7A Programming the User Interface

symbolics

7A Programming the User Interface

symbolics™

Cambridge, Massachusetts

Programming the User Interface, Volume A

999025

September 1986

This document corresponds to Genera 7.0 and later releases.

Symbolics, Inc.
4 New England Tech Center
555 Virginia Road
Concord, MA 01742

Text written and produced on Symbolics 3600-family computers by the Documentation Group of Symbolics, Inc.
Text masters produced on Symbolics 3600-family computers and printed on Symbolics LGP2 Laser Graphics Printers.
Cover design: Schafer|LaCasse
Printer: CSA Press
Printed in the United States of America.

Printing year and number: 88 87 86 9 8 7 6 5 4 3 2 1

Table of Contents

PART I.

User Interface Management System: an Overview

1. Guide to User Interface Documentation

1.1 New and Old Facilities

Genera 7.0 user interface facilities represent a major departure from tools offered for user interface programming in previous releases. Although the new facilities render many of the old ones obsolete, Genera still supports most of the old tools for the sake of compatibility with earlier releases. (For information on unsupported tools and incompatible changes: See the document *Converting to Genera 7.0.*) Consequently, the user interface documentation (Book 7) is divided into two major areas.

The first area, *Programming the User Interface, Volume A*, focuses on the new facilities for user interface programming. Thus, Part I of this book, the Overview, is an overview of the new facilities and does not generally refer to the old tools; Parts II through VIII, the Dictionaries, include, with one or two exceptions, only the new definitions.

The second area, *Programming the User Interface, Volume B*, corresponds to the pre-Genera 7.0 Book 7. This material is similar to the earlier book. The only changes are a series of minor corrections and improvements. Exceptions to this are changes to reflect the use of character styles instead of fonts, the implementation of mouse characters as structures, and a considerably expanded section on text scroll windows. (For more details: See the section "Changes to User Interface Programming Facilities in Genera 7.0" in *Genera 7.0 Release Notes.*

Much of the basic conceptual documentation on the window system in the old Book 7 (*Programming the User Interface, Volume B*) remains relevant, as does the reference documentation for most window init options and methods. We wish to emphasize, however, that many of the higher-level facilities in the old user interface management system – various menu facilities, the mouse-sensitive items facility, scroll windows, and text scroll windows – are maintained primarily for compatibility with pre-Genera 7.0 user programs.

The new system includes tools that are generally more powerful and easier to use than their old counterparts; in many cases, there are no counterparts in the old system. We encourage you, therefore, to concentrate your user interface programming efforts on facilities available in the new system. This will improve your productivity and better ensure the compatibility of your programs with future releases.

1.2 Levels of Detail

Just as the user interface facilities themselves are arranged in a functional hierarchy – from the high-level and general-purpose to the low-level and special-purpose – so too is the documentation hierarchical, from the general to the detailed.

At the highest level of abstraction is the introductory chapter to the overview, which outlines the major categories of user interface programming tools and describes the organizational hierarchy. See the section "Introduction to the User Interface Management System", page 7.

The subsequent chapters in the overview provide the next level of documentation detail. They discuss the major and minor groups of facilities, present tables listing the facilities included in each group, and include a variety of examples:

- See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.
- See the section "Overview of Command Processor Facilities", page 31.
- See the section "Overview of User Input Facilities", page 35.
- See the section "Overview of Program Output Facilities", page 47.
- See the section "Overview of Presentation Substrate Facilities", page 69.
- See the section "Overview of Window Substrate Facilities", page 87.

The overview concludes with a chapter outlining a sample application that illustrates the use of some of the key user interface programming facilities discussed in the preceding chapters: See the section "User Interface Application Example", page 91.

Parts II through VIII provide the greatest amount of detail. These are the dictionaries, containing reference documentation for each of the many Lisp objects discussed in the conceptual chapters. Each object is an entry in a dictionary of related objects. There are eight dictionaries, corresponding to the major functional categories among which the objects (facilities) are divided:

- See the section "Dictionary of Top-level Facilities for User Interface Programming", page 99.
- See the section "Dictionary of Command Processor Facilities", page 135.
- See the section "Dictionary User Input Facilities".

- See the section "Dictionary Program Output Facilities".
- See the section "Dictionary of Predefined Presentation Types", page 281.
- See the section "Dictionary of Presentation Substrate Facilities", page 347.
- See the section "Dictionary of Window Substrate Facilities", page 395.

Within each dictionary, the arrangement of facilities is alphabetical (package prefixes are ignored).

2. Introduction to the User Interface Management System

Genera's user interface management system provides a wide variety of tools for constructing user interfaces to application programs. This toolkit includes both high-level facilities for rapidly building a user interface framework and low-level facilities for changing the subtler details of user interface appearance and behavior. A number of additional tools lie in between; they provide varying degrees of power and flexibility in the several areas of user interface programming.

The system is largely hierarchical, with each layer of facilities built on the one below until, at the lowest level, the enabling substrate is reached. The position of any given tool in the hierarchy generally reflects its power and ease of use: the more powerful, easy-to-use tools are at the top; those providing less power but more flexibility, and also demanding a more advanced understanding of user interface programming, are further down.

The following outline shows the major categories and subcategories of facilities contributing to the user interface management system:

OUTLINE OF USER INTERFACE PROGRAMMING FACILITIES

Top-Level Facilities for User Interface Programming

- Frame-Up Layout Designer
- Program Framework Definition Facilities
- Program Command Facilities

Command Processor Facilities

- Basic Command Facilities
  - Command Definition Facilities
  - Command Processor Interface Facilities
- Advanced Command Facilities
  - Command Loop Management Facilities

Command Table Management Facilities
Command Accelerator Facilities

User Input Facilities

Basic User Input Facilities

Facilities for Accepting Single Objects
Facilities for Accepting Multiple Objects

Advanced User Input Facilities

Mouse Handler Facilities
Mouse Gesture Interface Facilities

Program Output Facilities

Basic Program Output Facilities

Presentation Output Facilities
Character Environment Facilities
List Formatting Facilities
Table Formatting Facilities
Graph Formatting Facilities
Graphic Output Facilities
Progress Indicator Facilities
Other Program Output Facilities

Advanced Program Output Facilities

Advanced Presentation Output Facilities
Redisplay Facilities
Facilities for Writing Formatted Output Macros

Presentation Substrate Facilities

Predefined Presentation Types

Presentation-Type Definition Facilities

Presentation Input Context Facilities

Presentation Input Blip Facilities

Other Presentation Facilities

Window Substrate Facilities

Dynamic Window Facilities

Dynamic Frame Facilities

This outline does not reflect two additional sources of facilities for building user interfaces. The first includes a variety of standard and special functions for program I/O documented elsewhere: See the section "Streams" in *Reference Guide to Streams, Files, and I/O.*

The second includes facilities for user interface programming provided by Symbolics prior to Genera 7.0. For more information on new versus old facilities and their respective documentation: See the section "Guide to User Interface Documentation", page 3.

What follows is a series of introductory sections to the major groups of user interface facilities. Because the presentation and Dynamic Window substrates are basic to understanding facilities occupying higher levels in the hierarchy, we start with them.

- **Presentation Substrate**

  The *presentation* system is central to most of the facilities provided for building user interfaces. This system provides a mechanism for specifying types, referred to as *presentation types*, for doing program I/O. The presentation-type mechanism is an extension of the Common Lisp type system and centralizes responsibility for parsing and printing data.

  In the presentation system, the printed (displayed) representation of a data object is distinct from its stored representation; that is, its appearance on the screen is specified independently of its internal structure. Consider, for example, the **integer** presentation type. It has a **:base** *presentation argument.* If it is appropriate for integer I/O to be in terms of binary integers, say, or octal integers, then specifying the appropriate base produces the desired result.

  The following two examples illustrate this point. Both use the Symbolics

Common Lisp function **accept** to read and inter an integer object. In the examples, the range of the integer sought is restricted to one between 0 and 100. In the first example, no base is specified, so the default base of 10 is used; a 10 is entered and returned. In the second, we specify an octal integer with the same range (note that **accept** adjusts the prompt accordingly). Again a 10 is entered and returned but, because of the base specification, the printed representation is in octal, that is, 12.

```
(accept '((integer 0 100))) ==>
Enter an integer greater than or equal to 0
and less than or equal to 100: 10
10
((INTEGER 0 100))

(accept '((integer 0 100) :base 8)) ==>
Enter an octal integer greater than or equal to 0
and less than or equal to 144: 12
10
((INTEGER) :BASE 8)
```

The above is a simple example. The degree of control you have over the printed representation of data types goes considerably beyond merely specifying predefined presentation arguments. You can write your own printer function controlling the user-visible appearance of any object. For example, you could make integers appear as graphic presentations. Such control over the printed representation of Lisp objects allows programs to *present* output and *accept* input in forms most meaningful to the application at hand.

The presentation system predefines a large number of presentation types, including all Common Lisp types. These predefined presentation types are included for documentation purposes with the presentation substrate facilities. This might suggest that they are low-level and specialized, which they are in some respects, but they are also used throughout the user interface programming hierarchy. Most program output and user input is in terms of presentation types.

Other presentation substrate facilities provide functions for creating new presentation types, including parsers, *help* facilities, and *completion* facilities. Through these tools you can extend and customize the presentation system to suit your application needs.

Together with Dynamic Windows and the window substrate, the presentation

substrate forms the basis of SemantiCue, Genera's smart input system. What makes this system "smart" is discussed in the following section on the window substrate.

- **Window Substrate**

The *window* system is the second major source of user interface substrate facilities. A window can be *static* or *dynamic*. Output to static windows is relative to an unchanging set of window coordinates; once a static window is full, it must be cleared entirely or partially before new output can be done without overwriting previous output. Dynamic Windows, on the other hand, are scrollable in both the vertical and horizontal dimensions; they have a definite origin (0, 0), but an indefinite length and width. Scrollability is a basic feature of Dynamic Windows and does not require the explicit use of special procedures as in the case of static windows.

Associated with the scrollability of Dynamic Windows are the concepts of *output history* and *viewport*. You do not have to clear a Dynamic Window to avoid overwriting previous output. New output, unless specifically directed otherwise, is appended to the bottom of the window's history, that is, at the end of all previous output to the window. The window is automatically scrolled so that the current viewport – the visible portion of the window – shows the new output. Previous output remains viewable by scrolling backwards through the history.

With the use of presentation types for doing output to a Dynamic Window, not only is the previous output retained and viewable, but its semantic content is also remembered. That is, links to the objects represented by displayed presentations are maintained so that the objects themselves remain accessible and usable as current program input. This capability is central to the SemantiCue input system. In the appropriate input context (established by your program), the displayed presentations are automatically mouse-sensitive. Automatic mouse-sensitivity is another point where Dynamic Windows depart from static windows; with a static window, mouse sensitivity must be provided through explicit procedures associated with output operations.

- **Advanced Command Facilities**

At the next level up from substrate facilities are advanced facilities for command management, user input, and program output. These facilities and the substrate facilities are available for when you need low-level tools for user interface programming. With the exception of the predefined presentation types, they are not intended for general use in most

applications; the general-use tools are in the "basic" and "top-level" categories.

The advanced command facilities allow you to write your own command loop. Three kinds of facilities are provided:

1. Tools for reading and parsing command input.
2. Tools for managing command tables.
3. Tools for defining and installing single-key command accelerators.

- **Advanced User Input Facilities**

The presentation and Dynamic Window substrates provide for the display of mouse-sensitive items in your programs' windows. Being able to use these items as program input or in other useful ways by clicking on them with the mouse depends on the availability of *mouse handlers*. Handlers, in addition to the presentation system and Dynamic Windows, form the third key ingredient of the SemantiCue input system.

Many mouse handlers are predefined in Genera, and it is possible that you will never have to define your own. However, with the Advanced User Input Facilities you can create your own handlers if the need arises. They come in two varieties: *translating mouse handlers* and *side-effecting mouse handlers*.

A translating mouse handler translates a displayed presentation of one type to an input object of another type. For example, you could define a translating mouse handler to extract a host object from a pathname presentation. Such a handler would return the "Host" object if a user clicked on the following displayed pathname: "Host:>dierdre>new-t-m-s.lisp".

By the way, the standard handler for inputting objects of a specified presentation type is a translating mouse handler. This predefined facility is known as the *identity handler*, because it "translates" a presentation object to itself, that is, the same object with the same presentation type. In the above example, if a pathname was sought rather than a host, the identity handler would automatically be available for clicking on the displayed presentation to return the pathname object.

A specialized type of translating handler translates presentation objects into Command Processor commands invoked on the objects. The facility for creating such handlers is considered top-level, and is introduced elsewhere:
See the section "Introduction to Top-Level Facilities for User Interface Programming", page 17.

A side-effecting mouse handler accomplishes some task independently of your main program, like showing information about a presentation object. For example, invoking a side-effecting mouse handler on a pathname presentation might display file attributes; nothing gets returned to your program, but the user has some additional information about the presentation object.

Also included in the Advanced User Input category are several ancillary facilities providing the interface between mouse characters and *mouse gestures.* A mouse gesture is the symbolic name, conventionally a keyword, corresponding to a mouse character. For example, **:select** is a gesture corresponding to #**\mouse-1**, that is, click-left. More than one gesture can correspond to the same mouse character. For example, another name for #**\mouse-1** is **:left**.

Mouse handlers are defined on a particular gesture. We say that a handler is "available on" the gesture. The interface facilities provide translation between mouse characters and gesture keywords. (For more information on mouse characters and related functions: See the section "Mouse Characters" in *Programming the User Interface, Volume B.*)

- **Advanced Program Output Facilities**

Advanced Program Output Facilities include macros and functions for

1. Creating *replayable* presentations.
2. Doing *incremental redisplay.*
3. Writing your own formatted output macros.

Replayable presentations are ones that can be rerun, in place, and displayed in a new format. You, the programmer, specify the redisplay options, called "viewspec choices". At runtime, a user of your program can click on the replayable presentation and call up a menu listing the viewspec choices. After exiting the menu the presentation is erased and redisplayed according to choices made by the user.

To see an example of a replayable presentation, invoke the Show Processes command in a Lisp Listener or **break** window. Now, with the mouse cursor anywhere in the displayed listing, click s-sh-Middle. This brings up a menu entitled "Output parameters" listing the viewspec choices. Try changing the selected choice from **None** to any of the others, click on **Done**, and watch what happens.

A set of inter-related facilities is provided for doing incremental redisplay of

program output. Output intended for redisplay is saved in an *output cache.* With the redisplay facilities, you can cache formatted or presented output and compare it against re-output of the same objects to check for changes. If changed, the cache is updated and the objects are redisplayed; if not, both the cache and the original display remain unaltered.

A large number of formatted output macros are already available among the Basic Program Output Facilities. Most programmers will not need to write their own, but if you do, we provide two facilities to help. The first is a macro for "snapshotting" the current values of lexical variables used within its body. The second is a function for determining the space needs of a specified continuation on a specified stream.

- **Basic Command Facilities**

Above the advanced facilities in the user interface hierarchy are basic facilities for defining commands, getting user input, and doing program output. It is at this level and the one above (top-level facilities) that application programmers find tools meeting most of their user interface needs.

The basic command facilities include two kinds of functions. The first lets you define Command Processor commands. As part of the definition process, you install you commands in a command table, for example, the "global" table that includes all the predefined Command Processor commands available in a Lisp Listener. Once defined, these may be invoked identically to the predefined commands. For example, if you define a new kind of Show File command, which you name Show Lisp File, and install it in the "global" table, the next time you select a Lisp Listener or enter a **break** loop, Show Lisp File will be one of the available commands.

The second kind of basic command facilities provides the interface between your programs and predefined or newly defined Command Processor commands. That is, these functions let you assemble Command Processor commands in your application code; when the code is run the commands are executed.

- **Basic User Input Facilities**

The basic function underlying most facilities for user input from Dynamic Windows is **accept**. Using this function and presentation types enables you to do typed input. ("Typed input" refers to object types, not typing at the keyboard.)

The output function that corresponds to **accept** is **present**. The **accept** functions within your programs determine the mouse-sensitivity of previously **present**ed output. Consider the following series:

```
(present "A:>ptolemy>solar-data.data") ==>
A:>ptolemy>solar-data.data
#<DISPLAYED-PRESENTATION 454412134>

(present #p"A:>copernicus>solar-data.data") ==>
A:>copernicus>solar-data.data
#<DISPLAYED-PRESENTATION 454412456>

(accept '((string))) ==>
Enter a string: A:>ptolemy>solar-data.data
"A:>ptolemy>solar-data.data"
STRING

(accept '((pathname))) ==>
Enter the pathname of a file: A:>copernicus>solar-data.data
#P"A:>copernicus>solar-data.data"
FS:LMFS-PATHNAME
```

In the first case, a string is presented, in the second a pathname. With the first **accept** function, an *input context* for **string** objects is established. Passing the mouse cursor over the string presentation, A:>ptolemy>solar-data.data, causes the presentation to be highlighted, thereby telling the user "We are looking for a string; this is a string; you can click on it and return it as a string object". This is what was done in the example, indicated by the italicized echo on the "Enter a string" prompt line.

In the **string** input context, passing the mouse cursor over the presented pathname, even though it looks the same as the string presentation, does not result in its highlighting. Despite its appearance, it represents a pathname object, acceptable only in contexts where pathname objects are sought. Such a context is established by the second call to **accept**. In that context, the pathname presentation is highlighted and the user can click on it to return the presentation object, that is, the pathname object #P"A:>copernicus>solar-data.data".

The interaction described above illustrates the kind of intelligence incorporated into SemantiCue, that is, what makes it a "smart" user input system. The Genera user interface relies extensively on this system. Using **accept**, **present**, and related functions lets you create similar interfaces to your programs.

In addition to **accept**, other facilities in the basic user input category provide the ability to prompt for and accept multiple objects. The accepted objects are returned when the function returns, or used to change the values of specified variables directly, before the function returns.

When a multiple-**accept** function is executed, either a series of *in-line prompts* or a separate window containing the prompts appears. The in-line prompts are so called because they appear in the same window that the function was called from, and remain in that window's output history. To see an example of in-line prompts, invoke the Set Window Options command in a Lisp Listener or **break** loop.

To see an example of a multiple-**accept** function generating a separate window for the prompts, evaluate the following:

```
(dw:accept-values '((integer :prompt "Half-life"
                             :default 24000)
                    (pathname :prompt "Log file")
                    (integer :prompt "Session number"))
          :prompt "Atomic experiment"
          :own-window t)
```

The window generated is equipped with its own scroll bar (for long prompt lists), as well as `Abort` and `Done` boxes on the bottom margin.

- **Basic Program Output Facilities**

The primary output facilities are those for *presenting* objects, the **present** function in particular. This function, and those based on it, output objects as *presentations*. A presentation includes not only the display itself, but also the object presented and its presentation type. When presentations are output to Dynamic Windows, the object type 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 mouse input to functions for **accept**ing objects. (For an example of a **present-accept** interaction: See the section "Introduction to Basic User Input Facilities", page 14.

In general, the display of a presented object depends on its presentation type. However, the display of any presented object can be modified independently of its type, and of what that type would normally dictate. If appropriate in your application, you could present a string as a graphic display, for example, and still have the string object be available for program input via the mouse.

Other basic output facilities include macros for controlling character output and a large number of formatting macros. The character output facilities provide control over character style or style components (family, face, and size). (For more information on character styles: See the section "Character Styles" in *Symbolics Common Lisp: Language Concepts.*) Other facilities let you specify underlining, filling, abbreviation, and truncation of character output.

The formatting macros are high-level facilities for creating textual lists, tables, and graphs. You provide the textual list facilities with a sequence of objects; they provide item delimiters, like commas, and a conjunction between the final two items. The table facilities let you create two-dimensional displays of simple or compound objects; they give you detailed control over layout. Two graph formatting facilities are available; both are for constructing hierarchical graphs showing the connections among object nodes.

Additional output facilities include a series of functions for graphic output – points, lines, arrows, strings, circles, polygons, and so on. The basic facilities also provide a set of methods and functions helpful for doing graphic displays on Dynamic Windows, including one to track the mouse.

Genera's display facilities in general, and the high-level formatting macros in particular, are collectively known as Showcase. The Showcase facilities are intended to make generating useful and attractive displays an easier-going task than if you had to do all the formatting yourself. You get to spend more time on application-specific needs for program output, and less on the requirements that most applications have in common.

- **Top-Level Facilities for User Interface Programming**

The top-level facilities include a utility for prototyping/designing the window and command interface to your program, the macro on which that utility is based, and additional facilities for enhancing the command interface.

The prototyping utility is called the Frame-Up Layout Designer. It offers you a choice of several standard configurations for the *program frame* that will form the basis of the window interface with your users. Alternatively, you can interactively construct the program frame, by modifying the initial configuration (displayed on entering the utility) or one of the standard configurations that Frame-Up provides.

Frame-Up offers options affecting the appearance and behavior of your program that correspond directly to options for

dw:define-program-framework, the macro on which Frame-Up is based. These options can be grouped into three areas:

1. Options affecting *panes* (subwindows) within the program frame. These options control the size and placement of program panes as well as their function, for example, whether a pane is a command-menu pane or one for displaying program output.

2. Options affecting your program's *command loop*. These options specify how program commands are defined, aspects of command table management, and the command loop function.

3. Miscellaneous options. Options in this area affect such things as the user-visible name of your application, the key it can be selected on, whether it is listed in the system menu, and so on. An important option in this group lets you specify your program's state variables. Doing so makes the variables accessible to methods you write for your program.

After you have designed a prototype program frame and specified whatever options are appropriate, Frame-Up writes out the **dw:define-program-framework** code corresponding to your specifications. The code is written to an editing buffer, where you can add the additional user interface features you desire, and the necessary links to your application.

Of the top-level facilities for enhancing the command interface to your program, one of the key capabilities provided is that of *presentation-to-command translation.* This capability lets your users click with the mouse on a displayed presentation, and have that gesture cause the execution of a command, using the presentation object as one of its arguments. The command executed can be a standard Command Processor command or, more likely, one you have specially created for your program. To see how this works, evaluate the following **present** function, or a similar one presenting a real pathname, in a Lisp Listener:

```
(present #p"y:>example>pathname.test")
```

Now, hold down the META key and place the mouse cursor over the presented pathname. Look at the top mouse documentation line and should see something like `m-Mouse-L: Edit File (file) Y:example>pathname.test`. This is the result of having a presentation-to-command translator available on the `m-Mouse-L` gesture. Clicking `m-Mouse-L` at this point executes the Edit File Command Processor command on the pathname object represented by the presented pathname.

Presentation-to-command translators are a special kind of mouse handler. For an introduction to mouse handlers generally: See the section "Introduction to Advanced User Input Facilities", page 12.

3. Overview of Top-Level Facilities for User Interface Programming

The following table lists the top-level facilities available for programming the user interface.

Table of Top-Level Facilities for User Interface Programming

Frame-Up Layout Designer

Program Framework Definition
dw:define-program-framework
dw:*program-frame*
dw::find-program-window
dw:get-program-pane

Program Command Definition
dw:define-program-command
define-presentation-to-command-translator

The implementation of these facilities is based on a model, or paradigm, for user interaction with application programs that, because of the unique nature of the enabling substrate, may be unfamiliar to you. (For more information on the substrate facilities: See the section "Overview of Presentation Substrate Facilities", page 69. See the section "Overview of Window Substrate Facilities", page 87.) We discuss this paradigm in the first section below, before considering the facilities themselves.

Reference documentation for the top-level facilities considered in the sections that follow is provided in a user interface dictionary: See the section "Dictionary of Top-level Facilities for User Interface Programming", page 99.

3.1 User Interaction Paradigm

User interfaces built on the presentation and Dynamic Window substrates provide a style of program interaction unlike that of conventional user interfaces. Central to the interaction paradigm is a Command Processor that is entirely based on these substrates and manages the user interface aspects of all commands, whether system commands or ones you create for your program. (For more on the

Command Processor: See the section "Overview of Basic Command Facilities", page 31.)

Top-level control for the program is provided by its *command loop.* The command loop for most programs is similar:

1. Read a command.
2. Execute the command.
3. Redisplay any modified data structures (that are already displayed).

The command reader part of the loop builds and then parses a complete "sentence", the command. Command sentences generally include "verbs", specifying the action to be performed (for example, Show File); "nouns", the objects on which the specified action is to be performed (for example, a pathname argument to the Show File command); and "modifiers", specializations introduced via optional, typically keyword, arguments. In our Show File example, the only possible modifier is the `:Output Destination` keyword. The complete sentence, then, is

```
Show File (file [default Q:>linda>library.text])        [verb]
Q:>linda>library.text                                   [noun]
(keywords) :Output Destination (a destination) Printer [modifier]
(a printer [default Asahi Shimbun]) ASAHI SHIMBUN
```

Users can construct command sentences from keyboard input, mouse input, or a mixture of the two. Mouse handling with respect to the Command Processor is synchronous, meaning that mouse and keyboard input can be interleaved in the construction of a command sentence (they use the same input buffer). Thus, for example, if the user types in the Show File command, the pathname argument can be supplied by clicking on a pathname presented earlier in the output.

Only presentations of a type appropriate to the command at hand are mouse-sensitive. Appropriate presentations are ones whose type matches that of the noun object needed to complete the command sentence. Also appropriate are presentations that can be *translated* into objects of the type needed. In the above example using Show File, presentations of the pathname type will be sensitive, as well as presentations for which *translating mouse handlers* are available that, if invoked (by clicking), will generate pathname objects. Sensitivity is indicated by highlighting – enclosure within a box – when the mouse cursor moves over a presentation of the right type. (For more information on mouse handlers: See the section "Overview of Mouse Handler Facilities", page 39.

Another kind of mouse handler is available for translating directly from a displayed presentation into a command executed using the object represented by the presentation, that is, the *presentation object,* as one of its arguments. For an

example, do a Show Directory listing in a Lisp Listener. Highlight one of the displayed pathnames by moving the mouse cursor over it and look at the top mouse documentation line: it informs you that by clicking left you can execute Show File on the highlighted pathname. This and other file-related commands – click right on a pathname to pop up the menu listing them – are available on various *mouse gestures* because *presentation-to-command translators* have been defined for them.

Translating mouse handlers provide one kind of *command acceleration*, and menus provide another. *Command menus* are especially useful, and used widely in Genera. A command menu contains a set of verbs or verb phrases that approximate, or are the same as, the names of the commands to which they correspond. Clicking on one of the verbs supplies, or solicits from the user, the noun(s) and modifiers for the rest of the sentence. Typically, a command menu is displayed when a program is selected and remains displayed until it is deselected. For examples, look at the Peek program (menu at top) or Zmail (menu in the middle).

Even though some commands cannot be entered with the mouse and others would be difficult or impossible to enter without some mouse-sensitive items to accelerate them (for example, graphic presentations), all are managed by a common command processing mechanism. This mechanism provides the same *help*, *mouse documentation*, and *completion* facilities to your commands that it supplies to system commands.

3.2 Frame-Up Layout Designer

The Frame-Up Layout Designer is an interactive code-building utility that helps you write the user interface to an application program. The code produced is written as a single definition using **dw:define-program-framework** (described in another section of this overview: See the section "Overview of Program Framework Definition Facilities", page 25.) What you are defining, precisely, is a *program flavor* having as its name the name of your program.

Frame-Up, then, is the interactive version of **dw:define-program-framework**. Using the latter facility, you can control, via keyword options, the specifics of the *program frame* – the window interface to your application – and various aspects of the command loop. (For an explanation of the command loop: See the section "User Interaction Paradigm", page 21.) In Frame-Up, the same options are offered as menu items.

During a session with Frame-Up, you configure the program frame by selecting one of several standard configurations or by editing a default or selected configuration. Editing operations include the splitting, swapping, sizing, and deletion of *panes* (subwindows) within the program frame.

Program panes, all based on a dynamic pane flavor, come in six varieties:

- Title panes
- Command-menu panes
- Display panes
- Interactor panes
- Lisp Listener panes
- Accept-values panes (another kind of menu pane for accepting variable, user-specified values)

For panes of each type, an appropriate set of options is available, controlling such factors as the pane name, height, whether a typeout window can appear, and the name of the function controlling redisplay of output to the pane.

In addition to pane options, Frame-Up provides program options for specifying the program name, key to be used for selecting the program, and factors related to Command Processor operations. Again, both these and the pane options are implemented as menu items that map to keyword options to **dw:define-program-framework**.

When you are done laying out your program frame and specifying interface options, you can preview the result and, if acceptable, exit to the editor buffer where you wish the interface code to be written. Using an editing command, you can then have Frame-Up write the **dw:define-program-framework** code corresponding to your interface into the buffer (it is appended to anything that was already in the buffer).

At this point you have the foundation and a good part of the superstructure of the user interface to your application. Of course, you have to write your program's commands and all of the application-specific code not already in place. Much of that code will manipulate your program's state variables. Note that Frame-Up, through **dw:define-program-framework**, has created a program flavor for your application. This means that your state variables can be set up as instance variables to the program flavor, and that you can access them directly in methods written for the program flavor.

Because of the close connection between the Frame-Up Layout Designer and **dw:define-program-framework**, you may find the overview of the latter facility helpful in understanding the former: See the section "Overview of Program Framework Definition Facilities", page 25. For complete documentation of Frame-Up: See the section "Frame-Up Layout Designer", page 103.

3.3 Program Framework Definition Facilities

All top-level user interface facilities are based on a model for application programs. Typically, a set of commands is made available to the user which, when invoked, implements the program-specific functions forming the core of the application. As each command is executed, displayed information affected by the invoked function(s) is updated and redisplayed. This sequence of events, from waiting for command input through execution and redisplay, is referred to as the command loop.

Soliciting user input and displaying or redisplaying program output are user interface functions separable to a large extent from the implementation details of particular applications. If those details form the core of the application, then the user interface functions can be thought of as the framework. The framework definition facilities let you abstract the user interface functions from your program and implement them at a high level. Central to this capability is the macro **dw:define-program-framework**.

A major function performed for you by **dw:define-program-framework** is that of establishing and managing the command loop for your program. As part of the services provided in this area, it sets up a command-definition macro specifically for your program. This macro is essentially the same as **dw:define-program-command**, which is considered in another section of this overview. (See the section "Overview of Program Command Facilities", page 28.) For example, say you have a game program named "nickel-dime"; the first part of the **dw:define-program-framework** definition for this application might look something like:

```
(dw:define-program-framework nickel-dime
  :pretty-name "Nickel & Dime Game"
  :command-definer define-n-d-command
  ... )
```

The value provided to the **:command-definer** keyword becomes the symbol for the command-definition macro that **dw:define-program-framework** creates for you. In other words, you could now write program commands using

```
(define-n-d-command (<command-name> <program-name> [keywords])
        <arglist>
        <body>)
```

In addition to establishing and managing the command loop, **dw:define-program-framework** provides control in two other key areas: management of screen real estate via a program frame (window); and management of your program's state variables. The former capability lets you specify the frame configuration(s) your program presents to the user. Specification details

include the types and sizes of various panes (subwindows) within the frame created by **dw:define-program-framework** for your program. (For a discussion of frames and panes: See the section "Frames" in *Programming the User Interface, Volume B.*)

State variables are program variables whose states (bindings) are preserved between activations of a program. They are managed through a keyword option to **dw:define-program-framework**. By using this option, program data, which you might otherwise store as special variables, are stored instead as instance variables. (For a discussion of variables: See the section "Kinds of Variables" in *Symbolics Common Lisp: Language Concepts.*)

The flavor to which the instance variables belong is your program itself; that is, **dw:define-program-framework** creates a *program flavor* unique to your program and having as its name the name of your program. An important and useful consequence of this is that program functions may be written as methods to the program flavor, and thereby have direct access to its instance variables, including your state variables. (For information on flavors and methods: See the section "Flavors" in *Symbolics Common Lisp: Language Concepts.*)

To illustrate these points, let's extend the nickel-dime game example begun above:

```
(dw:define-program-framework nickel-dime
  :pretty-name "Nickel & Dime Game"
  :command-definer define-n-d-command
  :panes ((title-pane :title)
          (command-pane :command-menu)
          (graphics-window :display)
          (message-window :interactor))
  :configurations '((first
                      (:layout
                        (first :column title-pane command-pane
                         graphics-window message-window))
                      (:sizes
                        (first (title-pane 0.05)
                               (command-pane :ask-window self
                                 :size-for-pane command-pane)
                               :then
                               (graphics-window 0.8)
                               (message-window 0.2)))))
  :state-variables ((game-flag)
                    (user-input)
                    (game-array)
                    (history-list)
                    ... )
  ... )
```

The program frame is specified first by the **:panes** option, which indicates the names and types of panes included; and second by the **:configurations** option, which controls details of pane layout and size.

The **:state-variables** option identifies program variables. Having been thus identified, these variables are lexically available in methods written for the program flavor `nickel-dime`. The following is a simple method to keep track of moves made in the game so far:

```
(defmethod (game-history nickel-dime) ()
   (setq history-list (append history-list
                        (cons game-array NIL))))
```

Three other facilities are provided for use in conjunction with **dw:define-program-framework**. These are **dw:*program-frame***, **dw::find-program-window**, and **dw:get-program-pane**.

The first, **dw:*program-frame***, is a variable bound to the currently exposed program frame. The following example was generated by selecting the Frame-Up Layout Designer – a example of a program created with **dw:define-program-framework** (the Flavor Examiner is another) – and pressing `SUSPEND` to enter a **break** loop:

```
Command: ,dw:*program-frame* ==>
#<PROGRAM-FRAME Frame-Up 1 3106337 exposed>
```

dw::find-program-window returns the program frame of a specified program flavor, whether it's exposed or not. Optionally, it creates and initializes an instance of the program if one does not already exist. Using **dw:get-program-pane** is how you access a particular pane of a program frame, rather than the frame as a whole.

Reference documentation for **dw:define-program-framework** and ancillary facilities is included in a user interface dictionary: See the section "Dictionary of Top-level Facilities for User Interface Programming", page 99. For an example and additional information on the use of certain options to **dw:define-program-framework**, particularly those implementing the command interface: See the section "User Interface Application Example", page 91. An advanced example is included in the file sys:examples;define-program-framework.lisp.

The Frame-Up Layout Designer is an interactive version of **dw:define-program-framework**. For an overview of this facility: See the section "Overview of the Frame-Up Layout Designer", page 23. For more detailed documentation: See the section "Frame-Up Layout Designer", page 103.

3.4 Program Command Facilities

Two key facilities are included in this category of top-level user interface tools. The first is **dw:define-program-command**; the second is **define-presentation-to-command-translator.**

The command-definition macro **dw:define-program-command** is intended for use only in conjunction with **dw:define-program-framework** (reviewed in another section: See the section "Overview of Program Framework Definition Facilities", page 25.) The macro not only lets you define commands for your program, but also specifies whether they are shown on a command-menu pane created by **dw:define-program-framework** for your program frame. Moreover, use of the two macros ensures that your commands are properly installed in the command table created for your program (by **dw:define-program-framework**). In other respects, **dw:define-program-command** is similar to the basic command-definition facility, **cp:define-command**. (For more information on **cp:define-command**: See the section "Overview of Basic Command Facilities", page 31.)

define-presentation-to-command-translator creates a mouse handler that lets your program's users click on a presentation and, through that action, cause a specified program command to be executed on the presentation object. To use the terminology presented in another section, it sets up the noun-verb order of the command sentence. Clicking on the noun initiates the completion and execution of the command sentence. The verb part of the sentence, that is, which command gets invoked on the noun, depends on the particular mouse gesture used.

The following example is taken from the system source. It defines the Delete File presentation-to-command translator:

```
(define-presentation-to-command-translator si:com-delete-file
                                           (fs:pathname
                                             :gesture nil)
  (path)
  (cp:build-command 'si:com-delete-file (ncons path)))
```

Note the use of **cp:build-command** in the body of this translator. This is the recommended way of interfacing to Command Processor commands from presentation-to-command-translators. Note also that the **:gesture** option to the translator is **nil**. This means that the translator is not available on any gesture, but only in the click-right menu available for all presentations.

For more on the role of presentation-to-command translators in the user interface: See the section "User Interaction Paradigm", page 21. For an overview of mouse handlers generally: See the section "Overview of Mouse Handler Facilities", page 39.

Reference documentation for these facilities is included in a user interface

dictionary: See the section "Dictionary of Top-level Facilities for User Interface Programming", page 99. For examples in the context of an application program: See the section "User Interface Application Example", page 91.

4. Overview of Command Processor Facilities

The facilities described here are divided into basic and advanced categories. The distinction is between functions that most application programmers are likely to use regularly, and those that they are not. The boundary is not a hard one, and we recommend that you look over both sections, especially if you are unfamiliar with Command Processor programming.

Reference documentation for the facilities discussed here is included in a user interface dictionary: See the section "Dictionary of Command Processor Facilities", page 135.

4.1 Basic Command Facilities

Table of Basic Command Facilities

Command Definition Facilities
cp:define-command

Command Processor Interface Facilities
cp:execute-command
cp:build-command
cp:*last-command-values*

As the above table shows, the basic command facilities are for

- Defining new Command Processor commands
- Providing an interface between your program and pre-existing Command Processor commands or those you newly define.

Only one basic facility is needed for defining Command Processor commands, **cp:define-command**. This macro lets you both create a command and install it into the *command table* of your choosing. For example, all predefined commands, those listed when you type "help" to the Command Processor prompt in a Lisp Listener, are in the "Global" command table. You may specify that your commands also be available in the "Global" command table, or in an application-specific command table. (For more information on command tables: See the section "Command Processor Command Tables" in *Programming the User Interface, Volume B.*)

If you are writing Command Processor commands intended specifically for use with a program you have created using **dw:define-program-framework**, you can do so with **cp:define-command**, but **dw:define-program-command** would be the better choice. The latter facility is intended for use with program definitions; it provides lexical access to a program's state variables, and other services as well: See the section "Overview of Program Command Facilities", page 28.

The objects listed in the table under Command Processor Interface Facilities allow you to use predefined Command Processor commands in your own code. The first, **cp:execute-command**, is used by programs to invoke Command Processor commands on a specified set of arguments. The second, **cp:build-command**, is used similarly by command translators (defined with **define-presentation-to-command-translator**). (For an example of a command translator showing the use of **cp:build-command**, and of **cp:define-command**: See the section "User Interface Application Example", page 91.)

The special variable **cp:*last-command-values*** provides access to the values returned by the most recently executed Command Processor command.

4.2 Advanced Command Facilities

One of the major advantages of using the top-level facilities for building the user interface to an application program is that they provide the command loop. (See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.) This relieves you of explicit responsibility for creating command prompts, reading and parsing commands, and so on. You can concentrate instead on the application-specific details of the commands themselves.

However, if you need some subtlety of command loop behavior not available in the default command loop functions used by **dw:define-program-framework**, then you can write your own functions with the aid of the facilities reviewed in this section. Note that this does not mean that you cannot or should not use **dw:define-program-framework** to build your user interface; it means only that you should make use of the **:top-level** and **:command-evaluator** keywords to that macro and supply your own command loop functions. For examples: See the section "User Interface Application Example", page 91.

The Advanced Command Facilities are divided into three subcategories:

- Command Loop Management Facilities
- Command Table Management Facilities
- Command Accelerator Facilities

4.2.1 Command Loop Management Facilities

The first subcategory of Advanced Command Facilities includes facilities for building command loops. A primary requirement is for reading and parsing commands, the function of the first six facilities listed below, from **cp:read-command** to **cp:read-accelerated-command.** They include command readers for regular commands, extended commands, accelerated commands, and so on.

Command Loop Management Facilities
- **cp:read-command**
- **cp:read-command-or-form**
- **cp:read-command-arguments**
- **cp:yank-and-read-full-argument-command**
- **cp:read-full-command**
- **cp:read-accelerated-command**
- **cp:echo-command**
- **cp:unparse-command**
- **cp:define-command-and-parser**
- **cp:turn-command-into-form**
- **cp::*default-blank-line-mode***
- **cp::*default-dispatch-mode***
- **cp::*default-prompt***

The other facilities listed in this subcategory provide a variety of useful services. For example, **cp:unparse-command** takes a command symbol and any arguments and returns the characters that would have been typed in to produce that command; you can use it, as the system does, to construct mouse documentation. **cp:define-command-and-parser** is a low-level, command-defining macro that lets you control how the command line is parsed. **cp:turn-command-into-form** takes a command name and a list of arguments, and constructs an evaluable form.

Finally, the three special variables – **cp::*default-blank-line-mode***, **cp::*default-dispatch-mode***, and **cp::*default-prompt*** – provide defaults for use by **cp:read-command** and **cp:read-command-or-form.**

4.2.2 Command Table Management Facilities

Command Table Management Facilities
- **cp:*command-table***
- **cp:make-command-table**
- **cp:find-command-table**
- **cp:install-commands**
- **cp:delete-command-table**
- **cp:command-in-command-table-p**

The Command Table Management Facilities are mostly self-explanatory. The current binding of the variable, **cp:*command-table***, is the command table used by the Command Processor to read commands. The next three facilities are functions for making and retrieving command table objects, and for installing commands into command tables. **cp:delete-command-table** removes a command table from the command table registry, and the predicate **cp:command-in-command-table-p** lets you test for the inclusion of a command in a command table.

For more information on command tables: See the section "Command Processor Command Tables" in *Programming the User Interface, Volume B.*

4.2.3 Command Accelerator Facilities

Command accelerators form the focus of the final subcategory of Advanced Command Facilities. Only one facility is provided, used for defining command accelerators:

Command Accelerator Facilities
cp:define-command-accelerator

Command accelerators are so called because they allow you to invoke, with a single key, a Command Processor command normally invoked by one or more words. For example, suppose in your application you define an Exit command to bury the program frame. You could put this command on the key E or X. A user would merely have to press the E or X key to exit the program.

When deciding whether to create new command accelerators, be aware that your program can inherit any command accelerators already existing in other command tables. If your program inherits these tables via the **:command-table** option to **dw:define-program-framework**, installed accelerators come along with the commands they accelerate. (See the function "**dw:define-program-framework**", page 124.)

If you wish to define and install your own accelerators, you can do so with **cp:define-command-accelerator.**

5. Overview of User Input Facilities

This section divides the user input programming facilities of SemantiCue into basic and advanced categories. As with similar distinctions made for Command Processor and program output facilities, we expect most programmers to make relatively heavy use of the basic facilities, and lighter use of the advanced facilities. Programming styles and needs differ, however, so the distinction is a somewhat arbitrary one, not to be taken too seriously.

If these facilities are new to you, you might find the introductory sections to this volume helpful in understanding the following discussion: See the section "Introduction to the User Interface Management System", page 7.

Reference documentation for all the user input facilities can be found in a user interface dictionary: See the section "Dictionary of User Input Facilities", page 163.

5.1 Basic User Input Facilities

The user input and program output facilities provided in Genera rely on the presentation-type system. This system is an extension of the Common Lisp type system and was created especially to facilitate user interface programming. The Basic User Input Facilities described in the following subsections are based on the presentation-type system and designed for use with Dynamic Windows. For more information on presentation types: See the section "Overview of Predefined Presentation Types", page 71. For more on Dynamic Windows: See the section "Overview of Window Substrate Facilities", page 87.

5.1.1 Facilities for Accepting Single Objects

Basic User Input Facilities can be categorized into those for accepting single objects and those for accepting multiple objects. Facilities in the first category are listed below.

Facilities for Accepting Single Objects
- **accept**
- **prompt-and-accept**
- **accept-from-string**
- **dw:menu-choose**
- **dw:menu-choose-from-set**

The primary facility for accepting input of presentation objects is the Symbolics

Common Lisp function **accept**. Objects can be accepted via keyboard or mouse input. Characters typed in at the keyboard in response to an **accept** prompt are parsed, and the object they represent is returned to the calling function. Alternatively, if the object has previously been output as a presentation and is in the current viewport of a Dynamic Window, the user can click on the object with the mouse and cause it to be returned directly (that is, no parsing is required).

Examples:

```
(accept '((string))) ==>
Enter a string: a string
"a string"
((STRING))

(accept '((string))) ==>
Enter a string [default a string]: a string
"a string"
((STRING))
```

In the first **accept** function, "a string" was typed at the keyboard. In the second **accept**, the user clicked on the keyboard-entered string of the first function. In both cases, the string object was returned.

Typically, not any kind of object is acceptable as input. Only an object of the presentation type specified in the current **accept** function can be input. The **accept** function establishes the current *input context*. For example, if the call to **accept** specified an integer presentation type, only a typed in or displayed integer would be acceptable. Numbers displayed as integer presentations would, in this input context, be mouse-sensitive, but those displayed as part of some other kind of presentation, for example, a file pathname, would not. Thus, **accept** controls the input context and thereby the mouse sensitivity of displayed presentations.

We say above that the range of acceptable input is, typically, restricted. How restricted 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. Consider the following example:

```
(accept '((or ((integer 1 4))
              ((dw:member-sequence
                 ("one" "two" "three" "four"))))))) ==>
Enter an integer greater than or equal to 1 and
less than or equal to 4 or one, two, three, or four: three
"three"
((DW:MEMBER-SEQUENCE ("one" "two" "three" "four")))
```

```
(accept '((or ((integer 1 4))
              ((dw:member-sequence
                 ("one" "two" "three" "four"))))))  ==>
Enter an integer greater than or equal to 1 and
less than or equal to 4 or one, two, three, or four: 4
4
((INTEGER 1 4))
```

The particular combination of types used above might not have any practical use, but it does begin to illustrate what the possibilities are. Notice that **accept** took care of devising a prompt. You could override this if you wanted to, but in most cases it comes up with something reasonable.

The parser used by **accept** for parsing strings into presentation objects is not part of the **accept** function itself. Rather, each presentation type has its own, type-specific parser that **accept** calls to parse objects of that type. The parser function is included in the form that defines a presentation type. You may write your own presentation types, including the parsers (and printers) that go with them, but a sizeable set of types has already been defined for you: See the section "Overview of Predefined Presentation Types", page 71. Each is documented in a user interface dictionary: See the section "Dictionary of Predefined Presentation Types", page 281.

Ancillary functions for accepting single objects include **prompt-and-accept** and **accept-from-string**. The first is the presentation-system equivalent of **prompt-and-read**. It is similar to **accept**, taking the same keyword options, but differs in its letting you use the **format** function to generate the input prompt. **accept-from-string** is the presentation-system equivalent of **read-from-string**.

Two **accept**-based menu facilities are included among the facilities for accepting single objects. The **dw:menu-choose** function is a menu-generating facility for use with Dynamic Windows. It displays a list of choices in a conventional menu format and returns the value associated (in your code) with the selected choice.

dw:menu-choose differs from the second listed menu facility, **dw:menu-choose-from-set**, in its ability to create menus of items in the "general list" form. (See the section "The Form of a Menu Item" in *Programming the User Interface, Volume B.*) **dw:menu-choose-from-set** is intended primarily for creating menus from a simple list of objects.

When considering menus for your applications, bear in mind that Dynamic Windows with displayed presentations can be regarded as menus of input possibilities. You may not need to construct a menu in the strict sense of **dw:menu-choose** to provide your users with the convenience that mouse acceleration of data entry provides.

5.1.2 Facilities for Accepting Multiple Objects

A second category of basic facilities for user input includes functions that return multiple objects to your program, rather than single objects. These are listed below:

Facilities for Accepting Multiple Objects
dw:accept-values
dw:accept-variable-values
dw:accepting-values

The function **dw:accept-values** is similar to **accept**. It differs in that it accepts a series of objects from the input stream, not just one object. The presentation type of each input object is specified independently. In the following example, an **integer** and a **pathname** object are sought:

```
(dw:accept-values '((integer :prompt "Half-life"
                             :default 24000)
                    (pathname :prompt "Log file"))
        :prompt "Atomic experiment") ==>
Atomic experiment
Half-life: 24000
Log file: Y:>curie>atom-data.log
ABORT aborts, END uses these values ==>
24000
#P"Y:>CURIE>atom-data.log"
```

The **dw:accept-variable-values** function is like **dw:accept-values**, but instead of returning a series of the user-entered values, it assigns these values to a set of special variables. It does this as the values are entered, not after the function returns. You have the option of constraining user choices for certain variables to a predefined set.

dw:accepting-values is a macro that takes all calls to **accept** within its body and puts the prompts into a single, multiple-prompt display like the one shown in the example above. It is the most versatile of the three and the one recommended for general use. One of its big advantages over the previous functions is that the multiple-prompt display can be modified at runtime, in response to values entered by your user to earlier prompts in the display. In other words, the values you solicit from your users can change "on the fly", at runtime, depending on the values already received. The following is a simple example:

```
(defun return-host-or-printer ()
  (fresh-line)
  (let (choice
        (stream *query-io*))
    (dw:accepting-values (stream :own-window t)
      (setq choice (accept '((member host printer))
                           :default 'printer
                           :stream stream
                           :prompt "Send file to host or printer?"))
      (case choice
        (host (accept 'neti:host :stream stream))
        (printer (accept 'sys:printer :stream stream))))))
```

For other examples, see the file sys:examples;accepting-values.lisp.

