-delta* are real numbers.

clim:translation-transformation-p *transform* *Generic Function*

Returns **t** if *transform* 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 returns **nil**.

clim:+transparent-ink+ *Constant*

When you draw a design that has areas of **clim:+transparent-ink+**, the former background shows through in those areas. Typically, **clim:+transparent-ink+** is used as one of the inks in a pattern so that parts of the pattern are transparent.

clim:tree-recompute-extent *record* *Generic Function*

You can use this function to ensure that the bounding rectangles for a tree of output records are up to date. Use this whenever the bounding rectangles of a number of children of a record have been changed, such as happens during table and graph formatting. **clim:tree-recompute-extent** computes the bounding rectangle large enough to contain all of the children of *record*, adjusts the bounding rectangle of *record* accordingly, and then calls **clim:recompute-extent-for-changed-child** on *record*.

If you write a new formatting facility that rearranges many of the descendants of an output record (for example, a new kind of graph formatting), you should call **clim:tree-recompute-extent** on the parent of the highest level record that was affected. It is preferable to use **clim:tree-recompute-extent** instead of repeatedly calling **clim:recompute-extent-for-changed-child**, because you will get better performance.

See the section "Concepts of CLIM Output Recording".

clim:type-or-string *type* *Clim Presentation Type Abbreviation*

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 **clim:accept** cannot parse as the representation of an object of type *type* is returned as a string.

type can be a presentation type abbreviation.

clim:unhighlight-highlighted-presentation *stream* &optional *(prefer-pointer-window* **t***)* *Function*

Unhighlights any highlighted presentations on *stream*.

If *prefer-pointer-window* is **t** (the default), this clears the highlighted presentation for the window that is located under the pointer. Otherwise it clears the highlighted presentation for the window *stream*.

clim:unread-gesture *gesture* &key *(:stream* ***standard-input****)* *Function*

Places the specified *gesture* back into *:stream*'s input buffer. The next **clim:read-gesture** request will return the unread gesture. The gesture supplied must be the most recent gesture read from the stream.

clim:*unsupplied-argument-marker* *Variable*

If you are building a command object that has required arguments that have not yet been supplied, use the value of **clim:*unsupplied-argument-marker*** as a placeholder for those arguments. When CLIM goes to execute a command that has any unsupplied arguments, it will first gather those arguments from the user via a menu or a dialog.

For example, if you have a Hardcopy File command that takes two required arguments, a pathname and a printer, you might write a translator as follows:

```
(clim:define-presentation-to-command-translator printer-to-hardcopy-file
    (printer com-hardcopy-file hardcopy
     :gesture :select
     :pointer-documentation "Hardcopy File")
    (object)
  (list clim:*unsupplied-argument* object))
```

clim:untransform-distance *transform dx dy* *Generic Function*

Applies the inverse of *transform* to the distance represented by *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.

clim:untransform-position *transform x y* *Generic Function*

Applies the inverse of *transform* to the point whose coordinates are *x* and *y,* and returns two values, the transformed X-coordinate and the transformed Y-coordinate.

clim:untransform-position is the spread version of **clim:untransform-region** in the case where the region is a point.

clim:untransform-rectangle* *transform x1 y1 x2 y2* *Generic Function*

Applies the inverse of *transform* to the rectangle specified by the four coordinate arguments, and returns four values that specify the minimum and maximum points of the transformed rectangle.

clim:untransform-rectangle* is the spread version of **clim:untransform-region** in the case where the region is a rectangle.

clim:untransform-region *transformation region* *Generic Function*

Applies the inverse of *transformation* to *region* and returns a new transformed region.

This is equivalent to:

```
(clim:transform-region
  (clim:invert-transformation transform) region)
```

clim:updating-output *(stream* &rest *args* &key *(:record-type* **''clim:standard-updating-output-record***) :unique-id (:id-test* **'#'eql***) :cache-value (:cache-test* **'#'eql***) :copy-cache-value :parent-cache :output-record :fixed-position :all-new* &allow-other-keys*)* &body *body*

Macro

Informs CLIM's incremental redisplay facilities of the characteristics of the output done by *body* to *stream*. Within **clim:updating-output**, you name a piece of output (with a unique id), and you state how to determine whether the output changes (with a cache value).

For related information, see the section "Using **clim:updating-output**".

:unique-id Provides a means to uniquely identify this output. If *:unique-id* is not supplied, CLIM will generate one that is guaranteed to be unique.

:id-test A function of two arguments that is used for comparing unique ids. It defaults to **eql**.

:cache-value
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 A function of two arguments that is used for comparing cache values. It defaults to **eql**.

:copy-cache-value
Controls whether the specified cache value should be copied using **copy-seq** before it is stored in the output record.

:fixed-position
Declares that the location of this output is fixed relative to its superior. When CLIM redisplays an output record which specified **:fixed-position t**, 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.

:all-new Indicates that all of the output done by *body* is new, and will never match output previously recorded.

:record-type
The type of output record that should be constructed. This defaults to CLIM's standard **clim:updating-output-record** class.

clim:user-command-table *Clim Command Table*

A command table reserved for user-defined commands.

clim-sys:using-resource *(variable resource* &rest *parameters)* &body *body* *Macro*

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 the body has been evaluated, **clim-sys:using-resource** returns the object in *variable* back to the resource. If some form in the body sets *variable* to **nil**, the object will not be returned to the resource. Otherwise, the body should not changes the value of *variable*.

See the section "Resources in CLIM".

clim:value-changed-callback *gadget client id value* *Generic Function*

This callback is invoked when the value of a gadget is changed, either by the user or programatically.

The default method (on **clim:value-gadget**) calls the function specified by the **:value-changed-callback** initarg with two arguments, the gadget and the new value.

Although you can specialize this function yourself, generally this function will simply call another programmer-specified callback function.

See the section "Using Gadgets in CLIM".

clim:value-gadget *Class*

The class used by gadgets that have a value; a subclass of **clim:basic-gadget**.

All subclasses of **clim:value-gadget** must handle the two initargs **:value** and **:value-changed-callback**, which are used to specify, respectively, the initial value and the value changed callback of the gadget. The value changed callback is either **nil** or a function of two arguments, the gadget and the new value.

clim:vertically *(*&rest *options* &key *:spacing* &allow-other-keys*)* &body *contents* *Macro*

The **clim:vertically** macro lays out one or more child panes vertically, from top to bottom. The **clim:vertically** macro serves as the usual interface for creating a **clim:vbox-pane**.