5.2 Advanced User Input Facilities

Facilities in this category are directed towards mouse manipulation of presentation objects, a key feature of the SemantiCue input system. The primary facilities in this category are those for defining mouse handlers. An ancillary set of facilities is provided for managing the interface between the symbolic mouse gestures used to invoke the handlers and the mouse characters to which the gestures correspond.

In the subsections that follow, we first present an overview of the mouse-handler definers and closely allied facilities. Then we look at the mouse gesture interface facilities. Following these two is an advanced concepts section. This section considers the important subject of *handler lookup*, that is, how SemantiCue finds the handlers applicable in any given input context, and some performance issues. The discussion is at a fairly advanced level, and is probably best put off until after you have a good working knowledge of handlers, presentation types, and the Dynamic Window system.

5.2.1 Mouse Handler Facilities

Facilities that let you or your users manipulate presentation objects with the mouse are referred to as mouse handlers. A large number of predefined mouse handlers are already included in SemantiCue. Clicking right on a displayed presentation in a Dynamic Lisp Listener throws up a menu of handlers applicable to the presentation object.

You define your own, application-specific handlers using the definition macros listed in the following table:

Mouse Handler Facilities
define-presentation-translator
define-presentation-action
dw:handler-applies-in-limited-context-p
dw:presentation-subtypep-cached
dw:delete-presentation-mouse-handler
dw:invalidate-type-handler-tables

Translating handlers, the kind generated with **define-presentation-translator**, are typically run when your program is waiting for input (of presentation objects). Given that some presentations are visible to your user, translators let the user click on a presentation of one type and use it as input of a different type, the type your program is seeking.

For example, say your program wants to input the version number of a file. You could define a translating handler that extracts the version number, an **integer** object, from a **pathname** presentation. Your program's users would then have the option of typing in a version number to the input prompt, or clicking on a **pathname** presentation that included a version number. Such a translator could be defined as follows:

```
(define-presentation-translator pathname-to-integer
   (pathname integer
             :gesture :middle
             :documentation "Return file version number")
   (pathname)
  (when (numberp (send pathname :version))
    (send pathname :version)))
```

After compiling this translator, try doing a Show Directory listing, then evaluate `(accept '((integer)))`. In this input context, move the mouse cursor over one of the pathnames and notice that the top mouse documentation line now says `Mouse-M: Return file version number; Mouse-R: Menu`. Clicking middle enters the file version number as an integer object.

define-presentation-to-command-translator is another translating-handler definition facility. It creates handlers for performing a single kind of translation, from presentations to Command Processor commands. This is considered a high-level facility and is discussed in another section: See the section "Overview of Program Command Facilities", page 28.

Side-effecting mouse handlers, the kind you create with **define-presentation-action**, are also typically run while your program is waiting for input, but do not themselves supply input. Rather, they run code outside the main control loop of your program to accomplish some action that is useful relative to the presentation which activates them.

A common use for side-effecting handlers is to display additional information about some presentation object. For example, if your program is providing graphic presentations of several key variables, it may be the case that to select of one of the variables to use as input, your user will require more information about the variables than can be included in the graphic representations. A side-effecting mouse handler could be used at this point to provide a display of all pertinent information about each of the available objects.

A major use made of side-effecting handlers by SemantiCue is to display menus of other handlers. The standard click-right menu for presentations, which shows handlers available in the current input context for the presentation at hand, is implemented in this fashion. Such handlers are created by specifying the **:defines-menu** option to **define-presentation-action.**

For some example mouse handler definitions: See the section "User Interface Application Example", page 91.

dw:handler-applies-in-limited-context-p and **dw:presentation-subtypep-cached** are related facilities used in **:tester** functions defined for translators. They restrict handler applicability to a specified input context. For more information: See the section "User-Defined Data Types as Presentation Types", page 82.

Other facilities concerned with mouse handlers include **dw:delete-presentation-mouse-handler** and **dw:invalidate-type-handler-tables.** The former eliminates a handler from your world. The latter is used when the presentation-type inheritance hierarchy has changed and the look-up tables controlling handler applicability need to be recomputed to reflect the change. For example, if you have defined a type that, depending on a flag, expands one way or another, then every time the flag changes you need to update the handler tables; **dw:invalidate-type-handler-tables** does this for you automatically the next time the tables are accessed.

5.2.2 Mouse Gesture Interface Facilities

Mouse Gesture Interface Facilities
dw:mouse-char-gesture
dw:mouse-char-gestures
dw:mouse-char-for-gesture

The Mouse Gesture Interface Facilities are ancillary to the mouse handlers. They provide the interface between mouse gestures, the symbolic names for mouse clicks – for example, **:left, :middle, :right** – and the mouse characters to which they correspond – **#\mouse-l, #\mouse-m, #\mouse-r.**

With these facilities, you can use predefined mouse gestures in your code where the symbolic names are required, or define and use new ones. Gestures are

required, in particular, for defining mouse handlers. Handlers are always defined on some gesture.

For more information about mouse characters and mouse character functions: See the section "Mouse Characters" in *Programming the User Interface, Volume B.*

5.2.3 Advanced Mouse Handler Concepts

5.2.3.1 How Mouse Handlers Are Found

You do not generally need to worry about the specifics of how SemantiCue decides what presentations to highlight or precisely which mouse handlers are available in a given input context. Still, it is helpful to understand the process conceptually, as it provides insight into some of the key aspects of SemantiCue's behavior. Also, this understanding is necessary when deciding how to correct some unexpected behaviors of the handlers you define.

When you move the mouse over a presentation, SemantiCue performs a multi-faceted search to find the right combination of 1) a presentation to highlight; 2) mouse handlers on that presentation; and 3) contexts to satisfy with the values returned by those handlers, were they invoked. The search includes the following activities:

- The presentation under the mouse is found. Presentations are arranged in a hierarchy; it is the innermost (smallest) presentation that is found at this stage. This presentation is found in the window's output history from the (x, y) position of the mouse.

- When the innermost *sensitive* presentation is found, the hierarchy of presentations is searched, from innermost to outermost, to find a presentation that has applicable handlers.

- What handlers apply to a presentation is determined by matching the *to-presentation-type* of the handler with the input context(s), and the *from-presentation-type* of the handler with the presentation-type of the presentation or the type of the object (the two are not necessarily the same).

  Type matching is based on the **dw:presentation-subtypep** function. That is, the presentation or object type must be a subtype of the type the handler accepts, and the type returned by the handler must be a subtype of the type wanted by the program.

- Different levels of the software may be looking for different presentation types. For example, when accepting a command, the parser for the **cp:command** presentation type accepts a **cp:command-name**, which in turn looks for some text. A handler defined with

define-presentation-to-command-translator may satisfy the **accept** of the **cp:command** presentation-type; clicking on an item output in a `Help` display may satisfy the **accept** of **cp:command-name**; while the translator on c-Mouse-M may provide some of the text that is being read by **cp:command-name**'s parser. The search proceeds so as to favor satisfying the outermost context.

- Mouse handlers whose gestures are assigned to mouse characters with the current set of shift keys down are considered. (However, other handlers may may also be examined to determine which other shifts have commands. This is so SemantiCue can generate the lower mouse documentation line.)

- The handlers for a particular gesture are sorted according to a precedence ordering. This ordering follows these rules:

    1. All handlers on `:gesture t` (that is, all handlers professing to handle all gestures) are handled before handlers on specific gestures.

    2. If the first `:gesture t` handler found has been defined with the `:exclude-other-handlers t` option, then no other handlers are considered, even if the handler with `:exclude-other-handlers t` does not pass the tests described below.

    3. The handlers are then sorted by priority, and considered in priority order. The first one for a particular gesture that passes the tests is the one that is used. See the macro "**define-presentation-action**", page 179.

- Finally, a mouse handler must pass a series of tests before it is considered applicable to a presentation. The tests include:

    1. If there is a predicate associated with the mouse handler's *from-presentation-type*, it is applied to the presentation's object. This predicate can come either from a **satisfies** clause in the type or its expansion, or from a **:typep** argument in the **define-presentation-type**.

    2. If there is a **:tester** for the mouse handler, it is called on the presentation, the context, the window, the handler, and the gesture.

    3. The handler must either have `:do-not-compose t`, or its value must not be the single value **nil**. (A single value of **nil** means the body decided not to handle the presentation). This returned value may then be tested by a predicate derived from the context presentation-type, in a manner similar to the predicate derived from the *from-presentation-type*, in number 1 above.

4. If the handler declares that it defines a menu (via the **:defines-menu** option), a check is made that there is at least one handler which is declared to be in that menu that applies in the current combination of context and presentation.

If any of these tests returns **nil**, the handler does not apply.

With respect to performance, it is important to realize that not all of this search is performed each time the mouse cursor crosses a presentation. Although several general principles related to handler efficiency exist, SemantiCue uses many performance techniques that complicate any straightforward analysis in this area. Most things happen considerably more quickly than the preceding description might suggest. Because of various forms of caching, this is especially true after a handler search has already occurred in a given context.

See the section "Some Efficiency Caveats for Mouse Handlers", page 44.

5.2.3.2 Some Efficiency Caveats for Mouse Handlers

Following are some caveats for making your mouse handlers efficient:

- Make handlers as specific as possible.

  Use the most specific types appropriate as your handler's *from-presentation-type* and *to-presentation-type*. Doing so will respectively restrict the number of presentations to which the handler potentially applies and the variety of input contexts in which it is potentially available.

  In particular, avoid handlers for **t** and **sys:expression** contexts. These apply in a wide variety of contexts, and the effect is cumulative; the more there are, the slower everything becomes. If you do define such handlers, pay particular attention to their efficiency. This also applies to translators from and to subtypes of **sys:expression**. See the section "Use of User-defined Data Types as Presentation Types".

- Keep presentation-type **:expander** and **:abbreviation-for** forms simple.

  These forms are evaluated a large number of times. They should avoid both consing and excessive computation. It is best if they are simple backquoted forms, as the system knows how to turn such consing into stack-consing, resulting in more speed and less work for the garbage collector.

  Also, avoid large type expansions. An **:expander** or **:abbreviation-for** clause with a large expansion, especially inside an **or**, results in much extra searching and possibly increased memory requirements for the handler lookup tables. Carried to an extreme, this could make all handler lookups

slow owing to excessive paging. If needed, use a more general type and a **satisfies** clause.

- Keep **:tester** forms fast.

  Bodies of translators can be slow so long as the **:tester** form returns **nil** in the cases where the body would be slow.

- Keep translators fast.

  Expensive computations are best done as commands, rather than as translators. Translators run when you move the mouse; commands do not run until you ask for them.

- If a slow translation is necessary, use `:do-not-compose t`.

  If you feel a slow operation must be done as a translator, use `:do-not-compose t`. This suppresses SemantiCue's evaluation of the result. Because it also suppresses any contextual checking of the result, use it sparingly.

- Avoid interpreted **satisfies** clauses.

  Write an auxiliary function and use that instead. **satisfies** clauses are run during mouse handling; running them interpreted creates a needless slowdown.

For some related information and examples: See the section "User-Defined Data Types as Presentation Types", page 82.

6. Overview of Program Output Facilities

Genera's program output facilities are collectively known as Showcase. Here we divide them into basic and advanced categories, as we have the Command Processor and user input facilities. The larger category is the first, the basic facilities, which includes a variety of functions for formatted output. The advanced facilities provide incremental redisplay capabilities and functions helpful when writing you own formatted output macros.

In the following sections we consider first the basic output facilities, then the advanced facilities. This is followed by two brief notes on output streams and naming conventions for program output macros.

Reference documentation for all Showcase facilities is included in a user interface dictionary: See the section "Dictionary of Program Output Facilities", page 201.

6.1 Basic Program Output Facilities

The basic program output facilities are distributed among the following categories:

- Basic Presentation Output Facilities
- Character Environment Facilities
- Textual List Formatting Facilities
- Table Formatting Facilities
- Graph Formatting Facilities
- Graphic Output Facilities
- Progress Indicator Facilities
- Other Facilities for Program Output

6.1.1 Basic Presentation Output Facilities

Program output and input facilities are necessarily tightly coupled. In Genera, the coupling is provided by presentation types and Dynamic Windows. All output of presentation objects is potentially available as user input, mouse-sensitive in the right input context.

Basic facilities for doing output of presentations are shown below:

Basic Presentation Output Facilities
present
present-to-string
dw:with-output-as-presentation

The primary facilities provided for presentation output are **present** and **dw:with-output-as-presentation**. **present** is the basic function for outputting presentation objects. The exact form that the output takes, that is, its printed representation, is not determined by **present**, however, but rather by the presentation type of the object being presented. The definition of the presentation type includes a printer function specifying the details of the output display. The following examples show presentation of **inverted-boolean** and **character-style** objects:

```
(present t '((inverted-boolean))) ==>No
#<DISPLAYED-PRESENTATION 444312267>

(present (si:parse-character-style '(:swiss :bold :large))) ==>
SWISS.BOLD.LARGE
#<DISPLAYED-PRESENTATION 425221252>
```

You have the option of defining your own presentation type, with its own printer function, but many, like the two example types above, have already been defined for you. (For a list of predefined types: See the section "Overview of Predefined Presentation Types", page 71. Reference documentation for each listed type is included in a user interface dictionary: See the section "Dictionary of Predefined Presentation Types", page 281.)

If you wish to output an object as a presentation of a predefined type, but want to modify the printed representation of the object, the **dw:with-output-as-presentation** macro provides such a capability. It uses your code to print an object rather than the printer of the presentation type. The following function of two arguments presents the first, *this*, as an object of presentation type *that*:

```
(defun present-this-as-that (this that
        &optional (stream *standard-output*))
  (send stream :clear-history)
  (dw:with-output-as-presentation (:single-box t
        :stream stream :type that :object this)
    (send stream :draw-circle 250 200 25)
    (send stream :draw-circle 270 200 25)))
```

Try calling this function with `"ABC"` as the first argument and `'((string))` as the second. Then, do `(accept '((string)))` and click on the graphic. You will see that a perfectly normal **string** object is returned, despite its unorthodox presentation.

The third function listed in the above table, **present-to-string**, is the presentation-system equivalent of **write-to-string**. The output is done in such a way as to ensure that the output object can subsequently be input as a presentation object (via **accept-from-string**).

6.1.2 Character Environment Facilities

Facilities providing control over the appearance of characters and lines of characters are listed in the following table:

Character Environment Facilities
- **with-character-style**
- **with-character-family**
- **with-character-face**
- **with-character-size**
- **with-underlining**
- **abbreviating-output**
- **filling-output**
- **indenting-output**

The first four facilities are macros affecting character style. A character style specification includes a character family, face, and size. Macros are provided to control each of these attributes individually or all together. The final character style of the output characters is the result of merging the macro-specified style against the default style set for the output stream. (For more information on character styles: See the section "Character Styles" in *Symbolics Common Lisp: Language Concepts.*)

The following example shows the use of **with-character-style** to italicize the column headings in a table:

```
(defun table-with-italicized-heads ()
  (fresh-line)
  (formatting-table ()
    (formatting-column-headings (())
      (with-character-face (:italic)
        (formatting-cell ()
          "Number")
        (formatting-cell ()
          "Square")))
    (loop for i from 1 to 10
          as square = (* i i)
          do
      (formatting-row ()
        (formatting-cell (nil :align :center)
          (princ i))
        (formatting-cell (nil :align :right)
          (princ square))))))

(table-with-italicized-heads) ==>
Number Square
   1        1
   2        4
   3        9
   4       16
   5       25
   6       36
   7       49
   8       64
   9       81
   10     100
NIL
```

The remaining facilities are also macros. **with-underlining** adds underlines to character output. **abbreviating-output** terminates a line of characters and supplies ellipses near the right edge of the output window. **filling-output** prevents the breaking of lines in the middle of words; it inserts newlines at appropriate points. **indenting-output** lets you insert space or a string at the beginning of each new line of character output.

Here's an example using **abbreviating-output**:

```
(defun abbrev-test (width height)
  (abbreviating-output (() :width width :height height
                           :show-abbreviation t)
    (loop for row from 1 to 20 do
      (terpri)
      (loop for col from 1 to 100 do
        (format T " ~d:~d" row col)))))

(abbrev-test 42 10) ==>
 1:1 1:2 1:3 1:4 1:5 1:6 1:7 1:8 1:9 1:10 ...
 2:1 2:2 2:3 2:4 2:5 2:6 2:7 2:8 2:9 2:10 ...
 3:1 3:2 3:3 3:4 3:5 3:6 3:7 3:8 3:9 3:10 ...
 4:1 4:2 4:3 4:4 4:5 4:6 4:7 4:8 4:9 4:10 ...
 5:1 5:2 5:3 5:4 5:5 5:6 5:7 5:8 5:9 5:10 ...
 6:1 6:2 6:3 6:4 6:5 6:6 6:7 6:8 6:9 6:10 ...
 7:1 7:2 7:3 7:4 7:5 7:6 7:7 7:8 7:9 7:10 ...
 8:1 8:2 8:3 8:4 8:5 8:6 8:7 8:8 8:9 8:10 ...
 9:1 9:2 9:3 9:4 9:5 9:6 9:7 9:8 9:9 9:10 ...
...
NIL
```

6.1.3 Textual List Formatting Facilities

Textual List Formatting Facilities
format-textual-list
formatting-textual-list
formatting-textual-list-element

Among the many high-level formatting facilities provided by Showcase, those listed above are for formatting "textual" lists. A textual list is simply a list of comma-separated items, for example "1, 2, 3, and 4". You provide the items for the list, and the facilities take care of inserting the commas and the "and" before the final item.

format-textual-list is the function for creating textual lists.
formatting-textual-list is the environment-binding macro for doing the same thing. What this and similar formatting macros provide that the functions do not is flexibility. In this case, the **format-textual-list** function requires that an explicit sequence object provide the items for formatting, for example:

```
(defun simple-list-formatter ()
  (fresh-line)
  (format-textual-list '(1 2 3 4) #'princ :conjunction "and"))

(simple-list-formatter) ==>
1, 2, 3, and 4
NIL
```

formatting-textual-list, on the other hand, lets you write code to sequence through the items using whatever data structure you choose, for example:

```
(defun simple-list-formatting ()
  (fresh-line)
  (formatting-textual-list (t :conjunction "and")
    (loop for i from 1 to 4
          do
      (formatting-textual-list-element ()
        (princ "Number ")
        (princ i)))))

(simple-list-formatting) ==>
Number 1, Number 2, Number 3, and Number 4
NIL
```

As shown in the above example, **formatting-textual-list-element** controls the printing of items for display by **formatting-textual-list**.

6.1.4 Table Formatting Facilities

Table Formatting Facilities
- **formatting-multiple-columns**
- **format-sequence-as-table-rows**
- **format-item-list**
- **formatting-item-list**
- **formatting-table**
- **formatting-column**
- **formatting-column-headings**
- **formatting-row**
- **formatting-cell**
- **format-cell**

The table formatting facilities shown above allow you to output tables of arbitrary complexity. The first four listed provide relatively fast and easy tools for generating tables. **formatting-multiple-columns**, for example, displays what would otherwise be a single column of output in a multiple-column format:

```
(defun quick-table ()
  (fresh-line)
  (formatting-multiple-columns ()
    (loop for i from 0 to 79
          do
      (present i 'integer)
      (terpri)))))

(quick-table) ==>
0  4  8  12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76
1  5  9  13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77
2  6  10 14 18 22 26 30 34 38 42 46 50 54 58 62 66 70 74 78
3  7  11 15 19 23 27 31 35 39 43 47 51 55 59 63 67 71 75 79
NIL
```

(If you try this function, be aware that your display might not look like the one above; the width of the output window affects the number of columns.)

format-sequence-as-table-rows takes a sequence of elements and outputs each element on its own row. **format-item-list** and **formatting-item-list** are also used for generating tables of simple items but, through a variety of keyword options, provide much finer control over the appearance of the table than do the first two facilities.

The remaining facilities, which are used together with **formatting-table** at the top level, provide the greatest flexibility for constructing tables. The following example creates a table of network servers:

```
(defun server-table ()
  (fresh-line)
  (formatting-table ()
    (formatting-column-headings ()
      (with-character-face (:italic)
        (with-underlining ()
          (formatting-cell ()
            (write-string "Protocol"))
          (formatting-cell ()
            (write-string "Medium"))
          (formatting-cell ()
            (write-string "No. of Arguments")))))
    (loop for server in neti:*servers* do
      (formatting-row ()
        (formatting-cell ()
          (format t "~a"
                  (neti:server-protocol-name server)))
        (formatting-cell ()
          (format t "~a"
                  (neti:server-medium-type server)))
        (formatting-cell (*standard-output* :align :right)
          (format t "~a"
                  (neti:server-number-of-arguments server)))))))
```

```
(server-table) ==>
```

Protocol	*Medium*	*No. of Arguments*
MANDELBROT	BYTE-STREAM	1
UNIX-RWHO	UDP	1
IEN-116	UDP	2
TCP-FTP	BYTE-STREAM	4
TFTP	UDP	1
CHAOS-FOREIGN-INDEX	CHAOS	1
RTAPE	BYTE-STREAM	1
CONVERSE	BYTE-STREAM	2
SEND	BYTE-STREAM	1
SMTP	BYTE-STREAM	3
CHAOS-MAIL	BYTE-STREAM	1
CONFIGURATION	BYTE-STREAM	1
NFILE	BYTE-STREAM-WITH-MARK	2
QFILE	CHAOS	1
CHAOS-SCREEN-SPY	CHAOS	1
NOTIFY	CHAOS	1
CHAOS-ROUTING-TABLE	CHAOS	1
CHAOS-STATUS	CHAOS	1
ECHO-XCN-TOKEN-LIST	TRANSACTION-TOKEN-LIST	1
3600-LOGIN	BYTE-STREAM	3
SUPDUP	BYTE-STREAM	3
TELNET	BYTE-STREAM	3
TTY-LOGIN	BYTE-STREAM	3
NAMESPACE-TIMESTAMP	DATAGRAM	3
NAMESPACE	BYTE-STREAM	0
BAND-TRANSFER	BYTE-STREAM	2
WHO-AM-I	DATAGRAM	2
PRINT-DISK-LABEL	BYTE-STREAM	1
EVAL	BYTE-STREAM	1
NAME	BYTE-STREAM	2
ASCII-NAME	BYTE-STREAM	2
LISPM-FINGER	DATAGRAM	1
UPTIME-SIMPLE	DATAGRAM	2
TIME-SIMPLE-MSB	DATAGRAM	2
TIME-SIMPLE	DATAGRAM	2
RESET-TIME-SERVER	DATAGRAM	0

```
NIL
```

6.1.5 Graph Formatting Facilities

Graph Formatting Facilities
format-graph-from-root
formatting-graph
formatting-graph-node
dw:find-graph-node

Two graph formatters are provided, **format-graph-from-root** and **formatting-graph**. Both are used for creating graphs showing the connections between nodes. The **format-graph-from-root** function generates a graph from your specification of a root node and its descendants. Here, for example, is a flavor-component grapher built on **format-graph-from-root**:

```
(defun graph-flavor-components (flavor-name)
  (labels ((component-flavors (flavor-name)
             (let* ((fl (flavor:find-flavor flavor-name)))
               (remove flavor-name
                 (cond  ((flavor::flavor-components-composed fl)
                         (flavor:flavor-all-components fl))
                        (t (flavor::compose-flavor-components
                             flavor-name)))))))
    (fresh-line)
    (format-graph-from-root flavor-name
                            #'(lambda (thing stream)
                                (present thing 'flavor:flavor
                                         :stream stream))
                            #'component-flavors
                            :dont-draw-duplicates t)))
```

If you run this function on complex flavors, by the way, you will get a chance to exercise the horizontal scrolling capability of Dynamic Windows. This also illustrates the point that the graph formatters (and **formatting-table** as well) have built-in the functionality provided by **dw::with-output-truncation** to other kinds of output. That is, output generated using these macros that exceeds the width of the output window does not wrap around as character output ordinarily would. Rather, the user's view of the output is truncated by the right margin of the window, but can be obtained by horizontal scrolling.

formatting-graph works similarly to **format-graph-from-root**, but lets you specify a number of nodes and their connections, not just one node and its descendants. This allows the creation of more complex graphs than possible to create with **format-graph-from-root**. (For an example: See the macro **formatting-graph-node**, page 242.) Creating node objects within **formatting-graph** is the job of **formatting-graph-node**.

Note that neither of the graph formatting facilities can be used for generating circular graphs.

6.1.6 Graphic Output Facilities

More than a dozen facilities are included in the graphic output category, shown in the table below:

Graphic Output Facilities
graphics:draw-string
graphics:draw-point
graphics:draw-arrow
graphics:draw-line
graphics:draw-lines
graphics:draw-cubic-spline
graphics:draw-circle
graphics:draw-ellipse
graphics:draw-triangle
graphics:draw-rectangle
graphics:draw-glyph
graphics:draw-polygon
graphics:draw-regular-polygon
graphics:draw-convex-polygon
graphics:draw-pattern

These facilities are straightforward functions for drawing strings, points, arrows, lines, and a variety of closed plane figures. The following example draws a couple of arrows, one with a solid shaft, the other dashed, into the point (x and y coordinates) you call it with:

```
(defun draw-arrow (x y)
  (dw:with-own-coordinates ()
    (graphics:draw-arrow 500 500 x y)
    (graphics:draw-arrow 700 500 x y
                         :arrow-head-length 20
                         :arrow-base-width 15
                         :dashed t)))
```

Here's one that uses two graphics functions, **graphics:draw-circle** and **graphics:draw-rectangle**, inside a **dw:tracking-mouse** macro:

```
(defun follow-the-mouse ()
  (dw:tracking-mouse (*standard-output*)
    (:who-line-documentation-string ()
     (if (zerop (tv:mouse-buttons))
         "Hold any button down to draw squares"
         "Release all buttons to draw circles"))
    (:mouse-motion (x y)
     (graphics:draw-circle x y 5))
    (:mouse-motion-hold (x y)
     (graphics:draw-rectangle x y (+ x 10) (+ y 10)))))
```

If you try the above function, you must press down one of the mouse buttons and hold it down to draw squares.

The other graphics functions are used in a manner similarly direct. If you have a need for such functions in your programs, try experimenting with the various options these functions take to get a feel for the range of possibilities.

The **graphics:draw-pattern** function lets you use a bit-array of some arbitrary pattern and have it displayed once or, optionally, as a (spatially) repeated pattern within a specified area of a window. The function **tv:make-binary-gray** can be used to create the pattern, as shown by the following example:

```
(defun ones-pattern ()
  (let ((raster (tv:make-binary-gray 8 8
                 '(#b00000000     ; The picture of what you
                   #b00001000     ; want the bit pattern
                   #b00111000     ; displayed to look like, in
                   #b00001000     ; this case the number 1.
                   #b00001000     ; Notice the #b in front of
                   #b00001000     ; each number to force the
                   #b00001000     ; reader into binary.
                   #b00111110))))
    (dw:with-own-coordinates ()
      (graphics:draw-pattern 300 300 raster :right  500
                                            :bottom 500))))
```

Keep in mind that any graphic display can be made the printed representation of a presentation object with the help of **dw:with-output-as-presentation**. Consequently, graphic displays can serve as sources of mouse-sensitive input to your program. (For an example: See the section "Overview of Basic Presentation Output Facilities", page 47.)

6.1.7 Progress Indicator Facilities

Facilities in this category of basic output facilities provide a way of communicating the progress of some operation to your users:

Noting Progress Facilities
tv:noting-progress
tv:note-progress
tv:dolist-noting-progress
tv:dotimes-noting-progress

Progress is indicated by the advance of a progress bar in the lower, right corner of the screen or, alternatively, by a wide bar across the entire width of the screen. (Which is determined by the setting of the "Progress area" option in the Set Screen Options command.) Also displayed is a string naming the operation being noted.

The general-purpose facility is **tv:noting-progress**, within which the **tv:note-progress** function is used. **tv:note-progress** is the one that decides when and how much progress has occurred. This is shown in the following example:

```
(tv:noting-progress ("Working Away By Fifths")
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 1 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 2 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 3 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 4 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 5 5)
  (sleep 1))
```

tv:dolist-noting-progress and **tv:dotimes-noting-progress** implement the Common

Lisp special forms **dolist** and **dotimes** in a noting-progress environment. They take care of most simple cases.

6.1.8 Other Facilities for Program Output

Facilities for program output in the "other" category include a number of methods for Dynamic Windows:

Other Facilities for Program Output
(flavor:method :clear-window dw:dynamic-window)
(flavor:method :clear-history dw:dynamic-window)
(flavor:method :clear-region dw:dynamic-window)
(flavor:method :delete-displayed-presentation dw:dynamic-window)
(flavor:method :visible-cursorpos-limits dw:dynamic-window)
(flavor:method :set-viewport-position dw:dynamic-window)
(flavor:method :y-scroll-position dw:dynamic-window)
(flavor:method :y-scroll-to dw:dynamic-window)
(flavor:method :x-scroll-position dw:dynamic-window)
(flavor:method :x-scroll-to dw:dynamic-window)
(flavor:method :with-output-recording-disabled dw:dynamic-window)
dw:with-own-coordinates
dw:tracking-mouse
dw::with-output-truncation
surrounding-output-with-border
dw:displayed-presentation-set-highlighting
dw:displayed-presentation-clear-highlighting

The methods listed equip you with the ability to control where in the output history of a Dynamic Window your output will appear. This is especially important in the case of dynamic as opposed to static windows, because it is often impossible to know precisely where the visible portion of the window is at the time any given output is displayed.

One way of handling this situation is with the **:visible-cursorpos-limits** method, as illustrated by the following example:

```
(defun graphic-output-to-dynamic-window-1 ()
  (let ((width 100) (height 50) (start-x 200) (start-y 150))
    (multiple-value-bind (x1 y1 x2 y2)
      (send *standard-output* :visible-cursorpos-limits)
      (send *standard-output* :draw-rectangle width height
                              (+ x1 start-x) (+ y1 start-y)))))
```

In this example, we are asking the window the coordinates of the current viewport, and using these as offsets to adjust where we send output.

The **dw:with-own-coordinates** macro has a similar purpose. That is, it allows you to avoid using absolute coordinates, and to use coordinates relative to the current viewport instead:

```
(defun graphic-output-to-dynamic-window-2 ()
  (let ((width 100) (height 50) (start-x 200) (start-y 150))
    (dw:with-own-coordinates ()
      (send *standard-output* :draw-rectangle width height
                                              start-x start-y))))
```

Another capability of **dw:with-own-coordinates** is the disabling of output recording. That is, through a keyword option to this macro, you can prevent output from being recorded in the output history of the window to which it is sent. (This capability is also provided by **(flavor:method :with-output-recording-disabled dw:dynamic-window).**) It defeats one of the main advantages of Dynamic Windows, but is sometimes useful, particularly when doing graphic output. Try calling the following example with **t**, to enable output recording, then **nil**, to disable it:

```
(defun moving-arrow (t-or-nil)
  (dw:with-own-coordinates (t :enable-output-recording t-or-nil)
    (do ((x 100 (+ x 4))
         (y 100 (+ y 2)))
        ((> x 500) 'done)
      (graphics:draw-arrow 100 100 x y :alu :flip
                           :arrow-base-width 20
                           :arrow-head-length 35)
      (graphics:draw-arrow 100 100 x y :alu :flip
                           :arrow-base-width 20
                           :arrow-head-length 35))
    (graphics:draw-arrow 100 100 500 300 :alu :flip
                         :arrow-base-width 20
                         :arrow-head-length 35)))
```

First note the speed with which the arrows are drawn. Now try scrolling backwards and forwards over the output and observe the effects.

This brings up another point concerning graphic presentations and backwards scrolling. Because they, like other presentations, are stored in the window's history by position, not with a timestamp, they are not redrawn in the order in which they were drawn originally. For example, if you use a **graphics:draw-rectangle** function to clear out a part of the window, then draw over that part, the **graphics:draw-rectangle** can happen *after* the later drawing when you scroll back to that portion, with the effect that parts of the graphics are missing from the re-presented display.

If this is a potential problem in your program, then make all of your overlapping

graphics part of a superior presentation (using **dw:with-output-as-presentation**), which then holds their collective place in the history. Inferiors are carefully redrawn in the order in which they were output originally.

Three other macros are included in this category. **dw:tracking-mouse** lets you track the current position of the mouse cursor, useful in graphic output applications, and other mouse events as well. In conjunction with the mouse handler facilities, it provides the primary interface between your programs and the mouse process. An example showing its use in a drawing function is presented elsewhere: See the section "Overview of Graphic Output Facilities", page 57.

dw::with-output-truncation is necessary for taking advantage of the horizontal scrolling capability of Dynamic Windows. With it you can prevent the wrapping of character output and let the user's view of the output be truncated by the right (or bottom) margin of the window. The truncated output is accessible through scrolling. This also applies to graphic presentations that would otherwise be too big if limited to the size of a window. (Note that **formatting-table** and the two graph formatters, **formatting-graph** and **format-graph-from-root**, include this kind of functionality as a built-in feature.)

For a simple demonstration of output truncation, try calling the following function with **t** and **nil**:

```
defun truncation-test (t-or-nil)
  (dw::with-output-truncation (t :horizontal t-or-nil)
    (loop repeat 100 do (write-char #\a))))
```

surrounding-output-with-border lets you enclose any other kind of output – tables, graphics, whatever – in a rectangular, oval, circular, or diamond-shaped border. To see the different shapes, try calling the following function with `:rectangle` , `:oval`, `:circle`, or `:diamond`:

```
(defun shape-test (shape)
  (fresh-line)
  (surrounding-output-with-border
    (*standard-output* :shape shape)
    (present tv:selected-window 'tv:window)))
```

The final two functions listed, **dw:displayed-presentation-set-highlighting** and **dw:displayed-presentation-clear-highlighting**, as their names suggest, let you highlight and clear the highlighting of displayed presentations. This highlighting, unrelated to mouse sensitivity, is done by either underlining the presentation or putting it into reverse video.

6.2 Advanced Program Output Facilities

The advanced Showcase facilities for program output are divided into three areas:

- Advanced Presentation Output Facilities
- Redisplay Facilities
- Facilities for Writing Formatted Output Macros

6.2.1 Advanced Presentation Output Facilities

Several macros, listed below, are provided for doing advanced presentation output:

Advanced Presentation Output Facilities
dw:with-output-to-presentation-recording-string
dw:with-replayable-output
dw:with-resortable-output

The first, **dw:with-output-to-presentation-recording-string**, is the presentation-system equivalent of the Common Lisp macro **with-output-to-string**. It works similarly, but you can subsequently output the string as a presentation, not just a string.

dw:with-replayable-output and **dw:with-resortable-output** are closely related, the latter being a special case of the former. **dw:with-replayable-output** lets you present all of the output generated in the body of the macro as a single presentation. This presentation is "replayable"; that is, it can be input as a whole, internally re-arranged in some fashion, and presented again as the same object.

dw:with-resortable-output takes a sequence and a sorting predicate, and constructs a **dw:with-replayable-output** macro to implement the sorting function. Users can click on the presented sequence and have it redisplayed in a different order.

To see an example of this, run the Show Processes command in a Lisp Listener. With the mouse cursor somewhere over the output display, press the SUPER and SHIFT keys and notice that the entire display is enclosed in a single box, indicating that it is a single presentation.

The top mouse documentation line informs you that the Edit viewspecs handler is available on s-sh-Mouse-M. Invoke the handler and a menu appears indicating what your choices are for sorting the displayed processes. Select one and watch the resort.

The following example is a function for presenting a resortable display of network servers. It is implemented similarly to the Show Processes command.

```
(defun format-servers (&optional (sort-by :none))
  (fresh-line)
  (dw:with-resortable-output
    ((servers sort-by :copy-of neti:*servers*)
     (:none #'ignore)
     (:protocol (lambda (s-1 s-2)
                  (string<
                    (string (neti:server-protocol-name s-1))
                    (string (neti:server-protocol-name s-2)))))
     (:medium (lambda (s-1 s-2)
                (string<
                  (string (neti:server-medium-type s-1))
                  (string (neti:server-medium-type s-2)))))
     (:arguments (lambda (s-1 s-2)
                   (< (neti:server-number-of-arguments s-1)
                      (neti:server-number-of-arguments s-2)))))
    ()
    (formatting-table ()
      (formatting-column-headings ()
        (with-character-face (:italic)
          (with-underlining ()
            (formatting-cell ()
              (write-string "protocol"))
            (formatting-cell ()
              (write-string "medium"))
            (formatting-cell ()
              (write-string "no. of arguments")))))
      (loop for server in servers do
        (formatting-row ()
          (formatting-cell ()
            (format t "~a"
                    (neti:server-protocol-name server)))
          (formatting-cell ()
            (format t "~a"
                    (neti:server-medium-type server)))
          (formatting-cell (*standard-output* :align :right)
            (format t "~a"
                    (neti:server-number-of-arguments server)))))))))
```

6.2.2 Redisplay Facilities

A set of inter-related facilities is provided for creating redisplayable output and doing incremental redisplay. The facilities are listed in the following table:

Redisplay Facilities
dw:redisplayable-present
dw:redisplayable-format
dw:independently-redisplayable-format
dw:with-redisplayable-output
dw:redisplayer
dw:do-redisplay

Redisplayable output is similar to ordinary output in the actual display; it differs in that, in addition to being output to a window for display, the output value is also stored in an *output cache* uniquely identified with that display. When the window is redisplayed, the new output value is first compared to the cached value and, if different, the cache is updated with the new value for display. This has efficiency advantages compared with non-cached output.

Incremental redisplay refers to the redisplay of individual pieces of the output to a window, rather than redisplaying the window as a whole. It works in the manner described above, except that each redisplayed piece of the window output is associated with its own output cache.

The first four facilities listed in the redisplay category are for creating redisplayable output. **dw:redisplayable-present** is used like **present** but creates a redisplayable presentation. Similarly, **dw:redisplayable-format** works as **format** does, but generates redisplayable output. **dw:independently-redisplayable-format** is like the previous function, except that each argument in the format-control string gets cached separately; hence its usefulness for incremental redisplay. Finally, the macro **dw:with-redisplayable-output** lets you make any output-producing code produce redisplayable output.

How you do redisplay once you have functions producing redisplayable output depends on whether you are taking advantage of **dw:define-program-framework**. If you are, then making a program pane use your redisplay function is simply a matter of supplying that function via the **:redisplay-function** keyword. If, additionally, incremental redisplay is what you want, then specify so with the **:incremental-redisplay** keyword.

If you are doing redisplay outside of **dw:define-program-framework**, then you need to create a redisplay object that you can pass to **dw:do-redisplay**, which, as its name says, does the redisplay. Creating a redisplay object is the job of **dw:redisplayer**; use this macro to enclose your redisplay function.

For examples showing the coordinated use of these facilities for incremental redisplay, see the file sys:examples;incremental-redisplay.lisp.

6.2.3 Facilities for Writing Formatted Output Macros

Facilities for Writing Formatted Output Macros
dw:continuation-output-size
dw:named-value-snapshot-continuation

The two facilities listed above help you write your own output formatting macros. Given a continuation (usually a closure) and a stream, **dw:continuation-output-size** tells you how much room, in spaces or pixels, the continuation will require on the stream. This is useful, for example, for making windows no larger than necessary to accommodate formatted displays. The reference documentation for this facility includes an example: See the function **dw:continuation-output-size**, page 209.

dw:named-value-snapshot-continuation is a macro that makes separate bindings for free variables referenced in its body; that is, it "snapshots" the free variables at the time the closure is constructed. This provides lexical separation between variables in the inner loops of a formatting function and variables with the same names in the outer loops. The reference documentation for **dw:named-value-snapshot-continuation** contains additional details on when and how to use this facility, including examples: See the macro **dw:named-value-snapshot-continuation**, page 252.

6.3 Output Streams for Program Output Facilities

The default stream for output is ***standard-output***, not ***terminal-io***. Avoid using the program output facilities with interactive streams like ***terminal-io*** and ***query-io***. Such streams should never be bound to output-only streams.

6.4 Naming Conventions for Program Output Macros

The naming of macros for program output has followed certain conventions. Facility names prefixed with "with-" are macros that bind the environment but do not directly participate in generating output. They establish a local environment for output. Code in the bodies of such facilities is responsible for creating the output. After output is completed, the local environment goes away.

A good example is **with-character-style**. Code in the body of the macro has the

job of generating characters. The macro ensures that they are output in the specified style. When the macro is finished executing, the default character style for the output stream used remains the same as before the macro was invoked.

Facility names in which the first word ends in "ing" are also macros that bind the local environment and let it go again when output is completed. In addition to this, however, they make a significant contribution to the output display, generally adding to whatever is generated in their bodies. **surrounding-output-with-border**, for example, makes an obvious contribution to the display.

7. Presentation Substrate Facilities

This section reviews presentation types and the facilities provided for their creation and manipulation. First we present some basic concepts – what a presentation is, what a presentation type is. Then we provide an overview of the predefined and documented presentation types available for use in your programs. This is followed by sections discussing the various groups of presentation substrate facilities other than the predefined types; there are four:

- Presentation-Type Definition Facilities
- Presentation Input Context Facilities
- Presentation Input Blip Facilities
- Other Presentation Facilities

Two concluding sections provides information and examples useful when defining your own presentation types. The first covers writing a presentation-type parser. The second discusses the implications of defining presentation types based on already defined data types, flavors, and structures. This discussion presents some advanced concepts. It will probably be most meaningful to you after you have had some experience with defining your own mouse handlers and presentation types. For related information on handlers: See the section "Advanced Mouse Handler Concepts", page 42.

Reference documentation for the Presentation Substrate Facilities is included in two dictionaries. The first covers the predefined presentation types: See the section "Dictionary of Predefined Presentation Types", page 281. The second covers all other presentation substrate facilities: See the section "Dictionary of Presentation Substrate Facilities", page 347.

To conclude these prefatory remarks, we call your attention to two system facilities for acquiring information about particular presentation types and presentations. These are the Show Presentation Type command and Presentation Inspector, respectively. The first is a Command Processor command that displays the argument list, supertypes, and subtypes of a specified type. The Presentation Inspector is an option on the Presentation debugging menu, itself an option on the standard click-right menu available for all displayed presentations. It allows you to explore the presentation you call it on from a variety of aspects.

7.1 Basic Presentation System Concepts

What is a presentation?

Conceptually, a presentation is the user-visible aspect of an object inside the machine. This encompasses both the way the program depicts the object for the user, and the gestures the user performs to depict the object to the program. As such, it forms the basis for the interface between a program and the user.

There can be many different ways for the program to depict an object to the user. The choice of how to present the object is determined not just by the type of object, but also by the semantics, or meaning, associated with the object by the programmer, the program, and the user. For example, a number might be an entry in a table describing last year's financial results in an accounting system, a pattern of bits to a numerologist, a slider controlling vibrato in a synthesizer voice editing program, or a color intensity control to the user of a color painting program. Some of these are values input, some are values output, and some are interactive.

A presentation is a particular *instance* of *presenting* an object to the user with a *presentation type.*

What is a presentation type?

A presentation type is what differentiates one presentation of an object from another. The type of presentation is specified by the implementor of each system, corresponding to the meaning assigned to each object in that system. In an accounting program, this may mean a dollar amount, and be read with exactly two digits after the decimal point. In a numerologist's program, it is just a number represented in binary; it can be displayed in hexadecimal, octal, or binary. If both systems are on the screen at once, the numbers from the numerology program do not belong in the accounting program, even though they both use numbers: the numerologist was not working with dollar amounts.

So, a presentation type is what distinguishes the different uses of a Lisp type – whether a number, a list, or a flavor instance – for the external user. Not only does this differentiation appear in the syntax used to express it, but it allows SemantiCue to make appropriate quantities available for selection with the mouse (via mouse sensitivity), without also making inappropriate ones available.

The parts of a presentation type

A presentation type is made up of three parts:

- The *name*

  The name is what identifies how presented objects of this type are printed, how they are read, and how this type relates to other types. This name may come predefined by the system, or be defined by the user.

- The *data arguments*

  The data arguments further distinguish which objects are being described by the type. They always denote a subset of the objects that would be denoted without the data arguments. For example, when asking for an integer, you can ask for an integer in a certain range by giving data arguments to the **integer** presentation type.

- The *presentation arguments*

  The presentation arguments do not distinguish between objects. Instead, they control how the objects are presented to or accepted from the user. For example, a presentation argument to the **integer** presentation type specifies the base in which an integer should be printed or read.

The syntax for distinguishing the different parts of a presentation type and examples are presented in the section on predefined presentation types: See the section "Overview of Predefined Presentation Types", page 71.

7.2 Predefined Presentation Types

Presentation types form the basis of the typing system for user input and program output. A large number of predefined presentation types exist; only a relatively few are used for program I/O. This is because every structure, flavor, and Common Lisp data type is also a presentation type. Most, however, are of little use in end-user-oriented application programs. Consider, for example, the Common Lisp types **hash-table** and **compiled-function**; you would not generally encounter these in end-user-visible places.

In this section, we list what we regard as the types most likely to be used by application programmers. Some, like **integer**, **string**, and **boolean**, are encountered frequently in all kinds of programs. Many others, like **sys:code-fragment** and **net:network**, are more specialized in their uses.

In any case, all of the types included here are also documented as individual entries in the Dictionary of Predefined Presentation Types. Also, many of them are defined in the file `sys:dynamic-windows;standard-presentation-types.lisp`, where you can look for models when defining your own types. The dictionary entry for each type notes whether it is one of those included in this file.

The documented types are divided into three groups:

1. Common Lisp Presentation Types
2. Symbolics Common Lisp Presentation Types
3. Other Presentation Types

Of course, the Common Lisp types form a subset of the Symbolics Common Lisp types, but for the purposes of the present discussion, we separated them out. The Other Presentation Types include the potentially useful types exported from packages other than Symbolics Common Lisp; most of them are in the specialized-use category.

The following table lists the useful Common Lisp presentation types:

Common Lisp Presentation Types
- **and**
- **character**
- **integer**
- **keyword**
- **member**
- **not**
- **null**
- **number**
- **or**
- **package**
- **pathname**
- **satisfies**
- **sequence**
- **string**
- **symbol**
- **symbol-name**
- **t**

Most of these Common Lisp types should be familiar as data types. As presentation types, they require some additional remarks. The first concerns syntax: there is a formal way to specify a presentation type and a shorthand way. The formal way is with a double set of parentheses as in the following **accept** function:

```
(accept '((integer))) ==>
Enter an integer: 14
14
((INTEGER))
```

The shorthand way is without the parentheses:

```
(accept 'integer) ==>
Enter an integer [default 14]: 14
14
INTEGER
```

In general, we have tried in the documentation to stick with the formal syntax, although you will encounter examples here and there that skip the parentheses.

The reason for the double set of parentheses is the second point we need to make. Presentation types can take arguments. There are two kinds of arguments to presentation types, *data arguments* and *presentation arguments*. Data arguments qualify the data type of a presentation object. They affect the subtype and supertype relationships of a type within a type family. Let's consider another example using the **integer** presentation type:

```
(accept '((integer 0 100))) ==>
Enter an integer greater than or equal to 0
and less than or equal to 100: 14
14
((INTEGER 0 100))
```

The **integer** type takes optional data arguments for specifying lower and upper range limits. The `14` returned by this **accept** is not an object of data type **integer**; it's of an **integer** subtype, those integers between 0 and 100.