:spacing is an integer that specifies how much space should be left between each child pane, in device units. *options* may include other pane initargs, such as space requirement options, **:foreground**, **:background**, **:text-style**, and so forth.

contents is one or more forms that produce the child panes. Each form in *contents* is of the form:

- A pane. The pane is inserted at this point and its space requirements are used to compute the size.
- A number. The specified number of device units should be allocated at this point.
- The symbol **clim:+fill+**. This means that an arbitrary amount of space can be absorbed at this point in the layout.
- 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 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:

```
(clim:vertically ()
  (9/10 (clim:make-clim-application-pane))
  (1/10 (clim:make-clim-interactor-pane)))
```

  would create a vertical stack of two stream panes. The application pane takes nine-tenths of the space, and the interactor pane takes one-tenth of the space.

See the section "Using the :LAYOUTS Option to CLIM:DEFINE-APPLICATION-FRAME".

clim:vbox-pane *Class*

The layout pane class that arranges its children in a vertical stack. **clim:vertically** generates a pane of this type.

In addition to the usual sheet initargs (the space requirement initargs, **:foreground**, **:background**, and **:text-style**), this class supports two other initargs:

:spacing An integer that specifies the amount of space to leave between each of the child panes, in device units.

:contents A list of panes that will be the child panes of the box pane.

clim:window-children *window* *Generic Function*

Returns a list of all of the windows that are children (inferiors) of *window*.

This is identical to **clim:sheet-children**, and is included only for compatibility with CLIM 1.1.

clim:window-clear *window* *Generic Function*

Clears the entire drawing plane of *window*, filling it with the background design. **clim:window-clear** also discards the window's output history and resets the cursor position to the upper left corner.

clim:window-erase-viewport *window* *Generic Function*

Clears the visible part of the drawing plane of *window*, filling it with the background design.

clim:window-event *Class*

The class that corresponds to any sort of window event.

clim:window-event-region *window-event* *Generic Function*

Returns the region that was affected by the window event *window-event*.

clim:window-expose *window* *Generic Function*

Makes the *window* visible on the screen.

This is identical to **(setf (clim:window-visibility** *window***) t)**, and is included only for compatibility with CLIM 1.1.

clim:window-inside-bottom *window* *Function*

Returns the coordinate of the bottom edge of the window *window*.

clim:window-inside-edges *window* *Generic Function*

Returns four values, the coordinates of the left, top, right, and bottom inside edges of the window *window*. The inside edges are computed by subtracted the window's margins from the window's outside edges. (The outside edges of a window can be obtained by calling **clim:bounding-rectangle*** on the window.)

clim:window-inside-height *window* *Function*

Returns the inside height of *window*.

clim:window-inside-left *window* *Function*

Returns the coordinate of the left edge of the window *window*.

clim:window-inside-right *window* *Function*

Returns the coordinate of the right edge of the window *window*.

clim:window-inside-size *window* *Generic Function*

Returns the inside width and height of *window* as two values.

clim:window-inside-top *window* *Function*

Returns the coordinate of the top edge of the window *window*.

clim:window-inside-width *window* *Function*

Returns the inside width of *window*.

clim:window-label *window* *Generic Function*

Returns the label (a string) associated with *window*, or **nil** if there is none. You can use **setf** of **clim:window-label** to give *window* a new label.

clim:window-parent *window* *Generic Function*

Returns the window that is the parent (superior) of *window*.

This is identical to **clim:sheet-parent**, and is included only for compatibility with CLIM 1.1.

clim:window-refresh *window* *Generic Function*

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.

clim:window-set-viewport-position *window x y* *Generic Function*

Moves the top-left corner of the window's viewport. This is how you scroll a window. This function is provided as a more convenient interface to **clim:scroll-extent** for the benefit of CLIM 1.1 programs.

clim:window-set-viewport-position, and all other functions that change the position of the viewport, also call **clim:note-viewport-position-changed** to notify the window that it has been scrolled.

clim:window-stack-on-bottom *window* *Generic Function*

Puts the *window* underneath all other windows that it overlaps.

clim:window-stack-on-top *window* *Generic Function*

Puts the *window* on top of all other windows that it overlaps, so you can see all of it.

clim:window-viewport *window* *Generic Function*

If the pane *window* is part of a scroller pane, this returns the region of *window*'s viewport. Otherwise it returns the region of *window* itself.

It is often more convenient to use this function instead of **clim:pane-viewport-region**.

clim:window-viewport-position *window* *Generic Function*

Returns two values, the X and Y coordinates of the top-left corner of the *window*'s viewport.

clim:window-visibility *stream* *Generic Function*

A predicate that returns true if the window is visible. You can use **setf** on **clim:window-visibility** to expose or deexpose the window.

Note that it is implementation-dependent whether this is true when the window is partially visible (or partially covered by an overlapping window).

clim:with-accept-help *options* &body *body* *Macro*

Binds the local environment to control `Help` and `control-?` documentation for input to **clim:accept**.

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 **clim:accept**.

:subhelp Specifies that *help-string* be used in addition to the default help documentation provided by **clim: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 **clim:with-accept-help** can override this. (That is, **:override** works from the outside in.)

:establish-unless-overridden
Specifies that the current help string be the help for this help type unless a higher-level call to **clim:with-accept-help** has already established a help string for this help type in the **:override** mode. This is what **clim: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 is evaluated.

See the section "Utilities for **clim:accept** Presentation Methods".

clim:with-activation-gestures *(additional-gestures* &key *:override)* &body *body*
Macro

Specifies gestures that terminate input during the evaluation of *body*. *additional-gestures* is a gesture spec or a form that evaluates to a list of gesture specs.

If *:override* is **t**, then then *additional-gestures* will override the existing activation gestures. If it is **nil** (the default), then *additional-gestures* will be added to the existing set of activation gestures.

See the **:activation-gestures** option to **clim:accept**. See also see the variable **clim:*standard-activation-gestures***.

clim:with-application-frame *(frame)* &body *body* *Macro*

Evaluates *body* with the variable *frame* bound to the current application frame.

clim:with-delimiter-gestures *(additional-gestures* &key *:override)* &body *body* *Macro*

Specifies gestures that terminate an individual token but not the entire input sentence during the evaluation of *body*. *additional-gestures* is a gesture spec or a form that evaluates to a list of gesture specs.