Presentation arguments are always keywords. Unlike data arguments, they do not affect the position of an object's type in the type hierarchy; that is, they have no impact on the internal representation of the object. Rather, they affect the appearance of the object, or some other aspect of its presentation unrelated to data type. Consider the **integer** type once again. You can change the printed representation of an integer by changing its base:

```
(accept '((integer 0 100) :base 16)) ==>
Enter a hexadecimal integer greater than or equal to 0
and less than or equal to 64: e
14
((INTEGER 0 100) :BASE 16)
```

:base is a presentation argument. Internally a `14` is still returned, but externally it is displayed as an `e`.

A third kind of argument to presentation types exists, called a *meta-presentation argument.*

Meta-presentation arguments to presentation types are arguments that are directly understood by **accept** or **present**. They are not dependent on the parser or printer of any particular presentation type, and can therefore be used as arguments to any type.

At present, a single meta-presentation argument is available, **:description.** Using this keyword option, you can control the prompt created by **accept** for soliciting

input of a given type. This allows you to make the prompt more appropriate to the current conceptual context. For example, instead of just asking for a integer, you could do something like:

```
(accept '((integer) :description "the number of copies")) ==>
Enter the number of copies: 5
5
((INTEGER) :DESCRIPTION "the number of copies")
```

So, the parentheses are intended to distinguish unequivocally between data arguments and presentation arguments. You have to use them when providing any kind of argument to a presentation type, but can skip them with naked types.

Of the other listed Common Lisp types, note that **t** is the supertype of all other types. Also note the compound types **and** and **or**; they provide a way of extending the type system by combining types. For example, suppose we wanted to accept only odd integers. A compound type using **and** would do the job:

```
(accept '((and ((integer)) ((satisfies oddp))))) ==>
Enter an integer satisfying ODDP: 53
53
((AND ((INTEGER))
      ((SATISFIES ODDP))))
```

satisfies is another Common Lisp type, used only within compound types based on **and**.

Several extensions via the **or** type are already among the predefined Symbolics Common Lisp types, listed below.

Symbolics Common Lisp Presentation Types
- **alist-member**
- **boolean**
- **character-face-or-style**
- **character-style**
- **character-style-for-device**
- **instance**
- **inverted-boolean**
- **null-or-type**
- **sequence-enumerated**
- **subset**
- **token-or-type**
- **type-or-string**

The three compound types mentioned above are **null-or-type**, **token-or-type**, and **type-or-string**. **sequence-enumerated** is also a compound type, one for accepting

or presenting a sequence of objects, each one of a specified presentation type. Using **alist-member** to accept an object is similar to using a menu; the object is represented by a user-visible string different from its internal representation. **subset** provides a way of accepting or presenting one or more objects in a set.

The **instance** presentation type is typical of the many Common Lisp types like **hash-table** and **compiled-function**, mentioned above, unlikely to be useful in many situations. For one thing, you cannot type the name of an instance at an **accept** function; it either has to be entered via a mouse click on a previously presented **instance** object, or through **accept**'s default mechanism. It is documented as a dictionary entry merely as an example of such types. You should not ordinarily need it.

The remaining presentation types, listed below, provide potentially useful I/O capabilities spread across a broad spectrum of system software and functionality. We encourage you to study this list, and the corresponding dictionary entries, for types of use in your applications. Only two of these types will be discussed further here, **sys:expression** and **dw:no-type**.

Other Presentation Types
- **dw:member-sequence**
- **dw:no-type**
- **dw:out-of-band-character**
- **dw:raw-text**
- **fs:directory-pathname**
- **fs:wildcard-pathname**
- **net:host**
- **net:local-host**
- **net:namespace**
- **net:namespace-class**
- **net:network**
- **neti:local-network**
- **net:object**
- **sys:printer**
- **neti:protocol-name**
- **neti:site**
- **net:user**
- **sct:system**
- **sct:system-version**
- **sys:code-fragment**
- **sys:expression**
- **sys:font**
- **sys:form**
- **sys:flavor-name**
- **sys:function-spec**

sys:generic-function-name
sys:stack-frame
time:time-interval
time:time-interval-60ths
time:timezone
time:universal-time
tv:window
zwei:buffer

sys:expression plays a key role in the presentation type system. We mentioned earlier that the number of presentation types is large, including all structures, flavors, and a variety of little-used (for program I/O) Common Lisp types. The **sys:expression** type is the link between these types and the presentation system. It is a supertype of all Common Lisp types (except **t**), and is the type from which they inherit their printer and parser functions when these are not otherwise defined for them. For example, the **instance** type inherits from **sys:expression** and, through **instance**, so do all flavors. (The undocumented **structure** presentation type plays an analogous role for all structures.)

sys:expression provides these types with a type history as well. In fact, some of the Common Lisp presentation types listed in an earlier table also make use of **sys:expression**'s type history. This is true of the **integer** presentation type. Through the **number** presentation type, to which it and all other numeric types are subtype, it has access to the history of **sys:expression** objects previously accepted.

The expression history is the source of default values offered when types inheriting this history are accepting objects. When used by **integer**, the expression history is "pruned" of non-integer objects; an appropriate default value can thereby be offered. Other types with access to the expression history benefit from a similar pruning process.

dw:no-type, as its name might suggest, is not really a presentation type. Rather, it is a bogus type for use by mouse handlers that are intended to be active only over blank areas of a window, not over presentations.

All presentation types listed in the tables in this section are documented in a separate dictionary: See the section "Dictionary of Predefined Presentation Types", page 281.

7.3 Presentation-Type Definition Facilities

The Presentation-Type Definition Facilities include the functions you need to define new presentation types. These are listed in the following table:

Presentation-Type Definition Facilities
- **define-presentation-type**
- **dw:read-char-for-accept**
- **dw:peek-char-for-accept**
- **dw:unread-char-for-accept**
- **dw:compare-char-for-accept**
- **dw:read-standard-token**
- **dw:with-accept-activation-chars**
- **dw:with-accept-blip-chars**
- **dw:with-accept-help**
- **dw:with-accept-help-if**
- **dw:completing-from-suggestions**
- **dw:suggest**
- **dw:complete-input**
- **dw:complete-from-sequence**

The primary facility in this category is **define-presentation-type**. It is this macro that establishes how a given type is parsed by **accept** and printed by **present**. In particular, the **define-presentation-type** macro for each type specifies a parsing and printing function for that type. The functions can either be written especially for the type or inherited from some other type. In either case, when **accept** or **present** is called, the respective parser or printer for the presentation type in question is used to input or output the object.

Writing presentation-type parsers is, in general, a more involved process than writing printers. All of the remaining facilities listed after **define-presentation-type** in the above table are for use in parser functions. As with presentation types themselves, they are intended for parsing input from Dynamic Windows. Thus, for example, **dw:read-char-for-accept** and **dw:peek-char-for-accept** are the Dynamic Window equivalents, respectively, of **zl:tyi** and **zl:tyipeek** for input from streams. (For information on the latter functions: See the section "Input Functions That Work on Streams" in *Reference Guide to Streams, Files, and I/O.*)

Beyond functions for input per se, other facilities in the Presentation-Type Definition subcategory let you provide *help* and *completion* services for the presentation types you define. Help messages are available to users during program input when they press the HELP key. Using **dw:with-accept-help** and **dw:with-accept-help-if**, you can augment the help displays for your presentation types. Similarly, with the listed completion facilities, you can customize the completion services available when users input objects to your programs.

For more information on writing parsers for presentation types, including examples: See the section "Writing a Presentation Type Parser", page 80. For more examples, see the file `sys:dynamic-windows;standard-presentation-types.lisp`.

7.4 Presentation Input Context Facilities

Facilities for manipulating presentation input contexts are listed below:

Presentation Input Context Facilities
dw:with-presentation-input-context
dw:clear-presentation-input-context
dw:presentation-input-context-option
dw:with-presentation-input-editor-context
dw:*presentation-input-context*

The primary facility in this subcategory of presentation substrate tools is the first listed, **dw:with-presentation-input-context**. This macro can be used to establish an input context just as **accept** establishes a context. In a sense, its relationship to **accept** is analogous to that of **dw:with-output-as-presentation** to **present**. (See the section "Overview of Basic Program Output Facilities", page 47.) It just provides the input context; you have to do your own input/parsing. The other facilities in this group provide additional help in manipulating the input context.

7.5 Presentation Input Blip Facilities

Presentation Input Blip Facilities
dw:echo-presentation-blip
dw:presentation-blip-object
dw:presentation-blip-options
dw:presentation-blip-presentation-type
dw::presentation-blip-mouse-char
dw:presentation-blip-typep
dw:presentation-blip-p
dw:presentation-blip-case
dw:presentation-blip-ecase

A presentation input blip is created by a translating mouse handler when a user clicks on a displayed presentation with the gesture appropriate for that handler. Conceptually, the blip represents how the user clicked on a sensitive presentation: it encodes the object, its presentation type, and the gesture used.

Do not confuse presentation blips with ordinary mouse blips. The former are generated by translating handlers in presentation input contexts established by **accept** or **dw:with-presentation-input-context**. Mouse blips, on the other hand,

are generated by clicking the mouse in non-presentation input contexts, for example, that established by `(send *terminal-io* :any-tyi)`. Do not mix presentation and non-presentation input contexts in your applications. (For more information on mouse blips: See the section "Mouse Blips" in *Programming the User Interface, Volume B.*)

The Presentation Input Blip Facilities are used within the blip clauses of a **dw:with-presentation-input-context** macro to manipulate input blips. The functions in this subcategory extract certain fields of the blip or test them in some way.

7.6 Other Presentation Facilities

Other Presentation Facilities
- **dw:presentation-type-p**
- **dw:presentation-subtypep**
- **dw:presentation-object**
- **dw:presentation-type**
- **dw:presentation-equal**
- **dw:describe-presentation-type**
- **dw:check-presentation-type-argument**
- **dw:with-presentation-type-arguments**
- **dw:with-type-decoded**
- **dw:presentation-type-name**
- **dw:presentation-type-default**

The Other Presentation Type facilities include a miscellany of useful functions. Perhaps the most important among these is **dw:presentation-subtypep**. This function tests to see if one type can be regarded as a subtype of another. Subtype considerations are key for determining the availability of presentation objects for input in a given context, and the applicability of mouse handlers to such objects.

In general, when the input context is for a supertype, all subtypes to that supertype are acceptable as input. Similarly, if a mouse handler is defined for the supertype, it is also active for all the subtypes. In both cases, the reverse is not true; that is, when a subtype is specified, a supertype is not acceptable.

In concrete terms, when you are accepting a **number**, any kind of number – **integer**, **ratio**, etc. – will do; when you are looking for an **integer**, any kind of integer will do, but not any kind of number. **dw:presentation-subtypep** and equivalent internal functions are the basis of such determinations.

The remaining facilities in this subcategory are for taking apart presentations and manipulating presentation-type arguments.

7.7 Writing a Presentation Type Parser

The parser for a presentation type has the responsibility for reading characters typed by the user, parsing the input, and returning an object of the proper type. Consequently, the parser defines the syntax of the presentation type.

The parser function is called with one positional argument, the stream on which to do I/O. It is also passed several keyword arguments, which the programmer can choose to use via &key in the parser's argument list.

The most important rule for writing a parser is that you must use the passed-in stream for all I/O operations in the body of the parser. Failure to follow this rule will cause your presentation type to malfunction in command lines, **dw:accepting-values** menus, and other contexts.

The following is a simple presentation type definition:

```
(define-presentation-type system-processes ()
   :parser ((stream)
            (dw:complete-from-sequence si:all-processes stream
                                       :name-key 'si:process-name
                                       :partial-completers '(#\sp)))
   :printer ((object stream)
             (princ object stream)))
```

This presentation type is a version of the **si:process** presentation type used by the standard system. It takes advantage of one of the input completion utilities, **dw:complete-from-sequence**. The caller provides **dw:complete-from-sequence** with a list of objects that form the possibilities set, and **dw:complete-from-sequence** takes care of reading input and returning the process object associated with the name that the user types.

The **:partial-completers** option to **dw:complete-from-sequence** causes individual words of process names to complete when the user types SPACE. This option is usually supplied for any completion set that can be broken down into space-separated tokens (for example, command names, window names, names of people).

For completing from a set of possible inputs when it is inconvenient to actually produce a list or vector of the possibilities, a parser uses **dw:completing-from-suggestions**, which defines a lexical function called **dw:suggest**, called to generate the set. To get control over the actual completion, that is, the mapping from substrings to possible inputs, use the **dw:complete-input** function.

The parser for a presentation type reads user input and defines a mapping between the characters typed and objects of that presentation type. In other words, the parser looks at the user's input and determines whether or not it can interpret it as an object of the appropriate type.

Such determinations can be made in a couple of ways. One way is by native syntactic evaluation. For example, when reading a number, make sure all the characters are digits and then multiply and add them to get the result. Another is by associative lookup, for example, names of objects in a list. If an object can be found that matches the input, the parser returns it. If not, the parser signals a special kind of error, called a parse-error.

A parse error should normally be a flavor of error built on **sys:parse-error**. The two most convenient error flavors for use in presentation type parsers are **dw:input-not-of-required-type**, which takes a presentation type and unparsable token, and **zl:parse-ferror**, which takes an arbitrary format string and arguments.

A parser must follow a certain discipline when reading keyboard input. This discipline includes using special functions to read all input. The most basic of these functions is **dw:read-standard-token**. It reads characters from the input stream and returns a string containing those characters. A simple table-lookup parser can be written using **dw:read-standard-token** as shown in the following example:

```
(define-presentation-type color ()
   :parser ((stream)
            (let* ((input (dw:read-standard-token stream))
                   (color (assoc input '(("blue" (0 0 1.0))
                                         ("red" (1.0 0 0))
                                         ("magenta" (1.0 0 1.0)))
                                 :test #'string-equal)))
              (if (null color)
                  (signal 'dw:input-not-of-required-type
                          :type 'color
                          :string input)
                  (second color)))))
```

The call to **dw:read-standard-token** reads all characters typed by the user until a delimiter is typed. When the string is returned, the parser looks it up in a table and returns the appropriate RGB color values. If no entry matching the input is found, the **dw:input-not-of-required-type** error is signalled.

The delimiters for a particular call to **dw:read-standard-token** are specified via the **dw:with-accept-blip-chars** and **dw:with-accept-activation-chars** macros. The "blip" characters are simply the set of characters that cause **dw:read-standard-token** to return when one of them is typed by the user at the end of the input line. It is a visible graphic character, like a space or period, and can be edited into and out of an input line.

The string returned by **dw:read-standard-token** contains all the characters typed up to, but not including, the terminating character. By default, there are no blip

characters, so in a simple call to **accept** with the **color** presentation type above, **dw:read-standard-token** does not return until the user types an activation character.

Activation characters are usually not graphic characters. They are not edited in the command line, but cause the command parser to return as soon as they are typed. By default, the available activation characters are RETURN or END.

A slightly more complicated example that involves reading tokens and individual characters follows. Characters are read by the **dw:read-char-for-accept** function and its companions, **dw:unread-char-for-accept** and **dw:peek-char-for-accept.** The "characters" returned by these functions can also be data structures whose exact contents are only meaningful to the input functions themselves. Therefore, all comparisons of characters read with these functions must be done with the **dw:compare-char-for-accept** function.

```
(define-presentation-type simple-character-style ()
   :parser ((stream)
            (let ((family (accept '((member fix swiss dutch))
                                  :prompt nil
                                  :stream stream)))
              (let ((delim (dw:read-char-for-accept stream)))
                (unless (dw:compare-char-for-accept delim #\.)
                  (sys:parse-ferror
                    "Character style components must be delimited with a period"))))))
```

Note that the recursive call to another presentation type parser, in this case **member**, left the delimiter that terminated its parse, the period, in the stream. A parser is responsible for reading intermediate delimiters and returning successfully with the final delimiter still in the stream.

In the case of a blip character delimiter, the delimiter will presumably be looked at by the caller of this presentation type, to make up a complex parse. For instance, the periods that separate the fields of a character style, or the spaces that separate the arguments to a command processor command.

In the case of an activation character, the delimiter will cause the enclosing calls to parsers to themselves activate. Thus, a return typed to terminate a command line percolates up through the parsers for the individual fields that make up the command until it causes the actual command parser to return.

7.8 User-Defined Data Types as Presentation Types

Whenever you use **defstruct**, **defflavor**, or **deftype**, you are creating a new data type. Often, when you have defined such a data type, you wish to present and

accept objects of the new type in you programs. Should you present them with the presentation type being the same as the data type, or invent a separate presentation type, instead? When addressing this issue, you need to consider mouse handlers both *from* the data type in question, and *to* the data type. It is important to remember that all data types are subtypes of **sys:expression**.

These issues also arise when defining presentation types in terms of data-types, for example, defining **odd-integer** to be an **:abbreviation-for** `((and integer ((satisfies oddp))))`.

Let's consider handlers *from* the new type first. If you define a handler from your type, this handler will apply to instances of your object that you **present**, or that are printed out by the debugger while you are debugging your program. This is as it should be. However, the handler might apply to other presentations as well. In the case of **odd-integer**, for example, whenever you move the mouse over an odd integer – whether it was explicitly presented as type **odd-integer** or not – the handler will apply. You may approve of this behavior, but if not, then do not define **odd-integer** in terms of **integer**; that way, the handler is only available when explicitly asked for.

In the case of flavors and structures, if you wish to restrict the applicability of handlers from your type to only those objects presented explicitly as this type, then define the presentation type with a name different from the flavor or structure.

Similar considerations apply to handlers *to* your type. If you define a translator with a *to-presentation-type* that is also a data type, remember that it will apply in the **sys:expression** input context, because such an object is also a type of expression. In particular, it will apply at the Lisp top level. Again, this may or may not be appropriate from your point of view. If not, the solutions are the same.

The following example should make both of these situations clearer. (The translator in this example is intentionally slow.)

```
(defstruct family-tree
  father
  mother)

(fs:define-canonical-type :family-tree "FAMILY-TREE"
  ((:unix :unix42 :vms :vms4) "FAT"))

(defun family-tree-file-p (pathname)
  (eql (send pathname :canonical-type) :family-tree))

(define-presentation-type family-tree-pathname ()
  :expander '((and pathname ((satisfies family-tree-file-p)))))
```

```
(define-presentation-translator pathname-to-family-tree
        (family-tree-pathname ; from type
         family-tree          ; to type
        :gesture :select)
        (pathname)
  (with-open-file (file pathname
                   :direction :input
                   :element-type 'characters)
    (read file)))

(cp:define-command (com-show-family-tree-directory
                    :command-table "GLOBAL") ()
  (loop for x in (directory (send (fs:user-homedir) :new-pathname
                                  :name :wild
                                  :type :family-tree
                                  :version :newest))
        do (present x 'family-tree-pathname)))

(cp:define-command (com-count-family-tree-generations
                    :command-table "GLOBAL")
        ((family-tree 'family-tree))
 (labels ((count (tree)
           (if (null tree)
               0
             (1+ (max (count (family-tree-father tree))
                      (count (family-tree-mother tree)))))))
   (format t "The number of generations is ~D."
           (count family-tree))))
```

In this example, pointing the mouse at a pathname that has the **:family-tree** canonical type highlights the presentation. This occurs even if the pathname is displayed as a result of a Show Directory command instead of a Show Family Tree Directory command; or if it results from evaluating

```
(fs:parse-pathname "ACME:>ui-programmer>my-family.family-tree)
```

in a Lisp Listener. If this behavior is unwanted, then the

```
:expander '((and pathname ((satisfies family-tree-file-p))))
```

should be omitted from the type's definition.

Also, the `pathname-to-family-tree` translator applies even if you are not looking specifically for a family tree. If you are accepting a **sys:expression**, perhaps in the Lisp Listener's command loop, you are offered the option of reading in the file.

The programmer's intent was to make the pathnames displayed by Show Family Tree Directory sensitive only in the **family-tree** input context, as produced by the Count Family Tree Generations command. There are two ways to get this behavior. The first is to rename the structure or the presentation type, for example,

```
(defstruct (family-tree-instance (:conc-name "FAMILY-TREE-")
           (:constructor make-family-tree))
  father
  mother)
```

However, sometimes it might be inconvenient to make the presentation type and the name different. In such cases, a **:tester** function can be supplied to the handler that uses **dw:handler-applies-in-limited-context-p** to limit the handler to cases where the program explicitly requests a **family-tree**.

Another problem with the translator, as written, arises when the input context is **family-tree** and the user moves the mouse over one of these pathnames. Because SemantiCue evaluates the body to see if the handler applies, the file is read in without the user even clicking the mouse. This problem could be prevented in two ways. First, we could put `:do-not-compose t` in the option list for the translator. This suppresses the evaluation of the body to check its return value. Unfortunately, it also means that if we did

```
(accept '((and family-tree ((satisfies dad-named-george-p)))))
```

the predicate **dad-named-george-p** would never be run.

Alternatively, the program could be remodularized, with a command that worked on files instead of **family-trees**. Here is our example rewritten, to take all of these considerations into account.

```
(defstruct family-tree
  father
  mother)

(fs:define-canonical-type :family-tree "FAMILY-TREE"
  ((:unix :unix42 :vms :vms4) "FAT"))

(defun family-tree-file-p (pathname)
  (eql (send pathname :canonical-type) :family-tree))

(define-presentation-type family-tree-pathname ()
  ;; It is not unreasonable for any pathname with the type
  ;; of :family-tree to be available as a
  ;; family-tree-pathname, so we leave this in.
  :expander '((and pathname ((satisfies family-tree-file-p)))))
```

```
(cp:define-command (com-show-family-tree-directory
                    :command-table "GLOBAL") ()
  (loop for x in (directory (send (fs:user-homedir)
                                  :new-pathname
                                  :name :wild
                                  :type :family-tree
                                  :version :newest))
        do (present x 'family-tree-pathname)))

(defun count-family-tree-generations (tree)
  (if (null tree)
      0
    (1+ (max (count-family-tree-generations
               (family-tree-father tree))
             (count-family-tree-generations
               (family-tree-mother tree))))))

(cp:define-command (com-count-family-tree-generations
                    :command-table "GLOBAL")
       ((family-tree 'family-tree))
 (format t "The number of generations is ~D."
         (count-family-tree-generations family-tree)))

(cp:define-command (com-count-family-tree-file-generations
                     :command-table "GLOBAL")
    ((pathname 'family-tree-pathname))
  (with-open-file (file pathname
                        :direction :input
                        :element-type 'character)
    (let ((family-tree (read file)))
      (format t "The number of generations is ~D."
              (count-family-tree-generations family-tree)))))
```

An advanced concepts section in the overview of mouse handler facilities presents information pertinent to the above discussion: See the section "Advanced Mouse Handler Concepts", page 42.

8. Window Substrate Facilities

The window system substrate is documented in two areas. We are concerned here with Dynamic Windows and facilities designed exclusively for Dynamic Windows. Many other window substrate facilities are available, however. These facilities, based on static windows, pre-date Genera 7.0 and are documented in *Programming the User Interface, Volume B*: See the section "Using the Window System" in *Programming the User Interface, Volume B.*

Virtually all of the facilities documented in that volume are not restricted in their use to static windows. They are equally useful, and in some cases necessary, for programming with Dynamic Windows. In fact, the Dynamic Window system is for the most part built on the static window system. Therefore, be aware that the facilities described in this section represent only a fraction of the window substrate, and that the documentation for needed init options, methods and other facilities is available in the other volume. The dictionary entry for **dw:dynamic-window** provides references to the relevant sections: See the flavor **dw:dynamic-window**, page 399.

The following table lists the Dynamic Window substrate facilities:

Dynamic Window Facilities
- **dw:dynamic-window**
- **dw:margin-borders**
- **dw:margin-white-borders**
- **dw:margin-whitespace**
- **dw:margin-drop-shadow-borders**
- **dw:margin-ragged-borders**
- **dw:margin-label**
- **dw:margin-scroll-bar**
- **(flavor:method :set-margin-components dw:margin-mixin)**
- **(flavor:method :set-borders dw:margin-mixin)**
- **(flavor:method :set-label dw:margin-mixin)**
- **(flavor:method :delayed-set-label dw:margin-mixin)**
- **(flavor:method :update-label dw:margin-mixin)**
- **dw:set-default-end-of-page-mode**

Dynamic Frame Facilities
- **dw:program-frame**

dw:dynamic-window is the basic window flavor in the Dynamic Window substrate. It is the dynamic equivalent of **tv:window**, the basic static window flavor. Unlike

tv:window, however, **dw:dynamic-window** has built into it a variety of desirable window features. **dw:dynamic-window** also refers to a resource of Dynamic Windows.

The basic Dynamic Window flavor supports an output-history, that is, presentation recording, is scrollable, includes a visible scroll bar, has a label, and is surrounded by a simple, one-pixel-wide border. The last three attributes – the scroll bar, label, and border – are margin components made available via a single mixin flavor, **dw:margin-mixin**.

Most of the remaining Dynamic Window facilities listed in the above table relate to margin components. They provide a set of flavors and methods allowing you to customize the appearance of your program's windows, from a variety of border designs to labels and scroll bars. The following example shows how to make a Dynamic Window with a customized set of margin components:

```
(defun dynamic-window-margin-example ()
  (let ((test (tv:make-window 'dw:dynamic-window
                :edges-from :mouse
                :margin-components
                '((dw:margin-borders :thickness 1)
                  (dw:margin-white-borders :thickness 3)
                  (dw:margin-borders :thickness 10)
                  (dw:margin-white-borders :thickness 8)
                  (dw:margin-borders :thickness 3)
                  (dw:margin-whitespace :margin
                                        :left :thickness 10)
                  (dw:margin-scroll-bar)
                  (dw:margin-whitespace :margin
                                        :bottom :thickness 7)
                  (dw:margin-scroll-bar :margin :bottom)
                  (dw:margin-whitespace :margin :left
                                        :thickness 10)
                  (dw:margin-label :margin :bottom
                                   :style (:sans-serif
                                           :italic :normal))
                  (dw:margin-whitespace :margin :top
                                        :thickness 10)
                  (dw:margin-whitespace :margin :right
                                        :thickness 13))
                :expose-p t
                :mouse-blinker-character #\mouse:fat-circle)))
    (send test :set-label "Margin Test Window")))
```

When you create this window and run the mouse cursor over it, you will notice the cursor changing shape. The shape, in this case a "fat circle", is specified via

the **:mouse-blinker-character** init option. Other available mouse blinker characters are listed in the section that follows.

Additional Dynamic Window methods are included in the program output category, because of their usefulness in that context: See the section "Overview of Other Facilities for Program Output", page 60.

Dynamic frame facilities considered to be substrate-level are limited to **dw:program-frame**. This is the building-block flavor used by the Frame-Up Layout Designer and **dw:define-program-framework** to create program frames. For an overview of these facilities and some frame functions: See the section "Overview of Top-level Facilities".) Also, as is the case with Dynamic Windows generally, static window system facilities for programming with frames are applicable to dynamic frames as well: See the section "Frames" in *Programming the User Interface, Volume B.* **dw:program-frame** is also a window resource.

Reference documentation for the facilities discussed in this section is included in a separate dictionary: See the section "Dictionary of Window Substrate Facilities", page 395.

8.1 Mouse-Blinker Characters

Through the **:mouse-blinker-character** init option to **dw:dynamic-window**, the mouse blinker, when moved over a Dynamic Window, can assume any of the shapes available in the mouse font (**fonts:mouse**). To see the glyphs included in this font, use the Show Font Command Processor command on "mouse". Each glyph in the font maps to a unique mouse-blinker character. The following lists these in the order in which they appear in the font:

#\mouse:up-arrow
#\mouse:right-arrow
#\mouse:down-arrow
#\mouse:left-arrow
#\mouse:vertical-double-arrow
#\mouse:horizontal-double-arrow
#\mouse:nw-arrow
#\mouse:times
#\mouse:fat-up-arrow
#\mouse:fat-right-arrow
#\mouse:fat-down-arrow
#\mouse:fat-left-arrow
#\mouse:fat-double-vertical-arrow
#\mouse:fat-double-horizontal-arrow

#\mouse:paragraph
#\mouse:nw-corner
#\mouse:se-corner
#\mouse:hourglass
#\mouse:circle-plus
#\mouse:paintbrush
#\mouse:scissors
#\mouse:trident
#\mouse:ne-arrow
#\mouse:circle-times
#\mouse:big-triangle
#\mouse:medium-triangle
#\mouse:small-triangle
#\mouse:inverse-up-arrow
#\mouse:inverse-down-arrow
#\mouse:filled-lozenge
#\mouse:dot
#\mouse:fat-times
#\mouse:small-filled-circle
#\mouse:filled-circle
#\mouse:fat-circle
#\mouse:fat-circle-minus
#\mouse:fat-circle-plus
#\mouse:down-arrow-to-bar
#\mouse:short-down-arrow
#\mouse:up-arrow-to-bar
#\mouse:short-up-arrow
#\mouse:boxed-up-triangle
#\mouse:boxed-down-triangle

Note that mouse-blinker characters are non-printing; that is, they are intended for on-line use only.

9. User Interface Application Example

This chapter presents a simple application illustrating the use of some of the major facilities available in Genera's user interface management system. The primary focus is on the command interface aspects of setting up an application. A separate examples file, `sys:examples;ui-application-example.lisp` contains the functions presented here in a more readily compilable form.

Suppose we have a simple flavor defined like this:

```
(defvar *employee-list* nil)

(defflavor employee (first-name last-name (retired-p nil))
                ()
 :readable-instance-variables
 :writable-instance-variables
 :initable-instance-variables)

(defmethod (make-instance employee) (&rest ignored)
 (push self *employee-list*))

(defmethod (employee-name employee) ()
 (format nil "~a ~a" first-name last-name))

(make-instance 'employee :first-name "Fred" :last-name "Flintstone")
(make-instance 'employee :first-name "Barney" :last-name "Rubble")
```

This is how we might define a presentation type to read objects of this flavor:

```
(define-presentation-type employee (() &key abbreviate)
   :no-deftype t          ;there's already a flavor, so don't try to
                          ;       define a type
   :history t             ;give us our own type history
   :parser ((stream &key type initially-display-possibilities)
            (dw:completing-from-suggestions
              (stream   :initially-display-possibilities
                                initially-display-possibilities
                        :partial-completers '(#\space)
                        :type type)
              (loop for emp in *employee-list*
                    do (dw:suggest (employee-name emp) emp))))
   :printer ((object stream &key acceptably)
             (when acceptably (write-char #\" stream))
          (cond (abbreviate (write-string (employee-last-name object) stream))
                (t (write-string (employee-name object) stream)))
          (when acceptably (write-char #\" stream)))
   :description "an employee")
```

The following handler turns **employee**s into strings by extracting their names. Thus, in the input context established by (accept 'string), **employee** displays are mouse-sensitive; clicking left (the **:select** gesture) on one is equivalent to typing the employee's name.

```
(define-presentation-translator employee-to-string
        (employee string
         :gesture :select
         :documentation "This employee's name")
                (employee)
  (employee-name employee))
```

The following handler is a side-effecting handler. It acts on **employee**s that are not currently retired (discriminating by using a **:tester** function) in *any* input context. (This is because **t**, meaning "any," is specified as the *to-presentation-type* for this handler.) This handler runs whenever a middle click is executed while the mouse is over an unretired **employee**. Since this happens without the application (command-loop) knowing about it, programming your interface in this style is not recommended. (Below you will see another example that shows you how to define a handler that makes changes that your program does know about.)

```
(define-presentation-action retire-employee
        (employee t
         :gesture :middle
         :tester ((employee)
                  (not (employee-retired-p employee)))
         :documentation "Retire this employee")
                (employee)
  (setf (employee-retired-p employee) t))
```

The following function formats a table of employee information:

```
(defun format-employees-list (stream)
  (fresh-line stream)
  (formatting-table (stream)
    (formatting-column-headings (stream :underline-p t)
      (with-character-face (:italic stream)
        (formatting-cell (stream) "name")
        (formatting-cell (stream) "retired?")))
    (loop for employee in *employee-list*
          do
      (formatting-row (stream)
        (formatting-cell (stream)
          (present employee 'employee
                 :stream stream))
        (formatting-cell (stream)
          (format stream "~:[no~;yes~]"
                (employee-retired-p employee)))))))
```

Here is a sample user interface framework for the application, created with the Frame-Up Layout Designer:

```
(dw:define-program-framework employee-editor
  :select-key #\*
  :command-definer t
  :command-table (:inherit-from 'nil :kbd-accelerator-p 'nil)
  :state-variables nil
  :panes
  ((pane-1 :title :size-from-output nil :redisplay-string nil
           :redisplay-function nil :height-in-lines 1
           :redisplay-after-commands nil)
   (pane-3 :display)
   (pane-2 :command-menu :equalize-column-widths nil
           :center-p nil :columns nil :rows nil
           :menu-level :top-level)
   (pane-4 :interactor :typeout-window nil :height-in-lines 4))
  :configurations
  '((dw::main (:layout (dw::main :column pane-1 pane-3 pane-2 pane-4))
     (:sizes
      (dw::main (pane-1 1 :lines) (pane-2 :ask-window self :size-for-pane pane-2)
       (pane-4 4 :lines) :then (pane-3 :even))))))
```

Notice that `:command-definer t` was specified in the above. (This is the default.) This means that a command-defining macro called `define-`*application*`-command` (in this case, `define-employee-editor-command`) will be defined for you. Use this macro to define commands associated with this application.

`define-`*application*`-command` has a syntax very similar to **cp:define-command**, differing in only several respects. First, commands defined using it will be installed in the command table of your application, not in the Global or User command table. Second, `define-`*application*`-command` takes the extra options **:kbd-accelerator**, **:menu-accelerator**, and **:menu-level** (explained in more detail below) that allow you to specify alternate ways of entering this command while in the application. Third, the command may refer to the state variables of the application.

Although a Command Processor command is defined when `define-`*application*`-command` is used, this does not constrain you to interacting with the application solely by typing commands on a command line. If you tell **dw:define-program-framework** that your application has an **:interactor** pane, then this interactor will, by default, read and execute commands defined with this macro. (There is no requirement that your application have an interactor, however.)

If you specify the option **:menu-accelerator** with some value as one of the options to `define-`*application*`-command`, the command will exist in the command-menu of your application, if one exists. `:menu-accelerator t` means the command name

should go in the menu. :menu-accelerator *name* means *name* should go in the menu. :menu-level *name* means "put this command in the menu with menu-level *name*." Thus, all the commands with :menu-accelerator *something* will be accessible from a command-menu.

Similarly, :kbd-accelerator *char* associates this command with a specific keystroke (as in the debugger). Your command-table must have :kbd-accelerator-p t for this to work. (This is the "Read single keystroke accelerators" option in Frame-Up.)

Using the above two options, you can construct interfaces that allow users to type commands as commands, click on commands in menus, and execute commands through single keystrokes. Using presentation-translators, you can allow the user to click on presentations to enter commands. (An example is presented below.) These facilities should give you enough flexibility to construct powerful user interfaces.

Here is an example command for the employee-editor application. It simply changes the status of an employee. It is available from the command menu as "Change Status".

```
(define-employee-editor-command
        (com-change-employee-status
          :menu-accelerator "Change Status")
        ((employee 'employee
         :confirm t
         :prompt "employee"))
  (setf (employee-retired-p employee)
    (not (employee-retired-p employee)))
  (fresh-line)
  (present employee 'employee)
  (write-string "'s status changed."))
```

This command is available from the command menu as "Show Employees", and simply calls the table-formatting routine defined above:

```
(define-employee-editor-command
        (com-show-employees :menu-accelerator t)
        ()
  (format-employees-list *standard-output*))
```

Note that while debugging your application, you can recompile the **dw:define-program-framework** form at any time. This causes your application to inherit any of the new options you may have added. Similarly, if you compile a command with a **:menu-accelerator**, when you reselect your application and restart the process (m-ABORT should do the job), the new item will be in the menu. This makes debugging and prototyping very easy.

Here is an example of the third way to enter commands. In this case, a translator from **employees** to commands is defined. This means that whenever the input context is **cp:command** (as it will be inside the command loop written for your application), objects presented as **employees** will be sensitive. The result of clicking left on them (the **:select** gesture) is to cause the Change Employee Status command to be executed with that employee as its argument.

Note that this translator applies in **cp:command** context. In string context, the translator defined above will still apply. Unlike the side-effecting mouse handler we defined above, this is the recommended way of interacting with your application through the mouse. When you use a translator, your application knows what the user did (there is no difference between clicking as above or typing Change Employee Status). When you use a side-effecting handler, your application does not know what happened. In this case, the display would not be updated to show that the employee was retired had you clicked middle on the employee, but it would be updated if you clicked left. This is because the click-left case (translating to a command) goes through the application properly.

```
(define-presentation-to-command-translator change-employee-status
                                           (employee)
  (employee)
  (cp:build-command 'com-change-employee-status
                    employee))
```

You can also write commands to manipulate your data structures that can be called from top-level command loops (that is, Lisp Listeners). Simply use **cp:define-command** instead. You will not be able to use the state variables of your application, though.

```
(cp:define-command (com-another-way-to-change-employee-status
        :command-table 'user)
    ((employee 'employee :prompt "employee"))
   (setf (employee-retired-p employee)
     (not (employee-retired-p employee)))
   (present employee 'employee)
   (write-string "'s status changed."))
```

Advanced applications may need more complex command loops than the one written by default by **dw:define-program-framework**. This is what the **:top-level** and **:command-evaluator** options to **dw:define-program-framework** are for.

The syntax for the **:top-level** option is `:top-level` (*function-name &rest args*). *Function-name* is the name of a function. It will be called with one argument, the program instance – consequently, a generic function may be appropriate for this option if the top-level function wishes to access state variables – followed by *args*. A slightly modified example from the system is:

```
:top-level (examiner-top-level :prompt "Flavor Examiner: ")

(defun examiner-top-level (program &rest options)
  ;; No point in making this a generic function,
  ;; although typically it would be.
  (examiner-help program (dw:get-program-pane 'command) nil)
  (apply #'dw:default-command-top-level program options))
```

What this does is define a top-level function for the flavor-examiner that ensures that the flavor-examiner's help message is displayed before the default command loop is entered. In this case, `:prompt "Flavor Examiner: "` is simply passed into **dw:default-command-top-level** which causes "Flavor Examiner: " to be used as the prompt.

The syntax for the **:command-evaluator** option is simply `:command-evaluator` *function-name*. *function-name* is the name of a function. It will be called with three arguments, the program instance (a generic function may be appropriate for this option if the function wishes to access state variables), the command symbol, and the arguments to the command. The function may do anything it wishes, but eventually should do `(apply command-symbol command-args)`. (For information on facilities used when writing command loops: See the section "Overview of Command Loop Management Facilities", page 33.

For a more advanced example of **dw:define-program-framework**, see the `calculator` program in the file `sys:examples;define-program-framework.lisp`. This program creates its own command-menu and command-menu handlers to simulate a four-function calculator. For this it uses two undocumented functions **dw:program-command-menu-item-list** and **dw:define-command-menu-handler**. See the referenced file for more information.

PART II.

Dictionary of Top-level Facilities for User Interface Programming

10. Dictionary Notes

This dictionary includes reference documentation for the following facilities:

Table of Top-Level Facilities for User Interface Programming

Frame-Up Layout Designer

Program Framework Definition
dw:define-program-framework
dw:*program-frame*
dw::find-program-window
dw:get-program-pane

Program Command Definition
dw:define-program-command
define-presentation-to-command-translator

The documentation for Frame-Up is presented first, followed by the remaining facilities in alphabetical order (package prefixes excluded).

For conceptual documentation: See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.

11. The Facilities

11.1 Frame-Up Layout Designer

11.1.1 Introduction

The Frame-Up Layout Designer is an interactive facility for helping you create the user interface to an application program. It is available on SELECT Q, through the System Menu, or from Zmacs.

More specifically, Frame-Up is the interactive version of **dw:define-program-framework**, a macro for defining a program's window and command interface. Frame-Up lets you configure a *program frame* and specify options for individual *panes* within the frame. (For more information on frames and panes: See the section "Frames" in *Programming the User Interface, Volume B.*) Other options, for the program as a whole, provide control over the program's command loop.

When you finish configuring the program frame and specifying pane and program options, Frame-Up creates the corresponding **dw:define-program-framework** code. This code is written to an editor buffer where it is available for hand editing. (For information on how to edit the frame configuration, see the above-referenced section on "Frames".) Alternatively, you can go back to Frame-Up, modify the interface, and have the new code written out in place of the old.

Three additional sections complete the Frame-Up Layout Designer documentation:

See the section "Getting Started with Frame-Up", page 103.

See the section "Frame-Up Commands", page 104.

See the section "Zmacs Commands for Frame-Up", page 114.

For an overview of the Frame-Up Layout Designer and related facilities, in particular, **dw:define-program-framework**: See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.

11.1.2 Getting Started

You can invoke the Frame-Up Layout Designer from the System Menu, via SELECT Q, or from a Zmacs editor buffer. Because the ultimate output provided by Frame-Up is editable Lisp code, it may be simplest to start off in a Lisp-mode buffer at the point where you want the **dw:define-program-framework** macro to be written. With the editor cursor at this point:

1. Enter the extended command (M-X) Create Program Definition.
2. Enter the program name.

The name entered in step 2 is the name argument to **dw:define-program-framework**. It is the name given to the program flavor created by this macro for your application.

After invoking the Frame-Up program, whether from an editor buffer or directly, an initial display appears including a starting configuration for the program frame and a menu of Frame-Up commands. Program- and frame-level commands are listed together on the left of the command menu, pane-level commands on the right. You could start with any of these, but if you are unfamiliar with Frame-Up, we recommend that you start with commands in the first category: See the section "Program and Frame Commands in Frame-Up", page 104.

11.1.3 Commands

11.1.3.1 Program and Frame Commands

Five Frame-Up Layout Designer commands are included in this category: Set Program Options; Select Configuration; Reset Configuration; Preview; and Done. The following subsections consider each in turn.

Set Program Options

The program options you can modify using the Set Program Options command are described below. (Where appropriate, references to the corresponding **dw:define-program-framework** options are given.)

Program name The name of the program flavor created by **dw:define-program-framework** for your application.

If you invoked Frame-Up from an editor buffer with the Create Program Definition extended command, the default value for this option is the name you supplied to that command.

Select key The key to use for selecting your program.

(See the macro "**dw:define-program-framework** 124.)

Name of command-defining macro

The name given to the macro created by **dw:define-program-framework** and used to define commands for your program.

The default, **t**, causes your program name to be used as part of this name. For example, if the name of your program is shell-game, the default command-defining macro will be define-shell-game-command.

You use the command macro created for you as you would **dw:define-program-command**; the syntax and keywords are the same.

(See the function "**dw:define-program-framework**", page 124.)

Read single-character command accelerators

Boolean option specifying whether your program accepts single-character command accelerators; the default is `No`.

If you enter `Yes` for this option, you have three possible sources of accelerators:

1. Accelerators you inherit when you inherit command tables using the program option discussed below.
2. Standard accelerators you supply to your program. (See the section "Overview of Advanced Command Facilities", page 32.)
3. Accelerators you define yourself. (See the section "Overview of Advanced Command Facilities", page 32.)

(See the macro "**dw:define-program-framework**", page 124.)

Inherit commands from command tables

The name(s) of command table(s) from which your program inherits commands and, if specified by the above option, command accelerators.

For example, supplying a value of `user` to this option results in all of the commands normally available in a Lisp Listener being available in your program, in addition to program commands you define yourself.

The default for this option – `'("colon full command" "standard arguments" "standard scrolling")` – enables use of extended (`m-x`) and colon full commands, standard single-character accelerators like `c-U`, and standard scroll keys like `SCROLL` and `m-SCROLL`. These are enabled only if you specify `Yes` to the **Read single-character command accelerators** option.

(See the macro "**dw:define-program-framework**", page 124.)

Select Configuration

The Select Configurations command gives you a choice of two standard configurations for your program frame. The first consists of a command-menu

pane and a listener pane; the second of a title, command-menu, display, and interactor pane. (For a description of pane types: See the section "Set Pane Options Frame-Up Command", page 107.)

You may select a standard configuration and then modify it using one or more of the pane-oriented commands: See the section "Pane Commands in Frame-Up", page 107.

Reset Configuration

The Reset command restores the original program frame. (The original frame is the one displayed when you first enter Frame-Up; it consists of a single display pane.)

Preview

The Preview command lets you see what the frame you have configured looks like on a full-screen display without having to compile your program. Without this command, to see your program frame you would have to exit Frame-Up, compile the **dw:define-program-framework** definition, and select your program. With it, you can look at the frame directly and, if unsatisfied with the result, continue editing the layout before writing out the interface code.

Done

The Done command signals the end of the Frame-Up session. What happens when you invoke this command depends on how you entered Frame-Up:

- If you entered Frame-Up from an editor buffer via the Create Program Definition or Edit Program Definition extended editor command, then Frame-Up returns you to that buffer and automatically writes out the **dw:define-program-framework** macro corresponding to the interface you configured.

  In the case of Edit Program Definition, the new code replaces the old code (**dw:define-program-framework** macro) that was already there.

- If you entered Frame-Up from the System Menu or via `SELECT Q`, then you are returned to the activity selected prior to entering Frame-Up.

  In this case, the **dw:define-program-framework** code corresponding to your interface is not written automatically to an editor buffer. You must select the buffer you wish the code to be written to and use the extended editor command Insert Program Definition.

11.1.3.2 Pane Commands

Five Frame-Up Layout Designer commands are available for manipulating panes: Set Pane Options; Set Pane Name; Split Pane; Swap Panes; and Delete Pane. The following subsections discuss each in turn.

(Note that, after finishing the Frame-Up session, further editing of the code affecting the appearance of program panes and the frame as a whole is possible. For more information: See the section "Frames" in *Programming the User Interface, Volume B.* In particular: See the section "**:layout** Constraint Frame Specification" in *Programming the User Interface, Volume B.* See the section "**:sizes** Constraint Frame Specification" in *Programming the User Interface, Volume B.*)

Set Pane Options

Pane options you can modify using the Set Pane Options command include the pane name and type. Other options depend on the pane type. Six types are available:

Accept-Values	Pane providing the features and services of a **dw:accept-variable-values** menu (the kind of menu used to display the pane options themselves).
Display	Pane for display of application-generated output.
Title	Pane for display of the program title.
Command-Menu	Pane for menu of program commands.
Interactor	Pane for interactive input/output.
Listener	Similar to an interactor, but taller. (Use this pane when you want the interaction history to be visible.)

The following subsections describe the other options available for each pane type. (For information on additional options that you can hand-edit into the **dw:define-program-framework** macro: See the macro "**dw:define-program-framework**", page 124.)

Accept Values Pane Options

Accept values function

Specifies a function for creating a **dw:accept-variable-values**-like display. The function may be written either as a generic function (using **defmethod**) to the program flavor or as a regular function (using **defun**). The function is passed two arguments: the current instance of the program flavor and the stream for I/O.

The multiple-**accept** display is created by wrapping the body of the function you write in a **dw:accepting-values** macro: See the macro **dw:accepting-values,** page 175. The wrapping is done for you by **dw:define-program-framework.** The general form of the function you write is

```
(defmethod (my-avv-function program) (stream)
  (setq state-var-1 (accept ...)
  (setq state-var-2 (accept ...)
  (setq state-var-3 (accept ...)
  ...)
```

For an example, see the program avv-pane-test in the file sys:examples;define-program-framework.lisp

If you include an Accept Values pane in your program frame but do not specify an Accept Values Function, the option defaults to an internal function that uses your program's state variables as the variables in the accept-values display. The state variables are those specified by the **:state-variables** option to **dw:define-program-framework:** See the macro "**dw:define-program-framework** 124.

This option maps to the **:accept-values-function** keyword option for **:accept-values** panes: See the macro "**dw:define-program-framework**", page 124.

Redisplay each time around command loop

Boolean option specifying whether to redisplay the pane after each command is executed. The default is Yes for Accept-Values panes, No for Display and Title panes.

This option maps to the **:redisplay-after-commands** keyword option for program panes: See the macro "**dw:define-program-framework**", page 124.

Set size of pane from contents

Boolean option specifying whether a pane is sized according to the space needs of output to that pane. The default is Yes for Accept-Values panes, No for Display and Title panes.

This option maps to the **:size-from-output** keyword option for program panes: See the macro "**dw:define-program-framework**", page 124.

Height in lines Fixes the pane height to the specified number of lines. The default value is 1 for Title panes and 4 for Interactor panes. No default is provided for Listener Panes.

This option maps to the **:height-in-lines** keyword option for program panes: See the macro **"dw:define-program-framework"**, page 124.

Display Pane Options

Redisplay output generator

This option specifies one of three possibilities for generating redisplay to the pane: no redisplay generator (`None`); a redisplay string (`String`); or a redisplay function (`Function`).

If you specify `String`, then the **Redisplay string** option appears: See the section "Display Pane Options", page 109.

If you specify `Function`, then both the **Redisplay function** and **Incremental redisplay** options appear. See the section "Display Pane Options", page 109.

Redisplay string Specifies a string written to the pane (starting at top) whenever the pane is redisplayed. This option is mutually exclusive with the **Redisplay function** option.

This option maps to the **:redisplay-string** keyword option for program panes: See the macro **"dw:define-program-framework"**, page 124.

Redisplay function

The function that runs whenever the pane is redisplayed. This option is mutually exclusive with the **Redisplay string** option.

The redisplay function may be written either as a generic function (using **defmethod**) to the program flavor or as a regular function (using **defun**). The function is passed two arguments: the current instance of the program flavor and the stream on which to do output.

This option maps to the **:redisplay-function** keyword option for program panes: See the macro **"dw:define-program-framework"**, page 124.

Incremental redisplay

Boolean option specifying whether redisplayed information is limited to items that have changed since the last redisplay, rather than the entire pane; the default is `No`.

If you specify `Yes`, you must write the appropriate redisplay function (see **Redisplay function**) above).

For information on incremental redisplay: See the section "Overview of Advanced Program Output Facilities", page 63. See also the file sys:examples;incremental-redisplay.lisp.

This option maps to the **:incremental-redisplay** keyword option for **:display** panes: See the macro "**dw:define-program-framework**", page 124.

Pane flavor The pane flavor to use for this pane; the default is **dw::dynamic-window-pane.**

This option maps to the **:flavor** keyword option for **:display** panes: See the macro "**dw:define-program-framework**", page 124.

Redisplay each time around command loop
Boolean option specifying whether to redisplay the pane after each command is executed. The default is Yes for Accept-Values panes, No for Display and Title panes.

This option maps to the **:redisplay-after-commands** keyword option for program panes: See the macro "**dw:define-program-framework**", page 124.

Typeout window Boolean option specifying whether a typeout (pull-down) window for ***terminal-io*** appears within the pane. The default is No.

This option maps to the **:typeout-window** keyword option for program panes: See the macro "**dw:define-program-framework**", page 124.

Height in lines Fixes the pane height to the specified number of lines. The default value is 1 for Title panes and 4 for Interactor panes. No default is provided for Listener Panes.

This option maps to the **:height-in-lines** keyword option for program panes: See the macro "**dw:define-program-framework**", page 124.

Set size of pane from contents
Boolean option specifying whether a pane is sized according to the space needs of output to that pane. The default is Yes for Accept-Values panes, No for Display and Title panes.

This option maps to the **:size-from-output** keyword option for program panes: See the macro "**dw:define-program-framework**", page 124.

Title Pane Options

Redisplay output generator

This option specifies one of three possibilities for generating redisplay to the pane: no redisplay generator (`None`); a redisplay string (`String`); or a redisplay function (`Function`).

If you specify `String`, then the **Redisplay string** option appears: See the section "Display Pane Options", page 109.

If you specify `Function`, then both the **Redisplay function** and **Incremental redisplay** options appear. See the section "Display Pane Options", page 109.

Redisplay string Specifies a string written to the pane (starting at top) whenever the pane is redisplayed. This option is mutually exclusive with the **Redisplay function** option.

This option maps to the **:redisplay-string** keyword option for program panes: See the macro **"dw:define-program-framework"**, page 124.

Redisplay function

The function that runs whenever the pane is redisplayed. This option is mutually exclusive with the **Redisplay string** option.

The redisplay function may be written either as a generic function (using **defmethod**) to the program flavor or as a regular function (using **defun**). The function is passed two arguments: the current instance of the program flavor and the stream on which to do output.

This option maps to the **:redisplay-function** keyword option for program panes: See the macro **"dw:define-program-framework"**, page 124.

Incremental redisplay

Boolean option specifying whether redisplayed information is limited to items that have changed since the last redisplay, rather than the entire pane; the default is `No`.

If you specify `Yes`, you must write the appropriate redisplay function (see **Redisplay function**) above).

For information on incremental redisplay: See the section "Overview of Advanced Program Output Facilities", page 63. See also the file `sys:examples;incremental-redisplay.lisp`.

This option maps to the **:incremental-redisplay** keyword option for **:display** panes: See the macro **"dw:define-program-framework"**, page 124.

Redisplay each time around command loop
Boolean option specifying whether to redisplay the pane after each command is executed. The default is `Yes` for Accept-Values panes, `No` for Display and Title panes.

This option maps to the **:redisplay-after-commands** keyword option for program panes: See the macro **"dw:define-program-framework"**, page 124.

Height in lines Fixes the pane height to the specified number of lines. The default value is `1` for Title panes and `4` for Interactor panes. No default is provided for Listener Panes.

This option maps to the **:height-in-lines** keyword option for program panes: See the macro **"dw:define-program-framework"**, page 124.

Set size of pane from contents
Boolean option specifying whether a pane is sized according to the space needs of output to that pane. The default is `Yes` for Accept-Values panes, `No` for Display and Title panes.

This option maps to the **:size-from-output** keyword option for program panes: See the macro **"dw:define-program-framework"**, page 124.

Command-Menu Pane Options

Menu geometry Specifies how the menu is to be laid out. You have three choices: you can let Frame-Up come up with a configuration (`Default`) that is in most cases reasonable; you can control the layout yourself by specifying menu `Rows`; or you can control layout by specifying menu `Columns`.

If you select `Rows`, then you are asked if you want to **Specify number of rows or row contents**. If `Number`, then enter a value in the **Number of rows** field that appears. If `Contents`, then enter one or more command names (strings) to be the **Items in row 1**, followed by the entering of one or more strings to be the **Items in row 2**, and so on, until all the rows are specified. This option maps to the **:rows** keyword option for **:command-menu** panes: See the macro **"dw:define-program-framework"**, page 124.

If you select Columns for the **Menu geometry** option, you proceed in a fashion analogous to that described for Rows. This option maps to the **:columns** keyword option for **:command-menu** panes: See the macro "**dw:define-program-framework**", page 124.

If you specify menu rows or columns by their contents, the string used to identify each command must be the same as that specified in the **:menu-accelerator** option to the command definer used for the program. (See the macro "**dw:define-program-command**", page 122.) The command definer is specified by one of the options in the Set Program Options command: See the section "Set Program Options Frame-Up Command", page 104.

Menu identifier Symbol identifying the command menu to appear in this pane if the program frame includes more than one. If only one command menu is available, the default value (:TOP-LEVEL) for this option is the appropriate choice.

This option maps to the **:menu-level** keyword option for **:command-menu** panes: See the macro "**dw:define-program-framework**", page 124.

Center menu items

Boolean option specifying whether command names are centered (left-right) in the command menu. The default is No, causing command names to be flush left in the column.

This option maps to the **:center-p** keyword option for **:command-menu** panes: See the macro "**dw:define-program-framework**", page 124.

Compress item columns

Boolean option specifying whether columns of command names are compressed on the left side of the pane or spread out over the full horizontal extent of the pane. The default is Yes (compressed to the left).

This option maps to the **:equalize-column-widths** keyword option for **:command-menu** panes: See the macro "**dw:define-program-framework**", page 124.

Interactor and Listener Pane Options

Typeout window Boolean option specifying whether a typeout (pull-down) window for ***terminal-io*** appears within the pane. The default is No.

This option maps to the **:typeout-window** keyword option for program panes: See the macro "**dw:define-program-framework**", page 124.

Height in lines Fixes the pane height to the specified number of lines. The default value is 1 for Title panes and 4 for Interactor panes. No default is provided for Listener Panes.

This option maps to the **:height-in-lines** keyword option for program panes: See the macro "**dw:define-program-framework**", page 124.

Set Pane Name

The Set Pane Name command lets you change the name of a pane. The arguments to this command are the current name of the pane and the new name.

Split Pane

The Split Pane command divides the specified pane in half. Arguments to this command are the pane to divide and whether the division is horizontal or vertical.

Splitting a pane horizontally causes the two daughter panes to appear in a column orientation, one on top of the other. Splitting a pane vertically causes the two daughter panes to appear in a row orientation, side-by-side.

Swap Panes

Use the Swap Pane command to exchange the position of two panes. The two panes must occur in either the same row or same column.

Delete Pane

This command deletes a specified pane from the configuration for the program frame.

11.1.4 Zmacs Commands for Frame-Up

11.1.4.1 Create Program Definition

The Create Program Definition command initiates a Frame-Up session from an editor buffer. When the session is terminated (via the Done command to Frame-Up), the **dw:define-program-framework** code corresponding to the configured interface is inserted into the buffer at point.

Create Program Definition is an extended (m-x) Zmacs command. When invoked, it firsts prompts you for the name of the program, then enters Frame-Up.

If you entered Frame-Up via SELECT Q or from the System Menu, then use the Insert Program Definition extended command to write the **dw:define-program-framework** code into an editor buffer.

11.1.4.2 Insert Program Definition

Insert Program Definition is an extended (m-x) Zmacs command for writing Frame-Up Layout Designer code into an editor buffer. Use it when you have entered Frame-Up via SELECT Q or from the System Menu, rather than through the Create Program Definition extended command.

When you exit from the Frame-Up session (via the Done command), select an editor buffer and use the Insert Program Definition command to write the **dw:define-program-framework** code corresponding to the configured interface. The code is inserted at point.

11.1.4.3 Edit Program Definition

You can use the Edit Program Definition extended (m-x) command to re-enter a Frame-Up session and make further modifications to the user interface configuration. This occurs after you have already written into your editor buffer the **dw:define-program-framework** macro corresponding to an earlier session. (The original code may have been written through either the Create Program Definition or Insert Program Definition extended command.)

When you terminate the new Frame-Up session (via the Done command), the code corresponding to the new interface configuration replaces the original code.

define-presentation-to-command-translator *name* *Macro*
(*presentation-type* &key *tester* (*gesture* **:select**)
documentation suppress-highlighting (*menu* **t**)
(*context-independent* **nil**) *priority*
exclude-other-handlers blank-area
do-not-compose) *arglist* &body *body*

Defines a mouse handler that translates from a displayed presentation object into a Command Processor command using that object as input.

name The name of the handler.

presentation-type
The type of the displayed presentation object for which the handler is intended.

:tester Specifies the parameter list and body for a tester function. The tester function determines whether the handler applies to the current presentation, if it is otherwise applicable based on the current presentation type and input context.

The parameter list consists of a positional argument – the current presentation object – and a subset of the keywords *presentation*, *input-context*, and *handler*. These keywords are the same as those available for inclusion in the argument list for the body of the handler, and are documented under *arglist* in the handler documentation; they are also documented separately: See the macro "**define-presentation-action**", page 179.

Note: inefficient testers can degrade the performance of your program. Tester functions must be capable of rapid execution. Also, do not use the body of your handler as an implicit tester if it does a large amount of consing or in other ways consumes resources; this will similarly affect program performance. For more information: See the section "Some Efficiency Caveats for Mouse Handlers", page 44.

For functions used in **:testers**: See the function **dw:handler-applies-in-limited-context-p**, page 192. See the function **dw:presentation-subtypep-cached**, page 199.

:gesture Specifies the mouse gesture on which the handler is available.

The gesture is specified by its symbolic name rather than as a mouse character. For example, symbolic names for #**mouse-l**, #**mouse-m**, and #**mouse-r** include **:left**, **:middle**,

and **:right**, respectively. (For lists of names assigned to these and other mouse gestures, use the function **dw:mouse-char-gestures**.) The default gesture is **:select**, which is the same as **:left**.

To assign your own symbolic name to a mouse character, use the following form:

```
(setf (dw:mouse-char-for-gesture symbol) #\mouse-x)
```

Specifying this option with **nil**, that is `:gesture nil`, results in the handler being unavailable on any gesture, only in a handler menu. See the macro "**define-presentation-action**", page 179.

Specifying this option with **t**, that is, `:gesture t`, results in the handler being available on all gestures. See the macro "**define-presentation-action**", page 179.

:documentation

Specifies a string or a function returning a string to be used as mouse and menu documentation for the handler.

It is often preferable not to supply this option and to use the default documentation instead. This is because the default documentation incorporates a string corresponding to the object the mouse is over, while the documentation you supply cannot. If the name of the handler is *handler-name*, the default documentation string will be "Handler Name *(presentation type) presentation object*".

:suppress-highlighting

Boolean option specifying whether to suppress highlighting of the presentation if this handler is the only applicable one. For example, the standard click-right menu handler uses this option. The default is **nil**.

:menu Specifies the name of a menu in which the handler is to be included. The default is **t**, the name of the standard click-right handler menu.

You can define you own handler menu with **define-presentation-action**: See the section "**define-presentation-action**", page 179.

:context-independent

Boolean option specifying whether handler behavior (that is, applicability to displayed presentations) is the same for all contexts in a nested-context structure (**accept** being called recursively); the default is **nil**.

This option is supplied with **t**, for example, if the handler's *to-presentation-type* is **t** (any context), and its contract is to print additional information about a particular presentation (that is, only the output matters).

Specifying this option **t**, when appropriate, allows more possibilities to be presented on different mouse gestures. Without it, a handler that applies in all contexts would be matched for a particular context, to the possible exclusion of other handlers in other contexts on other gestures. With it, you get the same behavior for this handler, and more possibilities as well.

For more information on context matching and related handler issues: See the section "How Mouse Handlers Are Found", page 42.

:priority Specifies a number adding to the priority of this handler relative to other applicable handlers defined on the same gesture; the default is `0`.

Handler applicability to displayed presentations depends on three factors: 1) the object type of the presentation; 2) the presentation type of the presentation; and 3) the current input context. A handler matching a displayed presentation in any of these factors is applicable and invokable.

In some cases, more than one applicable handler might be available on a given mouse gesture. In such cases, which handler is the one displayed for that gesture in the mouse documentation line is determined by handler precedence or priority. The system automatically assigns priotities according to the matching factors as follows: the priority is incremented by 1 when the object type matches; by 4 when the presentation type matches; and by 2 when the context type matches.

For example, in a Lisp Listener in the command-or-form context, an **accept** of a pathname appears something like the following:

```
(accept 'pathname)
Enter the pathname of a file [default
Q:>rel-7>sys>doc>uims>ui-dict2.sar]: ==>
Q:>rel-7>sys>doc>uims>ui-dict2.sar
#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"
FS:LMFS-PATHNAME
```

The default pathname was accepted causing it to be presented as both a pathname presentation (Q:>rel-7>sys>doc>uims>ui-dict2.sar) and a **sys:expression** presentation (`#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"`).

Two handlers defined on the **:select** gesture are applicable to both presentations. The first is **si:com-show-file**, applicable to expression presentations with a pathname object type, or pathname presentations of any object type. The second is **dw::quoted-expression**, applicable to expression presentations of any object type. The following table shows the priorities determined for them by the system relative to the two presentations in the above example:

	Pathname Presentation `Q:>rel-7>sys>doc>...`	Expression Presentation `#P"Q:>rel-7>sys>doc>...`
Show File	5	5
Quote Expression	0	4

For both presentations, the system-generated priority is highest for the show file handler. However, it was the system programmer's intent that the quoted expression handler should be displayed in the mouse documentation line whenever the mouse is over a presentation of the **sys:expression** type, regardless of what other applicable handlers might be available on the **:select** gesture. Therefore, in the definition for this handler, the value of the **:priority** option was made `1.5`. This is added to the system-generated priority of `4` in the bottom right cell of the table for a total score of `5.5`, enough to give this handler precedence.

:exclude-other-handlers

Boolean option, used with `:gesture t` handlers, specifying whether to exclude non-t handlers.

For example, any gesture selects a menu item. The translator that implements this has a **:tester** option that checks, among other things, for the keyword **:no-select** in the menu-item list: See the section "The "General List" Form of Item" in *Programming the User Interface, Volume B.* If the menu item includes the **:no-select** keyword, the translator does not apply. But, if `:exclude-other-handlers t` were not specified for this translator, other translators would still apply to the **:no-select** item's presentation, like the **:menu** (`Mouse-R`) gesture.

:exclude-other-handlers provides a way of saying "this translator implements the entire contract for the presentation it matches".

See the macro "**define-presentation-action**", page 179.

:blank-area

Boolean option specifying whether the handler is active when the mouse cursor is over areas of the screen in which no presentations are displayed; the default is **nil.**

To ensure that handlers intended to be active only in blank areas are not active over displayed presentations, use the **dw:no-type** presentation type as the [*from-*]*presentation-type* positional argument to the handler.

:do-not-compose

Boolean option specifying whether the value of *body* is computed to determine if the handler satisfies the current input context; the default is **nil.**

To see the need for this option, let's consider the default behavior. For example, if 1) you have a translating mouse handler that returns integer objects; 2) the mouse cursor is currently over the handler's *from-presentation-type* ; 3) any shift keys modifying the mouse gesture the handler is on are pressed; and 4) the current input context is for integers, the default system behavior would be to determine what the body of the handler returns. If it returns anything other than a single value of **nil**, then the handler is applicable; this fact is indicated in the mouse documentation line and the presentation is highlighted (if it's not already).

Now, if the input context in this situation was for odd integers, rather than for any integer – that is, `(accept '((and`

`integer ((satisfies oddp)))))` – by default this handler would still be run to see if it returns an *odd* integer, that is, that the returned object will satisfy the input context requirements. Only if this is the case will the handler be available. This is the motivation for the default behavior.

However, some translating handlers have side effects, for example, popping up a menu or asking a question. It is unlikely that you want such events occurring merely when a user of your program waves the mouse over a presentation. You want this behavior suppressed until the user actually clicks on the presentation. `:do-not-compose t` is how you express this intent.

As a general rule, avoid defining translators that have side effects. One way of doing this is by defining side-effecting handlers explicitly, with **define-presentation-action.**

arglist The argument list for the body of the handler. The argument list consists of one positional argument, the object that the mouse cursor is over, and keyword arguments from a predefined set.

The following predefined keywords are available for inclusion in the argument list to a mouse handler body. Their inclusion makes the named parameters available for use in the body. The parameter list can specify only those keywords that are explicitly used.

input-context
The current presentation-input context.

presentation
The presentation instance that the mouse cursor is over.

handler The handler object of which the body is a part.

mouse-char
The mouse character that triggered the handler. (This keyword cannot be used in the **:tester** function parameter list.)

window The window object in which the current presentation occurs.

The *body* of your translator must return at least one value, the presentation object. Optionally, it can also return keyword-value pairs that you define. In this case, you must return the presentation type of the object as well. The object is the first item returned, its presentation type the second; these are followed by the keyword-value pairs.

One predefined keyword is available, **:activate**. Supplied with **nil**, the activation of input entered via this handler is suppressed, with **t** it's promoted. For an example: See the macro **define-presentation-translator**, page 185.

The values returned by the translator will be used to construct a presentation blip. You do not make the blip; the handler takes care of this automatically. Any keywords the translator returns are included in the options field of the blip. Options can be extracted from blips with the **dw:presentation-blip-options** function. For an overview of this and related functions: See the section "Overview of Presentation Input Blip Facilities", page 78.

For an overview of **define-presentation-to-command-translator** and related facilities: See the section "Overview of Top-Level Facilities for User Interface Programming", page 21. For an overview of mouse handler definition facilities: See the section "Overview of Mouse Handler Facilities", page 39. For information on handler lookup and performance issues: See the section "How Mouse Handlers Are Found", page 42.

dw:define-program-command (*name program-name* &rest *options* &key (*keyboard-accelerator* **nil**) (*menu-accelerator* **nil**) (*menu-level* **'(:top-level)**) &allow-other-keys) *arglist* &body *body* *Macro*

Defines a Command Processor command for a program created with **dw:define-program-framework**. The definition generates two internal methods for the program flavor, one to parse the command and one to execute the command. These methods provide lexical access to your program's state variable both in the body of the command definition and in the command argument list; that is, you may use state variables as arguments.

name The name given to the command. To distinguish command names from other kinds of names, we recommend that the prefix `com-` be used, for example `com-exit-program`. The user-visible command does not include the prefix; in the above example, the user-visible command is `Exit Program`.

Like other commands, those you define using **dw:define-program-command** occupy the function namespace.

program-name
The symbol or string naming the program flavor (created by **dw:define-program-framework**) for which the command is being written.

:keyboard-accelerator
Specifies the key used to invoke the command; the default is **nil**. For example, if you are writing an *Exit Program* command, you might wish to specify #\e, the E key, as the keyboard accelerator.

This option may not be used if **nil** is specified for the **:kbd-accelerator-p** option to the **:command-table** keyword for **dw:define-program-framework**: See the macro "**dw:define-program-framework**", page 124.

:menu-accelerator
Specifies whether the command identifier is displayed in a command menu pane for the program; the default is **nil**.

To make the command available in a menu, supply a value of **t** or a string. **t** causes the user-visible name of the command to be displayed. If you provide a string, it's displayed instead of the user-visible name.

Note that the program frame must include a pane of the **:command-menu** type in order for the command identifier to be displayed: See the macro "**dw:define-program-framework**", page 124.

:menu-level
Specifies the command menu in which the command is to be displayed. You need to use this option explicitly only when more than one command menu pane has been specified in the **dw:define-program-framework** macro for your program. (See the macro "**dw:define-program-framework**", page 124.)

Additional keyword options to **dw:define-program-command** are the same as those documented under *name-and-options* in the dictionary entry for **cp:define-command**.

arglist The list of command arguments. Each element of the list is itself a list of the form (*arg-name presentation-type options*) where *arg-name* is the name of the argument; *presentation-type* is the presentation-type of the argument; and *options* are keyword options to the argument.

Permissible options are the same as those documented under *arguments* in the dictionary entry for **cp:define-command**.

For an overview of **dw:define-program-command** and related facilities: See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.

dw:define-program-framework *name* &key *pretty-name* (*command-definer* **nil**) (*command-table* **nil**) (*top-level* **'(dw:default-command-top-level)**) (*command-evaluator* **nil**) (*panes* **'(dw::main :listener)**) *selected-pane* *query-io-pane terminal-io-pane label-pane* (*configurations* **nil**) (*state-variables* **nil**) (*select-key* **nil**) (*system-menu* **nil**) (*size-from-pane* **nil**) *help* *Macro*

Defines a flavor specifying the screen interface, command interface, and state variables for a program.

name The name given to the program flavor created by **dw:define-program-framework**.

:pretty-name
Specifies the user-visible (that is, displayed) name of the program. If this option is not supplied, the displayed name is the program-flavor name specified by the *name* argument, but with hyphens removed and initial caps (for example, "my-program" becomes "My Program").

:command-definer
Specifies the symbol to be used when defining program commands. In the typical case, this option is supplied with a value of **t** (that is, `:command-definer t`; this results in the creation of a program command-definition macro invoked with the symbol *define-program-name-command*, where *program-name* is the *name* argument supplied to **dw:define-program-framework**.

The command-definition macro so created has the same syntax as **dw:define-program-command**, but with one exception: you do not have to supply the *program-name*. (See the macro **dw:define-program-command**, page 122.)

The **:command-definer** option defaults to **nil**, in which case no command-definition macro is created, and you must use **dw:define-program-command**.

:command-table

Specifies a list of options to the **cp:make-command-table** function. The latter form is used by **dw:define-program-framework** to do basic command table management. When supplied with the **:command-table** keyword, permissible options are limited to two: **:inherit-from** and **:kbd-accelerator-p.**

Supplying the name (symbol or string) of a command table to the **:inherit-from** option makes all the commands in that table available during the running of your program. For example, supplying a value of `"global"` or `"user"` results in all the commands in the global or user command table, respectively, being included in the application command table.

If your frame includes an accept-values pane, then one of the values to the **:inherit-from** option must be "accept-values-pane". (For more information on accept-values panes: See the macro "**dw:define-program-framework**", page 124.

Supplying a value of **t** to **:kbd-accelerator-p** allows you to specify single-key accelerators for program commands; the default is **nil**. Keyboard accelerators are specified via the command-definition macro created through the **:command-definer** option to **dw:define-program-framework**; via **dw:define-program-command**; or via **cp:define-command-accelerator**. Keyboard accelerators are also inherited when you use the **:inherit-from** option.

:top-level

Specifies the command loop function to be used for the program. The default provides the standard command loop.

For information on facilities available for writing your own command loop function: See the section "Overview of Advanced Command Facilities", page 32.

:command-evaluator

Specifies function called after a command is read. Arguments passed to the called function are the program instance, the command, and any command arguments. At some point before, during, or after the execution of application-specific tasks, the evaluator function should (**apply** *<command> <arguments>*).

:panes Specifies a list of panes to be included in the program frame. Each element of the list is itself a list of the form *(pane-name pane-type options)*. Six types of panes are available:

:title Pane for display of the program title (**:pretty-name** is the default).

:command-menu
Pane for menu of program commands.

:display Pane for display of application-generated output.

:interactor
Pane for interactive input/output.

:listener Similar to an interactor, but taller. (Use this pane when you want the interaction history to be visible.)

:accept-values
Pane providing the features and services of a **dw:accept-variable-values** menu. (If your frame includes an **:accept-values** pane, supply "accept-values-pane" as one of the values with the **:inherit-from** keyword to the **:command-table** option: See the macro "**dw:define-program-framework**", page 124.

The appearance and behavior of panes can be modified with a variety of keyword options; not all are appropriate for use with every pane type. Each option is listed below with a description of its purpose and an indication of the pane types for which it is appropriate:

:default-character-style
Specifies list of the form *(family face size)* to specify the style of characters displayed in the pane. The default style for **:display** panes is **(:fix :roman :normal)**; for **:title** panes **(:sans-serif :bold :large)**; and for **:command-menu** panes **(:jess :roman :normal)**. (For more information on available styles: See the section "Character Styles" in *Symbolics Common Lisp: Language Concepts*.)

This option is applicable to all pane types.

:height-in-lines
Specifies integer to fix the height of the pane to a number of text lines. The actual height in pixels

depends on the **:default-character-style** for the pane (see above).

This option is applicable to the **:title**, **:display**, **:interactor**, **:listener**, and **:accept-values** pane types.

:size-from-output
Boolean option specifying whether a pane is sized according to the space needs of output to that pane; the default is **t** for **:command-menu** and **:accept-values** pane types, **nil** for other pane types.

This option is applicable to the **:title**, **:command-menu**, **:display**, and **:accept-values** pane types.

:typeout-window
Boolean option specifying whether a typeout (pull-down) window for ***terminal-io*** appears within the pane; the default is **nil**.

This option is applicable to **:display**, **:interactor**, and **:listener** pane types.

:redisplay-string
Specifies a string written to the pane (starting at top) whenever the pane is redisplayed. This option is mutually exclusive with the **:redisplay-function** option (see below).

:redisplay-string is applicable to the **:title** and **:display** pane types.

:redisplay-function
Specifies name of user-defined function that runs whenever the pane is redisplayed. This option is mutually exclusive with the **:redisplay-string** option (see above).

The redisplay function may be written either as a generic function (using **defmethod**) to the program flavor or as a regular function (using **defun**). The function is passed two arguments: the current instance of the program flavor and the stream on which to do output.

:redisplay-function is applicable to the **:title** and **:display** pane types.

:redisplay-after-commands
Boolean option specifying whether output to the pane is to be redisplayed after each command is executed; the default is **t**.

This option is applicable to the **:title**, **:display**, and **:accept-values** pane types.

The following options are applicable only to the **:command-menu** pane type:

:menu-level
Specifies a unique identifier for each command menu in the program when more than one command menu is needed. The default value (for a single command menu) is **:top-level**.

:rows Specifies a list, each element of which is a list of command names (strings) to be included in the same row.

:columns
Specifies a list, each element of which is a list of command names (strings) to be included in the same column.

:equalize-column-widths
Boolean option specifying whether the widths of columns containing command names be equal; the default is **nil** (widths adjusted according to size of the outout in each column).

:center-p Boolean option specifying whether command names are centered (left-right) in the command menu; the default is **nil** (flush-left).

The following options are applicable only to the **:display** pane type:

:flavor Specifies the pane flavor to use for this pane; the default is **dw::dynamic-window-pane**.

:incremental-redisplay
Boolean option specifying whether redisplayed information is limited to items that have changed

since the last redisplay, rather than the entire pane. If **t**, you must write the appropriate redisplay function (see **:redisplay-function** above).

For information on incremental redisplay: See the section "Overview of Advanced Presentation Output Facilities", page 63.

The following option is applicable only to the **:accept-values** pane type:

:accept-values-function
Specifies a function for creating a **dw:accept-variable-values**-like display; it defaults to an internal one that operates on program state variables.

The function may be written either as a generic function (using **defmethod**) to the program flavor or as a regular function (using **defun**). The function is passed two arguments: the current instance of the program flavor and the stream for I/O.

The multiple-**accept** display is created by wrapping the body of the function you write in a **dw:accepting-values** macro: See the macro **dw:accepting-values**, page 175. The wrapping is done for you by **dw:define-program-framework**. The general form of the function you write is

```
(defmethod (my-avv-function program) (stream)
  (setq state-var-1 (accept ...)
  (setq state-var-2 (accept ...)
  (setq state-var-3 (accept ...)
  ...)
```

For an example, see the program `avv-pane-test` in the file `sys:examples;define-program-framework.lisp`

The **:default-character-style** keyword option is inherited from **dw:dynamic-window** (via **dw::dynamic-window-pane** on which all program panes are based by default). Many more keyword options exist, most of which, however, are inappropriate for use with panes created via

dw:define-program-framework. Among keywords that are appropriate, the following are most useful:

:blinker-p
Boolean option specifying whether a blinker appears in the pane. This option defaults to **t** for the **:interactor** and **:listener** pane types, **nil** for other pane types.

:more-p Boolean option specifying whether *more processing* is enabled. More processing lets the user control scrolling of character output to a window. The default is **t** for the **:display** and **:listener** pane types, **nil** for other pane types.

:end-of-page-mode
Specifies what happens when queued output exceeds the space available in the current viewport of the pane. There are three possibilities:

:scroll causes the pane to scroll automatically to accommodate the output.

:truncate causes scrolling to be the responsibility of the user, who must press the SCROLL key to see more output.

:wrap causes new output to appear at the top of the pane, rather than at the bottom as in the case of **:scroll** or **:truncate**.

:scroll-factor
Specifies the number of lines by which to scroll the pane when the **:end-of-page-mode** is **:scroll**.

:label Specifies string that appears as a label in the lower, left-hand corner of the pane (directly inside the border). The character style used is the default style for the pane. You may only use the **:label** option if not using the **:margin-components** option, described below.

:margin-components
Takes list of options specifying characteristics of pane margins. The default is for a 1-pixel-wide border and a margin between the border and displayed output to the pane of 4 pixels.

The defaults are implemented by the list `((dw:margin-borders) (dw:margin-white-borders 4))`. **dw:margin-borders** and **dw:margin-white-borders** are flavors for controlling the margin specifications of dynamic windows. For an overview of these and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

This option is applicable to all pane types.

:selected-pane
Designates pane selected (generally indicated by blinking cursor) when program is activated. If none is designated, this option defaults to an available pane in the following order of priority (highest to lowest): **:listener**, **:interactor**, **:display**.

:query-io-pane
Designates pane to which ***query-io*** is bound when program is active. If none is designated, this option defaults to an available pane in the following order of priority (highest to lowest): **:listener**, **:interactor**, **:display**.

:terminal-io-pane
Designates pane to which ***terminal-io*** is bound when program is active. If none is designated, this option defaults to an available pane in the following order of priority (highest to lowest): the pane with a **:typeout-window** option (see above), a **:listener** pane, a **:display** pane.

:label Designates pane on which program label is displayed if the program does not have a **:title** pane. If none is designated, this option defaults to an available pane in the following order of priority (highest to lowest): **:listener**, **:interactor**, **:display**.

:configurations
Specifies the layout and sizes of panes within the program frame. Program frames are built on a more basic type of window known as a *constraint frame*. The *constraint language* used to specify the layout and sizes of panes in a constraint frame is documented elsewhere: See the section "Specifying Panes and Constraints" in *Programming the User Interface, Volume B*.

In the default configuration, panes are vertically stacked in a single column and in the order specified by the **:panes** option (see above).

:state-variables
Specifies a list of program variables whose states are preserved between activations of the program. Each variable is itself a list consisting of the variable name and, optionally, a default value and presentation type. State variables are implemented as writeable instance variables to the program flavor.

:select-key
Specifies a character for selecting the program via the SELECT key.

:system-menu
Boolean option specifying whether the program appears on the System Menu. If **t**, the program appears both in the *Programs* column of the top-level menu and in the *Create* second-level menu; the default is **nil**.

:size-from-pane
Specifies the pane on which to base the size of the program frame; the default is **nil**.

:help Specifies the help message displayed when the HELP key is pressed while the program is selected. The value of this option can be either a string or a function. If it's a string, the string is displayed when the user presses HELP.

If it's a function, the function receives three arguments: the program flavor, the stream to which the help message should be output, and the string that has been typed so far.

For an overview of **dw:define-program-framework** and related facilities: See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.

For an example and additional information on the use of certain options to **dw:define-program-framework**, particularly those implementing the command interface: See the section "User Interface Application Example", page 91. More examples are available in the file sys:examples;define-program-framework.lisp.

dw::find-program-window *program-name* &rest *options* &key *(create-p* **t**) *(selected-ok* **t**) *program-state-variables* &allow-other-keys *Function*

Returns the window (frame) of a program (created via **dw:define-program-framework**).

program-name
The name of the program.

:create-p Boolean option specifying whether to create an instance of the program if one does not exist; the default is **t**.

:selected-ok
Boolean option specifying whether to return the program window if it is the currently selected activity; the default is **t**.

:program-state-variables
Specifies a list of initializations for the program's state variables. The list is of the form ((*<var-1>* *<val-1>*) (*<var-2>* *<val-2>*) ... (*<var-n>* *<val-n>*)).

If an instance of the program is created, its state variables are initialized according to this specification. If an instance already exists, its state variables are reset according to the specification.

Other keywords permitted are programmer-defined and system init options for the frame. If an instance of the program is created, it is initialized according to the keyword specifications.

For an overview of **dw::find-program-window** and related facilities: See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.

dw:get-program-pane *name* *Function*

Returns specified pane in a program frame created with **dw:define-program-framework**.

name The name of the pane as specified in the **:panes** option to **dw:define-program-framework**.

For an overview of **dw:get-program-pane** and related facilities: See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.

dw:*program* *Variable*

Bound to the currently active instance of a program flavor (created via **dw:define-program-framework**).

For an overview of **dw:*program*** and related facilities: See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.

dw:program-command-table *program* *Generic Function*

Returns the command table used by an instance of a program flavor (created via **dw:define-program-framework**).

program The program instance. (The currently active program instance can be accessed as the value of **dw:*program***.)

For an overview of **dw:program-command-table** and related facilities: See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.

dw:*program-frame* *Variable*

Bound to the program frame associated with the current instance of a program flavor (created via **dw:define-program-framework**).

Use this variable for access to the program frame from a generic function or method of the program flavor, or from a program command definition macro.

Example (for a program flavor named "my-program"):

```
(define-my-program-command (com-enable-secondary-commands
                                :menu-accelerator "More Commands"
                                :menu-level :main)
                           ()
  (send dw:*program-frame* :set-configuration 'secondary))
```

For access to a particular pane of the program frame, send a **:get-pane** message to **dw:*program-frame*** or use **dw:get-program-pane**.

For an overview of **dw:*program-frame*** and related facilities: See the section "Overview of Top-Level Facilities for User Interface Programming", page 21.

PART III.

Dictionary of Command Processor Facilities

12. Dictionary Notes

This dictionary includes reference documentation for both the basic and advanced Command Processor facilities listed in the following two tables:

Table of Basic Command Facilities

Command Definition Facilities
cp:define-command

Command Processor Interface Facilities
cp:execute-command
cp:build-command
cp:*last-command-values*

Table of Advanced Command Facilities

Command Loop Management Facilities
cp:read-command
cp:read-command-or-form
cp:read-command-arguments
cp:yank-and-read-full-argument-command
cp:read-full-command
cp:read-accelerated-command
cp:echo-command
cp:unparse-command
cp:define-command-and-parser
cp:turn-command-into-form
cp::*default-blank-line-mode*
cp::*default-dispatch-mode*
cp::*default-prompt*

Command Table Management Facilities
cp:*command-table*
cp:make-command-table
cp:find-command-table
cp:install-commands
cp:delete-command-table
cp:command-in-command-table-p

Command Accelerator Facilities
cp:define-command-accelerator

In the dictionary, the facilities are arranged in alphabetical order.

For conceptual documentation: See the section "Overview of Command Processor Facilities", page 31.

13. The Facilities

cp:build-command *command-name* &rest *command-arguments* *Function*

Constructs the internal representation of a Command Processor command within a **define-presentation-to-command-translator** macro; when the defined translator is activated, the command is invoked.

command-name
Symbol or string naming the command to invoke; if a string, it must be in the command table to which **cp:*command-table*** is currently bound.

command-arguments
Positional and keyword arguments to the named command.

Examples:

```
(cp:build-command "show file" "test-data.text")

(cp:build-command 'si:com-load-system "unifier"
                  :condition :always :automatic-answer t)
```

For an overview **cp:build-command** and related facilities: See the section "Overview of Basic Command Facilities", page 31.

cp:*command-table* *Variable*

Bound to the current command table, that is, the one used by the Command Processor when reading commands.

For an overview of **cp:*command-table*** and related facilities: See the section "Overview of Command Table Management Facilities", page 33.

cp:command-in-command-table-p *command-symbol command-table* *Function*

Determines the presence of a command in a Command Processor command table. The function returns **t** if the command is either in the specified command table, or in a table from which the specified table inherits.

command-symbol
The command symbol.

command-table
The command table.

For an overview of **cp:command-in-command-table-p** and related facilities: See the section "Overview of Command Table Management Facilities", page 33.

cp::*default-blank-line-mode* *Variable*

The default command processor blank line mode for **cp:read-command** and **cp:read-command-or-form**. This is a keyword that determines what action the command processor takes when you type a blank line:

:reprompt	Redisplay the prompt, if any. This is the default.
:beep	Beep.
:ignore	Do nothing.

The blank line mode used in Lisp Listeners and **zl:break** loops is the value of **cp:*blank-line-mode***.

cp::*default-dispatch-mode* *Variable*

The default command processor dispatch mode for **cp:read-command-or-form**; a keyword. Possible values are **:form-only**, **:form-preferred**, **:command-only**, and **:command-preferred**. For the meanings of these values: See the section "Setting the Command Processor Mode" in *User's Guide to Symbolics Computers*. The default is **:command-preferred**.

The dispatch mode used in Lisp Listeners and **zl:break** loops is the value of **cp:*dispatch-mode***.

cp::*default-prompt* *Variable*

The default command processor prompt option for **cp:read-command** and **cp:read-command-or-form**. The value of this variable is passed to the input editor as the value of the **:prompt** option. The value can be **nil**, a string, a function, or a symbol other than **nil** (but not a list): See the section "Displaying Prompts in the Input Editor" in *Reference Guide to Streams, Files, and I/O*. The default is "Command: ".

The prompt used in Lisp Listeners and **zl:break** loops is the value of **cp:*prompt***.

cp:define-command *name-and-options arguments* &body *body* *Macro*

Defines a Command Processor command.

name-and-options

Either the symbol to be used as the command name or a list whose first element is the name symbol and succeeding elements are alternating keyword-value pairs. To distinguish command names from other kinds of names, we recommend that the prefix *com-* be used; the user-visible command name will not include the prefix.

Following are the keywords that may be included in the *name-and-options* list:

:name Specifies the string serving as the user-visible command name. The default name is the result of calling **string-capitalize-words** on the print name of the symbol that is the first element of the *name-and-options* list; if the name begins with the substring "com-", the substring is omitted.

This option is useful for special capitalization within command names.

:command-table
Specifies the command table, or a symbol/string naming the command table, into which the command is to be installed. For example, to install a command into the "Global" command table, you could supply `'global`, `"global"`, or the form `(cp:find-command-table 'global)`.

This option is evaluated. If not supplied, the command is not installed in a command table; to install the command subsequently, use the function **cp:install-commands**.

A supported synonym for the **:command-table** option is **:comtab**.

For more information on command tables: See the section "Command Processor Command Tables" in *Programming the User Interface, Volume B.* For information on command table management facilities: See the section "Overview of Advanced Command Facilities", page 32.

:comtab Specifies the command table, or a symbol/string naming the command table, into which the command is to be installed. For example, to install a command into the "Global" command table, you could supply `'global`, `"global"`, or the form `(cp:find-command-table 'global)`.

This option is evaluated. If not supplied, the command is not installed in a command table; to install the command subsequently, use the function **cp:install-commands**.

A supported synonym for the **:comtab** option is **:command-table.**

For more information on command tables: See the section "Command Processor Command Tables" in *Programming the User Interface, Volume B.* For information on command table management facilities: See the section "Overview of Advanced Command Facilities", page 32.

:explicit-arglist
Specifies explicitly the argument list of the function implementing the body of the command. By default, the argument list of this function corresponds to the arguments specified as arguments to the command.

Typically, you do not need this option; however, it is useful when you want the command body to receive its arguments as an &rest *arg.*

:provide-output-destination-keyword
Boolean option specifying whether to provide the **:output-destination** keyword. The default is **t**; this allows the user of the command to redirect the output of the command to a place other than the screen.

To override the default action (if, for example, your command does not produce any useful output), specify a value of **nil** for this option.

:values Boolean option specifying whether the command returns values; the default is **nil.**

(Note that even if this option is **nil**, the values returned by executing the command are stored in **cp:*last-command-values***.)

arguments
The list of command arguments. Each element of the list is itself a list of the form (*arg-name presentation-type options*) where *arg-name* is the name of the argument; *presentation-type* is the presentation-type of the argument; and *options* are keyword options to the argument. (Note that *presentation-type* is evaluated, and should typically be quoted; for example, 'integer or 'pathname.)

Following are the keywords that may be included in the argument specification list:

:documentation
Specifies a string to use as the help message for the argument. The message is displayed if, after typing the command name and any preceding positional arguments, the user presses the HELP key.

Also displayed when the HELP key is pressed is information about the expected type of input. Such messages appear after the help messages you specify using this option. They are generated by the underlying **accept** functions used for doing command input.

:prompt Specifies either a string to be used as a prompt for the argument or a form that when evaluated returns such a string. If a default argument is displayed, the prompt appears before the default.

:prompt-mode
Specifies either the **:normal** or **:raw** mode for prompts. If **:normal**, the prompt you supplied using the **:prompt** option (or the default prompt) is transformed into a prompt suitable for a command line – it is surrounded with parentheses, the default is appended, and so on. If **:raw**, your prompt is used without transformation.

:default Specifies a form to be evaluated to determine the default value for the argument.

If no default is specified, the current default – taken from the presentation history – for the presentation type of the argument is used. (Access to the current default for a presentation type is available through **dw:presentation-type-default**.)

:mentioned-default
For a keyword argument, specifies a form to be evaluated and used as the default value for the argument, but only if the user types the argument name.

The form can refer to parameters defined for any positional arguments (but not keyword arguments) specified prior to this argument specification. At the time the form is evaluated, these parameters are bound to the values of arguments already accepted.

The default value used depends on what combination of **:default** and **:mentioned-default** options is supplied:

Both
: Use the value of **:mentioned-default** if the user types the name of the argument; otherwise, use the value of **:default.**

:mentioned-default only
: If the user types the argument name, use the value of **:mentioned-default**; otherwise, the default is **nil.**

:default only
: Use the value of **:default.**

Neither
: If the user does not type the argument name, the default is **nil.** If the user types the name, the argument has no default and the user has to supply a value.

:when Specifies a predicate to be evaluated at command-line reading time. This option provides simple control over what arguments the command line reads; if the predicate returns **nil,** the argument is not read. The predicate can refer to any positional arguments already read.

Example:

```
(cp:define-command (com-when-example)
  ((type '((member integer any)) :default 'integer)
   (number 'integer :when (eq type 'integer)))
  ...)
```

:name Specifies a string serving as the user-visible name of the argument. Note: this option is only valid for keyword arguments.

Example:

```
(cp:define-command (com-key-name
                    :command-table 'user)
  (&key (arg1 '((integer 1 10))
        :name "Copies"
        :prompt "Number of copies (1-10)"))
   (print arg1))
("Key Name" . COM-KEY-NAME)

==>Key Name (keywords) :Copies
(Number of copies (1-10)) 2
2
```

:default-type

Specifies the default presentation type of the object accepted as an argument value.

This option is useful only when used in conjunction with the **:default** option. When the type of the argument being read is ambiguous – for example, if you are using an **or** presentation type – specifying the **:default-type** option tells the Command Processor how to present the given default; that is, which presentation-type printer to use.

Example:

```
(number-or-string '((or integer string))
  :default 3 :default-type 'integer)
```

:provide-default

Boolean option specifying whether a default is provided for the argument. The default value for this option is `(not (null <default>))`. Consequently, `:default nil` implies `:provide-default nil` unless, as a special case, the presentation type being read is **boolean**.

This keyword is typically useful only if, as in the case of boolean arguments, **nil** happens to be a meaningful default for the type being read.

:display-default

Specifies whether the default is printed in the prompt. The default value for this option is **t**; however, if the **:provide-default** option is **nil**, no default is printed.

:confirm Boolean option specifying whether the argument requires confirmation by the user; the default is **nil**.

When `:confirm t` is specified, if the command line is terminated before the argument has been read, the prompt for the argument is printed (as well as the prompts and defaults for all unread arguments before this one on the command line), and the user must again terminate the command line.

This mechanism ensures that the user is aware that the argument is being specified automatically, and that the default value, if available, is displayed. (All destructive system commands, for example, Delete File, use `:confirm t` for their critical arguments.)

For an overview of **cp:define-command** and related facilities: See the section "Overview of Basic Command Facilities", page 31.

cp:define-command-accelerator *name command-table characters options arglist* &body *body* *Macro*

Defines single-key accelerators for Command Processor commands.

name Name for this accelerator.

command-table
Command table in which command and accelerator are included.

characters
List of characters (not necessarily more than one) serving as the single-key accelerators.

options List of keyword-value pairs. Possible keywords include:

:argument-allowed
Boolean option specifying whether this accelerator is allowed to take numeric arguments (for example, `c-3`). The default depends on whether you provide an *arglist*, **t** if you do, **nil** if you don't.

:activate Boolean option specifying whether the command defined by this accelerator executes immediately when the accelerator is typed; the default is **t**. If **nil**, the command requires confirmation and, possibly, additional args.

:echo Boolean option specifying whether the command

defined by this accelerator echoes on the command line as if it were typed. The default is the value supplied to the **:activate** option; this is because in the `:activate nil` case, the command is visible after you are finished editing and need not be repeated.

arglist List of arguments to the accelerated command. If **:argument-allowed** is **nil**, this arglist should be **nil** (no arguments allowed).

If **:argument-allowed** is **t**, the accelerator receives two arguments, *arg-p* and *arg*. *arg-p* means whether or not the user gave an argument to this accelerator; *arg* is the numeric *arg*. In this case, the *arglist* is typically just `(arg-p arg)`, but you can put anything here you want. This is just so that your *body* can make reference to these symbols under the names you chose.

body A form that returns a command (using **cp:build-command**, typically). It can make reference to the symbols bound in *arglist*.

A typical body might be:

```
(cp:build-command 'command-one
        :format (if arg-p :brief :detailed))
```

For an overview of **cp:define-command-accelerator** and related facilities:
See the section "Overview of Command Accelerator Facilities", page 34.

cp:define-command-and-parser *name-and-options arglist parser* &body *body* *Macro*

Defines a Command Processor command and command line parser.

name-and-options
Either the symbol to be used as the command name or a list whose first element is the name symbol and succeeding elements are alternating keyword-value pairs. To distinguish command names from other kinds of names, we recommend that the prefix *com-* be used; the user-visible command name will not include the prefix.