If *:override* is **t**, then then *additional-gestures* will override the existing delimiter gestures. If it is **nil** (the default), then *additional-gestures* will be added to the existing set of delimiter gestures.

See the **:delimiter-gestures** option to **clim:accept** .

clim:with-bounding-rectangle* *(left top right bottom) region* &body *body* *Macro*

Binds *left*, *top*, *right*, and *bottom* to the edges of the bounding rectangle of *region*, and then evaluates *body* in that context.

left	X coordinate of the left side of the region.
top	Y coordinate of the top side of the region.
right	X coordinate of the right side of the region.
bottom	Y coordinate of the bottom side of the region.
region	A bounded region or a sheet that has a bounded region. For example, a window, and output record, or a geometric object such as a line or an ellipse.

See the section "Bounding Rectangles in CLIM".

clim:with-command-table-keystrokes *(keystroke-var command-table)* &body *body* *Macro*

Binds *keystroke-var* to a list that contains all of the keystroke accelerators in the command table *command-table*, and then evaluates *body* in that context.

```
(clim:with-command-table-keystrokes (keystrokes command-table)
  (let ((command (clim:read-command-using-keystrokes
                   command-table keystrokes
                   :stream command-stream)))
    (if (and command (not (typep command 'clim:key-press-event)))
        (clim:execute-frame-command frame command)
        (clim:beep stream))))
```

In general, the keystroke accelerators you choose should not be any characters that a user can normally type in during an interaction. So for an application using a command-line interface style, they will typically be non-printing characters such as `#\control-E`.

This macro generates keystrokes suitable for use by **clim:read-command-using-keystrokes**.

clim:with-drawing-options *(medium* &key *:ink :clipping-region :transformation :line-style :line-unit :line-thickness :line-dashes :line-joint-shape :line-cap-shape :text-style :text-family :text-face :text-size)* *Macro*

Binds the state of *medium* to correspond to the supplied drawing *options*, and evaluates the *body* with the new drawing options in effect. Each option 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 output, or a stream.

Any call to a drawing function can supply a drawing option to override the prevailing one. In other words, the drawing functions effectively do a **clim:with-drawing-options** when drawing option arguments are supplied to them.

The default value specified for a drawing option is the value to which the corresponding component of a medium is normally initialized.

For information on the drawing options, see the section "Set of CLIM Drawing Options".

You can often realize performance improvements when doing graphical output by wrapping a single call to **clim:with-drawing-options** around calls to multiple drawing functions that use the same drawing options. See the macro **clim:with-medium-state-cached**.

clim:with-end-of-line-action *(stream action)* &body *body* *Macro*

Temporarily changes the end of line action for the duration of evaluation of *body*. The end of line action controls what happens if the 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 are usually the same thing.)

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 output.

:allow Ignore the text margin and just keep doing output.

clim:with-end-of-page-action *(stream action)* &body *body* *Macro*

Temporarily changes the end of page action for the duration of evaluation of *body*. The end of page action controls what happens if the cursor moves vertically out of the viewport.

The end of page action is one of:

: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 just keep doing output.

:wrap Wrap the text around (that is, go back to the top of the viewport). This is not currently implemented.

clim:with-first-quadrant-coordinates *(*&optional *stream x y)* &body *body* *Macro*

Binds the dynamic environment to establish a local coordinate system with the positive X-axis extending to the right and the positive Y-axis extending upward, with (0,0) at the current cursor position of *stream*.

clim:with-input-context *(type* &key *:override) (*&optional *object-var type-var event-var options-var) form* &body *clauses* *Macro*

Establishes an input context of type *type*. When *:override* is **nil** (the default), this invocation of **clim:with-input-context** adds its context presentation type to the extant context. In this way an application can solicit more than one type of input at the same time. When *:override* is **t**, it overrides the current input context rather than nesting inside the current input context.

After establishing the new input context, *form* is evaluated. If no pointer gestures are made by the end user during the evaluation of *form*, all of the values of *form* are returned. Otherwise, if the user invoked a translator by clicking on an object, one of the *clauses* is evaluated, based on the presentation type of the object returned by the translator. All of the values of that clause are are returned as the values of **clim:with-input-context**. During the evaluation of one of the *clauses*, *object-var* is bound to the object returned by the translator, *type-var* is bound to the presentation type returned by the translator, and *event-var* is bound to the event corresponding to the user's gesture. *options-var* is bound to any options that the translator might have returned, and will be either **nil** or a list of keyword-value pairs.

clauses is constructed like a **typecase** statement clause list whose keys are presentation types.

Note that, when one of the *clauses* is evaluated, nothing is inserted into the input buffer. If you want to insert input corresponding to the object the user clicked on, you must call **clim:replace-input** or **clim:presentation-replace-input**.

Only the arguments *type* and *:override* are evaluated.

```
(clim:with-input-context ('pathname)
                         (path)
    (read)
  (pathname
    (format t "~&The pathname ~A was clicked on." path)))
```

clim:with-input-editing *(*&optional *stream* &key *:input-sensitizer :initial-contents :class)* &body *body* *Macro*

Establishes a context in which the user can edit the input he or she types in on the stream *stream*. *body* is then evaluated in this context, and the values returned by *body* are returned as the values of **clim:with-input-editing**. *stream* defaults to ***standard-input***.

:class defaults to CLIM's standard input editing stream class. *:input-sensitizer*, if it is supplied, is a function of two arguments, a stream and a continuation. The *:input-sensitizer* function should call the continuation on the stream. For example, the implementation of **clim:accept** uses something like the following in order to make the user's input sensitive as a presentation for later use:

```
(flet ((input-sensitizer (continuation stream)
         (if (clim:stream-record-p stream)
             (clim:with-output-as-presentation (stream object type)
               (funcall continuation stream))
             (funcall continuation stream))))
  (clim:with-input-editing (stream :input-sensitizer #'input-sensitizer)
    ...))
```

:initial-contents is a string to use as the initial contents of the buffer for the stream to be edited.