Permissible keywords are the same as those listed under *name-and-options* in the dictionary entry for **cp:define-command**.

arglist The argument list of the function that implements the body of the command. It is a normal, Common Lisp argument list.

parser A form used to parse the command's arguments. This form has lexical access to the internal functions **cp:read-command-argument, cp:read-keyword-arguments,** and **cp:assign-argument-value.** It should use these functions to do the actual reading and assigning of values to command arguments:

cp:read-command-argument *presentation-type &rest options*
A **flet**ted function within **cp:define-command-and-parser.** *presentation-type* is the type of the argument. *options* are all options acceptable in a command argument specification to **cp:define-command.**

cp:read-keyword-arguments *&rest keyword-specs*
A **macrolet**ted macro within **cp:define-command-and-parser.** *keyword-specs* are command argument specifications identical to those you would use if you were writing the command using **cp:define-command.** Even if there are no keyword arguments, the parser should end with **cp:read-keyword-arguments**; any automatically generated keywords (for example, **:output-destination**) can thereby be read.

cp:assign-argument-value *argument-name value*
A **macrolet**ted macro within **cp:define-command-and-parser.** *Argument-name* is a symbol naming a command argument; *value* is its value. Each *argument-name* should correspond to an argument in **arglist** above.

Example:

```
(cp:define-command (com-this-is-a-test
        :command-table 'user)
    ((file 'pathname :default nil :prompt "file")
     &key
     (integer 'integer :default 17
        :mentioned-default 3 :prompt "the number"))
   (loop for i from 0 to integer do
          (print file)))

;;;is equivalent to
```

```
(cp:define-command-and-parser (com-this-is-a-test
          :command-table 'user)

        ;; The arglist of the function.
        ;; Note the presence (and need for) the
        ;; default value for INTEGER in the
        ;; argument list.
        (file &key (integer 17))

  ;; The argument parser.  It's just one big PROGN.
  ;; Note that it ends with read-keyword-arguments.
  (progn (cp::assign-argument-value file
           (cp::read-command-argument 'pathname
               :default nil :prompt "file"))
         (cp::read-keyword-arguments
           (integer 'integer :default 17
               :mentioned-default 3 :prompt "the number")))

  ;; The body of the command.
  (loop for i from 1 to integer do (print file)))
```

To see other examples, try **macroexpand**ing some **cp:define-command** definitions; they expand into **cp:define-command-and-parser** definitions.

For an overview of **cp:define-command-and-parser** and related facilities: See the section "Overview of Command Loop Management Facilities", page 33.

cp:delete-command-table *command-table-or-name* *Function*

Removes a Command Processor command table from the command table registry.

command-table-or-name

A command table object or the name (symbol or string) of a command table.

For an overview of **cp:delete-command-table** and related facilities: See the section "Overview of Command Table Management Facilities", page 33.

cp:echo-command *command-name arguments* *Function*

Echoes a Command Processor command and its arguments to ***standard-output***. (The echoed command is presented "acceptably", that is, in such a manner that is can subsequently be parsed by **accept**.)

command-name
The command name (symbol).

arguments
A list of command arguments.

For an overview of **cp:echo-command** and related facilities: See the section "Overview of Command Loop Management Facilities", page 33.

cp:execute-command *command-name* &rest *command-arguments* *Function*
Invokes a Command Processor command from within a program.

command-name
Symbol or string naming the command to invoke; if a string, it must be in the command table to which **cp:*command-table*** is currently bound.

command-arguments
Positional and keyword arguments to the named command.

Examples:

```
(cp:execute-command "show file" "test-data.text")

(cp:execute-command 'si:com-load-system "unifier"
                    :condition :always :automatic-answer t)
```

For an overview **cp:execute-command** and related facilities: See the section "Overview of Basic Command Facilities", page 31.

cp:find-command-table *name* &key (*if-does-not-exist* **:error**) *Function*
Returns the Command Processor command-table object specified by the command-table name.

name The name (symbol or string) of the command table.

:if-does-not-exist
Specifies what happens if the named command table is not found. Three values are possible:

nil The function returns **nil**.

:error An error message is returned and the debugger is entered; this is the default.

:create A new command table named *name* is created and returned.

For an overview of **cp:find-command-table** and related facilities: See the section "Overview of Command Table Management Facilities", page 33.

cp:install-commands *command-table new-commands* *Function*

Installs Command Processor commands into a command table.

command-table

Name (symbol or string) of the command table receiving the new commands. If it does not already exist, a command table will be created.

new-commands

A list of commands to install.

For an overview of **cp:install-commands** and related facilities: See the section "Overview of Command Table Management Facilities", page 33.

cp:*last-command-values* *Variable*

List of values returned by the most recently executed Command Processor command.

For an overview **cp:*last-command-values*** and related facilities: See the section "Overview of Basic Command Facilities", page 31.

cp:make-command-table *name* &rest *init-options* &key (*if-exists* **:error**) &allow-other-keys *Function*

Creates and returns a Command Processor command table object.

name The name (symbol or string) of the command table.

init-options

Keyword-values pairs that are init options to the (internal) command-table flavor from which the command table object is created. Permissible options and values are as follows:

:inherit-from

Specifies a list of command tables from which to inherit commands.

:command-table-delims

Specifies a list of characters to use as delimiters of words in command names for commands in the table. The default list is `(#\Space #\Tab #\Return)`.

:command-table-size

An initial estimate of the number of commands the table will include (to preclude the table from having to grow substantially).

:kbd-accelerator-p
Boolean option specifying whether single-key accelerators may be used for commands; the default is **t**.

:accelerator-case-matters
Boolean option specifying whether single-key accelerators, if allowed, are case sensitive; the default is **nil**.

:if-exists Specifies what happens if the command table named *name* already exists. Four values are possible:

nil No new command table is made and the existing command table is returned.

:supersede
The new command table is made and replaces the old command table.

:update-options
The existing command table remains but its options are updated to those newly specified in the call to **cp:make-command-table**.

:error An error message is returned and the debugger is entered.

Example:

```
(cp:make-command-table "shell-cmds" :inherit-from '("user")
                                    :kbd-accelerator-p nil)
```

For an overview of **cp:make-command-table** and related facilities: See the section "Overview of Command Table Management Facilities", page 33.

cp:read-accelerated-command &key (*command-table* **cp:*command-table***) (*stream* ***query-io***) (*help-stream* **stream**) (*echo-stream* **stream**) (*whostate* **nil**) (*prompt* **nil**) (*command-prompt* **cp::*full-command-prompt***) (*full-command-full-rubout* **nil**) (*special-blip-handler* **nil**) (*timeout* **nil**) (*input-wait* **nil**) (*input-wait-handler* **nil**) (*form-p* **nil**) (*handle-clear-input* **nil**) (*catch-accelerator-errors* **t**) (*unknown-accelerator-is-command* **nil**) *Function*

(*unknown-accelerator-tester* **nil**)
(*unknown-accelerator-reader* **nil**)
(*unknown-accelerator-reader-prompt* **nil**)
(*abort-chars* **nil**) (*suspend-chars* **nil**) (*status* **nil**)
(*intercept-function* **nil**) (*window-wakeup* **nil**)

Reads a Command Processor command input as a single-key accelerator.

The values returned by this function depend on whether a command or form is expected (see the **:form-p** option below). If the caller is expecting a command (**:form-p** is **nil**), the values returned are the command name, command arguments, and a flag. If the caller is expecting a form (**:form-p** is **t**), the values returned are the form and a flag.

Possible values for the returned flag are:

:command
A command was read.

:form A form was read.

:accelerator
An accelerator character was read.

:timeout A timeout expired.

:status The window's status changed.

:wakeup The window was asynchronously refreshed, selected, exposed, etc.

:unknown (or **nil**)
Something unknown was typed.

cp:read-accelerated-command accepts the following keyword options:

:command-table
Specifies the command table containing the accelerator; the default is the current binding of **cp:*command-table***.

:stream Specifies the stream from which to read the command; the default is ***query-io***.

:help-stream
Specifies the output stream for help messages; the default is the stream specified by the **:stream** option.

:echo-stream
Specifies the stream to which the input command is echoed; the default is the stream specified by the **:stream** option.

To suppress echoing, supply this option with #'**ignore**.

:whostate
Specifies a string to appear in the status line in place of "User Input".

:prompt Specifies a string to be used as the prompt, or a prompt option. (See the section "Displaying Prompts in the Input Editor" in *Reference Guide to Streams, Files, and I/O*.)

:command-prompt
Specifies a string to be used as the prompt if a command is to be read, that is, if the user types ":". The default is **cp::*full-command-prompt***, which is "Command: ".

:full-command-full-rubout
Boolean option specifying whether to return if CLEAR INPUT is pressed (or a series RUBOUTs back to the prompt) after m-X is typed. The default is **nil**, allowing the continuation m-X (extended) command parsing.

:special-blip-handler
Specifies a function called with mouse blips that are not presentation input blips. (See the section "Mouse Blips" in *Programming the User Interface, Volume B*.)

:timeout Specifies the length of time, in 60ths of a second, after which, if the user types nothing, **cp:read-accelerated-command** returns **:timeout** as the flag and **nil** for the other values.

:input-wait
Specifies a function testing for some condition while in the input-wait state. If this condition occurs, the **:input-wait-handler** is invoked.

:input-wait-handler
Specifies a function called after a condition satisfying the **:input-wait** function occurs.

:form-p Boolean option specifying whether a form or command is expected; the default is **nil**. If **t**, the function returns an evaluable form rather than the command name and arguments.

:handle-clear-input

Boolean option specifying whether #**clear-input** is treated specially; the default is **nil**. If **t** and the CLEAR INPUT key is pressed, the function clears the input buffer and reprompts.

:catch-accelerator-errors

Boolean option specifying that when an unknown accelerator character is typed, the function beeps and prints out a warning message. If **nil**, it signals the error flavor **cp::accelerator-error**; this is the default.

:unknown-accelerator-is-command

Specifies whether unknown accelerators are dispatched to the **:unknown-accelerator-reader** function.

The default is **nil**. Unknown accelerators that do not pass the **:unknown-accelerator-tester** function give errors (which may or may not get through to the user – see the **:catch-accelerator-errors** option).

If **t**, all unknown accelerators dispatch to the unknown-accelerator reader which should return a command.

A third value permitted for this option is **:alpha**, causing only unknown accelerators that are alphabetic characters to be dispatched to the unknown-accelerator reader.

:unknown-accelerator-tester

Specifies a function of one argument, the character typed, which should return something non-**nil** if this particular unknown accelerator is permitted. In this case, **:unknown** is returned as the flag and the value from this function is the first value. If **:form-p** is **nil**, the character is returned as the second value.

:unknown-accelerator-reader

Specifies a function of no arguments that should return a form. (The function can call **cp:read-command**, etc., but it should return a form.)

:unknown-accelerator-reader-prompt

Specifies a string to use as the prompt in this case, or a prompt option. (See See the section "Displaying Prompts in the Input Editor" in *Reference Guide to Streams, Files, and I/O.*)

:abort-chars
Specifies a list of "abort" characters; the default is **nil**.

If a list of characters is provided and the user types one, **sys:abort** is signalled.

:suspend-chars
Specifies a list of "abort" characters; the default is **nil**.

If a list of characters is provided and the user types one, a **break** loop is entered.

:status Specifies what happens if the window's status changes. Three values are permitted, **:selected**, **:exposed**, and **nil**.

If the value is **:selected** and the window is no longer selected, the function returns **:status**.

If the value is **:exposed** and the window is no longer exposed or selected, the function returns **:status**.

If the value is **nil**, the function continues to wait for input when the window is deexposed or deselected. This is the default.

:intercept-function
Specifies a function of one argument, a character, that gets called on each typed character that is one of **:abort-chars** or **:suspend-chars**.

:window-wakeup
Boolean option specifying whether to return **:wakeup** when an asynchronous window system condition like expose, select, or refresh occurs; the default is **nil**.

For an overview of **cp:read-accelerated-command** and related facilities:
See the section "Overview of Command Accelerator Facilities", page 34.

cp:read-command &optional (*stream* ***standard-input***) &key (*command-table* **cp:*command-table***) (*blank-line-mode* **cp::*default-blank-line-mode***) (*prompt* **cp::*default-prompt***) *Function*

Reads a Command Processor command from *stream*, terminated by RETURN or END.

If *stream* is not supplied or is **nil**, it defaults to ***standard-input***.

From the user's point of view, a command consists of a command name, positional arguments, and keyword arguments: See the section "Parts of a Command" in *User's Guide to Symbolics Computers.* **cp:read-command** offers completion over command names, keyword argument names, and some argument values, and it completes any unspecified command components when the command is terminated: See the section "Completion in the Command Processor" in *User's Guide to Symbolics Computers.*

cp:read-command prompts for arguments and gives information about what sort of values are expected. Some arguments have default values. The user can press HELP to see documentation appropriate to the current stage of entering the command: See the section "Help in the Command Processor" in *User's Guide to Symbolics Computers.* For a general description of how the user enters a command: See the section "Entering a Command" in *User's Guide to Symbolics Computers.*

If **:command-table** is supplied, it is a command table of the acceptable commands. The default command table is the value of **cp:*command-table***. The initial default is the "User" command table. See the section "Command Processor Command Tables" in *Programming the User Interface, Volume B.*

If **:blank-line-mode** is supplied, it is a keyword that determines what action the command processor takes when the user types a blank line:

:reprompt	Redisplay the prompt, if any.
:beep	Beep.
:ignore	Do nothing.

The default *blank-line-mode* is the value of **cp::*default-blank-line-mode***. The initial default is **:reprompt**.

If **:prompt** is supplied, it is a prompt option for the input editor to display at appropriate times. *prompt* can be **nil**, a string, a function, or a symbol other than **nil** (but not a list): See the section "Displaying Prompts in the Input Editor" in *Reference Guide to Streams, Files, and I/O.* The default prompt is the value of **cp::*default-prompt***. The initial default is "Command: ".

cp:read-command returns two values. The first is a symbol, the name of the command, which is defined as a function. The second is a list of the arguments, converted to the appropriate types. Usually you execute the command by applying the first value (the function) to the second (the arguments).

For an overview of **cp:read-command** and related facilities: See the section "Overview of Advanced Command Facilities", page 32.

cp:read-command-arguments *command-name* &key *initial-arguments* (*command-table* **cp:*command-table***) (*stream* ***standard-input***) (*prompt* **nil**) *Function*

Prompts for and returns the arguments to a Command Processor command.

command-name
The command name (symbol).

:initial-arguments
Specifies a list containing zero or more of the initial arguments to the command.

:command-table
Specifies the command table containing the command; the default is the current command table (bound to **cp:*command-table***).

:stream Specifies the input stream; the default is ***standard-input***.

:prompt Specifies a string, or a function returning a string, to be used as the prompt for the command arguments. The default value for this option is **nil**, causing the prompt to be derived from the user-visible name of the command.

Example:

```
(cp:read-command-arguments 'si:com-show-file :prompt
                                              "File for viewing")
```

For an overview of **cp:read-command-arguments** and related facilities:
See the section "Overview of Command Loop Management Facilities", page 33.

cp:read-command-or-form &optional (*stream* ***standard-input***) &key (*command-table* **cp:*command-table***) (*dispatch-mode* **cp::*default-dispatch-mode***) (*blank-line-mode* **cp::*default-blank-line-mode***) (*prompt* **cp::*default-prompt***) (*exception-chars* **nil**) (*environment* **si:*read-form-environment*** *environment-p*) *Function*

Reads a form or a Command Processor command from *stream*. This is an appropriate function to use at top level in a command loop that uses the command processor.

If *stream* is not supplied or is **nil**, it defaults to ***standard-input***.

If **:dispatch-mode** is specified, it is a keyword that indicates the command processor dispatch mode. The default is the value of **cp::*default-dispatch-mode***. The initial default is **:command-preferred**.

The actions that **cp:read-command-or-form** takes depend on *dispatch-mode*:

:form-only Calls **zl:read-form** to read a form from *stream*.

:command-only Calls **cp:read-command** to read a command from *stream*.

:form-preferred Calls **zl:read-form** unless the first character typed is a command dispatch character (by default, a colon). In that case calls **cp:read-command**.

:command-preferred
If the first character typed is a command dispatch character or an alphabetic character, calls **cp:read-command**; otherwise, calls **zl:read-form**. The user can evaluate a form that begins with an alphabetic character by first typing a form dispatch character (by default, a comma).

For a general description of how the user enters a command: See the section "Entering a Command" in *User's Guide to Symbolics Computers*.

If **:command-table** is supplied, it is a command table of the acceptable commands. The default command table is the value of **cp:*command-table***. The initial default is the "User" command table. See the section "Command Processor Command Tables" in *Programming the User Interface, Volume B*.

If **:blank-line-mode** is supplied, it is a keyword that determines what action the command processor takes when the user types a blank line:

:reprompt Redisplay the prompt, if any.

:beep Beep.

:ignore Do nothing.

The default *blank-line-mode* is the value of **cp::*default-blank-line-mode***. The initial default is **:reprompt**.

If **:prompt** is supplied, it is a prompt option for the input editor to display at appropriate times. *prompt* can be **nil**, a string, a function, or a symbol other than **nil** (but not a list): See the section "Displaying Prompts in the Input Editor" in *Reference Guide to Streams, Files, and I/O*. The default prompt is the value of **cp::*default-prompt***. The initial default is "Command: ".

cp:read-command-or-form returns a form. If **cp:read-command-or-form** calls **zl:read-form** to read from *stream*, it returns the form that **zl:read-form** returns. If it calls **cp:read-command**, it returns a list whose first element is a symbol, the name of the command, which is defined as a function. The remaining elements of the list are the arguments to the command, coerced to the appropriate types. Usually you execute the command by evaluating the returned list.

For an overview of **cp:read-command-or-form** and related facilities: See the section "Overview of Advanced Command Facilities", page 32.

cp:read-full-command

The m-X (extended) and colon-full-command Command Processor command accelerator.

cp:read-full-command is a function that is suitable for use as a command accelerator's function. However, because it is already installed on **#\: and #\m-x** in the "Colon Full Command" command-table, the best way to make use of this facility is to have the command tables in your applications that use accelerator characters inherit from "Colon Full Command".

For an overview of **cp:read-full-command** and related facilities: See the section "Overview of Command Loop Management Facilities", page 33.

cp:turn-command-into-form *command arguments* *Function*

Translates a Command Processor command into an evaluable form.

command
The command.

arguments
The arguments to the command.

For an overview of **cp:turn-command-into-form** and related facilities: See the section "Overview of Command Loop Management Facilities", page 33.

cp:unparse-command *command-name arguments* &optional *Function*
(*command-table* **cp:*command-table***)
(*acceptably* **t**)

Returns the input string corresponding to a Command Processor command and its arguments. (The string is created via a call to **present-to-string**.)

command-name
The command name (symbol).

arguments
The list of command arguments.

command-table
: The command table containing the named command; the default is the current command table.

acceptably
: Boolean argument passed through to **present-to-string** and specifying whether the output string can subsequently be parsed by **accept** and used for input.

For an overview of **cp:unparse-command** and related facilities: See the section "Overview of Command Loop Management Facilities", page 33.

cp:yank-and-read-full-command

The c-m-Y Command Processor command accelerator. It yanks back the last command typed for editing.

cp:yank-and-read-full-command is a function that is suitable for use as a command-accelerator's function. However, the easiest way to make use of this facility is to have the command tables in your applications that use accelerator characters inherit from "Colon Full Command".

For an overview of **cp:yank-and-read-full-command** and related facilities: See the section "Overview of Command Loop Management Facilities", page 33.

PART IV.

Dictionary of User Input Facilities

14. Dictionary Notes

This dictionary includes reference documentation for both the basic and advanced facilities provided for user input functions. These are listed in the following two tables:

Table of Basic User Input Facilities

Facilities for Accepting Single Objects
- **accept**
- **prompt-and-accept**
- **accept-from-string**
- **dw:menu-choose**
- **dw:menu-choose-from-set**

Facilities for Accepting Multiple Objects
- **dw:accept-values**
- **dw:accept-variable-values**
- **dw:accepting-values**

Table of Advanced User Input Facilities

Mouse Handler Facilities
- **define-presentation-translator**
- **define-presentation-action**
- **dw:handler-applies-in-limited-context-p**
- **dw:presentation-subtypep-cached**
- **dw:delete-presentation-mouse-handler**
- **dw:invalidate-type-handler-tables**

Mouse Gesture Interface Facilities
- **dw:mouse-char-gesture**
- **dw:mouse-char-gestures**
- **dw:mouse-char-for-gesture**

In the dictionary, the facilities are arranged in alphabetical order (package prefixes excluded).

For conceptual documentation: See the section "Overview of User Input Facilities", page 35.

15. The Facilities

accept *presentation-type* &key (*stream* ***query-io***) (*prompt* **:enter-type**) (*prompt-mode* **:normal**) *activation-chars additional-activation-chars blip-chars additional-blip-chars* (*inherit-context* **t**) (*default* **t**) (*provide-default* **'dw::unless-default-is-nil**) (*default-type* **dw:presentation-type**) (*display-default* **dw::prompt**) *present-default history* (*prompts-in-line* **dw::*accept-active***) (*initially-display-possibilities* **nil**) *input-sensitizer* (*handler-type* **'dw::parser**) *query-identifier* (*separate-inferior-queries* **nil**) *Function*

Reads printed representation of a Lisp object from a stream. If the representation is entered via a mouse gesture, it returns the object; if the representation is entered as a series of keyboard characters, it parses the series and returns the object.

presentation-type
Presentation type of the object to be accepted.

:stream Specifies stream from which object is read; the default is ***query-io***.

:prompt Specifies characteristics of the input prompt. Allowable values for this option are:

nil No prompt is printed.

string String to be used as prompt.

function Function to display a prompt string. It must take two positional arguments. The first is the stream on which the prompt is to be displayed. The second is a keyword indicating the origin of the function call; for available keywords and related information: See the section "Displaying Prompts in the Input Editor" in *Reference Guide to Streams, Files, and I/O*.

You typically provide a prompt function when you want the prompt to change dynamically. In such cases, you can **ignore** the second argument.

:enter-type
Causes the prompt "Enter a <*presentation type*>" to be used. The presentation type is that specified by the *presentation-type* argument to **accept**.

If **:prompt** is not **nil**, the default, if any, is displayed automatically after the prompt string. For example, a prompt string of "to file" for a presentation type of **pathname** is displayed as "to file (default Q:>foo.bar):".

If you provide a prompt string, whether **accept** provides trailing punctuation is determined by the **:prompt-mode** option.

:prompt-mode
Specifies whether a colon and space is appended to a user-supplied prompt. A value of **:normal** causes a trailing colon and space to be appended; a value of **:raw** does not. (See the function "**accept**", page 167.)

:activation-chars
Takes a list of characters that are used as activation characters for the duration of the call to **accept**. The default activators are #**return** and #**end**.

Activation characters signal the end of user input to the **accept** function. If input to the function is via the keyboard, the user must necessarily press an activation character to activate the **accept**.

If input is via a translating mouse handler, defined by **define-presentation-to-command-translator** or **define-presentation-translator**, then whether an activation character is necessary depends on whether the translator returns an `:activate t` keyword-value pair. See the macro **define-presentation-translator**, page 185.

:additional-activation-chars
Similar to **:activation-chars**; the list of characters supplied is added to the list of activators. Additional activation characters may be useful for activating **accept** when called recursively.

:blip-chars
Takes a list of characters that serve as delimiters of input fields for the duration of the call to **accept**.

:additional-blip-chars
Similar to **:blip-chars**; the list of characters supplied is added to the list of delimiters. Additional blip characters may be useful for terminating input fields when **accept** is called recursively.

:inherit-context
Boolean option specifying whether the current invocation of **accept** inherits the existing input context or establishes a new root node; the default value is **t**. This option is useful for controlling the input contexts at different levels in a recursive call to **accept**.

:default Specifies the object to be used as the default value for this **accept**. If no object is specified by this option – and a default is to be displayed (see the **:provide-default** option) – then the object offered is the one at the top of the presentation history for the presentation type specified in the *presentation-type* argument.

:provide-default
Specifies whether to provide a default value for this **accept**. If this option is not specified, a default value is displayed unless it is **nil**. If **nil** is a valid default that you want to be provided, then you must specify `:provide-default t`.

:default-type
Specifies the presentation type of the object offered as the default for completing the call to **accept**. The default for this option is the type specified by the *presentation-type* argument.

This option is useful for specifying explicitly the presentation type of the default when accepting compound presentation types, such as those created with the **or** presentation type. See the presentation type **or**, page 318.

:display-default
Controls the display of the default object. A value of **t** causes the default to be displayed whether or not a prompt is displayed; **nil** suppresses the display of the default whether or not a prompt is displayed.

The default value for this option, **dw::prompt**, causes the default to be displayed when a prompt is displayed, and the default to be suppressed when a prompt is not displayed.

:present-default
Boolean option specifying whether the default object is presented and accepted. This option is for the internal use of **dw:accept-variable-values** and related facilities.

:history Specifies which presentation-type history to use for yanking purposes. A value of **nil**, the default, causes the history of the type specified by the *presentation-type* argument to be used.

Aside from providing another presentation type, you may also supply as the value to this option a history object. This would be appropriate if you constructed the presentation-type history yourself, rather than letting the presentation substrate do it for you.

:prompts-in-line
Boolean option specifying whether prompt is displayed in-line with parentheses or with a trailing colon. The default is **t** if the **accept** was called recursively, **nil** otherwise.

:initially-display-possibilities
Boolean option specifying whether to display the objects that could be used as input in the current context; the default is **nil**. If **t**, the possibilities are presented before the prompt appears. This is the same list of possibilities that is displayed when the user presses HELP after the initial prompt appears.

:input-sensitizer
This option is used internally by **dw:accept-variable-values** and related facilities.

:handler-type
This option is used internally by **dw:accept-variable-values** and related facilities.

:query-identifier
Specifies a unique identifier for this call to **accept**; the default is derived from the prompt.

This option is only used when the **accept** is part of a multiple-**accept** form. See the function "**dw:accepting-values**", page 175.

:separate-inferior-queries
Boolean option specifying whether recursive calls to **accept** go on separate lines when executing an **dw:accept-values** function; the default is **nil**.

For an overview of **accept** and related facilities: See the section "Overview of Facilities for Accepting Single Objects", page 35.

accept-from-string *presentation-type string* &rest *args* &key *index* (*start* **0**) *end* &allow-other-keys *Function*

Reads the printed representation of a Lisp object from a string and returns the object with a specified presentation type. This function is the presentation-system equivalent of the Common Lisp function **read-from-string**.

presentation-type
Presentation type of the object to be accepted.

string String from which to accept the object.

args Keyword options to **accept**.

:index [Not implemented]

:start Specifies the position of the first character to be parsed. The default is `0`, the position of the first character.

:end Specifies the position of the first character not to include in the parsing of the string.

Examples:

```
(accept-from-string 'string "Test 1") ==>
"Test 1"
STRING

(accept-from-string 'integer "Test 2" :start 5) ==>
2
INTEGER
```

For an overview of **accept-from-string** and related facilities: See the section "Overview of Facilities for Accepting Single Objects", page 35.

dw:accept-values *descriptions* &key (*prompt* **nil**) (*near-mode* **'(:mouse)**) (*stream* ***query-io***) (*own-window* **nil**) (*temporary-p* **dw::own-window**) (*initially-select-query-identifier* **nil**) *Function*

Reads a series of printed representations of Lisp objects from a stream and returns one value for each object read. The objects may entered via mouse gestures or as keyboard input.

descriptions
List of descriptions. Each description is a list of a presentation type and a set of the keyword options; available keywords are those allowed by **accept**.

Note that when the same presentation type appears in more than one description, they must be distinguished by the **:prompt** option. Failing to do so results in the same value being returned for all occurrences of that type.

Example:

```
(dw:accept-values '((integer :prompt "Half-life"
                             :default 24000)
                    (pathname :prompt "Log file")
                    (integer :prompt "Session number"))
                  :prompt "Atomic experiment")
```

:prompt Specifies a string, or a function returning a string, serving as the prompt or heading for the whole series of input prompts that follow.

:near-mode
Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)** in *Programming the User Interface, Volume B.*

This option is applicable only when the value of the **:own-window** option is **t**.

:stream Specifies the stream to be used for input and output; the default is ***query-io***.

:own-window
Specifies whether the input/output interaction occurs in a separate, momentary window or runs "in place" in the current window like ordinary input/output; the default is **nil**.

:temporary-p
Specifies whether the menu window is temporary or

momentary. If the value of the **:own-window** option is **t**, then the default is a temporary window; if the value of **:own-window** is **nil**, then this option is inapplicable.

:initially-select-query-identifier
Specifies that a particular field is pre-selected when the user interaction begins. The field to be selected is tagged by the **:query-identifier** option to **accept**, passed through to **accept** by **dw:accept-values**. Use this tag as the value for the **:initially-select-query-identifier** keyword, as shown in the following example:

```
(dw:accept-values '((integer :prompt "Number of times"
                             :query-identifier fred)
                    (boolean :prompt "Backwards"))
                  :initially-select-query-identifier 'fred)
```

When the initial display is output, the mouse cursor appears after the prompt of the tagged field, just as if the user had selected that field by clicking on it. Note that the default value,if any, for the selected field is not displayed.

For an overview of **dw:accept-values** and related facilities: See the section "Overview of Facilities for Accepting Multiple Objects", page 38.

dw:accept-variable-values *variables* &key (*prompt* **"Choose Variable Values"**) (*near-mode* **'(:mouse)**) (*delayed* **t**) (*stream* ***query-io***) (*own-window* **nil**) (*temporary-p* **dw::own-window**) (*initially-select-query-identifier* **nil**) **Function**

Provides a menu-like facility for setting the values of special variables to values provided by the user. The value for each variable is read via a call to **accept** using a specified presentation type.

(Usage note: **dw:accept-variable-values** is intended for use with special variables, not local ones. As such, it is useful for conversion from **tv:choose-variable-values** but is, in general, less appropriate for new applications of multiple-**accept** technology. For the latter, we recommend using **dw:accept-values** and **dw:accepting-values**.)

variables A list of variable descriptions. Each description is a list of a variable name, a prompt string, and a presentation type.

Example:

```
(dw:accept-variable-values
        '((*a* "Number" integer)
          (*b* "File" pathname)
          (*c* "Printer" sys:printer))) ==>
Choose Variable Values
Number:an integer
File: the pathname of a file
Printer: a printer
ABORT aborts, END uses these values
```

:prompt Specifies a string, or a function returning a string, serving as the prompt or heading for the whole series of input prompts that follow.

:near-mode
Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)** in *Programming the User Interface, Volume B.*

This option is applicable only when the value of the **:own-window** option is **t**.

:delayed Boolean option specifying whether variables are updated with user-supplied values after the entire accept-variable-values interaction is complete, or individually after input to each variable field is terminated; the default is **t**.

:stream Specifies the stream to be used for input and output; the default is ***query-io***.

:own-window
Specifies whether the input/output interaction occurs in a separate, momentary window or runs "in place" in the current window like ordinary input/output; the default is **nil**.

:temporary-p
Specifies whether the menu window is temporary or momentary. If the value of the **:own-window** option is **t**, then the default is a temporary window; if the value of **:own-window** is **nil**, then this option is inapplicable.

:initially-select-query-identifier
[Not implemented]

For an overview of **dw:accept-variable-values** and related facilities: See the section "Overview of Facilities for Accepting Multiple Objects", page 38.

dw:accepting-values (&optional (*stream* '***query-io***) &key (*own-window* **nil**) (*display-exit-boxes* **(not dw::own-window)**) (*temporary-p* **dw::own-window**) (*label* **"Multiple accept"**) (*near-mode* **'(:mouse)**) (*initially-select-query-identifier* **nil**) (*resynchronize-every-pass* **nil**) &body *body*) *Macro*

Causes all calls to **accept** within *body* to appear in a single, **dw:accept-variable-values**-like menu that can be modified dynamically.

stream Stream for input and output; the default is ***query-io***.

:own-window
Specifies whether the input/output interaction occurs in a separate, momentary window or runs "in place" in the current window like ordinary input/output; the default is **nil**.

:display-exit-boxes
Boolean option specifying whether the Abort-End exit message is displayed. The default is to display it unless the interaction is in its own window (see the **:own-window** option).

:temporary-p
Specifies whether the menu window is temporary or momentary. If the value of the **:own-window** option is **t**, then the default is a temporary window; if the value of **:own-window** is **nil**, then this option is inapplicable.

:label Specifies a string to serve as the title of the interaction menu. This option is applicable only if the value of the **:own-window** option is **t**.

:near-mode
Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)** in *Programming the User Interface, Volume B.*

This option is applicable only when the value of the **:own-window** option is **t**.

:initially-select-query-identifier

Specifies that a particular field is pre-selected when the user interaction begins. The field to be selected is tagged by the **:query-identifier** option to **accept**; use this tag as the value for the **:initially-select-query-identifier** keyword, as shown in the following example:

```
(let (a b c)
  (dw:accepting-values (*query-io*
        :initially-select-query-identifier 'the-tag)
    (setq a (accept 'pathname :prompt "The file"))
    (setq b (accept 'integer :prompt "The number"
        :query-identifier 'the-tag))
    (setq c (accept 'sys:printer
             :prompt "The printer")))
  (format t "Printing ~D copies of
              file ~A on ~A" b a c))
```

When the initial display is output, the mouse cursor appears after the prompt of the tagged field, just as if the user had selected that field by clicking on it. Note that the default value, if any, for the selected field is not displayed.

:resynchronize-every-pass

Boolean option specifying whether to redisplay after each value is accepted; the default is **nil**.

You can use this option to alter dynamically the multiple-accept display. 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 ()
  (fresh-line)
  (let ((flag 1))
    (dw:accepting-values
        (t :resynchronize-every-pass t)
      (if (= flag 1)
          (setq flag (accept 'integer :default flag))
          (accept 'string)
          (accept 'pathname)))))
```

As the example shows, to use this option effectively, the controlling variable(s) must be initialized outside the lexical scope of the **dw:accepting-values** macro.

body The body is run in order to generate the initial prompt/value display. The body (or some part of it) is re-run each time a change is made; so the dependencies that later calls to **accept** may have on earlier ones will be correctly resolved. Because the body is run repeatedly, you must be careful of side-effects in the body code.

Also, because the stream carries the state information, all input/output calls within the body must use the stream specified in the **dw:accepting-values** options list.

Good examples:

```
(let (a b c)
  (dw:accepting-values (*query-io*
                        :prompt "Good Example")
    (setq a (accept 'pathname :prompt "The file"))
    (setq b (accept 'integer :prompt "The number"))
    (setq c (accept 'sys:printer
                    :prompt "The printer")))
  (format t "Printing ~D copies of
               file ~A on ~A" b a c))

(multiple-value-bind (a b c)
  (dw:accepting-values ()
    (values
      (accept 'pathname :prompt "The file")
      (accept 'integer :prompt "The number")
      (accept 'sys:printer
              :prompt "The printer")))
  (format t "Printing ~D copies of
               file ~A on ~A" b a c))
```

Poor example:

```
(let ((the-list nil))
  (dw:accepting-values ()
    (push
      (list
        (accept 'pathname :prompt "The file")
        (accept 'sys:printer :prompt "The printer"))
      the-list))
  (format t "The list = ~S" the-list))
```

The above example is a poor one because the output list will have an unpredictable number of elements; this detracts from its usefulness.

A useful presentation type to use with **accept** functions in the body of a **dw:accepting-values** macro is **alist-member**. Its usefulness derives from the keyword options available for inclusion in the item lists contributing to the alists. Three options exist: **:documentation**, **:style**, and **:selected-style**.

The value of the **:documentation** keyword is a string that appears in the mouse documentation line when the mouse cursor is over the item (that is, the item is highlighted).

:style specifies the character style for the item when it is displayed. **:selected-style** specifies the character style of the item when it is selected, that is, after it has been clicked on. The **:selected-style** defaults to the boldface version of the unselected style.

Use of the **alist-member** presentation type with **dw:accepting-values** is illustrated by the following example:

```
(defun filter-a-v ()
  (let ((low-pass-list
          '(("Mean" :value :mean
             :documentation "1  1  1 mask"
             :style (:swiss :roman :normal)
             :selected-style (:dutch :bold nil))
            ("Gaussian" :value :gauss
             :documentation "1  2  1 mask"
             :style (:swiss :roman :normal)
             :selected-style (:dutch :bold nil))))
        (edge-list
          '(("Laplacian, HP" :value :lpl-hp
             :documentation "-1  3  -1 mask"
             :style (:swiss :roman :normal)
             :selected-style (:dutch :bold nil))
            ("Laplacian, ED" :value :lpl-ed
             :documentation "-1  2  -1 mask"
             :style (:swiss :roman :normal)
             :selected-style (:dutch :bold nil)))))
    (dw:accepting-values (*query-io* :own-window t)
      (fresh-line)
      (setq lo-pass-f (accept '((alist-member
                                  :alist ,low-pass-list)
                                  :description "a low-pass filter")))
      (setq edge-f (accept '((alist-member
                               :alist ,edge-list)
                               :description "a hi-pass/edge filter"))))))
```

For an overview of **dw:accepting-values** and related facilities: See the section "Overview of Facilities for Accepting Multiple Objects", page 38.

For additional examples, see the file `sys:examples;accepting-values.lisp`

define-presentation-action *name* (*from-presentation-type* *to-presentation-type* &key *tester* (*gesture* **:select**) *documentation suppress-highlighting* (*menu* **t**) (*context-independent* **nil**) *priority* *exclude-other-handlers blank-area defines-menu*) *arglist* &body *body* *Macro*

Defines a side-effecting mouse handler for performing actions on a displayed presentation object that are independent of the main body and command loop of an application.

name The name of the handler.

from-presentation-type
The type of the displayed presentation object.

to-presentation-type
A presentation type. This argument establishes the input context in which the handler is active. The value usually supplied is **t**, meaning that the handler is potentially available in any input context.

:tester Specifies the parameter list and body for a tester function. The tester function determines whether the handler applies to the current presentation, if it is otherwise applicable based on the current presentation type and input context.

The parameter list consists of a positional argument – the current presentation object – and a subset of the keywords *presentation*, *input-context*, and *handler*. These keywords are the same as those available for inclusion in the argument list for the body of the handler, and are documented under *arglist* in the handler documentation; they are also documented separately: See the macro "**define-presentation-action**", page 179.

Note: inefficient testers can degrade the performance of your program. Tester functions must be capable of rapid execution. Also, do not use the body of your handler as an implicit tester if it does a large amount of consing or in other ways consumes resources; this will similarly affect program

performance. For more information: See the section "Some Efficiency Caveats for Mouse Handlers", page 44.

For functions used in **:testers**: See the function **dw:handler-applies-in-limited-context-p**, page 192. See the function **dw:presentation-subtypep-cached**, page 199.

:gesture Specifies the mouse gesture on which the handler is available.

The gesture is specified by its symbolic name rather than as a mouse character. For example, symbolic names for **#\mouse-l**, **#\mouse-m**, and **#\mouse-r** include **:left**, **:middle**, and **:right**, respectively. (For lists of names assigned to these and other mouse gestures, use the function **dw:mouse-char-gestures**.) The default gesture is **:select**, which is the same as **:left**.

To assign your own symbolic name to a mouse character, use the following form:

```
(setf (dw:mouse-char-for-gesture symbol) #\mouse-x)
```

Specifying this option with **nil**, that is `:gesture nil`, results in the handler being unavailable on any gesture, only in a handler menu. See the macro "**define-presentation-action**", page 179.

Specifying this option with **t**, that is, `:gesture t`, results in the handler being available on all gestures. See the macro "**define-presentation-action**", page 179.

:documentation
Specifies a string or a function returning a string to be used as mouse and menu documentation for the handler.

It is often preferable not to supply this option and to use the default documentation instead. This is because the default documentation incorporates a string corresponding to the object the mouse is over, while the documentation you supply cannot. If the name of the handler is *handler-name*, the default documentation string will be "Handler Name (*presentation type*) *presentation object*".

:suppress-highlighting
Boolean option specifying whether to suppress highlighting of the presentation if this handler is the only applicable one.

For example, the standard click-right menu handler uses this option. The default is **nil**.

:menu Specifies the name of a menu in which the handler is to be included. The default is **t**, the name of the standard click-right handler menu.

You can define you own handler menu with **define-presentation-action**: See the macro "**define-presentation-action**", page 179.

:context-independent
Boolean option specifying whether handler behavior (that is, applicability to displayed presentations) is the same for all contexts in a nested-context structure (**accept** being called recursively); the default is **nil**.

This option is supplied with **t**, for example, if the handler's *to-presentation-type* is **t** (any context), and its contract is to print additional information about a particular presentation (that is, only the output matters).

Specifying this option **t**, when appropriate, allows more possibilities to be presented on different mouse gestures. Without it, a handler that applies in all contexts would be matched for a particular context, to the possible exclusion of other handlers in other contexts on other gestures. With it, you get the same behavior for this handler, and more possibilities as well.

For more information on context matching and related handler issues: See the section "How Mouse Handlers Are Found", page 42.

:priority Specifies a number adding to the priority of this handler relative to other applicable handlers defined on the same gesture; the default is `0`.

Handler applicability to displayed presentations depends on three factors: 1) the object type of the presentation; 2) the presentation type of the presentation; and 3) the current input context. A handler matching a displayed presentation in any of these factors is applicable and invokable.

In some cases, more than one applicable handler might be available on a given mouse gesture. In such cases, which handler is the one displayed for that gesture in the mouse

documentation line is determined by handler precedence or priority. The system automatically assigns priorities according to the matching factors as follows: the priority is incremented by 1 when the object type matches; by 4 when the presentation type matches; and by 2 when the context type matches.

For example, in a Lisp Listener in the command-or-form context, an **accept** of a pathname appears something like the following:

```
(accept 'pathname)
Enter the pathname of a file [default
Q:>rel-7>sys>doc>uims>ui-dict2.sar]: ==>
Q:>rel-7>sys>doc>uims>ui-dict2.sar
#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"
FS:LMFS-PATHNAME
```

The default pathname was accepted causing it to be presented as both a pathname presentation (Q:>rel-7>sys>doc>uims>ui-dict2.sar) and a **sys:expression** presentation (`#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"`).

Two handlers defined on the **:select** gesture are applicable to both presentations. The first is **si:com-show-file**, applicable to expression presentations with a pathname object type, or pathname presentations of any object type. The second is **dw::quoted-expression**, applicable to expression presentations of any object type. The following table shows the priorities determined for them by the system relative to the two presentations in the above example:

	Pathname Presentation `Q:>rel-7>sys>doc>...`	Expression Presentation `#P"Q:>rel-7>sys>doc>...`
Show File	5	5
Quote Expression	0	4

For both presentations, the system-generated priority is highest for the show file handler. However, it was the system programmer's intent that the quoted expression handler should be displayed in the mouse documentation line whenever the mouse is over a presentation of the **sys:expression** type, regardless of what other applicable

handlers might be available on the **:select** gesture. Therefore, in the definition for this handler, the value of the **:priority** option was made `1.5`. This is added to the system-generated priority of `4` in the bottom right cell of the table for a total score of `5.5`, enough to give this handler precedence.

:exclude-other-handlers

Boolean option, used with `:gesture t` handlers, specifying whether to exclude non-**t** handlers.

For example, any gesture selects a menu item. The translator that implements this has a **:tester** option that checks, among other things, for the keyword **:no-select** in the menu-item list: See the section "The "General List" Form of Item" in *Programming the User Interface, Volume B.* If the menu item includes the **:no-select** keyword, the translator does not apply. But, if `:exclude-other-handlers t` were not specified for this translator, other translators would still apply to the **:no-select** item's presentation, like the **:menu** (`Mouse-R`) gesture.

:exclude-other-handlers provides a way of saying "this translator implements the entire contract for the presentation it matches".

See the macro "**define-presentation-action**", page 179.

:blank-area

Boolean option specifying whether the handler is active when the mouse cursor is over areas of the screen in which no presentations are displayed; the default is **nil**.

To ensure that handlers intended to be active only in blank areas are not active over displayed presentations, use the **dw:no-type** presentation type as the [*from-*]*presentation-type* positional argument to the handler.

:defines-menu

Specifies the handler menu that that this handler invokes. That is, when this option is specified, it means that this handler is to produce a menu of other handlers that apply to the presentation at hand. Other handlers are included in this menu by specifying their **:menu** options with the menu named by **:defines-menu**.

The following example is for the Presentation debugging menu, available on s-Mouse-R for all presentations, in all input contexts (both the *from-* and *to-presentation-type*s are **t**):

```
(define-presentation-action presentation-debugging-menu
   (t t
      :documentation "Presentation debugging menu"
      :gesture :presentation-debugging-menu
      :menu (t :style (nil :italic nil))
      :defines-menu :presentation-debugging
      :context-independent t
      :blank-area t)
    (ignore &rest args)
   (return-from presentation-debugging-menu
     (apply #'dw::call-presentation-menu
            :presentation-debugging args)))
```

Note the body: except for the keyword identifying the menu, **:presentation-debugging**, this is the same for all side-effecting handlers that generate handler menus. The function creating the menu is **dw::call-presentation-menu**. Use it exactly as shown in the example.

arglist The argument list for the body of the handler. The argument list consists of one positional argument, the object that the mouse cursor is over, and keyword arguments from a predefined set.

The following predefined keywords are available for inclusion in the argument list to a mouse handler body. Their inclusion makes the named parameters available for use in the body. The parameter list can specify only those keywords that are explicitly used.

input-context
: The current presentation-input context.

presentation
: The presentation instance that the mouse cursor is over.

handler The handler object of which the body is a part.

mouse-char
The mouse character that triggered the handler. (This keyword cannot be used in the **:tester** function parameter list.)

window The window object in which the current presentation occurs.

For an overview of **define-presentation-action** and related facilities: See the section "Overview of Mouse Handler Facilities", page 39. For information on handler lookup and performance issues: See the section "How Mouse Handlers Are Found", page 42.

define-presentation-translator *name (from-presentation-type to-presentation-type* &key *tester (gesture* **:select**) *documentation suppress-highlighting (menu* **t**) *(context-independent* **nil**) *priority exclude-other-handlers blank-area do-not-compose) arglist* &body *body* *Macro*

Defines a mouse handler that translates from a displayed presentation object of a certain type to a returned presentation object of a different type. Typically, the "translation" is a matter of extracting a nested object, for example, a host object from a pathname object.

name The name of the handler.

from-presentation-type
The type of the displayed presentation object.

to-presentation-type
The type of the returned presentation object.

:tester Specifies the parameter list and body for a tester function. The tester function determines whether the handler applies to the current presentation, if it is otherwise applicable based on the current presentation type and input context.

The parameter list consists of a positional argument – the current presentation object – and a subset of the keywords *presentation*, *input-context*, and *handler*. These keywords are the same as those available for inclusion in the argument list for the body of the handler, and are documented under *arglist* in the handler documentation; they are also documented separately: See the macro **"define-presentation-action"**, page 179.

Note: Inefficient testers can degrade the performance of your

program. Tester functions must be capable of rapid execution. Also, do not use the body of your handler as an implicit tester if it does a large amount of consing or in other ways consumes resources; this will similarly affect program performance. For more information: See the section "Some Efficiency Caveats for Mouse Handlers", page 44.

For functions used in **:testers**: See the function **dw:handler-applies-in-limited-context-p**, page 192. See the function **dw:presentation-subtypep-cached**, page 199.

:gesture Specifies the mouse gesture on which the handler is available.

The gesture is specified by its symbolic name rather than as a mouse character. For example, symbolic names for **#\mouse-l**, **#\mouse-m**, and **#\mouse-r** include **:left**, **:middle**, and **:right**, respectively. (For lists of names assigned to these and other mouse gestures, use the function **dw:mouse-char-gestures**.) The default gesture is **:select**, which is the same as **:left**.

To assign your own symbolic name to a mouse character, use the following form:

```
(setf (dw:mouse-char-for-gesture symbol) #\mouse-x)
```

Specifying this option with **nil**, that is `:gesture nil`, results in the handler being unavailable on any gesture, only in a handler menu. See the macro "**define-presentation-action**", page 179.

Specifying this option with **t**, that is, `:gesture t`, results in the handler being available on all gestures. See the macro "**define-presentation-action**", page 179.

:documentation

Specifies a string or a function returning a string to be used as mouse and menu documentation for the handler.

It is often preferable not to supply this option and to use the default documentation instead. This is because the default documentation incorporates a string corresponding to the object the mouse is over, while the documentation you supply cannot. If the name of the handler is *handler-name*, the default documentation string will be "Handler Name (*presentation type*) *presentation object*".

:suppress-highlighting
Boolean option specifying whether to suppress highlighting of the presentation if this handler is the only applicable one. For example, the standard click-right menu handler uses this option. The default is **nil**.

:menu Specifies the name of a menu in which the handler is to be included. The default is **t**, the name of the standard click-right handler menu.

You can define you own handler menu with **define-presentation-action**: See the function "**define-presentation-action**", page 179.

:context-independent
Boolean option specifying whether handler behavior (that is, applicability to displayed presentations) is the same for all contexts in a nested-context structure (**accept** being called recursively); the default is **nil**.

This option is supplied with **t**, for example, if the handler's *to-presentation-type* is **t** (any context), and its contract is to print additional information about a particular presentation (that is, only the output matters).

Specifying this option **t**, when appropriate, allows more possibilities to be presented on different mouse gestures. Without it, a handler that applies in all contexts would be matched for a particular context, to the possible exclusion of other handlers in other contexts on other gestures. With it, you get the same behavior for this handler, and more possibilities as well.

For more information on context matching and related handler issues: See the section "How Mouse Handlers Are Found", page 42.

:priority Specifies a number adding to the priority of this handler relative to other applicable handlers defined on the same gesture; the default is `0`.

Handler applicability to displayed presentations depends on three factors: 1) the object type of the presentation; 2) the presentation type of the presentation; and 3) the current input context. A handler matching a displayed presentation in any of these factors is applicable and invokable.

In some cases, more than one applicable handler might be available on a given mouse gesture. In such cases, which handler is the one displayed for that gesture in the mouse documentation line is determined by handler precedence or priority. The system automatically assigns priorities according to the matching factors as follows: the priority is incremented by 1 when the object type matches; by 4 when the presentation type matches; and by 2 when the context type matches.

For example, in a Lisp Listener in the command-or-form context, an **accept** of a pathname appears something like the following:

```
(accept 'pathname)
Enter the pathname of a file [default
Q:>rel-7>sys>doc>uims>ui-dict2.sar]: ==>
Q:>rel-7>sys>doc>uims>ui-dict2.sar
#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"
FS:LMFS-PATHNAME
```

The default pathname was accepted causing it to be presented as both a pathname presentation (Q:>rel-7>sys>doc>uims>ui-dict2.sar) and a **sys:expression** presentation (`#P"Q:>rel-7>sys>doc>uims>ui-dict2.sar.newest"`).

Two handlers defined on the **:select** gesture are applicable to both presentations. The first is **si:com-show-file**, applicable to expression presentations with a pathname object type, or pathname presentations of any object type. The second is **dw::quoted-expression**, applicable to expression presentations of any object type. The following table shows the priorities determined for them by the system relative to the two presentations in the above example:

	Pathname Presentation `Q:>rel-7>sys>doc>...`	Expression Presentation `#P"Q:>rel-7>sys>doc>...`
Show File	5	5
Quote Expression	0	4

For both presentations, the system-generated priority is highest for the show file handler. However, it was the system programmer's intent that the quoted expression

handler should be displayed in the mouse documentation line whenever the mouse is over a presentation of the **sys:expression** type, regardless of what other applicable handlers might be available on the **:select** gesture. Therefore, in the definition for this handler, the value of the **:priority** option was made `1.5`. This is added to the system-generated priority of `4` in the bottom right cell of the table for a total score of `5.5`, enough to give this handler precedence.

:exclude-other-handlers

Boolean option, used with `:gesture t` handlers, specifying whether to exclude non-**t** handlers.

For example, any gesture selects a menu item. The translator that implements this has a **:tester** option that checks, among other things, for the keyword **:no-select** in the menu-item list: See the section "The "General List" Form of Item" in *Programming the User Interface, Volume B.* If the menu item includes the **:no-select** keyword, the translator does not apply. But, if `:exclude-other-handlers t` were not specified for this translator, other translators would still apply to the **:no-select** item's presentation, like the **:menu** (`Mouse-R`) gesture.

:exclude-other-handlers provides a way of saying "this translator implements the entire contract for the presentation it matches".

See the macro "**define-presentation-action**", page 179.

:blank-area

Boolean option specifying whether the handler is active when the mouse cursor is over areas of the screen in which no presentations are displayed; the default is **nil.**

To ensure that handlers intended to be active only in blank areas are not active over displayed presentations, use the **dw:no-type** presentation type as the [*from-*]*presentation-type* positional argument to the handler.

:do-not-compose

Boolean option specifying whether the value of *body* is computed to determine if the handler satisfies the current input context; the default is **nil.**

To see the need for this option, let's consider the default behavior. For example, if 1) you have a translating mouse handler that returns integer objects; 2) the mouse cursor is currently over the handler's *from-presentation-type* ; 3) any shift keys modifying the mouse gesture the handler is on are pressed; and 4) the current input context is for integers, the default system behavior would be to determine what the body of the handler returns. If it returns anything other than a single value of **nil**, then the handler is applicable; this fact is indicated in the mouse documentation line and the presentation is highlighted (if it's not already).

Now, if the input context in this situation was for odd integers, rather than for any integer – that is, `(accept '((and integer ((satisfies oddp)))))` – by default this handler would still be run to see if it returns an *odd* integer, that is, that the returned object will satisfy the input context requirements. Only if this is the case will the handler be available. This is the motivation for the default behavior.

However, some translating handlers have side effects, for example, popping up a menu or asking a question. It is unlikely that you want such events occurring merely when a user of your program waves the mouse over a presentation. You want this behavior suppressed until the user actually clicks on the presentation. `:do-not-compose t` is how you express this intent.

As a general rule, avoid defining translators that have side effects. One way of doing this is by defining side-effecting handlers explicitly, with **define-presentation-action**.

arglist The argument list for the body of the handler. The argument list consists of one positional argument, the object that the mouse cursor is over, and keyword arguments from a predefined set.

The following predefined keywords are available for inclusion in the argument list to a mouse handler body. Their inclusion makes the named parameters available for use in the body. The parameter list can specify only those keywords that are explicitly used.

input-context
The current presentation-input context.

presentation
The presentation instance that the mouse cursor is over.

handler The handler object of which the body is a part.

mouse-char
The mouse character that triggered the handler. (This keyword cannot be used in the **:tester** function parameter list.)

window The window object in which the current presentation occurs.

The *body* of your handler must return at least one value, the presentation object. Optionally, it can also return keyword-value pairs that you define. In this case, you must return the presentation type of the object as well. The object is the first item returned, its presentation type the second; these are followed by the keyword-value pairs.

One predefined keyword is available, **:activate**. Supplied with **nil**, the activation of input entered via this handler is suppressed, with **t** it's promoted. The following example is taken from the system code:

```
(define-presentation-translator command-name-to-command
   (cp:command-name cp:command)
   (command-name)
  (values
    `(,command-name) 'command :activate nil))
```

This translator allows commands displayed as command-name presentations – for example, in the display generated when you press HELP after entering the first word of a command to the command processor prompt – to be used as command object input. Because `:activate nil` is provided, the command is not executed immediately after clicking on its name; the user must press RETURN to activate the command.

The values returned by the translator will be used to construct a presentation blip. You do not make the blip; the handler takes care of this automatically. Any keywords the translator returns are included in the options field of the blip. Options can be extracted from blips with the **dw:presentation-blip-options** function. For an overview of this and related functions: See the section "Overview of Presentation Input Blip Facilities", page 78.

For an overview of **define-presentation-translator** and related facilities: See the section "Overview of Mouse Handler Facilities", page 39. For

information on handler lookup and performance issues: See the section "How Mouse Handlers Are Found", page 42.

dw:delete-presentation-mouse-handler *name* *Function*

Removes an already defined presentation mouse handler.

name The name of the mouse handler to remove.

For an overview of **dw:delete-presentation-mouse-handler** and related facilities: See the section "Overview of Mouse Handler Facilities", page 39.

dw:handler-applies-in-limited-context-p *context* *Function*
limiting-context-type

This function is intended for use in the **:tester** forms of mouse handlers. It takes a *context* as provided in the **:context** keyword argument to a tester, and a presentation type to use as the *limiting-context-type*. It returns **t** if and only if the presentation type in the context is a subtype of the *limiting-context-type*. Because of caching, it is much faster than using **dw:presentation-subtypep** for this purpose, and it provides the convenience of extracting the presentation type from the context. See the function **dw:presentation-subtypep-cached**, page 199.

This function is typically used with translating handlers whose *to-presentation-type* is a subtype of **sys:expression**. For example, a translator from a `.bin` pathname to a `.lisp` pathname may be intended for use only in the pathname input context, not when any Lisp object is acceptable. By putting **dw:handler-applies-to-limited-context-p** in the **:tester** of the handler, the handler can be limited to contexts that are looking for some type of pathname.

Example:

```
(define-presentation-translator source-file-pathname
   (pathname pathname
             :tester ((ignore &key context)
                      (dw:handler-applies-in-limited-context-p
                        context 'pathname)))
   (pathname)
  (send (send (send pathname :generic-pathname) :get
              :qfasl-source-file-unique-id)
        :new-pathname :version :newest))
```

For an overview of **dw:handler-applies-in-limited-context-p** and related facilities: See the section "Overview of Mouse Handler Facilities", page 39.

dw:invalidate-type-handler-tables *Function*

Invalidates presentation mouse handler lookup tables. The next time the tables are accessed, they are updated by this function to reflect any changes in the type hierarchy affecting handler applicability.

This function gets called by the system whenever a new presentation type is defined. You need to call it directly only if your presentation-type definitions change dynamically at runtime, for example, through a flag in the **:abbreviation-for** option. However, because the updating of the handler lookup tables does not occur in real time, you should avoid such usage.

For an overview of **dw:invalidate-type-handler-tables** and related facilities: See the section "Overview of Mouse Handler Facilities", page 39.

dw:menu-choose *item-list* &key (*prompt* **nil**) (*default* **nil**) (*presentation-type* **nil**) (*printer* **nil**) (*near-mode* **'(:mouse)**) (*superior* **tv:mouse-sheet**) (*center-p* **dw::*default-menu-center-p***) (*character-style* **'(:jess :roman :large)**) (*momentary-p* **t**) (*temporary-p* **dw::momentary-p**) *Function*

Constructs a menu from a list of items and returns the value associated with the selected item; also returned are the item and the mouse character that was used to select it.

item-list A quoted list of items to include in the menu display. A menu item can have various forms: See the section "The Form of a Menu Item" in *Programming the User Interface, Volume B.*

If you wish to control the mouse documentation associated with an item, use the "general list" form and include the **:documentation** menu-item option.

Example:

```
(dw:menu-choose '(("First Choice" :value 1
                   :documentation "Mouse Doc One")
                  ("Second Choice" :value 2
                   :documentation "Mouse Doc Two")))
```

The other available menu-item option is **:style**, specifying the character style of the individual item. This contrasts with the **:character-style** option to **dw:menu-choose** as a whole, which specifies the style for all items. If both are specified, the locally specified **:style** prevails. For an example, see the **:character-style** option; descriptions of this and other options to **dw:menu-choose** follow.

:prompt Specifies a string to use as a title for the menu. The menu title appears at the top of the menu.

:default Specifies an item in the *item-list* that is the currently selected (highlighted) item when the menu is first displayed.

If the item list is a simple one containing symbols, then specify the default item by its symbol as shown below:

Simple example:

```
(dw:menu-choose '(a b c) :default 'a)
```

If the items are themselves lists, then supply the default item in a fashion similar to that shown in the following example:

```
(setq item-list '(("One" :value 1) ("Two" :value 2)))
(dw:menu-choose item-list :default (first item-list))
```

:presentation-type
Specifies the presentation type of the items presented in the menu. This results in the printer for that presentation type being used to display the items in the menu.

Because each item (element) in *item-list* is passed to the presentation type's printer, using this option is, in general, only appropriate when *item-list* is a simple list of objects (as opposed to a "general list"). In this case, you might also consider using **dw:menu-choose-from-set** rather than **dw:menu-choose.**

:printer Specifies a function of two arguments for printing the menu items. The arguments are an object – one element of the *item-list* – and a stream. If both this option and the **:presentation-type** option are specified, the printer used is the one specified by this option, not that of the presentation type.

Example:

```
(dw:menu-choose '((a :value test)
                  (b :value 17))
                :printer
                #'(lambda (object stream)
                    (format stream
                      "xxx~Axxx" (car object))))
```

The example function pops up a menu displaying two choices, "xxxAxxx" and "xxxBxxx". Clicking on "xxxAxxx" returns TEST; clicking on "xxxBxxx" returns 17.

:near-mode

Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)** in *Programming the User Interface, Volume B.*

:superior

Specifies the window that is the superior of the menu window; the default is **tv:mouse-sheet.**

:center-p Boolean option specifying whether items displayed in the menu are centered, left to right. The default is **nil**, which causes the items to be flush left.

:character-style

Specifies the character style for display of menu items. The default is (:jess :roman :large).

If the **:style** option for an individual item is specified, this locally overrides the **:character-style** specified for the menu as a whole, but does not affect other items. The example below illustrates this:

```
(dw:menu-choose '(("First Choice" :value 1
                                  :style (nil nil :normal))
                  ("Second Choice" :value 2))
                :character-style
                  '(:serif :bold :very-large))
```

:momentary-p

Boolean option specifying whether the menu is momentary or temporary; the default is momentary. If you wish to make the menu temporary, supply a value of **nil** to this option and **t** to the **:temporary-p** option.

A momentary window is de-activated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

:temporary-p

Boolean option specifying whether the menu is temporary or momentary; the default is momentary. If you wish to make the menu temporary, supply a value of **t** to this option and **nil** to the **:momentary-p** option.

A momentary window is de-activated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

For an overview of **dw:menu-choose** and related facilities: See the section "Overview of Facilities for Accepting Single Objects", page 35.

dw:menu-choose-from-set *list presentation-type* &key (*printer* **nil**) (*prompt* **nil**) (*default* **nil**) (*near-mode* **'(:mouse)**) (*superior* **tv:mouse-sheet**) (*center-p* **dw::*default-menu-center-p***) (*character-style* **'(:jess :roman :large)**) (*momentary-p* **t**) (*temporary-p* **dw::momentary-p**) *Function*

Constructs a menu from a list of objects of a specified presentation type and returns the selected object.

This function is similar to **dw:menu-choose**, but is intended primarily for presenting a simple list of items in menu format, not items of the "general list" form that **dw:menu-choose** handles.

list The list of objects.

presentation-type

The presentation type used to present the objects (but see the **:printer** option below).

Examples:

```
(dw:menu-choose-from-set '(a b c) 'symbol)


(dw:menu-choose-from-set '(#p"sys:site;foo.bar"
                           #p"y:>doughty>a.b") 'pathname)


(setq item-list '("One" "Two"))
(dw:menu-choose-from-set item-list 'string)
```

:printer Specifies a function of two arguments for printing menu items. The arguments are an object – one element of *list* –

and a stream. If specified, this printer is used for displaying menu items rather than that of the specified *presentation-type*.

Example:

```
(dw:menu-choose-from-set '(#p"sys:site;config.data"
                           #p"y:>doty>examples.lisp") 'pathname
                         :printer
                          #'(lambda (object stream)
                              (write-string
                                (send object :name)
                                stream)))
```

The example function creates a menu displaying the choices "CONFIG" and "EXAMPLES". Pathname objects are still returned when clicked on; just the appearance in the menu has changed.

:prompt Specifies a string to use as a title for the menu. The menu title appears at the top of the menu.

:default Specifies an item to be the currently selected (highlighted) item when the menu is first displayed.

Examples:

```
(dw:menu-choose-from-set '(a b c) 'symbol :default 'a)

(setq item-list '("One" "Two"))
(dw:menu-choose-from-set item-list 'string
                         :default (first item-list))
```

:near-mode
Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)** in *Programming the User Interface, Volume B.*

:superior
Specifies the window that is the superior of the menu window; the default is **tv:mouse-sheet.**

:center-p Boolean option specifying whether items displayed in the menu are centered, left to right. The default is **nil**, which causes the items to be flush left.

:character-style
Specifies the character style for display of menu items. The default is (:jess :roman :large).

:momentary-p
Boolean option specifying whether the menu is momentary or temporary; the default is momentary. If you wish to make the menu temporary, supply a value of **nil** to this option and **t** to the **:temporary-p** option.

A momentary window is de-activated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

:temporary-p
Boolean option specifying whether the menu is temporary or momentary; the default is momentary. If you wish to make the menu temporary, supply a value of **t** to this option and **nil** to the **:momentary-p** option.

A momentary window is de-activated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

For an overview of **dw:menu-choose-from-set** and related facilities: See the section "Overview of Facilities for Accepting Single Objects", page 35.

dw:mouse-char-for-gesture *gesture* *Function*
Returns the mouse character associated with a gesture. You can use this function to assign a new gesture symbol to a mouse character.

gesture An existing or new gesture symbol.

To assign your own symbolic name to a mouse character, use the following form:

```
(setf (mouse-char-for-gesture gesture) #\mouse-x)
```

Conventionally, the *gesture* symbol is a keyword.

For an overview of **dw:mouse-char-for-gesture** and related facilities: See the section "Overview of Mouse Gesture Interface Facilities", page 41.

For information on mouse characters: See the section "Mouse Characters" in *Programming the User Interface, Volume B.*

dw:mouse-char-gesture *mouse-char* *Function*

Returns the standard gesture associated with a mouse character.

mouse-char
The mouse character (for example, #**mouse-m**).

For an overview of **dw:mouse-char-gesture** and related facilities: See the section "Overview of Mouse Gesture Interface Facilities", page 41.

For information on mouse characters: See the section "Mouse Characters" in *Programming the User Interface, Volume B.*

dw:mouse-char-gestures *mouse-char* *Function*

Returns a list of gestures associated with a mouse character.

mouse-char
The mouse character (for example, #**mouse-m**).

For an overview of **dw:mouse-char-gestures** and related facilities: See the section "Overview of Mouse Gesture Interface Facilities", page 41.

For information on mouse characters: See the section "Mouse Characters" in *Programming the User Interface, Volume B.*

dw:presentation-subtypep-cached *subtype supertype* *Function*

Determines whether one presentation type is a subtype of another presentation type.

subtype The putative subtype presentation type.

supertype The putative supertype presentation type.

This is like **dw:presentation-subtypep**, but it encaches the result of the lookup. (See the function **dw:presentation-subtypep**, page 382.) It is intended primarily for use in the **:tester** forms of mouse handlers. Although it is generally more convenient to use **dw:handler-applies-in-limited-context-p**, more complex testers may need **dw:presentation-subtypep-cached**, on which the former is based:

```
(defun handler-applies-in-limited-context-p
       (context limiting-context-type)
  (let ((context-type
          (presentation-input-context-presentation-type
            context)))
    (presentation-subtypep-cached
      context-type limiting-context-type)))
```

See the function **dw:handler-applies-in-limited-context-p**, page 192.

For an overview of **dw:presentation-subtypep-cached** and related facilities: See the section "Overview of Mouse Handler Facilities", page 39.

prompt-and-accept *presentation-type-or-args* &optional *format-string* &rest *format-args* *Function*

Prompts for and accepts user input. (This function is similar to **accept**; it differs in its reliance on the **format** function for creating the input prompt.)

presentation-type-or-args
: Presentation type of the input object or, alternatively, a list of keyword-value pairs.

  Available keywords are the same as those for **accept** with one exception. This is the **:type** keyword, specifying the presentation type of the input object. If keywords are provided, one of them must be **:type**.

format-string
: Control string for the **format** function.

format-args
: Arguments for the format specifiers included in the *format-string*.

Example:

```
(prompt-and-accept '(:type string :default "J. Doe")
                   "Please enter your name")
```

For an overview of **prompt-and-accept** and related facilities: See the section "Overview of Facilities for Accepting Single Objects", page 35.

PART V.

Dictionary of Program Output Facilities

16. Dictionary Notes

This dictionary includes reference documentation for both the basic and advanced facilities provided for doing program output. These are listed in the following two tables:

Table of Basic Program Output Facilities

Basic Presentation Output Facilities
- **present**
- **present-to-string**
- **dw:with-output-as-presentation**

Character Environment Facilities
- **with-character-face**
- **with-character-family**
- **with-character-size**
- **with-character-style**
- **abbreviating-output**
- **filling-output**
- **indenting-output**
- **with-underlining**

Textual List Formatting Facilities
- **format-textual-list**
- **formatting-textual-list**
- **formatting-textual-list-element**

Table Formatting Facilities
- **formatting-multiple-columns**
- **format-sequence-as-table-rows**
- **format-item-list**
- **formatting-item-list**
- **formatting-table**
- **formatting-column**
- **formatting-column-headings**
- **formatting-row**
- **formatting-cell**
- **format-cell**

Graph Formatting Facilities
- **format-graph-from-root**
- **formatting-graph**
- **formatting-graph-node**
- **dw:find-graph-node**

Graphic Output Facilities
- **graphics:draw-string**
- **graphics:draw-point**
- **graphics:draw-arrow**
- **graphics:draw-line**
- **graphics:draw-lines**
- **graphics:draw-cubic-spline**
- **graphics:draw-circle**
- **graphics:draw-ellipse**
- **graphics:draw-triangle**
- **graphics:draw-rectangle**
- **graphics:draw-glyph**
- **graphics:draw-polygon**
- **graphics:draw-regular-polygon**
- **graphics:draw-convex-polygon**
- **graphics:draw-pattern**

Progress Indicator Facilities
- **tv:noting-progress**
- **tv:note-progress**
- **tv:dolist-noting-progress**
- **tv:dotimes-noting-progress**

Other Facilities for Program Output
- **(flavor:method :clear-window dw:dynamic-window)**
- **(flavor:method :clear-history dw:dynamic-window)**
- **(flavor:method :clear-region dw:dynamic-window)**
- **(flavor:method :delete-displayed-presentation dw:dynamic-window)**
- **(flavor:method :visible-cursorpos-limits dw:dynamic-window)**
- **(flavor:method :set-viewport-position dw:dynamic-window)**
- **(flavor:method :y-scroll-position dw:dynamic-window)**
- **(flavor:method :y-scroll-to dw:dynamic-window)**
- **(flavor:method :x-scroll-position dw:dynamic-window)**
- **(flavor:method :x-scroll-to dw:dynamic-window)**
- **(flavor:method :with-output-recording-disabled dw:dynamic-window)**
- **dw:with-own-coordinates**
- **surrounding-output-with-border**
- **dw:tracking-mouse**

dw::with-output-truncation
dw:displayed-presentation-set-highlighting
dw:displayed-presentation-clear-highlighting

Table of Advanced Program Output Facilities

Advanced Presentation Output Facilities
dw:with-output-to-presentation-recording-string
dw:with-replayable-output
dw:with-resortable-output

Redisplay Facilities
dw:redisplayable-present
dw:redisplayable-format
dw:independently-redisplayable-format
dw:with-redisplayable-output
dw:redisplayer
dw:do-redisplay

Facilities for Writing Formatted Output Macros
dw:continuation-output-size
dw:named-value-snapshot-continuation

In the dictionary, the facilities are arranged in alphabetical order (package prefixes excluded).

For conceptual documentation: See the section "Overview of Program Output Facilities", page 47.

17. The Facilities

abbreviating-output (&optional *stream* &key *width height* *lozenge-returns show-abbreviation* *abbreviate-initial-whitespace*) &body *body* *Macro*

Binds local environment such that character output is abbreviated. That is, output exceeding a specified width or height (in characters) is truncated.

stream The output stream; the default is ***standard-output***.

:width Specifies the width, in characters, beyond which abbreviation occurs, or **t** or **nil**. If **nil**, the default, individual lines are not truncated. If **t**, the width used is the value returned by the stream's **:size-in-characters**.

:height Specifies the height, in lines, beyond which abbreviation occurs, or **t** or **nil**. If **nil**, the default, no truncation occurs. If **t**, the height used is the value returned by the stream's **:size-in-characters**.

:lozenge-returns
Boolean option specifying whether **#\return** characters at line truncations are displayed within a lozenge, rather than causing a newline; the default is **nil**.

:show-abbreviation
Boolean option specifying whether an ellipsis (...) is displayed where output truncation occurs. The default is **nil**, meaning that there is no explicit indication that truncation has occurred.

:abbreviate-initial-whitespace
Boolean option specifying that initial whitespace (spaces, tabs, newlines) be suppressed; the default is **nil**.

Example:

```
(defun abbrev-test (width height lozenge-p)
  (abbreviating-output (() :width width :height height
                           :lozenge-returns lozenge-p
                           :show-abbreviation t)
    (loop for row from 1 to 20 do
      (terpri)
      (loop for col from 1 to 100 do
        (format T " ~d:~d" row col)))))

(abbrev-test 42 10 nil)
```

The *body* code continues to run normally to completion, even though its output to *stream* may be truncated.

Within **abbreviating-output**, the **:set-cursorpos** operation is restricted. Only the *x* position may be specified, and then, only in characters.

For an overview of **abbreviating-output** and related facilities: See the section "Overview of Character Environment Facilities", page 49.

:clear-history of **dw:dynamic-window** *Method*

Eliminates all items in the output history of the window, and resets the viewport to the top of the history.

For an overview of **(flavor:method :clear-history dw:dynamic-window)** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

:clear-region *left top right bottom* of **dw:dynamic-window** *Method*

Clears the output display in a rectangular area of the window. Specify the region in terms of absolute window coordinates.

left The x-coordinate for the left edge of the cleared area.

top The y-coordinate for the top edge of the cleared area.

right The x-coordinate for the right edge of the cleared area.

bottom The y-coordinate for the bottom edge of the cleared area.

For an overview of **(flavor:method :clear-region dw:dynamic-window)** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

:clear-window of **dw:dynamic-window** *Method*

Scrolls the window forward in the vertical dimension far enough to eliminate previous output from the current display. Note that only the display is affected, not the window's output history.

For an overview of **(flavor:method :clear-window dw:dynamic-window)** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

dw:continuation-output-size *continuation stream* &optional (*unit* **:pixel**) *Function*

Determines the amount of space a specified continuation would require for output on a specified stream. Four values are returned: *width*, *height*, *cursor-motion-x*, and *cursor-motion-y*.

The *continuation* is **funcall**ed with a single argument, an internal stream, which tracks the cursor motion caused by the output code of *continuation*.

continuation
The continuation to run.

stream The output stream. This must be supplied, even though no output is actually sent to it, because information about the stream is necessary. For example, if a **:string-length** message is involved, the default character style for the stream is needed information.

unit The unit of measure. The default is **:pixel**; the other possible value is **:character**.

Example:

```
(defmacro centering-about-point ((stream x y) &body body)
  `(centering-about-point-internal
     (zl:named-lambda centering-about-point (,stream) ,@body)
     ,stream ,x ,y))

(defun centering-about-point-internal (continuation stream x y)
  (multiple-value-bind (width height)
      (dw:continuation-output-size continuation stream)
    (let ((start-x (- x (round width 2)))
          (start-y (- y (round height 2))))
      (dw:in-sub-window (stream start-x start-y width height)
        (funcall continuation stream))
      ;; Drawing the lines is just to verify the centering
      (graphics:draw-line start-x start-y (+ start-x width) (+ start-y height))
      (graphics:draw-line (+ start-x width) start-y start-x (+ start-y height))
      )))
```

```
;;; Some code to test it
(defun test-centering ()
  (send *standard-output* :clear-window)
  (multiple-value-bind (left top right bottom)
      (send *standard-output* :visible-cursorpos-limits)
    (let ((center-x (round (+ left right) 2))
          (center-y (round (+ top bottom) 2)))
      (centering-about-point (*standard-output* center-x center-y)
        ;; Surround with border just to show the bounding box of the output
        (dw:surrounding-output-with-border (*standard-output*)
          ;; Generate some output
          (cp:execute-command "Show Flavor Handler"
                              ':tyo 'dw:dynamic-window
                              :code :detailed))))
    ;; Drawing the lines is just to verify the centering
    (graphics:draw-line left top right bottom)
    (graphics:draw-line right top left bottom)
    ;; Pause to read a character before the command prompt
    ;;clobbers our carefully crafted output.
    (read-char)))

;; In a full-size Lisp Listener, try
;; (with-character-size (:large) (test-centering))
```

For an overview of **dw:continuation-output-size** and related facilities: See the section "Overview of Facilities for Writing Formatted Output Macros", page 66.

:delete-displayed-presentation *displayed-presentation* of **dw:dynamic-window** *Method*

Deletes an already displayed presentation from a Dynamic Window's output history and display.

displayed-presentation
The presentation to delete.

For an overview of **(flavor:method :delete-displayed-presentation dw:dynamic-window)** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

dw:displayed-presentation-clear-highlighting *Generic Function*
displayed-presentation window &optional
(*highlighting-mode* **:underline**)

Eliminates highlighting of a displayed presentation (see **dw:displayed-presentation-set-highlighting**).

displayed-presentation
The highlighted presentation.

window The window displaying the presentation.

highlighting-mode
The mode in which the displayed presentation is highlighted, either **:underline** (the default for **dw:displayed-presentation-set-highlighting**) or **:inverse-video**.

For an overview of **dw:displayed-presentation-clear-highlighting** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

dw:displayed-presentation-set-highlighting *Generic Function*
displayed-presentation window &optional
(*highlighting-mode* **:underline**)

Highlights a displayed presentation.

displayed-presentation
The presentation to highlight.

window The window displaying the presentation.

highlighting-mode
Either **:underline** (the default) or **:inverse-video**.

For an overview of **dw:displayed-presentation-set-highlighting** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

dw:do-redisplay *redisplay-piece* &optional (*stream* ***standard-output***) &key *full-set-cursorpos truncate-p* *Generic Function*

Causes incremental redisplay from a redisplay object (created by **dw:redisplayer**: See the macro **dw:redisplayer**, page 258.). It runs the code in the *body* of the redisplayer, doing output to *stream* with respect to the display cache points described under **dw:with-redisplayable-output**: (See the macro **dw:with-redisplayable-output**, page 273.)

redisplay-piece
The redisplay object.

stream The output stream; the default is ***standard-output***.

:full-set-cursorpos
Boolean option specifying whether the cursor will move backwards or sideways, rather than in strict tty style, so that a special stream is necessary; the default is **nil**.

:truncate-p
Option specifying whether to do the redisplay with the output stream in truncate mode. With `:truncate-p nil`, the default, the output window rescrolls to update separate parts of the display. With `:truncate-p t`, some updating happens off-screen.

A third value permitted for this option is **:if-necessary**. In this case, **dw:do-redisplay** simulates, if necessary, some cursor motion on behalf of the output stream.

dw:do-redisplay is one of a number of facilities used to do incremental redisplay. For examples, see the file sys:examples;incremental-redisplay.lisp.

For an overview of **dw:do-redisplay** and related facilities: See the section "Overview of Advanced Presentation Output Facilities", page 63.

tv:dolist-noting-progress (*var listform name* &optional (*progress-note-variable* **'tv:*current-progress-note***) (*process* **'sys:current-process**)) &body *body* *Macro*

Binds local environment such that the progress of a **dolist** special form is noted by a progress bar displayed in the status line at the bottom of the screen.

var A variable bound to each successive element in *listform* on each successive iteration.

listform The list.

name A string naming the operation being noted. This string is displayed with the progress bar.

progress-note-variable
The variable bound to the note object; the default is **tv:*current-progress-note***.

process The process on whose behalf the progress is noted; the default is **sys:current-process**. This is used to determine the precedence of notes.

Example:

```
(defun note-element-printing (list)
  (tv:dolist-noting-progress (element list "Printing elements")
    (print element)
    (sleep 1)))
```

For an overview of **tv:dolist-noting-progress** and related facilities: See the section "Overview of Progress Indicator Facilities", page 59.

tv:dotimes-noting-progress (*var countform name* &optional (*progress-note-variable* **'tv:*current-progress-note***) (*process* **'sys:current-process**)) &body *body* *Macro*

Binds local environment such that the progress of a **dotimes** special form is noted by a progress bar displayed in the status line at the bottom of the screen.

var A variable bound to the count (0, 1, 2, and so on) on each successive iteration.

countform
The number of iterations.

name A string naming the operation being noted. This string is displayed with the progress bar.

progress-note-variable
The variable bound to the note object; the default is **tv:*current-progress-note***.

process The process on whose behalf the progress is noted; this is used to determine the precedence of notes.

Example:

```
(defun note-square-roots (n)
  (tv:dotimes-noting-progress
    (count n "Calculating square roots")
    (sqrt count)))
```

For an overview of **tv:dotimes-noting-progress** and related facilities: See the section "Overview of Progress Indicator Facilities", page 59.

graphics:draw-arrow *from-x from-y to-x to-y* &key (*stream* ***standard-output***) (*alu* **:draw**) (*arrow-head-length* **graphics::*default-arrow-length***) (*arrow-base-width* **graphics::*default-arrow-width***) (*dashed* **nil**) *dash-spacing dash-length initial-dash-phase* (*draw-partial-dashes* **t**) *Function*

Draws an arrow.

from-x The x-coordinate for the base of the arrow.

from-y The y-coordinate for the base of the arrow.

to-x The x-coordinate for the tip of the arrow.

to-y The y-coordinate for the tip of the arrow.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

- **:draw** Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.
- **:erase** Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.
- **:flip** Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:arrow-head-length
Specifies the length, in pixels, of the arrowhead; the default is 12.

:arrow-base-width
Specifies the width, in pixels, of the base of the arrowhead; the default is 5.

:dashed Boolean option specifying whether the shaft of the arrow is drawn as a series of dashes; the default is **nil**.

:dash-spacing
Specifies the distance, in pixels, between dashes.

:dash-length
Specifies the length, in pixels, of individual dashes.

:initial-dash-phase
Specifies the offset, in pixels, of the start of the first dash from the starting point of the line.

:draw-partial-dashes
Boolean option specifying whether a partial dash is drawn at the end of a dashed line so that it reaches its specified end-point; the default is **t**.

For an overview of **graphics:draw-arrow** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-circle *center-x center-y radius* &key (*stream* ***standard-output***) (*alu* **:draw**) (*inner-radius* **0**) (*thickness* **1**) (*start-angle* **0**) (*end-angle* **(* pi 2)**) (*filled* **t**) (*mask* **nil**) *Function*

Draws a circle.

center-x The x-coordinate for the center of the circle.

center-y The y-coordinate for the center of the circle.

radius The radius of the circle.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

- **:draw** Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.
- **:erase** Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.
- **:flip** Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:inner-radius
Specifies the inner radius of a (filled) ring figure. When using this option, do not specify the **:thickness** and **:filled** options.

:thickness
Specifies the thickness, in pixels, of the line drawn; the default is 1.

If you specify this option to a value greater than 1, the resulting figure is unfilled, regardless of the value specified by the **:filled** option. The extra thickness is added to the interior of the figure.

:start-angle
[Not implemented]

:end-angle
[Not implemented]

:filled Boolean option specifying whether all pixels within the figure are turned on, or only the outline pixels; the default is **t**.

:mask [Not implemented]

For an overview of **graphics:draw-circle** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-convex-polygon *points* &key (*stream* ***standard-output***) (*alu* **:draw**) (*mask* **nil**) (*filled* **t**) *Function*

Draws a convex polygon. This function is more efficient than **graphics:draw-polygon** for drawing convex polygons. If you cannot guarantee that the *points* argument will describe a convex figure, use the latter function.

points A list of points in the form (*x1 y1 x2 y2 ... xn yn*); these form the points of the polygon.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:mask [Not implemented]

:filled Boolean option specifying whether all pixels within the figure are turned on, or only the outline pixels; the default is **t**.

For an overview of **graphics:draw-convex-polygon** and related functions:
See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-cubic-spline *points* &key (*stream* ***standard-output***) (*alu* **:draw**) (*thickness* **1**) (*filled* **nil**) (*start-relaxation* **:relaxed**) (*end-relaxation* **graphics::start-relaxation**) (*mask* **nil**) *Function*

Draws a cubic spline through a series of points.

points A list of points in the form (*x1 y1 x2 y2 ... xn yn*).

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:thickness
Specifies the thickness, in pixels, of the line drawn; the default is 1.

:filled Boolean option specifying whether all pixels within the figure are turned on, or only the outline pixels; the default is **t**.

[Not implemented for **graphics:draw-cubic-spline**.]

:start-relaxation
Determines the derivative of the curve at its starting point (*x1 y1*) and, by default, its ending point (*xn yn*). Three values are possible:

:relaxed The derivatives at the end-points are set to values that continue the trend of the curve established by neighboring points.

:cyclic Forces the derivatives at the two end-points of the curve to be equal. When the starting and ending points are equal, this value results in a smooth, continuous curve.

:anti-cyclic
Forces the derivatives at the two end-points of the curve to be equal in magnitude but opposite in sign. When the starting and ending points are equal, this value causes the curve to come to a point.

:end-relaxation
By default, this is set to the value of **:start-relaxation**, which is always what you want.

:mask [Not implemented]

For an overview of **graphics:draw-cubic-spline** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-ellipse *center-x center-y semi-major-x semi-major-y semi-minor-x semi-minor-y* &key (*stream* ***standard-output****) (*alu* **:draw**) (*filled* **nil**) (*mask* **nil**) (*thickness* **1**) *Function*

Draws an ellipse. The ellipse is aligned rectangularly; that is, the axes of the ellipse must lie along the vertical and horizontal axes of the display window.

center-x The horizontal center of the ellipse.

center-y The vertical center of the ellipse.

semi-major-x
The x-coordinate of one end-point of the major axis.

semi-major-y
The y-coordinate of one end-point of the major axis.

semi-minor-x
The x-coordinate of one end-point of the minor axis.

semi-minor-y
The y-coordinate of one end-point of the minor axis.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:filled Boolean option specifying whether all pixels within the figure are turned on, or only the outline pixels; the default is **t**.

:mask [Not implemented]

:thickness
Specifies the thickness, in pixels, of the line drawn; the default is 1.

For an overview of **graphics:draw-ellipse** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-glyph *index font x y* &key (*stream* ***standard-output***) (*alu* **:draw**) *Function*

Draws a figure referenced by a font array.

index The index into the font array.

font The font.

x The x-coordinate where the glyph is drawn (left edge).

y The y-coordinate where the glyph is drawn (baseline).

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

Example:

```
(defun glyph-example ()
  (graphics:draw-glyph (char-code #\∞) fonts:mouse 100 100))
```

To see the elements of a font, use the Show Font command. To see the list of loaded fonts, press the HELP key to the Show Font command. For more information on fonts, including information on how to create your own: See the section "Font Editor" in *Text Editing and Processing*.

For an overview of **graphics:draw-glyph** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-line *start-x start-y end-x end-y* &key (*stream* ***standard-output***) (*alu* **:draw**) (*draw-end-point* **t**) (*mask* **nil**) (*thickness* **1**) (*dashed* **nil**) *dash-spacing dash-length initial-dash-phase* (*draw-partial-dashes* **t**) *Function*

Draws a line.

start-x The x-coordinate of the starting point.

start-y The y-coordinate of the starting point.

end-x The x-coordinate of the ending point.

end-y The y-coordinate of the ending point.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned

on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:draw-end-point
Boolean option specifying whether to draw the final point; the default is **t**.

:mask [Not implemented]

:thickness
Specifies the thickness, in pixels, of the line drawn; the default is 1.

:dashed Boolean option specifying whether the line is drawn as a series of dashes; the default is **nil**.

:dash-spacing
Specifies the distance, in pixels, between dashes.

:dash-length
Specifies the length, in pixels, of individual dashes.

:initial-dash-phase
Specifies the offset, in pixels, of the start of the first dash from the starting point of the line.

:draw-partial-dashes
Boolean option specifying whether a partial dash is drawn at the end of a dashed line so that it reaches its specified end-point; the default is **t**.

For an overview of **graphics:draw-line** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-lines *points* &key (*stream* ***standard-output***) (*alu* **:draw**) (*mask* **nil**) (*thickness* **1**) (*draw-end-point* **t**) (*closed* **nil**) (*dashed* **nil**) *dash-spacing dash-length initial-dash-phase* *Function*

(*draw-partial-dashes* **t**)

Draws a connected series of line segments.

points A list of points in the form (*x1 y1 x2 y2 ... xn yn*).

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:mask [Not implemented]

:thickness
Specifies the thickness, in pixels, of the line drawn; the default is 1.

:draw-end-point
Boolean option specifying whether to draw the final point; the default is **t**.

:dashed Boolean option specifying whether the lines are drawn as series of dashes; the default is **nil**.

:dash-spacing
Specifies the distance, in pixels, between dashes.

:dash-length
Specifies the length, in pixels, of individual dashes.

:initial-dash-phase
Specifies the offset, in pixels, of the start of the first dash from the starting point of the line.

:draw-partial-dashes
Boolean option specifying whether a partial dash is drawn at the end of a dashed line so that it reaches its specified end-point; the default is **t**.

For an overview of **graphics:draw-lines** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-pattern *left top pattern* &key (*stream* ***standard-output***) (*alu* **:draw**) (*right* **nil**) (*bottom* **nil**) (*pattern-left* **0**) (*pattern-top* **0**) (*copy-pattern* **nil**) *Function*

Draws a pattern of display bits/pixels. To copy the pattern over a wide area, use the **:right** and **:bottom** options to this function.

left The left edge of the window area where the pattern is drawn.

top The top edge of the window area where the pattern is drawn.

pattern The raster array containing the pattern to be drawn. You can provide your own raster – the **tv:make-binary-gray** function is useful here (see example below) – or use one of the standard, background-gray patterns: **tv:25%-gray**, **tv:33%-gray**, **tv:50%-gray**, or **tv:75%-gray**.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:right Specifies the right edge of the window area where the pattern is drawn.

:bottom Specifies the bottom edge of the window area where the pattern is drawn.

:pattern-left
Specifies the column in the pattern array that is the first one drawn. This controls the pattern's phase in the horizontal dimension.

:pattern-top
Specifies the row in the pattern array that is the first one drawn. This controls the pattern's phase in the vertical dimension.

:copy-pattern
Boolean option specifying whether to copy the pattern array to another array, used when displaying the output history of the window.

The default is **nil**, appropriate when the pattern array remains constant. However, if the pattern array can change between calls to this function, and you wish the originally displayed pattern to be the one retained in the output history, then set this option to **t**.

Examples:

```
(defun standard-gray-pattern ()
  (dw:with-own-coordinates ()
    (graphics:draw-pattern 300 100 tv:50%-gray :right  500
                                               :bottom 300)))

(defun ones-pattern ()
  (let ((raster (tv:make-binary-gray 8 8
                 '(#b00000000     ; The picture of what you
                   #b00001000     ; want the bit pattern
                   #b00111000     ; displayed to look like, in
                   #b00001000     ; this case the number 1.
                   #b00001000     ; Notice the #b in front of
                   #b00001000     ; each number to force the
                   #b00001000     ; reader into binary.
                   #b00111110))))
    (dw:with-own-coordinates ()
      (graphics:draw-pattern 300 300 raster :right  500
                                            :bottom 500))))
```

For an overview of **graphics:draw-pattern** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-point *x y* &key (*stream* ***standard-output***) (*alu* **:draw**) *Function*

Draws a point.

x The point's x-coordinate.

y The point's y-coordinate.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

For an overview of **graphics:draw-point** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-polygon *points* &key (*stream* ***standard-output***) (*alu* **:draw**) (*mask* **nil**) (*filled* **t**) *Function*

Draws a polygon connecting a set of points.

points A list of points in the form (*x1 y1 x2 y2 ... xn yn*); these form the points of the polygon.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:mask [Not implemented]

:filled Boolean option specifying whether all pixels within the figure are turned on, or only the outline pixels; the default is **t**.

For an overview of **graphics:draw-polygon** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-rectangle *left top right bottom* &key (*stream* ***standard-output***) (*alu* **:draw**) (*filled* **t**) (*thickness* **1**) (*pattern* **t**) (*mask* **nil**) *Function*

Draws a rectangle.

left The x-coordinate of the left side of the rectangle.

top The y-coordinate of the top side of the rectangle.

right The x-coordinate of the bottom side of the rectangle.

bottom The y-coordinate of the bottom side of the rectangle.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:filled Boolean option specifying whether all pixels within the figure are turned on, or only the outline pixels; the default is **t**.

:thickness
Specifies the thickness, in pixels, of the line drawn; the default is 1.

To make use of this option, you must specify the **:filled** option with a value of **nil**. The additional thickness is added to the interior of the figure.

:pattern Specifies a raster array containing a pattern to be drawn within the rectangle. You can provide your own raster – see the example given for **graphics:draw-pattern** – or use one of the standard, background-gray patterns: **tv:25%-gray**, **tv:33%-gray**, **tv:50%-gray**, or **tv:75%-gray**.

This option overrides the `:filled t` option.

:mask [Not implemented]

For an overview of **graphics:draw-rectangle** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-regular-polygon *start-x start-y end-x end-y number-of-sides* &key (*stream* ***standard-output***) (*alu* **:draw**) (*handedness* **:left**) (*mask* **nil**) (*filled* **t**) *Function*

Given the starting and ending coordinates for a single side and the number of sides, draws a regular polygon.

start-x The x-coordinate of the starting point for side 1.

start-y The y-coordinate of the starting point for side 1.

end-x The x-coordinate of the ending point for side 1.

end-y The y-coordinate of the ending point for side 1.

number-of-sides
The total number of sides.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:handedness
Specifies whether the polygon is drawn to the **:left** or **:right** of side 1. The default is **:left**, meaning that, if you were located at (*start-x start-y*) and facing (*end-x end-y*), the polygon would be drawn to your left.

:mask [Not implemented]

:filled Boolean option specifying whether all pixels within the figure are turned on, or only the outline pixels; the default is **t**.

For an overview of **graphics:draw-regular-polygon** and related functions:
See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-string *string start-x start-y* &key (*toward-x* **(1+ graphics::start-x)**) (*toward-y* **graphics::start-y**) (*stream* ***standard-output***) (*alu* **:draw**) (*stretch-p* **nil**) (*mask* **nil**) *Function*

Draws a character string.

string The string.

start-x The x-coordinate where drawing of the string begins. This argument specifies the left edge of the first character.

start-y The y-coordinate where drawing of the string begins. This argument specifies the baseline of the first character.

:toward-x
The x-coordinate toward which the string is drawn. The default value is one greater than the starting x-coordinate, meaning that the string is drawn to the right; its deviation from the horizontal is determined by the **:toward-y** option.

:toward-y
The y-coordinate toward which the string is drawn. The default value is equal to the starting y-coordinate, meaning that the string is drawn horizontally.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:stretch-p

Boolean option specifying whether the characters are spaced evenly between the starting (*start-x start-y*) and ending (**:toward-x** <*end-x*> **:toward-y** <*end-y*>) coordinates.

If the space provided is greater than that required by the default spacing between characters of the given style, then additional spacing is inserted; the string is stretched. If the space is less than that required by the default spacing, space is eliminated; the string is compressed.

The default is **nil**, meaning that the default spacing for the character style in question is used, regardless of the distance between the starting and ending coordinates.

:mask [Not implemented]

To control the character style in which the string is drawn, use this function inside the body of a character-style macro, for example, **with-character-style**.

For an overview of **graphics:draw-string** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

graphics:draw-triangle *x1 y1 x2 y2 x3 y3* &key (*stream* ***standard-output***) (*alu* **:draw**) (*filled* **t**) (*mask* **nil**) *Function*

Draws a triangle.

x1 The x-coordinate of the first point of the triangle.

y1 The y-coordinate of the first point of the triangle.

x2 The x-coordinate of the second point of the triangle.

y2 The y-coordinate of the second point of the triangle.

x3 The x-coordinate of the third point of the triangle.

y3 The y-coordinate of the third point of the triangle.

:stream Specifies the output stream; the default is ***standard-output***.

:alu Specifies the drawing mode. Possible values for this option are:

:draw Pixels specified by the graphics function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the graphics function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the graphics function are turned on if they were previously off, and off if they were previously on.

:filled Boolean option specifying whether all pixels within the figure are turned on, or only the outline pixels; the default is **t**.

:mask [Not implemented]

For an overview of **graphics:draw-triangle** and related functions: See the section "Overview of Graphic Output Facilities", page 57.

filling-output (&optional *stream* &key *fill-column* (*fill-on-spaces* **t**) *after-line-break after-line-break-initially-too*) &body *body* *Macro*

Binds local environment such that character output is filled; that is, **filling-output** makes sure that any output done within its body does not split "words" across lines.

"Words" are separated by spaces. When a line is broken to keep from wrapping past the end of a line, the line break is made at a space.

stream The output stream; the default is ***standard-output***.

:fill-column

Specifies the length, in pixels, of filled lines. **:fill-column** is the cursorpos of the right end of the line.

If **:fill-column** is unspecified, line length is determined as follows: If the underlying stream supports the

:visible-cursorpos-limits message, as do all Dynamic Windows, the right-hand cursorpos limit is used. Otherwise, if the underlying stream supports the **:inside-size** message, the inside size is used. If neither of the two preceding messages are supported, simple character counting is used, and lines are filled to 95 characters in width.

:after-line-break
Specifies a string to be sent to *stream* after line breaks; the string appears at the beginning of each new line.

:after-line-break-initially-too
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**.

:fill-on-spaces
Boolean option specifying whether lines are to be filled at spaces, the default behavior.

If **nil**, lines are filled at the text resulting from sending the **:conditional-string-out** message (undocumented) to the internal stream supporting **filling-output**.

Example:

```
(defun filling-test ()
  (fresh-line)
  (filling-output (*standard-output*
                   :fill-column 420
                   :after-line-break "Hi Dad, "
                   :after-line-break-initially-too t)
    (loop for i from 1 to 100
          do
      (Format t "Hello Mom! "))))
```

For an overview of **filling-output** and related facilities: See the section "Overview of Character Environment Facilities", page 49.

dw:find-graph-node *redisplay-helper-stream id* &key (*key* #'**identity**) (*test* #'**eql**) *Generic Function*

Searches for a node object given its symbol and the output stream on which it is to be displayed. The function returns the object if it finds it, **nil** otherwise.

Node objects are created with **formatting-graph-node**: See the macro **formatting-graph-node**, page 242. Also, see that facility for an example.

redisplay-helper-stream
The output stream for the node. This should be the same stream in use by **formatting-graph**.

id The symbol for the node: See the macro **formatting-graph-node**, page 242.

:key Specifies a function applied to a node object before comparison with *id*.

:test Specifies the function used to compare node objects with the one specified by *id*.

For an overview of **dw:find-graph-node** and related facilities: See the section "Graph Formatting Facilities", page 56.

format-cell *object printer* &key (*stream* ***standard-output****) *align* *Function*
Controls the printing of a table element within a **formatting-table** or **formatting-item-list** macro (see **formatting-table** for an example).

object The table element.

printer The function used to display each element. The function is passed two arguments, the *object* and the output stream. You can have the function write to the stream any information you want included in the table for that element.

:stream Specifies the output stream; the default is ***standard-output****.

:align Specifies how elements of a column should be aligned. The default (**nil**), causes the elements to be flush-left in the column. The other possible values are **:right** (flush-right) and **:center** (centered).

For an overview of **format-cell** and related facilities: See the section "Overview of Table Formatting Facilities", page 52.

format-graph-from-root *root-object object-printer inferior-producer* *Function*
&key (*stream* ***standard-output****)
(*dont-draw-duplicates* **nil**) (*key* #'**identity**) (*test* #'**eql**) (*root-is-sequence* **nil**) (*orientation* **:vertical**) (*direction* **:after**)
(*default-drawing-mode* **:line**) (*cutoff-depth* **nil**)

(*balance-evenly* **nil**) (*border*
'(:shape :rectangle)) (*row-spacing* **40**)
(*within-row-spacing* **20**) (*column-spacing* **20**)
(*within-column-spacing* **10**)

Constructs and displays a tree graph.

root-object
The root element of the set, from which the tree can be derived.

object-printer
A function used to display each tree node. 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 extract the inferiors from a node object. It is passed one argument, the node in question.

:stream Specifies the output stream; the default is ***standard-output***.

:dont-draw-duplicates
Boolean option specifying whether items that are duplicated in the tree are drawn only once (with all the reference lines drawn to the same object) or multiple times (once for each occurrence in the tree); the default is **nil**. (See the **:test** and **:key** options.)

:key Specifies the function used to extract the node object attribute used for duplicate comparison. The default is **identity**, that is, the object itself.

:test Specifies the test function used for duplicate detection. The default is **eql**.

:root-is-sequence
Specifies that the value supplied for *root-object* is a sequence. Each element of the sequence becomes a separate root. (The resulting graphs might not themselves be separate if the **:dont-draw-duplicates** option is **t**.)

:orientation
Specifies **:vertical** or **:horizontal** orientation for the graph display.

:direction

Specifies whether new nodes should be drawn above, below, left, or right of the current node. Possible values are **:after** and **:before**; the default is **:after**.

For `:orientation :horizontal`, **:after** means to the right, **:before** to the left. For `:orientation :vertical`, **:after** means below, **:before** means above.

:default-drawing-mode

Specifies the drawing mode used to connect nodes of the tree. The default is **:line** , which connects the nodes with solid lines. Other modes are **:dashed-line**, **:arrow**, and **:dashed-arrow**.

:cutoff-depth

Specifies how many levels of each branch of the tree should be explored.

:balance-evenly

Specifies whether the subtrees of the tree should all be the same size (width or height, depending on **:orientation**), the size of the largest subtree. The default is **nil**.

:border Specifies the shape and thickness, in pixels, of the border drawn around each node. The default is `(:shape :rectangle :thickness 1)`. Other possible shapes are **:circle**, **:oval**, and **:diamond**. **nil** means no border.

Abbreviations:

Full Form	Abbreviated Form
`:border (:shape` *xxxx*`)`	`:border` *xxxx*
`:border (:thickness` *n*`)`	`:border` *n*

:row-spacing

For **:vertical** orientation, specifies the spacing, in pixels, between rows of tree nodes; the default is `40`.

:within-row-spacing

For **:horizontal** orientation, specifies the spacing, in pixels, between columns of tree nodes; the default is `20`.

:column-spacing

For **:vertical** orientation, specifies the spacing, in pixels, between columns of tree nodes; the default is `20`.

:within-column-spacing
For **:horizontal** orientation, specifies the spacing, in pixels, between rows of tree nodes; the default is 10.

Examples:

```
(defun format-graph-from-root-example-1 ()
  (fresh-line)
  ;; you wouldn't actually bother to write this let, but it makes
  ;; for a clearer example.
  (let ((root (pkg-find-package "hardcopy"))
        (print-function #'princ)
        (inferior-producer #'si:pkg-used-by-list))
    (format-graph-from-root root
                            print-function
                            inferior-producer)))


;;; Try calling the following flavor-component grapher first
;;; first on simple flavors like net:object and tv:minimum-window.
;;; More complex flavors let you exercise the horizontal scrolling
;;; capability of Dynamic Windows.