See the section "Input Editing and Built-in Keystroke Commands in CLIM".

clim:with-input-editor-typeout *(*&optional *stream* &key *:erase)* &body *body* *Macro*

If, when you are inside of a call to **clim:with-input-editing**, you want to perform some sort of typeout, it should be done inside **clim:with-input-editor-typeout**. *stream* is the input editing stream and *body* is the code that will do output to the stream. *stream* defaults to ***standard-input***.

For more information on the input editor, see the section "The Structure of the CLIM Input Editor".

clim:with-input-focus *(stream)* &body *body* *Macro*

Temporarily gives the keyboard input focus to the given window (which is most often an interactor pane). By default, a frame will give the input focus to the **clim:frame-query-io** pane.

For example, suppose you want the user to supply some input in a ‘‘pop up’’, after which the window ‘‘pops down’’ again. The following function will do this.

```
(defun do-pop-up-text-editing (window)
  (setf (clim:window-visibility window) t)
  (unwind-protect
      (clim:with-input-focus (window)
        (clim:with-input-editing (window)
          (clim:with-activation-gestures ('(:end) :override t)
            (unwind-protect
                (clim:read-token window)
              ;; Eat the activation gesture if it's still there
              (clim:read-gesture :stream window :peek-p t :timeout 0)))))
    (setf (clim:window-visibility window) nil)))
```

clim:with-local-coordinates *(*&optional *stream x y)* &body *body* *Macro*

Binds the dynamic environment to establish a local coordinate system with the positive X-axis extending to the right and the positive Y-axis extending downward, with (0,0) at the current cursor position of *stream*.

clim-sys:with-lock-held *(place* &optional *state)* &body *forms* *Macro*

Evaluates *body* while holding the lock named by *place*. *place* is a reference to a lock created by **clim-sys:make-lock**.

On systems that do not support locking, **clim-sys:with-lock-held** is equivalent to **progn**.

See the section "Locks in CLIM".

clim:with-medium-state-cached *(medium)* &body *body* *Macro*

Declares that all of the drawing operations within *body* will use exactly the same drawing options. This allows CLIM back-ends to cache the state of the medium so that the medium does not need to be ''decoded'' for each drawing operation. Used in conjunction with **clim:with-drawing-options**, this can result in substantial performance improvements when doing graphical output, whether or not output recording is enabled or disabled.

clim:with-menu *(menu* &optional *associated-window* &rest *options* &key *:label :scroll-bars)* &body *body* *Macro*

Binds *menu* to a temporary window, exposes the window on the same screen as the *associated-window*, runs the body, and then deexposes the window. The values returned by **clim:with-menu** are the values returned by *body*.

menu The name of a variable which is bound to the window to be used for the menu.

associated-window
A window that this window is associated with, typically a pane of an application frame. If not supplied, *associated-window* will default to the top-level window of the current application frame.

:label A string to use to label the menu. The default is **nil**, meaning that there is no label.

:scroll-bars
Indicates whether the new window should have scroll bars. One of **nil**, **:none**, **:vertical**, **:horizontal**, or **:both**. The default is **:vertical**.

Example

This example shows how to use **clim:with-menu** with **clim:menu-choose-from-drawer** to draw a temporary menu.

```
(defun choose-compass-direction ()
  (labels ((draw-compass-point (stream ptype symbol x y)
             (clim:with-output-as-presentation (stream symbol 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)
      (clim:menu-choose-from-drawer
        menu 'clim:menu-item #'draw-compass))))
```

clim:with-menu can also be used to allocate a temporary window for other uses. You can use **clim:position-sheet-carefully** and **clim:size-frame-from-contents** to set the size and position of windows created using **clim:with-menu**.

clim:with-new-output-record *(stream* &optional *record-type record* &rest *initargs)* &body *body* *Macro*

Creates a new output record of type *record-type* (which defaults to CLIM's default ''linear'' output record) and then captures the output of *body* into 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).

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 initialization arguments that are passed to **clos:make-instance** when the new output record is created.

clim:with-new-output-record returns the output record it creates.

See the section "Concepts of CLIM Output Recording".

clim:with-output-as-gadget *(stream* &rest *options)* &body *body* *Macro*

Evaluates *body* to create a gadget, creates a gadget output record containing the gadget, and installs the output record into the output history of the *stream*. The returned value of *body* must be the gadget.

The options in *options* are passed as CLOS initargs to the call to **clim:invoke-with-new-output-record** that is used to create the gadget output record.

For example, the following could be used to create an output record containing a radio box that itself contains several toggle buttons:

```
(clim:with-output-as-gadget (stream)
  (let* ((radio-box
           (clim:make-pane 'clim:radio-box
             :client stream :id 'radio-box)))
    (dolist (item sequence)
      (clim:make-pane 'clim:toggle-button
        :label (princ-to-string (item-name item))
        :value (item-value item)
        :id item :parent radio-box))
    radio-box))
```

A more complex (and somewhat contrived) example of a push button that calls back into the presentation type system to execute a command might be as follows:

```
(clim:with-output-as-gadget (stream)
  (clim:make-pane 'clim:push-button
    :label "Click here to exit"
    :activate-callback
      #'(lambda (button)
          (declare (ignore button))
          (clim:throw-highlighted-presentation
            (make-instance 'clim:standard-presentation
              :object `(com-exit ,clim:*application-frame*)
              :type 'command)
            clim:*input-context*
            (make-instance 'clim:pointer-button-press-event
              :sheet (clim:sheet-parent button)
              :x 0 :y 0
              :modifiers 0
              :button clim:+pointer-left-button+)))))
```

clim:with-output-as-presentation *(stream object type* &key *:modifier :single-box (:allow-sensitive-inferiors* **t***) :parent :record-type)* &body *body* *Macro*

Gives separate access to the two aspects of **clim:present**: recording the presentation and drawing the visual representation. This macro generates a presentation from the output done in the *body* to the *stream*. The presentation's underlying object is *object*, and its presentation type is *type*. For information on the syntax of specifying a presentation type, see the section "How to Specify a CLIM Presentation Type".

All arguments of this macro are evaluated.

clim:with-output-as-presentation returns a presentation.

Note that CLIM captures the presentation type for its own use, so you should not modify it once you have handed it to CLIM.

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.

For background information, see the section "Presentation Types in CLIM".

stream

The stream to which output should be sent. The default is ***standard-output***.

:modifier

Specifies a function of one argument (the new value) that can be called in order to store a new value for *object* after the user edits the presentation. The default is **nil**.

:single-box

Controls how CLIM determines whether the pointer is pointing at this presentation and controls how this presentation is highlighted when it is sensitive.

The possible values are:

t
: If the pointer's position is inside the bounding rectangle of this presentation, it is considered to be pointing at this presentation. This presentation is highlighted by highlighting its bounding rectangle.

nil
: If the pointer is pointing at a visible piece of output (text or graphics) drawn as part of the visual representation of this presentation, it is considered to be pointing at this presentation. This presentation is highlighted by highlighting every visible piece of output that is drawn as part of its visual representation. This is the default.

:position
: Like **t** for determining whether the pointer is pointing at this presentation, like **nil** for highlighting.

:highlighting
: Like **nil** for determining whether the pointer is pointing at this presentation, like **t** for highlighting.

Supplying **:single-box :highlighting** is useful when the default behavior produces an ugly appearance (for example, a very jagged highlighting box).

Supplying **:single-box :position** is useful when the visual representation of a presentation consists of one or more small graphical objects with a lot of space between them. In this case the default behavior offers only small targets that the user might find difficult to position the pointer over.

:allow-sensitive-inferiors

When *:allow-sensitive-inferiors* is **nil**, it indicates that nested calls to **clim:present** or **clim:with-output-as-presentation** inside this one should not generate presentations. The default is **t**.

:record-type

This option is useful when you have defined a customized record type to replace CLIM's default record type. It specifies the class of the output record to be created.

clim:with-output-recording-options *(stream* &key *:draw :record)* &body *body Macro*

Used to disable output recording and/or drawing on the given *stream*, within the extent of *body*.

If *:draw* is **nil**, output to the stream is not drawn on the viewport, but can still be recorded in the output history. If *:record* is **nil**, output recording is disabled but output otherwise proceeds normally.

clim:with-output-to-output-record *(stream* &optional *record-type record* &rest *init-targs)* &body *body Macro*

This is similar to **clim:with-new-output-record** except that the new output record is not inserted into the output record hierarchy. That is, when you use **clim:with-output-to-output-record** no drawing on the stream occurs and nothing is put into the stream's normal output history. Unlike in facilities such as **with-output-to-string**, *stream* must be an actual stream, but no output will be done to it.

record-type is the type of output record to create, which defaults to CLIM's default output record type. *init-args* are CLOS init keywords which are used to initialize the record.

If *record* is supplied, it is a variable which will be bound to the new output record while *body* is evaluated.

See the section "Concepts of CLIM Output Recording".

clim:with-output-to-pixmap *(medium-var medium* &key *:width :height)* &body *body Macro*

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.

In Genera, you may call stream output functions (such as **write-char** and **write-string**) from within *body*, but not all CLIM implementations will necessarily support this.

:width and *:height* are integers that give the width and height of the pixmap. If they are unsupplied, the result pixmap will be just large enough to contain all of 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 **clim:copy-from-pixmap**.

clim:with-output-to-postscript-stream *(stream-var file-stream* &key *:device-type (:orientation* **:portrait***) :multi-page :scale-to-fit :header-comments (:destination* **:printer***))* &body *body* *Macro*

Within *body*, *stream-var* is bound to a stream that produces PostScript code. This stream is suitable as a stream or medium argument to any CLIM output utility. A PostScript program describing the output to the *stream-var* stream will be written to *file-stream*.

:device-type — The class of PostScript display device to use. The only device class currently provided is is suitable for generating PostScript for the printers such as the Apple LaserWriter.

:orientation — Specifies the orientation (portrait or landscape) of the output. It can be either **:portrait** or **:landscape**. The default is **:portrait**.

:multi-page — When supplied as **t**, any output that is larger than the size of a page will automatically be broken up into several pages. (No hints are given as to how the resulting pages should be pieced together; you're on your own here.) This defaults to **nil**.

:scale-to-fit — When supplied as **t**, this causes the output to be scaled so that it fits on a single page. This defaults to **nil**. It does not make sense to supply both *:multi-page* and *:scale-to-fit*.

:header-comments — This allows you to specify some PostScript header comment fields for the resulting PostScript program. The argument should be a list of alternating keyword and value pairs. These are the supported keywords:

:title — Specifies a title for the document, as it will appear in the `"%%Title:"` header comment.

:for — Specifies who the document is for. The associated value will appear in a `"%%For:"` document comment.

:destination One of either **:printer** or **:document**. When it is **:printer**, the output file will include a PostScript ''showpage'' command that causes the output to be sent to the printer. When it is **:document**, the ''showpage'' command will not be included in the output file, making it suitable for inclusion in other documents.

See the section "Hardcopy Streams in CLIM".

Note: The PostScript programs written by this implementation do not conform to the conventions described under "Appendix C: Structuring Conventions" of the *PostScript Language Reference Manual*. Software tools which attempt to determine information about these PostScript programs based on `"%%"` comments within them will be unsuccessful.

clim:with-presentation-type-decoded *(name-var* &optional *parameters-var options-var) type* &body *body* *Macro*

The specified variables are bound to the components of the presentation type specifier, the forms in *body* are evaluated, and the values of the last form are returned. The value of the *type* must be a presentation type specifier. *name-var*, if non-**nil**, is bound to the presentation type name. *parameters-var*, if present and non-**nil**, is bound to a list of the parameters. *options-var*, if present and non-**nil**, is bound to a list of the options.

clim:with-presentation-type-options *(type-name type)* &body *body* *Macro*

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 evaluated 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*. t*ype-name* is not evaluated.

clim:with-presentation-type-parameters *(type-name type)* &body *body* *Macro*

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 value of the form *type* must be a presentation type specifier whose name is *type-name*. t*ype-name* is not evaluated. The forms in *body* are evaluated in the scope of these variables and the values of the last form are returned.

clim:with-radio-box *(*&rest *options* &key *(:type* **':one-of***)* &allow-other-keys*)* &body *body* *Macro*

Creates a radio box or a check box whose buttons are created by the forms in *body*. The macro **clim:radio-box-current-selection** (or **clim:check-box-current-selection**) can be wrapped around one of forms in *body* in order to indicate that that button is the current selection. If *:type* is **:one-of**, this macro creates a radio box. If *:type* is **:some-of**, it creates a check box.

See the section "Using Gadgets in CLIM".