(defun graph-flavor-components (flavor-name)
  (labels ((component-flavors (flavor-name)
             (let* ((fl (flavor:find-flavor flavor-name)))
               (remove flavor-name
                       (cond  ((flavor::flavor-components-composed fl)
                               (flavor:flavor-all-components fl))
                              (t (flavor::compose-flavor-components
                                   flavor-name)))))))
    (fresh-line)
    (format-graph-from-root flavor-name
                            #'(lambda (thing stream)
                                (present thing 'flavor:flavor
                                         :stream stream))
                            #'component-flavors
                            :dont-draw-duplicates t)))
```

Do not use **format-graph-from-root** to draw circular graphs.

For an overview of **format-graph-from-root** and related facilities: See the section "Overview of Graph Formatting Facilities", page 56.

format-item-list *list* &key (*stream* ***standard-output****) *printer* *Function*
presentation-type (*fresh-line* **t**) (*return-at-end* **t**)
(*order-columnwise* **t**) (*optimal-number-of-rows*
si:*optimal-number-of-rows*)
(*additional-indentation* **2**)
(*equalize-column-widths* **nil**) *max-width*
max-height

Displays the elements of a list in a tabular format.

list The list of items to display.

:stream Specifies the output stream; the default is ***standard-output****.

:printer Specifies the function used to display the items; the default printer is **princ**. The function is passed two arguments, an item and the output stream.

This option and the **:presentation-type** option are mutually exclusive. For

:presentation-type
Specifies the presentation type used to display the items. Items are output via calls to **present**. Items output as presentations can be used as mouse-sensitive input in the proper input context.

This option and the **:printer** option are mutually exclusive.

:fresh-line
Boolean option specifying whether a **fresh-line** operation should be performed on the output stream before the table is displayed; the default is **t**.

:return-at-end
Boolean option specifying whether a newline should be printed on the output stream when the table display is completed; the default is **t**.

:order-columnwise
Boolean option specifying whether table items are ordered as a series of columns, the default, or rows.

```
Column-wise ordering     Row-wise ordering
      1  4  7                 1  2  3

      2  5  8                 4  5  6

      3  6  9                 7  8  9
```

:optimal-number-of-rows
Specifies the number of rows in the table. If the number of rows specified is too small or too large to accommodate the list of items supplied, the appropriate number of rows closest to that specified is used.

:additional-indentation
Specifies the number of characters by which the left margin of the table is indented; the default is 2.

:equalize-column-widths
Boolean option specifying whether all columns have the same width (that of the widest column); the default is **nil**.

:max-width
Specifies the maximum width, in pixels, of the table display.

:max-height
Specifies the maximum height, in pixels, of the table display.

For an overview of **format-item-list** and related facilities: See the section "Overview of Table Formatting Facilities", page 52.

format-sequence-as-table-rows *sequence printer* &rest *options* *Function*
&key (*stream* ***standard-output***)
&allow-other-keys

Displays the elements in a sequence as a series of table rows.

sequence The sequence to be displayed. Each element of the sequence become one row in the resulting table.

printer The function used to display each element. The function is passed two arguments, an element of the sequence and an output stream. You can have the function write to the stream any information you want included in the table row for that item.

options [Reserved]

:stream Specifies the output stream; the default is ***standard-output***.

Additional keyword options available for this function are the same as those to **formatting-table**.

For an overview of **format-sequence-as-table-rows** and related facilities:
See the section "Overview of Table Formatting Facilities", page 52.

format-textual-list *sequence function* &rest *args* &key (*separator* "," *finally if-two filled after-line-break conjunction* (*stream* ***standard-output***) *Function*

Outputs a sequence of items as a textual list; for example, "1 2 3 4" becomes "1, 2, 3, and 4":

```
(defun simple-list-formatter ()
  (fresh-line)
  (format-textual-list '(1 2 3 4) #'princ :conjunction "and"))
```

sequence The sequence to output.

function The function used to print sequence elements.

:separator
Specifies the characters to use to separate elements of a textual list. The default is ", " (comma followed by a space).

:finally Specifies the separator to be used between the next-to-last and last elements of the list. The default is **nil**, meaning use the regular separator (specified by the **:separator** option).

A typical value is " and ".

:if-two Specifies the separator to use when there are only two elements in the list. A typical value is " and ".

:filled Specifies whether the list should be "filled"; the default is **nil**.

A filled list is one containing newline characters at appropriate points to prevent wrapping of output from right margin to left. Thus, specifying `:filled t` for a long list results in two or more separate lines of output – each of a length less than the width of the output window – rather than one long, wrapped line. Line breaks come between list elements, not within.

Another value permitted for this option is **:before**. This is

like **t**, except that in the case where a line break occurs at a **:separator**, the break is made before the separator rather than after.

:after-line-break
In `:filled t` mode, specifies the string to insert at the beginning of each new line. This is useful for specifying leading indentation, etc. (See the **:filled** option.)

:conjunction
Specifies a string to use in the position between the last two elements. Typical values are `"and"` and `"or"`.

This option is similar to the **:finally** option, but does not affect the separator between the last two elements, unless only two elements occur. That is, the **:conjunction** option takes care of the two-element case; the **:if-two** option is not necessary if you use this option.

:stream Specifies the output stream; the default is ***standard-output***.

For an overview of **format-textual-list** and related facilities: See the section "Overview of Textual List Formatting Facilities", page 51.

formatting-cell (&optional *stream* &key *align*) &body *body* *Macro*
Binds local environment to control the printing of a table element within a **formatting-table** macro (see the latter facility for examples).

stream The output stream; the default is ***standard-output***.

:align Specifies how elements of a column should be aligned. The default (**nil**), causes the elements to be flush-left in the column. The other possible values are **:right** (flush-right) and **:center** (centered).

For an overview of **formatting-cell** and related facilities: See the section "Overview of Table Formatting Facilities", page 52.

formatting-column (&optional *stream* &rest *options*) &body *body* *Macro*
Controls column layout within a **formatting-table** macro (see the latter facility for examples).

stream Specifies the output stream; the default is ***standard-output***.

options [Reserved]

For an overview of **formatting-column** and related facilities: See the section "Overview of Table Formatting Facilities", page 52.

formatting-column-headings (&optional *stream* &rest *options*) &body *forms* *Macro*

Controls the display of column headings within a **formatting-table** macro.

Example:

```
(defun table-with-column-headings
       (&optional (column-one-label "Number"))
  (fresh-line)
  (formatting-table ()
     (formatting-column-headings ()
       (formatting-cell () (write-string column-one-label))
       (formatting-cell () "Square"))
     (loop for i from 1 to 10
          as square = (* i i)
          do
       (formatting-row ()
         (formatting-cell ()
           (princ i))
         (formatting-cell ()
           (princ square))))))
```

stream Specifies the output stream; the default is ***standard-output***.

options The following option is available:

underline-p
Boolean option specifying whether column headings are underlined; the default is **nil**.

For an overview of **formatting-column-headings** and related facilities: See the section "Overview of Table Formatting Facilities", page 52.

formatting-graph (&optional *stream* &key (*orientation* **:vertical**) (*balance-evenly* **nil**) (*row-spacing* **40**) (*within-row-spacing* **20**) (*column-spacing* **20**) (*within-column-spacing* **10**) (*default-drawing-mode* **:line**)) &body *body* *Macro*

Binds local environment to output graph connecting node objects generated in the body of the macro. The node objects are created by the macro **formatting-graph-node**.

stream The output stream; the default is ***standard-output***.

:orientation
Specifies **:vertical** or **:horizontal** orientation for the graph display.

:balance-evenly
Specifies whether the subtrees of the tree should all be the same size (width or height, depending on **:orientation**), the size of the largest subtree. The default is **nil**.

:row-spacing
For **:vertical** orientation, specifies the spacing, in pixels, between rows of tree nodes; the default is `40`.

:within-row-spacing
For **:horizontal** orientation, specifies the spacing, in pixels, between columns of tree nodes; the default is `20`.

:column-spacing
For **:vertical** orientation, specifies the spacing, in pixels, between columns of tree nodes; the default is `20`.

:within-column-spacing
For **:horizontal** orientation, specifies the spacing, in pixels, between rows of tree nodes; the default is `10`.

:default-drawing-mode
Specifies the drawing mode used to connect nodes of the tree. The default is **:line** , which connects the nodes with solid lines. Other modes are **:dashed-line**, **:arrow**, and **:dashed-arrow**.

Example:

```
(defun simple-graph (stream)
  (fresh-line stream)
  (formatting-graph (stream :orientation :horizontal)
    (let ((node-a (formatting-graph-node (stream)
                    (surrounding-output-with-border
                      (stream :shape :rectangle :thickness 3)
                      (princ 'a stream)))))
      (formatting-graph-node (stream :connections '(:right ,node-a)
                                     :drawing-mode :dashed-line)
        (surrounding-output-with-border
          (stream :shape :rectangle :thickness 3)
          (princ 'b stream))))))
```

If you want to try this example, compile it first. For a more complex example: See the macro **formatting-graph-node**, page 242.

Do not use **formatting-graph** to draw circular graphs.

For an overview of **formatting-graph** and related facilities: See the section "Overview of Graph Formatting Facilities", page 56.

formatting-graph-node (&optional *stream* &key *id connections* (*drawing-mode* **t**)) &body *body* *Macro*

Binds local environment to create node objects for use by the **formatting-graph** macro. For an example, see the dictionary entry for the latter facility.

stream The output stream; the default is ***standard-output***.

:id Specifies a symbol for the node. A node symbol is used as an argument to **dw:find-graph-node**: See the generic function **dw:find-graph-node**, page 231.

:connections
Specifies the connections between this node and one or more other nodes. The specification is a list in the form ((*key node-object-1*) (*key node-object-2*) ... (*key node-object-n*)), where key is one of **:left**, **:right**, **:above**, or **:below**.

:drawing-mode
Specifies the drawing mode used to connect this node with other nodes of the tree. This specification locally overrides the **:default-drawing-mode** specified by **formatting-graph**. Possible modes are **:line**, **:dashed-line**, **:arrow**, and **:dashed-arrow**.

Example:

```
(defun graph-1 (list unique-id-p)
  (let ((stream *standard-output*))
    (fresh-line stream)
    (formatting-graph (stream)
      (labels ((do-one (contents &rest connections)
                 (let ((node nil))
                   (when unique-id-p
                     (let ((already-there
                             (dw:find-graph-node stream contents)))
                       (when already-there
                         (dw::connect-graph-nodes
                           stream already-there connections)
                         (setq node already-there))))
                   (unless node
                     (setq node (formatting-graph-node
                                  (stream
                                    :id contents
                                    :connections connections)
                                  (surrounding-output-with-border
                                    (stream)
                                    (prin1 contents stream)))))
                   (when (consp contents)
                     (dolist (sublist contents)
                       (do-one sublist :after node))))))
        (do-one list)))))

(graph-1 '(a (b c) c) nil)
(graph-1 '(a (b c) c) t)
(graph-1 '(#1=(x y z) (w #1#) y) nil)
(graph-1 '(#1=(x y z) (w #1#) y) t)
```

For an overview of **formatting-graph-node** and related facilities: See the section "Overview of Graph Formatting Facilities", page 56.

formatting-item-list (&optional *stream* &key *inter-row-spacing* *inter-column-spacing* *row-wise* *output-row-wise* *n-rows* *n-columns* *inside-width* *inside-height* *max-width* *max-height*) &body *body* *Macro*

Binds local environment to output a list of items created in the body of the macro as a table.

Example:

```
(defun formatting-list-example ()
  (formatting-item-list (t :n-columns 3)
    (fresh-line)
    (loop for (p) in si:active-processes
          do
      (when p
        (formatting-cell ()
          (write-string (si:process-name p)))))))
```

stream The output stream; the default ***standard-output***.

:inter-row-spacing
Specifies the number of pixels between rows; the default is `0`.

:inter-column-spacing
Specifies the number of pixels between columns of the table; the default is the width of two spaces.

:row-wise
Boolean option specifying that the table is built by rows, that is, that the each succeeding item in the list be placed in the same row, one column after the previous item (except for line breaks); the default is **t**. **nil** specifies that each item be placed in the same column, one row below the previous item.

:output-row-wise
Boolean option specifying that the table be displayed row-by-row. The default is **nil**, causing the table to be displayed in the order in which it is constructed (see the **:row-wise** option.)

:n-rows Specifies the number of rows the table should have.

:n-columns
Specifies the number of columns the table should have.

:inside-width
Specifies the exact width, in pixels, of the table display.

:inside-height
Specifies the exact height, in pixels, of the table display.

:max-width
Specifies the maximum width, in pixels, of the table display.

:max-height
Specifies the maximum height, in pixels, of the table display.

For an overview of **formatting-item-list** and related facilities: See the section "Overview of Table Formatting Facilities", page 52.

formatting-multiple-columns (&optional *stream* &key *number-of-columns*) &body *body* *Macro*

Binds local environment such that the lines of text generated by the body of the macro are output in a multiple-column format.

stream The output stream; the default is ***standard-output***.

:number-of-columns
Specifies the number of columns into which the items are arranged. If this is unspecified, it uses as many columns as will fit, based on the stream's **:inside-size**.

Example:

```
(defun test-columns (&optional (stream *standard-output*))
  (loop for hundreds from 0 to 100 by 100 do
    (terpri stream)
    (formatting-multiple-columns (stream)
      (loop for j from 1 to 20 do
        (format stream "~d ~r~%" (+ j hundreds) (+ j hundreds))))))
```

Usage note: you should not use **formatting-table** within **formatting-multiple-columns**. Instead, use the **:multiple-columns** option to **formatting-table**: See the macro "**formatting-table**", page 246.

For an overview of **formatting-multiple-columns** and related facilities: See the section "Overview of Table Formatting Facilities", page 52.

formatting-row (&optional *stream* &rest *options*) &body *body* *Macro*

Controls row layout within a **formatting-table** macro (see the latter facility for examples).

stream The output stream; the default is ***standard-output***.

options [Reserved]

For an overview of **formatting-row** and related facilities: See the section "Overview of Table Formatting Facilities", page 52.

formatting-table (&optional *stream* &key *equalize-column-widths* *extend-width* *extend-height* *inter-row-spacing* *inter-column-spacing* *multiple-columns* (*multiple-column-inter-column-spacing* **dw::inter-column-spacing**) (*equalize-multiple-column-widths* **nil**) (*output-multiple-columns-row-wise* **nil**)) &body *body* *Macro*

Binds local environment to output items in a tabular format.

This macro must be used in conjunction with at least two others. The first, **formatting-row** or **formatting-column**, controls whether items are output as table rows or table columns, respectively. The second, **formatting-cell** or **format-cell**, controls the printing of each item. Contrast the output of the following two examples:

```
(defun row-oriented-table-formatting ()
  (fresh-line)
  (formatting-table ()
    (loop for i from 1 to 10
              as square = (* i i)
          do
      (formatting-row ()
        (formatting-cell ()
          (princ i))
        (formatting-cell ()
          (princ square))))))

(defun column-oriented-table-formatting ()
  (fresh-line)
  (formatting-table ()
    (loop for i from 1 to 10
              as square = (* i i)
          do
      (formatting-column ()
        (format-cell i #'princ)
        (format-cell square #'princ)))))
```

stream The output stream; the default is ***standard-output***.

:equalize-column-widths
Boolean option specifying whether all columns have the same width (that of the widest column); the default is **nil**.

:extend-width
Specifies whether the spacing of table columns is extended; the default is **nil**. Alternative values are **t**, meaning make use of the full horizontal space available, or a number, indicating the number of pixels over which to extend the table.

:extend-height
Specifies whether the spacing of table rows is extended; the default is **nil**. Alternative values are **t**, meaning make use of the full vertical space available, or a number, indicating the number of pixels over which to extend the table.

:inter-row-spacing
Specifies the minimum number of pixels inserted between rows of the table; the default is 0. This value will be the actual number of pixels inserted unless overridden by the **:extend-height** option.

:inter-column-spacing
Specifies the minimum number of pixels inserted between columns of the table; the default is the width of a space. This value will be the actual number of pixels inserted unless overriden by the **:extend-width** option.

:multiple-columns
Boolean option specifying that table rows be distributed among a series of two or more columns.

For example,

```
Set Point #1:   50
Set Point #2:   36
Set Point #3:   65
Set Point #4:   45
```

becomes

```
Set Point #1:   50      Set Point #3:   65
Set Point #2:   36      Set Point #4:   45
```

The arrangement of rows and columns generated is such that the number of columns is maximixed, the number of rows is minimized, and the hole, if any, left in the lower right corner of the table is the smallest possible.

:multiple-column-inter-column-spacing
Specifies the number of pixels to insert between columns in a

multiple-column display (**:multiple-columns** option is **t**). It defaults to the value of the **:inter-column-spacing** option.

:equalize-multiple-column-widths
Boolean option specifying whether all columns in a multiple column display (**:multiple-columns** option is **t**) have the same width (that of the widest column); the default is **nil**.

:output-multiple-columns-row-wise
Boolean option specifying whether columns in a multiple-column display (**:multiple-columns** option is **t**) are displayed by outputting all the elements in one row followed by all in the next, and so on. The default is **nil**, meaning that the order of display is "column-wise": first all the elements in one column are output, then all the elements in the next, and so on.

The resulting display is the same no matter which way this flag is set; only the order in which the elements are displayed is changed. This affects the order in which calls to **formatting-row** are made within the body of the table-formatting macro. In the default case, calls are made in the order given; in the alternative case, call order is unpredictable.

For an overview of **formatting-table** and related facilities: See the section "Overview of Table Formatting Facilities", page 52.

formatting-textual-list (&optional *stream* &key (*separator* ",") *Macro*
finally if-two filled after-line-break conjunction)
&body *body*

Binds local environment to output a sequence of items as a textual list. This macro must be used in conjunction with the **formatting-textual-list-element** macro specifying the printing function.

Example:

```
(defun simple-list-formatting ()
  (fresh-line)
  (formatting-textual-list (t :conjunction "and")
    (loop for i from 1 to 4
          do
      (formatting-textual-list-element ()
        (princ i)))))
```

stream 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).

:finally Specifies the separator to be used between the next-to-last and last elements of the list. The default is **nil**, meaning use the regular separator (specified by the **:separator** option).

A typical value is " and ".

:if-two Specifies the separator to use when there are only two elements in the list. A typical value is " and ".

:filled Specifies whether the list should be "filled"; the default is **nil**.

A filled list is one containing newline characters at appropriate points to prevent wrapping of output from right margin to left. Thus, specifying `:filled t` for a long list results in two or more separate lines of output – each of a length less than the width of the output window – rather than one long, wrapped line. Line breaks come between list elements, not within.

Another value permitted for this option is **:before**. This is like **t**, except that in the case where a line break occurs at a **:separator**, the break is made before the separator rather than after.

:after-line-break
In `:filled t` mode, specifies the string to insert at the beginning of each new line. This is useful for specifying leading indentation, etc. (See the **:filled** option.)

:conjunction
Specifies a string to use in the position between the last two elements. Typical values are "and" and "or".

This option is similar to the **:finally** option, but does not affect the separator between the last two elements, unless only two elements occur. That is, the **:conjunction** option takes care of the two-element case; the **:if-two** option is not necessary if you use this option.

For an overview of **formatting-textual-list** and related facilities: See the section "Overview of Textual List Formatting Facilities", page 51.

formatting-textual-list-element (&optional *stream*) &body *body* *Macro*

Controls the printing of items output as textual list elements within a **formatting-textual-list** macro.

Example:

```
(formatting-textual-list (t :conjunction "and")
  (loop for i from 1 to 4 doing
    (formatting-textual-list-element () (princ i))))
```

stream The output stream; the default is ***standard-output***.

For an overview of **formatting-textual-list-element** and related facilities: See the section "Overview of Textual List Formatting Facilities", page 51.

indenting-output (*stream indentation*) &body *body* *Macro*

Binds local environment to control the insertion of spaces or other characters at the beginning of each newline output to a stream.

stream The output stream. As a special case, **t** and **nil** are abbreviations for ***standard-output***.

indentation

What gets inserted at the beginning of each line output to the stream. Three possibilities exist:

integer The number of spaces to indent.

string A string to print at the beginning of each line.

function A function to print a string. The function receives one argument, the output stream. Because the system calls this function with other streams, for example, with a dummy stream to determine the space requirements of the output, it should output something of the same size each time it is called.

You should either begin the body with `(terpri` *stream*`)`, or equivalent, or pre-position the stream to the same place as the indentation.

Examples:

```
(defun simple-indenter ()
  (indenting-output (t 10)
    (loop for i from 1 to 5
          do
      (terpri)
      (format t "This is indented line ~d" i))))
```

The **trace** special form uses **indenting-output** as follows:

```
(flet ((indent (stream)
         (loop for n from 1 below trace-level do
           (write-char (if ... #\| #\sp) stream))))
  (indenting-output (*trace-output* #'indent)
    (terpri *trace-output*)
    ...))
```

For an overview of **indenting-output** and related facilities: See the section "Overview of Character Environment Facilities", page 49. For a related facility: See the macro **sys:with-indentation** in *Reference Guide to Streams, Files, and I/O.*

dw:independently-redisplayable-format *stream format-string* &rest *format-args* *Macro*

Outputs a formatted string such that each format argument is independently redisplayable. (See the macro **dw:with-redisplayable-output**, page 273.)

stream The output stream; the default is ***standard-output***.

format-string
The format-control string. (See the function **format** in *Reference Guide to Streams, Files, and I/O.*)

format-args
The format arguments.

The *format-string* is parsed at compile time, resulting in a series of calls to **dw:redisplayable-format** or **format**. Some restrictions result:

- *stream* may not be **nil**, although **format** would permit it.
- **format** commands that need all the **format** arguments, like conditionals, iterations, or gotos, cannot be used.

dw:independently-redisplayable-format is one of a number of facilities used to do incremental redisplay. For examples, see the file `sys:examples;incremental-redisplay.lisp`.

For an overview of **dw:independently-redisplayable-format** and related facilities: See the section "Overview of Advanced Presentation Output Facilities", page 63.

dw:named-value-snapshot-continuation *name var-list* &body *body* *Macro*

Generates a lexical closure of its *body*, except that it snapshots the current values of lexical variables used free within *body*.

name The internal-function name for the generated lexical closure. This supplies the *X* in names like `(:INTERNAL SOMETHING 2 X)`.

var-list The lambda-list for the generated lexical closure.

dw:named-value-snapshot-continuation can be of use, for example, when collecting closures within an iteration. The following code

```
(defun print-reverse-of-list (list)
  (let ((list-of-closures ()))
    (dolist (x list)
      (push (lambda (stream) (print x stream))
            list-of-closures))
    (dolist (closure list-of-closures)
      (funcall closure *standard-output*))))

(print-reverse-of-list '(1 2 3))
```

would print three occurrences of 3. This is because the first **dolist** might **macroexpand** into something like

```
(let ((x) (temp list))
  (prog nil
   loop (when (null temp) (return))
        (setq x (pop temp))
        (push (lambda (stream) (print x stream))
              list-of-closures)
        (go loop)))
```

where each `(lambda ...)` snapshots exactly the same binding of x. Unfortunately, this means that each time through the loop, x – the same x in *all* the closures – gets **setq**'d. A way around this is to introduce a new binding for the x at the point the closure is produced:

```
(let ((x)
      (temp list))
  (prog nil
        (when (null temp) (return))
        (setq x (pop temp))
        (push (let ((x x))
                (lambda (stream) (print x stream)))
              list-of-closures)))
```

Here the x snapshotted by the closure is different for each closure, achieving the desired effect.

dw:named-value-snapshot-continuation processes the *body*, identifying freely-referenced lexical variables which need such snapshotting. It also does special processing for **self** and instance variables referenced within flavor methods. As a result, the above fragment could be written

```
(defun print-reverse-of-list (list)
  (let ((list-of-closures ()))
    (dolist (x list)
      (push (dw:named-value-snapshot-continuation
              writer (stream)
              (print x stream))
            list-of-closures))
    (dolist (closure list-of-closures)
      (funcall closure *standard-output*))))
```

and then `(print-reverse-of-list '(1 2 3))` would correctly print 3, 2, 1.

For an overview of **dw:named-value-snapshot-continuation** and related facilities: See the section "Overview of Facilities for Writing Formatted Output Macros", page 66.

tv:note-progress *numerator* &optional (*denominator* **1**) (*note* **tv:*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 **tv:noting-progress** macro (for examples, look at the dictionary entry for that facility). The progress bar is updated by fractional amounts between 0 and 1.

numerator
: The numerator of the fraction by which to update the bar.

denominator
: The denominator of the fraction by which to update the bar; the default is 1.

note The note object (bound to the *variable* supplied to **tv:noting-progress**).

For an overview of **tv:note-progress** and related facilities: See the section "Overview of Progress Indicator Facilities", page 59.

tv:noting-progress (*name* &optional (*variable* **'tv:*current-progress-note***) (*process* **'sys:current-process**)) &body *body* *Macro*

Binds local environment such that the progress of an operation performed within the body of the macro is noted by a progress bar displayed in the status line at the bottom of the screen. The function **tv:note-progress** does the updating of the progress bar.

name A string naming the operation being noted. This string is displayed above the progress bar.

variable The variable bound to the note object; the default is **tv:*current-progress-note***. This variable is an argument to **tv:note-progress**.

process The process on whose behalf the progress is noted; the default is **sys:current-process**. This is used to determine the precedence of notes.

Examples:

```
(tv:noting-progress ("Working Away By Tenths")
  (loop for i from .1 to 1.0 by .1
        do
    (tv:note-progress i)
    (sleep 1)))
```

```
(tv:noting-progress ("Working Away By Fifths")
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 1 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 2 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 3 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 4 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 5 5)
  (sleep 1))
```

For an overview of **tv:noting-progress** and related facilities: See the section "Overview of Progress Indicator Facilities", page 59.

present *object* &optional (*presentation-type* **(type-of dw::object)**) *Function*
&key (*stream* ***standard-output***) (*acceptably* **nil**) (*sensitive* **t**) (*form* **nil**) (*location* **nil**) (*single-box* **nil**) (*allow-sensitive-inferiors* **t**)

Outputs a presentation object to a stream. If the stream supports presentation remembering, the presented object is accessible via the mouse in the appropriate input context; if not, the printed representation is the same, but the object is not mouse-sensitive.

object The object to be presented.

presentation-type
The presentation type to be used in presenting the object; the default is the type of the object.

:stream Specifies stream on which the object is presented; the default is ***standard-output***.

:acceptably
Boolean option specifying whether to present the object in such a way that it can later be parsed by **accept**; the default is **nil**. This option is useful when output is to strings or files, but not necessary when output is to Dynamic Windows.

:sensitive
Boolean option specifying whether the presentation is mouse-sensitive; the default is **t**. This option is useful for explicitly preventing mouse sensitivity of objects presented to Dynamic Windows.

:form Specifies a form that can be passed to **setf** to store a new value in place of the current output value. This option and **:location** are mutually exclusive.

The form supplied for this option is used by a predefined, side-effecting mouse handler (available on `c-m-Mouse-R`) to modify the contents of structure slots.

:location Specifies a locative that can be used to store a new value in place of the current output value. This option and **:form** are mutually exclusive.

The locative supplied for this option is used by a predefined, side-effecting mouse handler (available on `c-m-mouse-R`) to modify the contents of structure slots.

:single-box
Specifies that mouse-sensitivity of objects output in a series of inferior calls to this form be indicated by a single, large box for highlighting rather than the sum of all the individual boxes. This option is used mostly with graphic presentations.

:allow-sensitive-inferiors
Boolean option specifying whether nested calls to **present** or **dw:with-output-as-presentation** from inside this presentation – for example, when presenting the individual elements of a Lisp list – generate presentation objects. The default is **t**.

For an example: See the macro "**dw:with-output-as-presentation**", page 268.

For an overview of **present** and related facilities: See the section "Overview of Basic Presentation Output Facilities", page 47.

present-to-string *object* &optional (*presentation-type* **(dw::decode-old-presentation-type (type-of dw::object) :atomic-ok t)**) &key (*string* **nil**) *index acceptably for-context-type* *Function*

Outputs a presentation object to a string.

object The object to be presented.

presentation-type
The presentation type to be used in presenting the object; the default is the type of the object.

:string Specifies a string to which the object is presented. The default is **nil**, causing a new string object to be created.

:index The character position in the string array where the presenting of the object begins; the default is position `0`. Use this option only if the **:string** option is non-**nil**.

:acceptably
Boolean option specifying whether the object should be presented in such a way that it can later be parsed by **accept**; the default is **nil**.

:for-context-type
Specifies the input context on whose behalf the presentation is made. This affects the printing of the string. For example, the Command Processor uses this option to ensure that command names are preceded by colons when in the `'command-or-form` context (and form-preferred dispatch mode).

For an overview of **present-to-string** and related facilities: See the section "Overview of Basic Presentation Output Facilities", page 47.

dw:redisplayable-format *stream format-string* &rest *format-args* *Function*

Outputs a formatted string redisplayably. This simply calls **format** within a caching point for incremental redisplay. (See the macro **dw:with-redisplayable-output**, page 273.) *format-string* is used as the cache-id; the list *format-args* is used as the cache-value.

stream The output stream; the default is ***standard-output***.

format-string
The format-control string. (See the function **format** in *Reference Guide to Streams, Files, and I/O.*)

format-args
The format arguments.

dw:redisplayable-format is one of a number of facilities used to do incremental redisplay. For examples, see the file sys:examples;incremental-redisplay.lisp.

For an overview of **dw:redisplayable-format** and related facilities: See the section "Overview of Advanced Presentation Output Facilities", page 63.

dw:redisplayable-present *object* &optional *presentation-type* &key (*stream* ***standard-output****) (*unique-id* **nil**) &allow-other-keys *Function*

Presents an object redisplayably. This simply calls **present** within a caching point for incremental redisplay. (See the macro **dw:with-redisplayable-output**, page 273.) The *object* itself is used as the cache-value.

object The object to present.

presentation-type
The presentation type to display the object as; the default is the Lisp object type of the object, that is, **(type-of** *object*).

:stream Specifies the output stream; the default is ***standard-output****.

:unique-id
Identifies the particular incremental redisplay cache. This may be any object, as long as it is unique with respect to the *id-test* predicate among all such ids in the current incremental redisplay.

Other keyword options to **dw:redisplayable-present** are the same as those to **present**, to which they are passed: See the function **present**, page 255.

dw:redisplayable-present is one of a number of facilities used to do incremental redisplay. For examples, see the file sys:examples;incremental-redisplay.lisp.

For an overview of **dw:redisplayable-present** and related facilities: See the section "Overview of Advanced Presentation Output Facilities", page 63.

dw:redisplayer (&optional *stream*) &body *body* *Macro*

Creates a redisplay object out of its *body* which can be used to do incremental redisplay on *stream*. Provide the redisplay object as the *redisplay-piece* argument to **dw:do-redisplay**: See the generic function **dw:do-redisplay**, page 211.

stream The output stream; the default is ***standard-output***.

dw:redisplayer is one of a number of facilities used to do incremental redisplay. For examples, see the file sys:examples;incremental-redisplay.lisp.

For an overview of **dw:redisplayer** and related facilities: See the section "Overview of Advanced Presentation Output Facilities", page 63.

:set-viewport-position *new-left new-top* of **dw:dynamic-window** *Method*

Scrolls the window to a specified location in the window's output history. Specify the location in terms of absolute window coordinates.

new-left The x-coordinate for the viewport's left edge.

new-top The y-coordinate for the viewport's top edge.

For an overview of **(flavor:method :set-viewport-position dw:dynamic-window)** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

surrounding-output-with-border (&optional *stream* &key (*shape* **:rectangle**) (*thickness* **1**) (*margin* **1**) (*pattern* **t**) (*filled* **nil**) (*width* **nil**) (*height* **nil**) (*move-cursor* **t**)) &body *body* *Macro*

Binds local environment such that output generated in the body of the macro is enclosed within a border. The border is sized to just surround the output.

stream The output stream; the default is ***standard-output***.

:shape Specifies the shape of the border; the default is **:rectangle**. Other possible shapes are **:circle**, **:oval**, and **:diamond**.

:thickness
Specifies the thickness, in pixels, of the border; the default is 1.

:margin Specifies the minimum whitespace, in pixels, between the border and the enclosed output.

:pattern Specifies the pattern to be used in drawing the border. At present, only the **:rectangle** shape uses the pattern specified.

Example:

```
(defun pattern-test ()
  (fresh-line)
  (surrounding-output-with-border
    (*standard-output* :shape :rectangle
                       :pattern tv:50%-gray)
    (present tv:selected-window 'tv:window)))
```

If the **:filled** option is **t**, the pattern is drawn throughout the rectangular area and XORed with the unfilled values of the area's pixels.

For more information on how to specify patterns: See the function **graphics:draw-pattern**, page 223.

:filled Boolean option specifying whether the shaped enclosed by the border is filled; the default is **nil**. If **t**, filling occurs by XORing the turned-on and unfilled values of the pixels in the filled area.

If a pattern is specified by the **:pattern** option, filling occurs by XORing the pattern values and unfilled values of the pixels. In general, the best results are achieved by leaving the **:pattern** option unspecified if you intend to fill.

:width Specifies the the maximum width, in pixels, of the border; the default (**nil**) places no limit on the maximum.

:height Specifies the the maximum height, in pixels, of the border; the default (**nil**) places no limit on the maximum.

:move-cursor
Boolean option specifying whether a newline is performed at the end of *body*.

Example:

```
(defun shape-test (shape fill-p)
  (fresh-line)
  (surrounding-output-with-border
    (*standard-output* :shape shape
                       :filled fill-p)
    (present tv:selected-window 'tv:window)))
```

To see how the differently shaped borders look, try calling the above with the various shape keywords.

For an overview of **surrounding-output-with-border** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

dw:tracking-mouse (&optional *stream* &key (*whostate* "**Track Mouse**") (*who-line-documentation-string* **nil**)) &body *clauses* *Macro*

Tracks the mouse in the user process. User-supplied routines are executed when mouse events occur, such as position changes and the pressing or releasing of a button.

stream The output stream; the default is ***standard-output***.

:whostate
Specifies the string displayed in the run-state slot of the status line. The default value is `"Track Mouse"`.

:who-line-documentation-string
Specifies the mouse documentation string.

clauses Keyword-value pairs supplying routines (the values) executed when the mouse event indicated by the keyword occurs. Some keywords provide arguments. Available keywords and their arguments are described below:

:presentation (*presentation*)
Smallest presentation under mouse, or **nil**; called when the mouse moves.

:presentation-hold (*presentation*)
Same as **:presentation**, but used if a mouse button is still down.

:mouse-motion (*x y*)
Position of the mouse; called when the mouse moves.

:mouse-motion-hold (*x y*)
Same as **:mouse-motion**, but used if a mouse button is still down.

:who-line-documentation-string ()
Allows dynamic control of mouse documentation line; called whenever anything changes.

:release-mouse () Called when all mouse buttons are up after some were down.

:keyboard (*char*)
Called when user presses a keyboard key (rather than clicking).

:presentation-click (*presentation mouse-char*)
Called when a mouse button is pressed. *presentation* is the smallest presentation under mouse, or **nil**; *mouse-char* is the mouse-character object corresponding to the mouse gesture used.

:mouse-click (*mouse-char x y*)
Called when a mouse button is pressed. Arguments are the mouse position and the mouse-character object corresponding to the mouse gesture used.

To see the macro in action, try the following example:

```
;;; To run this function create two lisp listeners side by side.
;;; In the first Lisp Listener type the form:
;;;     (setq *LL1* *terminal-io*).
;;; Click left on the second Lisp Listener and enter the form:
;;;     (mouse-1 *terminal-io*).