For example, the following creates a radio box with three buttons in it, the second of which is initially selected.

```
(clim:with-radio-box ()
  (clim:make-pane 'clim:toggle-button :label "Mono")
  (clim:radio-box-current-selection
    (clim:make-pane 'clim:toggle-button :label "Stereo"))
  (clim:make-pane 'clim:toggle-button :label "Quad"))
```

The following simpler form can also be used when you do not need to control the appearance of each button closely:

```
(clim:with-radio-box ()
  "Mono" "Stereo" "Quad")
```

clim-sys:with-recursive-lock-held *(place* &optional *state)* &body *forms* *Macro*

Evaluates *body* while holding the recursive lock named by *place*. *place* is a reference to a recursive lock created by **clim-sys:make-recursive-lock**.

On systems that do not support locking, **clim-sys:with-recursive-lock-held** is equivalent to **progn**.

See the section "Locks in CLIM".

clim:with-room-for-graphics *(*&optional *stream* &key *:height (:first-quadrant* **t***) (:move-cursor* **t***) :record-type)* &body *body* *Macro*

Binds the dynamic environment to establish a local coordinate system for doing graphics output onto *stream*. *body* is evaluated to produce the output.

If *:first-quadrant* is **t** (the default), a local Cartesian coordinate system is established with the origin (0,0) of the local coordinate system placed at the current cursor position; (0,0) is in the lower left corner of the area created.

If *: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).

If *:height* is supplied, it should be a 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 **clim:standard-sequence-output-record**.

clim:with-rotation *(medium angle* &optional *origin)* &body *body* *Macro*

Establishes a rotation on *medium* that rotates clockwise by *angle* (in radians), and then evaluates *body* with that transformation in effect. If *origin* is supplied, the rotation is about that point. The default for *origin* is (0,0).

This is equivalent to using **clim:with-drawing-options** with the **:transformation** keyword argument supplied:

```
(clim:with-drawing-options
   (medium :transformation
           (clim:make-rotation-transformation angle origin))
 body)
```

clim:with-scaling *(medium sx* &optional *sy)* &body *body* *Macro*

Establishes a scaling transformation on *medium* that scales by *sx* in the X direction and *sy* in the Y direction, and then evaluates *body* with that transformation in effect. If *sy* is not supplied, it defaults to *sx*.

This is equivalent to using **clim:with-drawing-options** with the **:transformation** keyword argument supplied:

```
(clim:with-drawing-options
   (medium :transformation
           (clim:make-scaling-transformation sx sy))
 body)
```

clim:with-sheet-medium *(medium sheet)* &body *body* *Macro*

Within the body, the variable *medium* is bound to *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 as the when the body has been exited. The values of the last form of the body are returned as the values of **clim:sheet-medium**.

The *medium* argument is not evaluated, and must be a symbol that is bound to a medium.

See the section "Sheet Output Protocols".

clim:with-text-face *(medium face)* &body *body* *Macro*

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**. The default for *medium* is ***standard-output***.

This is the same as:

```
(clim:with-drawing-options (medium :text-face face)
  body)
```

Using **clim:with-text-family** affects **clim:medium-text-style**.

See the section "CLIM Text Style Objects".

clim:with-text-family *(medium family)* &body *body* *Macro*

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**. The default for *medium* is ***standard-output***.

This is the same as:

```
(clim:with-drawing-options (medium :text-family family)
  body)
```

Using **clim:with-text-family** affects **clim:medium-text-style**.

See the section "CLIM Text Style Objects".

clim:with-text-size *(medium size)* &body *body* *Macro*

Binds the current text size of *medium* to correspond to the new size text *size*, within the *body*. *size* is one of the logical sizes (**:normal**, **:small**, **:large**, **:very-small**, **:very-large**, **:smaller**, **:larger**), or a real number representing the size in printer's points, or **nil**. The default for *medium* is ***standard-output***.

This is the same as:

```
(clim:with-drawing-options (medium :text-size size)
  body)
```

Using **clim:with-text-size** affects **clim:medium-text-style**.

See the section "CLIM Text Style Objects".

clim:with-text-style *(medium style)* &body *body* *Macro*

Binds the current text style of *medium* to correspond to the new text *style*, within the *body*. *style* is a text style object. The default for *medium* is ***standard-output***.

This is the same as:

```
(clim:with-drawing-options (medium :text-style style)
  body)
```

Using **clim:with-text-style** affects **clim:medium-text-style**.

See the section "CLIM Text Style Objects".

clim:with-translation *(medium dx dy)* &body *body* *Macro*

Establishes a scaling transformation on *medium* that scales by *dx* in the X direction and *dy* in the Y direction, and then evaluates *body* with that transformation in effect.

This is equivalent to using **clim:with-drawing-options** with the **:transformation** keyword argument supplied:

```
(clim:with-drawing-options
   (medium :transformation
           (clim:make-translation-transformation dx dy))
 body)
```

clim-sys:without-scheduling &body *forms* *Macro*

Evaluates *body* in a context that is guaranteed to be free from interruption by other processes. This returns the value of the last form in *body*.

On systems that do not support multi-processing, **clim-sys:without-scheduling** is equivalent to **progn**.

clim:write-token *token stream* &key *:acceptably* *Function*

clim:write-token is the opposite of **clim:read-token**: given the string *token*, it writes it to the stream *stream*. If *:acceptably* is **t** and there are any characters in *token* that are delimiter gestures (see the macro **clim:with-delimiter-gestures**), then **clim:write-token** will surround the token with quotation marks, `#\"`.

It is appropriate to use **clim:write-token** instead of **write-string** inside of **clim:present** methods.

CLIM Glossary

Glossary of CLIM Terminology

adopted *(of a sheet)* A sheet is said to be *adopted* when it has a parent sheet.

affine transformation
See *transformation*.

ancestors The parent of a *sheet* or an *output record*, and all of the ancestors of the parent, recursively.

applicable *(of a presentation translator)* 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 In general, a program that interacts directly with a *user* to perform some specific task. In CLIM, a frame is the central mechanism for presenting an application's user interface. Also, a Lisp object that holds the information associated with such a program, including the panes of the user interface and application state variables.

background The *design* that is used when erasing, that is, drawing on a *medium* using **clim:+background-ink+**.

bounding rectangle The smallest *rectangle* that surrounds a bounded *region* and contains every point in the region; it may contain additional points as well. The sides of a bounding rectangle are parallel to the coordinate axes. Also, a Lisp object that represents a *bounding rectangle*.

coordinates A pair of real numbers that identify a point in the *drawing plane*.

cache value During *incremental redisplay*, CLIM uses the *cache value* to determine whether or not a piece of output has changed.

children *(of a sheet or output record)* The direct descendants of a *sheet* or an *output record*.

color An object representing the intuitive definition of a color, such as black or red. Also, a Lisp object that represents a *color*. See also *ink*.

command An object that represents a user interaction. Each command has a name, which is a symbol. A command can also have arguments, both positional and keyword arguments. A user may enter a command in several different ways, by typing, by using a menu, or by directly clicking the mouse on some output. Also, a Lisp object that represents a *command*; a cons of the command name and the list of the command's arguments.

command-defining macro A Lisp macro for defining the commands specific to a particular application frame, defined by CLIM when the application frame is defined.

command-line name The name that the end user sees and uses during command line interactions. This is not the same thing as the *command name*. For example, the command **com-show-chart** might have a command-line name of `"Show Chart"`.

command name A symbol that names a CLIM *command*.

command parser The part of CLIM's command loop that performs the conversion of strings of characters (usually what the user typed) into a command.

command table A way of collecting and organizing a group of related commands, and defining the interaction styles that can be used to invoke those commands. Also, a Lisp object that represents a *command table*.

completion A facility provided by CLIM for completing user input from a set of possibilities.

compositing *(of designs)* The creation of a new *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 *(of transformations)* 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 sensitive input The facility in CLIM that allows an interface to describe what sort of input it expects by specifying the input's type. That type is called a *presentation type*.

cursor The place in the *drawing plane* of a stream where the next piece of text output will appear.

degrafted *(of a sheet)* Not *grafted*, that is, not attached to any *display server*.

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. A *design* is the generalization of a *region* into the color domain.

direct manipulation A style of interaction with an application where the user indicates the desired task by performing an analogous action in the interface. For instance, deleting a file by dragging an icon representing the file over another icon that looks like a trash can, or connecting two components in a circuit simulation by drawing a line representing a connection between two icons representing the components.

disabled *(of a sheet)* Not *enabled*.

disowned *(of a sheet)* Not *adopted*, that is, not having any parent sheet.

display function The function, associated with a particular *pane* in an *application frame*, that performs the "appropriate" output for that pane.

display server A device on which output can appear, such as an X console or a PostScript printer.

displayed output record An *output record* that corresponds to a visible piece of output, such as text or graphics. A leaf in an output record tree.

drawing plane An imaginary two-dimensional plane, on which graphical output occurs, that extends infinitely in all directions and has infinite resolution. A drawing plane contains an arrangement of colors and opacities that is modified by each graphical output operation. The drawing plane provides an idealized version of the graphics you draw. CLIM transfers the graphics from the drawing plane to a host window by a process called *rendering*.

enabled *(of a sheet)* A sheet is said to be *enabled* when it is actively participating in the windowing relationship with its parent.

event Some sort of significant event, such as a user gesture (such as moving the pointer, pressing a pointer button, or typing a keystroke) or a window configuration event (such as resizing a window). Also, a Lisp object that represents an *event*.

flipping ink An *ink* that interchanges occurrences of two *designs*, such as might be done by "XOR" on a monochrome display. Also, a Lisp object that represents a *flipping ink*.

foreground The *design* that is used when drawing on a *medium* using **clim:+foreground-ink+**.

formatted output Output that obeys some high level constraints on its appearance, such as being arranged in a tabular format, or justified within some margins. The CLIM facility that provides a programmer the tools to produce such output.

frame See *application frame*.

frame manager An object associated with a *port* that controls the realization of the look and feel of an *application frame*. It is responsible for creating panes and gadgets, laying out menus and dialogs, and doing other tasks related to look and feel.

fully specified *(of a text style)* Having components none of which are **nil**, and not having a relative size (that is, neither **:smaller** nor **:larger**).

gesture Some sort of input action by a user, such as typing a character or clicking a pointer button. User gestures are frequently represented by event objects.

gesture name A symbol that gives a name to a class of *gestures*, for example, **:select** is commonly used to indicate a left pointer button click.

gesture spec A platform-independent way of specifying some sort of input gesture, such as a pointer button press or a key press. For example, the `control-D` ''character'' is specified by the gesture spec `(:D :control)`.

graft A kind of *mirrored sheet* that represents a host window, typically a ''root'' window.

grafted *(of a sheet)* Having an ancestor sheet that is a *graft*. A grafted sheet will be visible on a *display server*, unless it is clipped or occluded by some other window on the same display.

highlighted A visual indication that the presentation under the pointer is *sensitive*, and can thus be entered as input to the program currently accepting input. Highlighting might appear as a box drawn around the presentation, or as a different appearance of the presentation, such as a bold text style.

incremental redisplay
The redrawing of part of some output while leaving other output unchanged. The CLIM facility that implements this behavior.

indirect ink An ink whose exact behavior depends on the context in which it is used. Drawing with an *indirect ink* is the same as drawing with another *ink* named directly. **clim:+foreground-ink+** and **clim:+background-ink+** are both indirect inks.

ink Any member of the class *design* supplied as the **:ink** argument to a CLIM drawing function.

input context A state in which a program is expecting input of a certain type. A command that takes arguments as input specifies the presentation type of each argument. When the command is given, an input context is set up in which presentations that are appropriate are sensitive (they are highlighted when the pointer passes over them). Also, a Lisp object that represents an *input context*.

input editor The CLIM facility that allows a *user* to modify typed-in input.

input editing stream
A CLIM stream that supports *input editing*.

interactive stream A stream that supports both input from and output to the user in an interactive fashion.

keyboard accelerator
A single key or key chord (that is, set of keys pressed together) used to issue an entire command.

line style Advice to CLIM's rendering substrate on how to draw a path, such as a line or an unfilled ellipse or polygon. Also, a Lisp object that represents a *line style*.

medium A destination for output, having a *drawing surface*, two designs called the medium's *foreground* and *background*, a *transformation*, a *clipping region*, a *line style*, and a *text style*. Also, a Lisp object that represents a *medium*.

mirror The host window system object associated with a *mirrored sheet*, such as a window on Genera or an X11 *display server*.

mirrored sheet A special class of *sheet* that is attached directly to a window on a *display server*. A *graft* is one kind of a *mirrored sheet*. On some platforms, many of the gadgets (such as scroll bars and push buttons) will be mirrored sheets as well.

mixed mode interface
An interface to an application that allows more than one style or mode of interaction. For example a user might be able to enter the same command by typing it, by clicking on a menu button, or by dragging an icon.

opacity A way of controlling how new graphical output covers previous output, such as fully opaque to fully transparent, and levels of translucency between. Also, a Lisp object that represents an *opacity*.

output history The highest level *output record* for an *output recording stream*.

output record An object that remembers the output performed to an *output recording stream*. Also, a Lisp object that represents an *output record*.

output recording The process of remembering the output performed to a stream. Output recording is the basis for other CLIM facilities, such as formatted output, incremental redisplay, and context-sensitive input.

output recording stream
A CLIM stream that remembers and can replay its output.

pane A *sheet* or window that appears as the child of some other window or *frame*. A composite pane can hold other panes; a leaf pane cannot. An application frame is normally composed of a hierarchy of panes.

parameterized presentation type
A *presentation type* whose semantics are modified by parameters. A parameterized presentation type is always a subtype of the presentation type without parameters. For example, **(integer 0 10)** is a parameterized type indicating an integer in the range of zero to ten. Parameterized presentation types are analogous to Common Lisp types that have parameters.

parent The direct ancestor of a *sheet* or an *output record*.

patterning The process of creating a bounded rectangular arrangement of *designs*, like a checkerboard. A *pattern* is a *design* created by this process.

pixmap An ''off-screen window'', that is, an object that can be used for graphical output, but is not visible on any display device.

point A *region* that has dimensionality 0, that is, has only a position. Also, a Lisp object that represents a *point*.

pointer An input device that enables pointing to an area of the screen, such as a mouse, tablet, or other pointing device.

pointer documentation
Short documentation associated with a presentation that describes what will happen when any button on the pointer is pressed.

port A 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 A location on a plane, such as CLIM's abstract drawing plane. A pair of real number values *(x,y)* that represent a *position*.

presentation An association between an object and a *presentation type* with some output on an *output recording stream*. Also, a Lisp object that represents a presentation, consisting of at least three things: the underlying application object, its presentation type, and its displayed appearance.

presentation method
A method that supports some part of the behavior of a presentation type, such as the **clim:accept** method (which parses keyboard input into an object of this type) and the **clim:present** method (which displays the object as a presentation which will be sensitive in matching contexts).

presentation tester A predicate that restricts the applicability of a *presentation translator*. It is used only to prevent a translation that would otherwise happen.

presentation-to-command translator
A particular type of *presentation translator* where the ''to type'' (that is, the type of presentation resulting from the translation) is a command.

presentation translator
A mapping from an object of one *presentation type*, an *input context*, and a *gesture* to an object of another presentation type. In effect, a translator broadens an input context so that some presentations are sensitive when the program seeking input is

expecting a different type. A presentation translator enables the user to enter a presentation of a related type which can be translated into input of the expected type.

presentation type A description of a class of *presentations*. The semantic type of an object to be displayed to the user.

presentation type specifier
The syntax for specifying a presentation type to CLIM functions such as **clim:accept**, **clim:present**, and others. There are three patterns for specifying presentation types, the first being simply its name, and the other two enabling you to specify its parameters and options.

programmer A person who writes application programs using CLIM.

rectangle A four-sided polygon whose sides are parallel to the coordinate axes. Also, a Lisp object that represents a *rectangle*.

redisplay See *incremental redisplay*.

region A set of mathematical points in the plane; a mapping from an *(x,y)* pair into either true or false (meaning member or not a member, respectively, of the region). In CLIM, all regions include their boundaries (that is, they are closed) and have infinite resolution. Also, a Lisp object that represents a *region*.

rendering The process of drawing a shape (such as a line or a circle) on a display device. Rendering is an approximate process, since an abstract shape exists in a continuous coordinate system having infinite precision, whereas display devices must necessarily draw discrete points having some measurable size.

repainting The act of redrawing all of the sheets or output records in a ''damage region'', such as occurs when a window is raised from underneath occluding windows.

replaying The process of redrawing a set of *output records*.

sensitive *(of a presentation)* Relevant to the current input context. A *presentation* is *sensitive* if some action will take place when the user clicks on it with the pointer, that is, there is at least one *presentation translator* that is *applicable*. In this case, the presentation will usually be *highlighted*.

sheet The basic unit of windowing in CLIM. A sheet's attributes always include a region and a mapping to the coordinate system of its parent, and may include other attributes, such as a medium and event handling. Also, a Lisp object that represents a *sheet*.

sheet region The *region* that a *sheet* occupies.

sheet transformation
A *transformation* that maps the coordinates of a *sheet* to the coordinate system of its parent, if it has a parent.

stream A kind of *sheet* that implements the stream protocol (such as doing textual input and output, and maintaining a *text cursor*).

text face The component of a *text style* that specifies a variety or modification of a *text family*, such as bold or italic.

text family The highest level component of a *text style* that specifies the common appearance of all the characters. Characters of the same family have a typographic integrity so that all characters of the same family resemble one another, such as **:sans-serif**.

text size The component of a *text style* that specifies the size of text.

text style A description of how textual output should appear, consisting of family, face code, and size. Also, 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* created by this process.

top level sheet The single, topmost sheet associated with an entire application frame. All the panes of the frame are descendants of this sheet.

transformation A mapping from one coordinate system onto another that preserves straight lines. General transformations include all the sorts of transformations that CLIM uses, namely, translations, scaling, rotations, and reflections. Also, a Lisp object that represents a *transformation*.

unique id During *incremental redisplay*, the *unique id* is an object used to uniquely identify a piece of output. The output named by the *unique id* will often have a *cache value* associated with it. When incremental redisplay finds a unique id that it has seen before and its cache value has changed since the last time redisplay was done, then CLIM will redraw that piece of output.

user A person who uses an application program that was written using CLIM.

view A way of displaying a presentation. Views can serve to mediate between presentations and gadgets. Also, a Lisp object that represents a *view*.

viewport The region of the *drawing plane* that is visible on a window.