(defvar *LL1*)

(defun mouse-1 (window)
  (dw:tracking-mouse (window)
    (:who-line-documentation-string ()
     (if (zerop (tv:mouse-buttons))
         "Buttons up"
         "Buttons Down"))
    (:release-mouse ()
     (format *LL1* "~&Mouse key released"))
    (:mouse-motion (x y)
        (format *LL1* "~&Mouse motion(~d,~d)" x y))
    (:mouse-motion-hold (x y)
        (format *LL1* "~&Mouse motion hold(~d,~d)" x y))
    (:mouse-click (button x y)
       (graphics:draw-rectangle x y (+ x 10) (+ y 10))
       (selector button char-mouse-equal
       (#\mouse-left (format *LL1* "~&Left click"))
       (#\mouse-middle-2 (return-from mouse-1 "~&That's All Folks!"))))))
```

For an overview of **dw:tracking-mouse** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

:visible-cursorpos-limits &optional (*unit* **:pixel**) of **dw:dynamic-window** *Method*

Returns the left, top, right, and bottom limits of the current viewport. The limits are returned as absolute window locations.

unit The unit of measure for the viewport limits; the default is **:pixel**. The alternative is **:character**. The character used is the space character in the window's default character style.

For an overview of **(flavor:method :visible-cursorpos-limits dw:dynamic-window)** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

with-character-face (*face* &optional (*stream* **t**) &key *bind-line-height*) &body *body* *Macro*

Binds the local environment such that character output is in the specified face.

face The face to be used for character output, for example, **:bold** or **:italic**.

stream Output stream; the default is ***standard-output***.

:bind-line-height

Boolean option specifying whether the height, in pixels, of the line containing the character output is based on the size of the default character style or of the style specified in the macro. Whether you specify **t** or **nil** (the default) depends on the context of the output. To see the difference, run the following function first with **nil**, then with **t**:

```
(defun line-height-binder (bind)
  (format t "~&Foo")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (write-string "bar"))
  (write-string "baz")
  (dotimes (j 2) (terpri))
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (format t "~&Frob one~%Frob two~%"))
  (format t "Last line"))
```

In this example, the difference is apparent; much more subtle are the differences produced when only the character family or face is changed, as opposed to its size.

The height of the default character style is determined from the height of the space character.

To see a list of valid character style faces, evaluate the variable **si:*valid-faces***. For more information on character styles: See the section "Character Styles" in *Symbolics Common Lisp: Language Concepts*.

For an overview of **with-character-face** and related facilities: See the section "Overview of Character Environment Facilities", page 49.

with-character-family (*family* &optional (*stream* **t**) &key *bind-line-height*) &body *body* *Macro*

Binds the local environment such that character output is in the specified family.

style The family to be used for character output, for example, **:serif** or **:jess**.

stream Output stream; the default is ***standard-output***.

:bind-line-height

Boolean option specifying whether the height, in pixels, of the line containing the character output is based on the size of the default character style or of the style specified in the macro. Whether you specify **t** or **nil** (the default) depends on the context of the output. To see the difference, run the following function first with **nil**, then with **t**:

```
(defun line-height-binder (bind)
  (format t "~&Foo")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (write-string "bar"))
  (write-string "baz")
  (dotimes (j 2) (terpri))
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (format t "~&Frob one~%Frob two~%"))
  (format t "Last line"))
```

In this example, the difference is apparent; much more subtle are the differences produced when only the character family or face is changed, as opposed to its size.

The height of the default character style is determined from the height of the space character.

To see a list of valid character style families, evaluate the variable **si:*valid-families***. For more information on character styles: See the section "Character Styles" in *Symbolics Common Lisp: Language Concepts.*

For an overview of **with-character-family** and related facilities: See the section "Overview of Character Environment Facilities", page 49.

with-character-size (*size* &optional (*stream* **t**) &key *bind-line-height*) &body *body* *Macro*

Binds the local environment such that character output is of the specified size.

size The size of character output, for example, **:very-small** or **:very-large**.

stream Output stream; the default is ***standard-output***.

:bind-line-height

Boolean option specifying whether the height, in pixels, of the line containing the character output is based on the size of the default character style or of the style specified in the macro. Whether you specify **t** or **nil** (the default) depends on the context of the output. To see the difference, run the following function first with **nil**, then with **t**:

```
(defun line-height-binder (bind)
  (format t "~&Foo")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (write-string "bar"))
  (write-string "baz")
  (dotimes (j 2) (terpri))
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (format t "~&Frob one~%Frob two~%"))
  (format t "Last line"))
```

In this example, the difference is apparent; much more subtle are the differences produced when only the character family or face is changed, as opposed to its size.

The height of the default character style is determined from the height of the space character.

To see a list of valid character style sizes, evaluate the variable **si:*valid-sizes***. For more information on character styles: See the section "Character Styles" in *Symbolics Common Lisp: Language Concepts.*

For an overview of **with-character-face** and related facilities: See the section "Overview of Character Environment Facilities", page 49.

with-character-style (*style* &optional (*stream* **t**) &key *bind-line-height*) &body *body* *Macro*

Binds the local environment such that character output is in the specified style.

style List of the form (*:family :face :size*) specifying character style.

stream Output stream; the default is ***standard-output***.

:bind-line-height

Boolean option specifying whether the height, in pixels, of the line containing the character output is based on the size of the default character style or of the style specified in the macro. Whether you specify **t** or **nil** (the default) depends on the context of the output. To see the difference, run the following function first with **nil**, then with **t**:

```
(defun line-height-binder (bind)
  (format t "~&Foo")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (write-string "bar"))
  (write-string "baz")
  (dotimes (j 2) (terpri))
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (format t "~&Frob one~%Frob two~%"))
  (format t "Last line"))
```

In this example, the difference is apparent; much more subtle are the differences produced when only the character family or face is changed, as opposed to its size.

The height of the default character style is determined from the height of the space character.

A character style specifies three style components: family, face, and size. To see lists of valid families, faces, and sizes, evaluate the respective variables **si:*valid-families***, **si:*valid-faces***, and **si:*valid-sizes***. (The same information is presented in another section: See the section "Available Character Styles" in *Symbolics Common Lisp: Language Concepts*.) Note that not all permutations of family, face, and size are legitimate character styles.

A partially specified character style is merged against the default character style for the window. (See the init option **(flavor:method :default-character-style tv:sheet)** in *Programming the User Interface, Volume B.* See the section "Merging Character Styles" in *Symbolics Common Lisp: Language Concepts.*) Consider the following example (which has to be compiled):

```
(defun character-style-merge ()
  (dw:with-own-coordinates (t)
    (with-character-style ('(nil :bold :large) t)
      (graphics:draw-string "CURRENT DATA" 100 100
                            :alu :flip))))
```

The character style specification in the above example only specifies two components, face and size. The **nil** supplied in the family component slot means that the default family for the window is used. If you wish to keep the defaults for two of the components, then you can use one of the following macros:

with-character-family
with-character-face
with-character-size

You can determine the character style corresponding to a particular TV font by using the **si:backtranslate-font** function. (See the function **si:backtranslate-font** in *Programming the User Interface, Volume B.*)

Example:

```
(si:backtranslate-font 'fonts:bigfnt)
#<CHARACTER-STYLE FIX.ROMAN.VERY-LARGE 260250707>
#<STANDARD-CHARACTER-SET 260000540>
0
#<B&W-SCREEN-DISPLAY-DEVICE 260302767>
```

The example shows that **fonts:bigfnt** corresponds to the **(:fix :roman :very-large)** character style.

For more information on character styles: See the section "Character Styles" in *Symbolics Common Lisp: Language Concepts.*

For an overview of **with-character-style** and related facilities: See the section "Overview of Character Environment Facilities", page 49.

dw:with-output-as-presentation (&key *stream object type form location single-box* (*allow-sensitive-inferiors* t)) &body *body* *Macro*

Outputs an object as a presentation object; in effect, allows you to rewrite the printer function (used locally) for a presentation type. The following example illustrates this point:

```
(defun present-this-as-that (this that
        &optional (stream *standard-output*))
  (send stream :clear-history)
  (dw:with-output-as-presentation (:single-box t
        :stream stream :type that :object this)
    (send stream :draw-circle 250 200 25)
    (send stream :draw-circle 270 200 25)))
```

Try calling this function with `"ABC"` as the first argument and `'string` as the second. Now, do `(accept 'string)` and click on the graphic.

Note the `:single-box t` option used in the above example. This is nearly always appropriate when using this macro for graphic presentations.

Following are the keyword arguments recognized by **dw:with-output-as-presentation**. Note that some of them are required.

:stream Specifies stream on which the object is presented; the default is ***standard-output***.

:object Specifies the presentation object of the output presentation. If you do not use this option, then you must supply either the **:form** or **:location** option.

:type Specifies the type of the presentation. You must provide this option.

:form Specifies a form that can be passed to **setf** to store a new value in place of the current output value. This option and **:location** are mutually exclusive.

The form supplied for this option is used by a predefined, side-effecting mouse handler (available on `c-m-Mouse-R`) to modify the contents of structure slots.

:location Specifies a locative that can be used to store a new value in place of the current output value. This option and **:form** are mutually exclusive.

The locative supplied for this option is used by a predefined,

side-effecting mouse handler (available on c-m-mouse-R) to modify the contents of structure slots.

:single-box
Specifies that mouse-sensitivity of objects output in a series of inferior calls to this form be indicated by a single, large box for highlighting rather than the sum of all the individual boxes. This option is used mostly with graphic presentations.

:allow-sensitive-inferiors
Boolean option specifying whether nested calls to **present** or **dw:with-output-as-presentation** from inside this presentation – for example, when presenting the individual elements of a Lisp list – generate presentation objects. The default is **t**.

Example:

```
(defun sensitive-inferior-test (sensitive-p)
  (dw:with-output-as-presentation
    (:object fl
     :type 'sys:flavor-name
     :allow-sensitive-inferiors sensitive-p)
    (format t "The flavor ~S." fl)))
```

Try **setq**ing fl to some flavor then calling sensitive-inferiors-test with **t**, then **nil**. You should find that in the first case both the entire presentation and the flavor name are individually sensitive depending on where you have the mouse cursor; in the latter case, only the entire presentation is sensitive.

For an overview of **dw:with-output-as-presentation** and related facilities: See the section "Overview of Basic Presentation Output Facilities", page 47.

:with-output-recording-disabled *continuation xstream* of **dw:dynamic-window** *Method*

Disables output recording on a specified window for a specified continuation.

continuation
The continuation, a function of one argument, the output stream.

xstream The window whose output recording is disabled.

Example:

```
(defun draw-circles (stream)
  (loop repeat 50
        do
    (graphics:draw-circle
      (random 500)
      (random 500) 10 :stream stream)))

(send *standard-output*
      :with-output-recording-disabled
      #'draw-circles *terminal-io*)
```

You could incorporate this method into a macro as follows:

```
(defmacro with-output-recording-disabled
          ((stream) &body body)
  `(send ,stream :with-output-recording-disabled
         (dw::named-continuation
           with-output-recording-disabled (,stream)
           ,@body)
         ,stream))

;;; Uses the macro.
(defun new-draw-circles
       (&optional (stream *standard-output*))
  (with-output-recording-disabled (stream)
    (loop repeat 50
          do
      (graphics:draw-circle
        (random 500) (random 500) 10 :stream stream))))
```

For an overview of **(flavor:method :with-output-recording-disabled dw:dynamic-window)** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

dw:with-output-to-presentation-recording-string (*stream*) &body *body* *Macro*

Binds local environment to output to a string, the way **with-output-to-string** does, except that the string records presentations resulting from calls to **present** and **dw:with-output-as-presentation**. If the resulting string is subsequently printed (via **princ** or **present**) to a stream supporting presentations, the recorded presentations are re-presented to that stream.

stream The output stream; the default is ***standard-output***.

dw:with-output-to-presentation-recording-string is distinguished from **present-to-string** as follows:

w-o-to-p-r-string	**present-to-string**
Returns a presentation-recording string	Returns an ordinary string
Arbitrary body writing to string	Single object to be presented

Example:

```
(defun test-pr-string ()
  (let ((string (dw:with-output-to-presentation-recording-string
                  (*standard-output*)
                  (dolist (symbol '(butcher baker candlestick-maker))
                    (write-string " ")
                    (present symbol 'symbol)))))
    (format T "~&~s~%These should be mouse-sensitive: {~a}~%"
            string string)
    (accept 'symbol)))
```

For an overview of **dw:with-output-to-presentation-recording-string** and related facilities: See the section "Overview of Advanced Presentation Output Facilities", page 63.

dw::with-output-truncation (&optional *stream* &rest *options*) &body *body* *Macro*

Binds the local environment to allow textual output to extend beyond the bottom and right borders of the output window.

stream The output stream; the default is ***standard-output***.

To access text extending beyond the margins of the output window, the window needs vertical and horizontal scroll bars. For information on how to equip Dynamic Windows with scroll bars (and other margin components): See the flavor **dw:dynamic-window**, page 399.

options Two options are available:

:horizontal

Boolean option specifying whether truncation occurs in the horizontal dimension; the default is **t**.

"Truncation" here means that output exceeding the width of the window extends beyond the right

margin of the current window viewport; the margin truncates the user's view of the output. If **nil**, the output wraps to the next line.

:vertical Boolean option specifying whether truncation occurs in the vertical dimension. The default is **t**, meaning that output exceeding the height of the window extends below the bottom margin of the current window viewport.

Example:

```
(defun truncation-test (t-or-nil)
  (dw::with-output-truncation (t :horizontal t-or-nil)
    (loop repeat 100 do (write-char #\a))))
```

For an overview of **dw::with-output-truncation** and related facilities: See the section "Overview of Character Environment Facilities", page 49.

dw:with-own-coordinates (&optional *stream* &key *left top right bottom* (*clear-window* **t**) (*erase-window* **nil**) (*enable-output-recording* **t**)) &body *body* *Macro*

Binds the local environment such that output to a Dynamic Window is in a refreshed area, and the coordinate system is relative to the current viewport, not the window's origin.

stream The output stream; the default is ***standard-output***.

:left Specifies the x-coordinate at the beginning of the area to be erased when the **:erase-window** option is **t**.

:top Specifies the y-coordinate at the beginning of the area to be erased when the **:erase-window** option is **t**.

:right Specifies the x-coordinate at the end of the area to be erased when the **:erase-window** option is **t**.

:bottom Specifies the y-coordinate at the end of the area to be erased when the **:erase-window** option is **t**.

:clear-window
Boolean option specifying whether the window is scrolled to a clear area before output begins; the default is **t**.

:erase-window
Boolean option specifying that the output window be erased before output begins; the default is **nil**.

If this option is **t**, use the **:left**, **:top**, **:right**, and **:bottom** keywords to specify the coordinates of the area to be erased. If no coordinates are specified, they default to the coordinates of the current viewport. Output begins at the top of the erased area.

:enable-output-recording
Boolean option specifying whether output is retained in the output history of the window; the default is **t**.

This option is useful with animated graphic presentations that, because of the time required for redisplay, can impede scrolling through a window's history.

For an overview of **dw:with-own-coordinates** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

dw:with-redisplayable-output (&key *stream cache-value unique-id* (*cache-test* #'**eql**) *copy-cache-value* (*id-test* #'**eql**)) &body *body* *Macro*

Introduces a caching point for incremental redisplay. If this is used outside the dynamic scope of an incremental redisplay, it has no particular effect. However, when incremental redisplay is occurring, the supplied *cache-value* is compared with the value stored in the cache identified by *unique-id*. If the values differ, the code in *body* runs, and *cache-value* is saved for next time. If the cache values are the same, the code in *body* is not run, because the current output is still valid.

:stream Specifies the output stream; the default is ***standard-output***.

:cache-value
Specifies the value to be compared each time against the value stored in the cache.

:unique-id
Identifies the particular incremental redisplay cache. This may be any object, as long as it is unique with respect to the *id-test* predicate among all such ids in the current incremental redisplay.

:cache-test
Specifies the test used to compare *cache-value* against the value saved in the cache. The default is **eql**.

:copy-cache-value
Boolean option specifying whether to **copy-seq** the cache value before saving it in the cache. Use this, for example, when the cache value is a stack list which must be copied before being stored away somewhere.

:id-test Specifies the test used to locate the cache identified by *unique-id* among the caches used by the current incremental redisplay. The default is **eql**.

dw:with-redisplayable-output is one of a number of facilities used to do incremental redisplay. For examples, see the file sys:examples;incremental-redisplay.lisp.

For an overview of **dw:with-redisplayable-output** and related facilities: See the section "Overview of Advanced Presentation Output Facilities", page 63.

dw:with-replayable-output (&rest *parameters*) &body *body* *Macro*
Binds the local environment such that all of the output generated by *body* becomes a single, replayable presentation.

The code in *body* is snapshotted (using **dw:named-value-snapshot-continuation**) so that it can be rerun (replayed) in an altered environment; this results in a new printed representation. The user specifies the new output parameters at runtime via the Edit Viewspecs mouse handler. This handler is invoked by clicking s-sh-Middle on a replayable presentation.

parameters
A list of variable specifications in the style of **dw:accept-variable-values**. That is, each item in the list is a list of the form (*variable-name prompt-string presentation-type*).

The parameters are used to construct an **dw:accept-variable-values** menu which pops up in response to the mouse gesture Edit Viewspecs (s-sh-Middle). The values of the variables can then be changed by the user, and the presentation rerun with the new values.

Example:

```
;;; Compile and run this code, then Edit Viewspecs by
;;; clicking s-sh-Middle on its output.
(defun wrpo ()
  (fresh-line)
  (let ((style '(:fix :roman :normal))
        (width 50)
        (start 1))
    (dw:with-replayable-output
      ((style "Character style" character-style)
       (width "Width in characters" ((integer 5 120)))
       (start "Starting from" integer))
      (with-character-style (style)
        (let ((fill-width
                (* width (send *standard-output* :char-width))))
          (filling-output (() :fill-column fill-width)
            (loop repeat 50
                  for i from start
                  do (format T " ~r" i)))))))))
```

Note that other presentations on the screen are not moved to account for changes in the size of the target presentation due to replaying. It is possible for gaps or overwriting to occur. If this is a problem, then consider the facilities provided for incremental redisplay. See the Advanced Presentation Output Facilities section, referenced below, for more information.

dw:with-replayable-output is similar to **dw:with-output-as-presentation** in the sense that it lets you define a presentation-type printer "on the fly", that is, not as part of a presentation type. In the case of **dw:with-replayable-output**, you are writing a printer that can be modified by the user at runtime, via the Edit Viewspecs handler. This is not the only way to provide users with the ability to alter displayed presentations; you can use the **:viewspec-choices** option to **define-presentation-type** to provide the same capability with regard to all presentations of the defined type: See the macro "**define-presentation-type**", page 366.

dw:with-resortable-output is a specialization of **dw:with-replayable-output** for re-ordering and redisplaying lists: See the macro **dw:with-resortable-output**, page 276.

For an overview of **dw:with-replayable-output** and related facilities: See the section "Overview of Advanced Presentation Output Facilities", page 63.

dw:with-resortable-output ((*list key* &key *copy-of*) &rest *sort-clauses*) *other-parameters* &body *body* *Macro*

Binds the local environment such that all of the output generated by *body* becomes a single, replayable presentation. The list can be output in one of several orders specified by sorting predicates. Which sorting predicate is used can be specified by users at runtime via the Edit Viewspecs mouse handler, available on s-sh-Middle.

list A variable holding the sequence of items for output.

key A variable holding an identifier for selecting which of the *sort-clauses* is used.

:copy-of Specifies a list to be copied and sorted instead of *list*. The value is copied by **copy-seq**. Typically, the value of this option is *list*. Use it when you do not want the order of the original list to be destroyed by sorting.

sort-clauses
An **ecase** body, selecting on sort keys (the value of *key*), and returning a sort predicate.

other-parameters
Other parameters included in the *parameters* argument passed to **dw:with-replayable-output**: See the macro **dw:with-replayable-output**, page 274. These and the sorting options appear in the **dw:accept-variable-values** display created by the Edit Viewspecs handler.

To see this macro in action, execute the Command Processor command Show Processes or Show Directory. Both use **dw:with-resortable-output** and produce output resortable via the Edit Viewspecs handler (s-sh-Middle).

Another example:

```
(defun sortable-output ()
  (let ((data (make-array 10))
        (how :alpha)
        (style '(:swiss nil nil)))
    (dotimes (i 10)
      (setf (aref data i)
            (list i (format nil "~r" i))))
    (dw:with-resortable-output
      ;; list and key
      ((data how)
       ;; sort clauses
       (:alpha
         (lambda (x y)
           (string-lessp (second x) (second y))))
       (:length
         (lambda (x y)
           (< (string-length (second x))
              (string-length (second y)))))
       (:number
         (lambda (x y)
           (< (first x) (first y)))))
      ;; other parameter
      ((style "Character style" character-style))
      ;; body
      (with-character-style (style)
        (format t "~&Here come the data, sorted by ~(~a~): " how)
        (format-textual-list data
                             (lambda (x stream)
                               (princ (second x) stream)))))))
```

For an overview of **dw:with-resortable-output** and related facilities: See the section "Overview of Advanced Presentation Output Facilities", page 63.

with-underlining (&optional *stream*) &body *body* *Macro*

Binds the local environment such that character output is underlined.

stream Output stream; the default is ***standard-output***.

Example:

```
(defun underline-example ()
  (fresh-line)
  (with-underlining ()
    (princ 12345)
    (sleep 2)
    (princ 56789)))
```

For an overview of **with-underlining** and related facilities: See the section "Overview of Character Environment Facilities", page 49.

:x-scroll-position of **dw:dynamic-window** *Method*

Returns four values:

1. The absolute location of the current viewport's left edge.
2. The viewport's horizontal extent.
3. The window's minimum x-coordinate (typically `0`).
4. The absolute location of the viewport's right edge.

For an overview of **(flavor:method :x-scroll-position dw:dynamic-window)** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

:x-scroll-to *position type* of **dw:dynamic-window** *Method*

Scrolls the window to a specified x-coordinate.

position The x-coordinate to scroll to.

type The type of scrolling operation. Three possibilities exist:

:absolute
The *position* argument specifies an absolute window location.

:relative The *position* argument specifies a location, in pixels, relative to the current position of the cursor.

:relative-jump
The *position* argument specifies a location, in characters, relative to the current position of the cursor. The width of a character in pixels depends on the default character style for the window; the width of the space character is used.

For an overview of **(flavor:method :x-scroll-to dw:dynamic-window)** and

related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

:y-scroll-position of **dw:dynamic-window** *Method*

Returns four values:

1. The absolute location of the current viewport's top edge.
2. The viewport's vertical extent.
3. The window's minimum y-coordinate (typically `0`).
4. The absolute location of the viewport's bottom edge.

For an overview of **(flavor:method :y-scroll-position dw:dynamic-window)** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

:y-scroll-to *position type* of **dw:dynamic-window** *Method*

Scrolls the window to a specified y-coordinate.

position The y-coordinate to scroll to.

type The type of scrolling operation. Three possibilities exist:

:absolute
The *position* argument specifies an absolute window location.

:relative The *position* argument specifies a location, in pixels, relative to the current position of the cursor.

:relative-jump
The *position* argument specifies a location, in lines, relative to the current position of the cursor. The height of a line in pixels depends on the default character style for the window.

For an overview of **(flavor:method :y-scroll-to dw:dynamic-window)** and related facilities: See the section "Overview of Other Facilities for Program Output", page 60.

PART VI.

Dictionary of Predefined Presentation Types

18. Dictionary Notes

This dictionary includes reference documentation for the following presentation types:

Table of Predefined Presentation Types

Common Lisp Presentation Types

- **and**
- **character**
- **integer**
- **keyword**
- **member**
- **not**
- **null**
- **number**
- **or**
- **package**
- **pathname**
- **satisfies**
- **sequence**
- **string**
- **symbol**
- **symbol-name**
- **t**

Symbolics Common Lisp Presentation Types

- **alist-member**
- **boolean**
- **character-face-or-style**
- **character-style**
- **character-style-for-device**
- **instance**
- **inverted-boolean**
- **null-or-type**
- **sequence-enumerated**
- **subset**
- **token-or-type**
- **type-or-string**

Other Presentation Types

dw:member-sequence
dw:no-type
dw:out-of-band-character
dw:raw-text
dw:replayable-output
fs:directory-pathname
fs:wildcard-pathname
net:host
net:local-host
net:namespace
net:namespace-class
net:network
neti:local-network
net:object
sys:printer
neti:protocol-name
neti:site
net:user
sct:system
sct:system-version
sys:code-fragment
sys:expression
sys:font
sys:form
sys:flavor-name
sys:function-spec
sys:generic-function-name
sys:printer
sys:stack-frame
time:time-interval
time:time-interval-60ths
time:timezone
time:universal-time
tv:window
zwei:buffer

In the dictionary, the types are arranged in alphabetical order (package prefixes excluded).

For conceptual documentation: See the section "Overview of Predefined Presentation Types", page 71.

19. The Facilities

alist-member (&key *alist*) &key *convert-spaces-to-dashes* **nil** *Presentation Type*
Type for accepting or presenting an association list item.

:alist Data option specifying the list of items. The usual form of item is a dotted pair of the print string and its object: ((*String-1* . *object-1*) (*string-2* . *object-2*) ... (*string-n* . *object-n*)).

Alternatively, items can be in the "general list" form: See the section "The Form of a Menu Item" in *Programming the User Interface, Volume B.* One of the advantages of this form is that documentation for each item can be added that will appear if the user asks for help (presses the HELP key) during an **accept** of this type. Documentation is specified with the **:documentation** keyword. See the examples section of **alist-member**.

Two other keywords are permitted in an item list. The first is **:style**, specifying the character style of the presented item.

The second is **:selected-style**. This keyword may only be used when **alist-member** is part of a **dw:accepting-values** function. It specifies the character style of the item when it is selected, that is, after it has been clicked on. The **:selected-style** defaults to the boldface version of the unselected style. For an example: See the macro **dw:accepting-values**, page 175.

:convert-spaces-to-dashes
Presentation option specifying whether spaces in the print string should be converted to dashes; the default is **nil.**

Examples:

```
(accept '((alist-member :alist (("Item 1" . a) ("Item 2" . b)))
          :convert-spaces-to-dashes t)) ==>
Enter Item-1 or Item-2: Item-2
B
((ALIST-MEMBER :ALIST (("Item 1" . A) ("Item 2" . B)))
:CONVERT-SPACES-TO-DASHES T)
```

```
(present 'b '((alist-member :alist (("Item 1" . a) ("Item 2" . b)))
              :convert-spaces-to-dashes t)) ==>
Item-2
#<DISPLAYED-PRESENTATION 444272462>

(defun filter-alist-example ()
  (let ((operator-alist
          '(("Gaussian" :value :gauss
             :documentation "low-pass filter")
            ("Laplacian, HP" :value :lpl-hp
             :documentation "high-pass filter")
            ("Laplacian, ED" :value :lpl-ed
             :documentation "edge detector")
            ("Roberts" :value :rbts
             :documentation "edge detector")
            ("Prewitt, Hz" :value :prw-hz
             :documentation "horizontal edge detector")
            ("Prewitt, Vt" :value :prw-vt
             :documentation "vertical edge detector")
            ("Sobel, Hz" :value :sbl-hz
             :documentation "horizontal edge detector")
            ("Sobel, Vt" :value :sbl-vt
             :documentation "vertical edge detector"))))
    (accept `((alist-member :alist ,operator-alist)
              :description "a 2-dimensional image filter"))))

(filter-alist-example) ==>
Enter a 2-dimensional image filter: HELP
You are being asked to enter a 2-dimensional image filter.

These are the possible 2-dimensional image filters:
Gaussian        low-pass filter
Laplacian, ED   edge detector
Laplacian, HP   high-pass filter
Prewitt, Hz     horizontal edge detector
Prewitt, Vt     vertical edge detector
Roberts         edge detector
Sobel, Hz       horizontal edge detector
Sobel, Vt       vertical edge detector
```

```
Enter a 2-dimensional image filter: Laplacian, HP
:LPL-HP
((ALIST-MEMBER :ALIST
  (("Gaussian" :VALUE :GAUSS :DOCUMENTATION
       "low-pass filter")
   ("Laplacian, HP" :VALUE :LPL-HP :DOCUMENTATION
       "high-pass filter")
   ("Laplacian, ED" :VALUE :LPL-ED :DOCUMENTATION
       "edge detector")
   ("Roberts" :VALUE :RBTS :DOCUMENTATION
       "edge detector")
   ("Prewitt, Hz" :VALUE :PRW-HZ :DOCUMENTATION
       "horizontal edge detector")
   ("Prewitt, Vt" :VALUE :PRW-VT :DOCUMENTATION
       "vertical edge detector")
   ("Sobel, Hz" :VALUE :SBL-HZ :DOCUMENTATION
       "horizontal edge detector")
   ("Sobel, Vt" :VALUE :SBL-VT :DOCUMENTATION
       "vertical edge detector")))
 :DESCRIPTION "a 2-dimensional image filter")
```

Because the prompt generated by **accept** for input of **alist-member** items can sometimes be awkward, you may want to use the meta-presentation argument **:description** to change it. (See the section "Overview of Predefined Presentation Types", page 71.) This was done in the (filter-alist-example) above.

The filter example also demonstrates the advantage of providing an alist of the general list form. The **:documentation** provided in the alist can add much useful information to the display.

A type history is not available for the **alist-member** presentation type.

alist-member is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

and (&rest *types*) *Presentation Type*

Compound type for accepting or presenting an object of two or more presentation types. Typically, the second and subsequent types are derived via the **satisfies** presentation type.

types Data arguments specifying the contributing presentation types.

Examples:

```
(accept '((and sys:expression (satisfies symbolp)))) ==>
Enter the representation of any Lisp
object satisfying SYMBOLP: ramjet
RAMJET
((AND SYS:EXPRESSION
     (SATISFIES SYMBOLP)))

(accept '((and ((integer)) ((satisfies oddp))
                           ((satisfies plusp))))) ==>
Enter an integer satisfying ODDP and
PLUSP [default 9]: 21
21
((AND ((INTEGER))
      ((SATISFIES ODDP))
      ((SATISFIES PLUSP))))
```

The compound presentation type in the first example is equivalent to the **symbol** presentation type and is, in fact, how that type is defined.

and can combine any number of **satisfies** types with an initial, non-**satisfies** type. The second example above shows an initial integer type used with two **satisfies** types to solicit input of odd, positive integers.

Note that the compound type has access to the type history of the initial presentation type, if one exists. However, it does not automatically use the value at the top of the history as the default value in an **accept** function. Rather, it uses the item most recently added to the type history that also satisfies the **satisfies** function(s).

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

boolean *Presentation Type*

Type for accepting or presenting a yes-or-no answer, where "yes" is **t** and "no" is **nil**.

Examples:

```
(accept '((boolean))) ==>
Enter Yes or No: No
NIL
((BOOLEAN))


(present t '((boolean))) ==>Yes
#<DISPLAYED-PRESENTATION 444300153>
```

A type history is not available for the **boolean** presentation type.

boolean is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

See also the **inverted-boolean** presentation type.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

zwei:buffer &key (*create-p :if-forced*) *Presentation Type*

Type for accepting or presenting Zmacs editor buffers.

:create-p Presentation option specifying whether to create the buffer entered in response to an **accept** prompt if it does not already exist.

The default is **:if-forced**. This provides the user the option of changing the input or creating the new buffer, rather than just creating the buffer as in the case of `:create-p t`.

Examples:

```
(accept '((zwei:buffer))) ==>
Enter an editor buffer
[default ui-dict15.sar >sys>doc>uims Q:]: HELP ==>
You are being asked to enter an editor buffer.

These are the possible editor buffers:
  *Buffer-1*
  *Definitions-1*
  doc-29-55.lisp >sys>doc>patch>doc-29 Q:
  miscui2.sar >sys>doc>miscui Q:
  standard-presentation-types.lisp >sys>dynamic-windows Q:
  ui-dict15.sar >sys>doc>uims Q:
```

```
Enter an editor buffer
[default ui-dict15.sar >sys>doc>uims Q:]: *Buffer-1*
#<NON-FILE-BUFFER "*Buffer-1*" 47700004>

(accept '((zwei:buffer) :create-p t)) ==>
Enter an editor buffer
[default ui-dict15.sar >sys>doc>uims Q:]: foo.test
#<NON-FILE-BUFFER "foo.test" 47700567>
((ZWEI:BUFFER) :CREATE-P T)

(present (zwei:make-buffer 'zwei:non-file-buffer)
         '((zwei:buffer))) ==>*Buffer-2*
#<DISPLAYED-PRESENTATION 274672153>
```

The **zwei:buffer** presentation type uses the special variable **zwei:*buffer-history*** to provide its type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

character *Presentation Type*

Type for accepting or presenting single characters.

Examples:

```
(accept '((character)) ==>
Enter a character: R
#\R
((CHARACTER))

(accept '((character)) ==>
Enter a character: r
#\r
((CHARACTER))

(accept '((character)) ==>
Enter a character: %
#\%
((CHARACTER))

(accept '((character)) ==>
Enter a character: 3
#\3
```

```
(present #\, '((character))) ==>,
#<DISPLAYED-PRESENTATION 445346702>
((CHARACTER))
```

Use the **character** presentation type for normal, editable character input. To accept characters that would be mistaken as input-editor commands, for example **#\c-b**, use **dw:out-of-band-character** instead.

There is no type history for the **character** presentation type.

character is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

character-face-or-style *(*&key *device (against-default si:*standard-default-character-style*))* &key *for-attribute-list* Presentation Type

Type for accepting either a fully specified character style, or just the face component. (The device argument, although implemented as a keyword, is required.)

:device Data option specifying the device for the character style. There are four possibilities: **si:*b&w-screen***, **lgp:*lgp-printer***, **lgp:*lgp2-printer***, and **dmp1:*dmp1-printer***.

:against-default
Data option specifying a default character style against which the input character style is merged. See the section "Merging Character Styles" in *Symbolics Common Lisp: Language Concepts*.

:for-attribute-list
Presentation option specifying whether the character style should be presented in list form, for example, `(:fix :bold :normal)`. The default is **nil**. Supply a value of **t** when presenting a character style for inclusion in the attribute list of file.

Examples:

```
 (accept '((character-face-or-style
:device ,si:*b&w-screen*))) ==>
Enter a character face or style: BOLD
#<CHARACTER-STYLE NIL.BOLD.NIL 155157247>
((CHARACTER-FACE-OR-STYLE :DEVICE
#<B&W-SCREEN-DISPLAY-DEVICE 154221604>))

(accept '((character-face-or-style
:device ,si:*b&w-screen*))) ==>
Enter a character face or style: DUTCH.ROMAN.NORMAL
#<CHARACTER-STYLE DUTCH.ROMAN.NORMAL 154174235>
((CHARACTER-FACE-OR-STYLE :DEVICE
#<B&W-SCREEN-DISPLAY-DEVICE 154221604>))
```

The **character-face-or-style** presentation type does not support a type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

character-style (&key *against-default*) &key *for-attribute-list* *Presentation Type*
Type for accepting or presenting character styles.

:against-default
Data option specifying a default character style against which the input character style is merged. See the section "Merging Character Styles" in *Symbolics Common Lisp: Language Concepts.*

:for-attribute-list
Presentation option specifying whether the character style should be presented in list form, for example, `(:fix :bold :normal)`. The default is **nil**. Supply a value of **t** when presenting a character style for inclusion in the attribute list of file.

When accepting a character style, the user is prompted for the family, face, and size, in that order. The first two entries must be terminated by a period, the last by `RETURN` or `END`.

Examples:

```
(accept '((character-style))) ==>
Enter a valid character style: SWISS.BOLD.LARGE
#<CHARACTER-STYLE SWISS.BOLD.LARGE 264231477>

(present (si:parse-character-style '(:swiss :bold :large))) ==>
SWISS.BOLD.LARGE
#<DISPLAYED-PRESENTATION 425221252>
```

The **character-style** presentation type supports a type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

character-style-for-device *(*&key *device* (*against-default* *si:*standard-default-character-style**) (*allow-relative* **t**) (*allow-device-font nil*)) &key *for-attribute-list* (*provide-subhelp t*) Presentation Type

Type for accepting or presenting character styles for a specified device. (The device argument, although implemented as a keyword, is required.)

:device Data option specifying the device for the character style. There are four possibilities: **si:*b&w-screen***, **lgp:*lgp-printer***, **lgp:*lgp2-printer***, and **dmp1:*dmp1-printer***.

:against-default
Data option specifying a default character style against which the input character style is merged. See the section "Merging Character Styles" in *Symbolics Common Lisp: Language Concepts.*

:allow-relative
Data option specifying whether relative style specifications, such as **smaller** or **larger**, are permitted. See the section "Merging Character Styles" in *Symbolics Common Lisp: Language Concepts.*

:allow-device-font
Data option specifying whether a device font is permitted; the default is **nil.**

Device fonts are applicable only to the black-and-white screen device (**si:*b&w-screen***). For a list of possibilities, press HELP after entering "device-font." in an **accept** of this presentation type, with this option specified **t.**

For more information about device fonts: See the section "Mapping a Character Style to a Font" in *Symbolics Common Lisp: Language Concepts.*

:for-attribute-list
Presentation option specifying whether the character style should be presented in list form, for example, `(:fix :bold :normal)`. The default is **nil**. Supply a value of **t** when presenting a character style for inclusion in the attribute list of file.

:provide-subhelp
Presentation option specifying whether to provide a HELP display; the default is **t**. Disable this if a higher-level call provides help.

Examples:

```
(accept '((character-style-for-device
:device ,si:*b&w-screen*))) ==>
Enter a character style: FIX.BOLD.TINY
#<CHARACTER-STYLE FIX.BOLD.TINY 154222436>
((CHARACTER-STYLE-FOR-DEVICE :DEVICE
#<B&W-SCREEN-DISPLAY-DEVICE 154221604>))

(accept '((character-style-for-device
:device ,lgp:*lgp2-printer* :allow-relative t))) ==>
Enter a character style [default FIX.BOLD.TINY]: SWISS.BOLD.SAME
#<CHARACTER-STYLE SWISS.BOLD.SAME 15212221>
((CHARACTER-STYLE-FOR-DEVICE :DEVICE
#<LGP2-DISPLAY-DEVICE 154173651> :ALLOW-RELATIVE T))

(accept '((character-style-for-device
:device ,si:*b&w-screen* :allow-device-font t))) ==>
Enter a character style: DEVICE-FONT.BIGFNT
#<CHARACTER-STYLE DEVICE-FONT.BIGFNT.NORMAL 14251534>
((CHARACTER-STYLE-FOR-DEVICE :DEVICE
#<B&W-SCREEN-DISPLAY-DEVICE 154221604> :ALLOW-DEVICE-FONT T))
```

character-style-for-device is a subtype of **character-style**, from which it inherits a type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sys:code-fragment *Presentation Type*

Type for accepting or presenting pieces of Lisp code. This presentation type is a subtype of **sys:form**, and intended primarily for accessing code fragments in editor buffers. The following example, the definition of a translating mouse handler for editor commands, uses **sys:code-fragment** as the *from-presentation-type* argument:

```
(zwei:define-presentation-to-editor-command-translator
       typeout-menu-arglist-from-buffer
       (sys:code-fragment "Arglist" *standard-comtab*
        :gesture :hyper-meta-middle)
                                (function-spec)
  (when (and (sys:validate-function-spec function-spec)
             (fdefinedp function-spec))
    `(typeout-menu-arglist ,function-spec)))
```

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

fs:directory-pathname &key *(default-version :newest)* *default-type nil) (default-name nil) dont-merge-default* *(direction :read)* *(format :normal)* Presentation Type

Type for accepting or presenting directory pathnames.

This presentation type can be useful if you need to distinguish unequivocally between directory pathname presentations and file pathname presentations. For example, if you can arrange for the availability to your users of some **fs:directory-pathname** presentations, then mouse handlers performing directory-related functions can be defined that do not have to test whether a given **pathname** presentation is a directory pathname, or extract directory objects from **pathname** presentations.

fs:directory-pathname is a subtype of the **pathname** presentation type, from which it inherits a printer, parser, and type history. It also takes the same keyword arguments, as follows:

:default-version

Presentation option specifying the default version number of an accepted file. The default value for this option is **:newest**, the newest file version.

:default-type

Presentation option specifying the default file type, for example, "lisp", "text", "data", and so on. The default value for this option is **nil**.

:default-name
Presentation option specifying the default file name. The default value for this option is **nil**.

:dont-merge-default
Presentation option specifying whether to prevent merging of a partially specified pathname entered by the user against the default pathname. The default value for this option is **nil**, meaning that merging occurs when appropriate; that is, parts of the pathname not entered by the user are supplied from the default.

Suppression of merging against the default and providing a different default (against which merging may or may not be enabled) are different issues. To deal with the latter, use the **:default** option to **accept**: See the function "**accept**", page 167. An example follows:

```
(accept '((pathname) :default-type nil)
        :default (send (fs:default-pathname)
                       :new-pathname :type nil
                       :version :newest))
```

:direction
Presentation option specifying either **:read** (the default) or **:write**. The value supplied is passed through to **fs:complete-pathname** and affects completion behavior. (See the function **fs:complete-pathname** in *Reference Guide to Streams, Files, and I/O.*)

Use the default (**:read**) if the user is likely to enter the pathname of an already existing file when prompted by **accept**, **:write** otherwise.

:format Presentation option specifying the output format of the pathname. There are four choices:

:normal For example, `S:>mb>dw-pgms>fancy-windows.lisp`. This is the default format.

:directory
For example, `>mb>dw-pgms>`. The host, file name, and file type are not displayed.

:dired For example, `fancy-windows.lisp`. Only the file name and type are displayed.

:editor For example, `fancy-windows.lisp >mb>dw-pgms S`. The display format is that used by Zmacs.

For examples illustrating the use of these keywords in pathname presentations: See the presentation type **pathname**, page 320.

fs:directory-pathname is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sys:expression &rest *options* *Presentation Type*

Type for accepting or presenting expressions. An expression is the readable, printed representation of a Lisp object. The expression is not evaluated.

options Presentation options controlling the generation of the printed representation. They are listed in the following table, along with the special variables providing each option with its default value. (Note that these options are the same as those available for the Common Lisp function **write**.)

Option	*Special Variable*
:escape	***print-escape***
:pretty	***print-pretty***
:abbreviate-quote	***print-abbreviate-quote***
:radix	***print-radix***
:base	***print-base***
:circle	***print-circle***
:level	***print-level***
:length	***print-length***
:case	***print-case***
:gensym	***print-gensym***
:array	***print-array***
:readably	***print-readably***
:array-length	***print-array-length***
:string-length	***print-string-length***
:bit-vector-length	***print-bit-vector-length***
:structure-contents	***print-structure-contents***

The special variables are documented together in another section: See the section "Output Functions" in *Reference Guide to Streams,*

Files, and I/O. Consult the documentation for the individual variables to find out what they do and what values they can have. These values are the same that can be supplied with the corresponding presentation options to **sys:expression**.

Examples:

```
(accept '((sys:expression))) ==>
Enter the representation of any Lisp object
[default (ACCEPT '((SYS:EXPRESSION)))]: setq
SETQ
((SYS:EXPRESSION))

(accept '((sys:expression))) ==>
Enter the representation of any Lisp object
[default (ACCEPT '((SYS:EXPRESSION)))]: (+ 33 900)
(+ 33 900)
((SYS:EXPRESSION))

(present (net:find-object-named :network "DNA")
'((sys:expression))) ==>#<DNA-NETWORK DNA 13702517>
#<DISPLAYED-PRESENTATION 275045641>

(accept '((sys:expression))) ==>
Enter the representation of any Lisp object
[default (ACCEPT '((SYS:EXPRESSION)))]:
'#<DISPLAYED-PRESENTATION 275045641>
'#<DISPLAYED-PRESENTATION 275045641>
SYS:FORM
```

The **sys:expression** type occupies a unique position in the data type hierarchy, namely, the highest spot but for one, that occupied by **t**. This means that, except for **t**, **sys:expression** is supertype to all other Symbolics Common Lisp types.

For all data types not explicitly defined as presentation types (via **define-presentation-type**), **sys:expression** serves as the access point to the presentation system. It provides these types with a parser, printer, and type history. In fact, it provides one or more of these functions to many defined presentation types as well.

sys:expression's history includes all previously accepted Lisp objects. This is why, in the **accept** examples above, the default is always `(ACCEPT '((SYS:EXPRESSION)))`; this expression is the most recently accepted one.

When accessed by other types, **sys:expression**'s type history is pruned to objects of the accessing type. For example, **number** and types descended from **number** do not maintain their own type histories. When a previously accepted value is needed to provide, say, a default value in an **accept** of an **integer**, the expression history is pruned to integer objects of which the most recently accepted is used as the default.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sys:flavor-name *Presentation Type*

Type for accepting or presenting symbols that name flavors.

Examples:

```
(accept '((sys:flavor-name))) ==>
Enter a flavor name: DW:PROGRAM-FRAME
DW:PROGRAM-FRAME
((SYS:FLAVOR-NAME))

(present 'dw:margin-mixin '((sys:flavor-name))) ==>DW:MARGIN-MIXIN
#<DISPLAYED-PRESENTATION 275147735>

(accept '((sys:flavor-name))) ==>
Enter a flavor name [default DW:PROGRAM-FRAME]: DW:MARGIN-MIXIN
DW:MARGIN-MIXIN
((SYS:FLAVOR-NAME))
```

The **sys:flavor-name** presentation type supports a type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sys:font *Presentation Type*

Type for accepting or presenting loaded fonts.

Examples:

```
(accept '((sys:font))) ==>
Enter a loaded font: HELP ==>
You are being asked to enter a loaded font.
```

```
There are 87 possible loaded fonts.  Do you want to see them all?
(Y or N) Yes.
These are the possible loaded fonts:
  5X5         DUTCH20B   HL14I     MEDFNTBI         TR12BI
  BIGFNT      DUTCH20BI  HL18      MEDFNTI          TR12I
  BIGFNTB     DUTCH20I   HL18B     MOUSE            TR14
  BIGFNTBI    EINY7      HL18BI    NARROW           TR14B
  BIGFNTI     EUREX12I   HL18I     SWISS12-CCAPS    TR14I
  BOXFONT     EUREX24I   HL8       SWISS12B-CCAPS   TR18
  CPTFONT     HIPPO12    HL8B      SWISS20          TR18B
  CPTFONTB    HL10       HL8BI     SWISS20B         TR8
  CPTFONTBI   HL10B      HL8I      SWISS20BI        TR8B
  CPTFONTC    HL10BI     JESS13    SWISS20I         TR8BI
  CPTFONTCB   HL10I      JESS13B   SYMBOL12         TR8I
  CPTFONTCC   HL12       JESS13I   TINY             TVFONT
  CPTFONTI    HL12B      JESS14    TR10             TVFONTB
  DUTCH14     HL12BI     JESS14B   TR10B            TVFONTBI
  DUTCH14B    HL12I      JESS14I   TR10BI           TVFONTI
  DUTCH14BI   HL14       MATH12    TR10I
  DUTCH14I    HL14B      MEDFNT    TR12
  DUTCH20     HL14BI     MEDFNTB   TR12B

Enter a loaded font: DUTCH20
#<FONT DUTCH20 260074563>
SYS:FONT

(accept '((sys:font))) ==>
Enter a loaded font [default DUTCH20]: SWISS20
#<FONT SWISS20 260160676>
((SYS:FONT))

(present (si:get-font si:*b&w-screen* si:*standard-character-set*
'(:jess :roman :normal))) ==>JESS13
#<DISPLAYED-PRESENTATION 440305757>
```

The **sys:font** presentation type supports a type history.

sys:font is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sys:form &key (*environment* **si:*read-form-environment***) *Presentation Type*

Type for accepting or presenting Lisp forms.

:environment

Presentation option specifying the lexical environment of an input form. (For more on environments: See the section "Lexical Environment Objects and Arguments" in *Symbolics Common Lisp: Language Concepts.*)

```
(accept '((sys:form))) ==>
Enter A Lisp expression to be evaluated
[default (ACCEPT '((SYS:FORM)))]: (symbolp t)
(SYMBOLP T)
((SYS:FORM))

(present '(symbolp t) '((sys:form))) ==>(SYMBOLP T)
#<DISPLAYED-PRESENTATION 275141170>

Command: (SYMBOLP T)
T
```

Presented forms are evaluable. In the above examples, run in the command-or-form context, the *(SYMBOLP T)* form was entered to the `Command:` prompt by clicking left on the output of the preceding **present** function. This form was immediately evaluated. Contrast this behavior with that of **sys:expression** presentations; presented forms are quoted and not evaluable directly.

The **sys:form** presentation type inherits its printer and type history from **sys:expression**.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sys:function-spec (&key *defined-p*) &key (*partial-completers* '(#**space**)) *Presentation Type*

Type for accepting or presenting valid function specs. (For information on function specs: See the section "Function Specs" in *Symbolics Common Lisp: Language Concepts.*)

:defined-p

Data option restricting function specs to those that are defined; the default is **nil**.

:partial-completers
Presentation option specifying a list of characters to be used as completers of function-spec tokens during input; the default list is (#\space).

Examples:

```
(present '+ '((sys:function-spec))) ==>+
#<DISPLAYED-PRESENTATION 275374421>

(accept '((sys:function-spec))) ==>
Enter a valid function spec: +
+
((SYS:FUNCTION-SPEC))

(accept '((sys:function-spec))) ==>
Enter a valid function spec [default +]: (:PROPERTY alpha bravo)
(:PROPERTY ALPHA BRAVO)
((SYS:FUNCTION-SPEC))

(accept '((sys:function-spec :defined-p t))) ==>
Enter a defined function spec: (:PROPERTY alpha bravo)
(:PROPERTY ALPHA BRAVO) is not a defined function spec.
Type RUBOUT to correct your input. [Abort]

(defun (:property alpha bravo) () 1) ==>
(:PROPERTY ALPHA BRAVO)

(accept '((sys:function-spec :defined-p t))) ==>
Enter a defined function spec
[default (:PROPERTY ALPHA BRAVO)]: (:PROPERTY ALPHA BRAVO)
(:PROPERTY ALPHA BRAVO)
((SYS:FUNCTION-SPEC :DEFINED-P T))
```

The **sys:function-spec** presentation type supports a type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sys:generic-function-name &key *show-compatible-message* *Presentation Type*
Type for accepting or presenting function specs for generic functions.

:show-compatible-message
Presentation option specifying whether to also print, if

defined, the name of the compatible message for the generic function. (Compatible messages are specified by an option to **defgeneric**: See the section "Defining a Compatible Message for a Generic Function" in *Symbolics Common Lisp: Language Concepts.*)

Examples:

```
(accept '((sys:generic-function-name))) ==>
Enter a generic function name: HELP
You are being asked to enter a generic function name.

There are 11630 possible generic function names.
Do you want to see them all? (Y or N) No. [Thanks, anyway.]

Enter a generic function name: DW:DO-REDISPLAY
DW:DO-REDISPLAY
((SYS:GENERIC-FUNCTION-NAME))

(present 'sys:print-self '((sys:generic-function-name))) ==>
SYS:PRINT-SELF
#<DISPLAYED-PRESENTATION 275755254>

(present 'sys:print-self '((sys:generic-function-name)
:show-compatible-message t)) ==>SYS:PRINT-SELF (:PRINT-SELF)
#<DISPLAYED-PRESENTATION 275755527>
```

The **sys:generic-function-name** presentation type supports a type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

net:host *Presentation Type*

Type for accepting or presenting a network host.

Examples:

```
(accept '((net:host))) ==>
Enter the name of a host: Harpagornis
#<LISPM-HOST HARPAGORNIS 53344734>
((NET:HOST))
```

```
(accept '((net:host))) ==>
Enter the name of a host [default HARPAGORNIS]: laurent
#<MSDOS-HOST YVES-ST-LAURENT 533601167>
((NET:HOST))

(present (si:parse-host "owl") '((net:host))) ==>OWL
#<DISPLAYED-PRESENTATION 275435731>

(accept '((net:host))) ==>
Enter the name of a host [default YVES-ST-LAURENT]: OWL
#<LISPM-HOST OWL 13707365>
((NET:HOST))
```

The **net:host** presentation type has its own parser and type history; it inherits its printer via **net:object**, to which it is subtype, from **sys:expression**.

net:host is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

instance (&optional (*flavor* '*)) *Presentation Type*

Type for accepting or presenting flavor instances.

flavor Data argument specifying what flavor this is an instance of; the default leaves the flavor unspecified.

Examples:

```
(present (tv:make-window 'dw:dynamic-window) 'instance) ==>
Dynamic Window 1
#<DISPLAYED-PRESENTATION 444315574>

(accept '((instance))) ==>
Enter the representation of any Lisp object: Dynamic Window 1
#<DYNAMIC-WINDOW Dynamic Window 1 1200437 deactivated>
INSTANCE
```

```
(accept '((instance 'dw:dynamic-window))
        :prompt "Enter an instance") ==>
Enter an instance [default Dynamic Window 1]: Dynamic Window 1
#<DYNAMIC-WINDOW Dynamic Window 1 1200437 deactivated>
((INSTANCE 'DW:DYNAMIC-WINDOW))
```

The **instance** presentation type inherits its printer and parser functions – as well as a type history – from the **sys:expression** presentation type. Thus, in the first **accept** function above, the prompt says to "Enter the representation of any Lisp object". We override this by providing our own prompt in the second call to **accept.**

In the first **accept** form, the entered *Dynamic Window 1* is in italics because it was entered via a mouse click on the presentation created by the **present** function. If we had tried to type in "dynamic window 1", **accept** would have returned the object DYNAMIC when the first space character was typed.

instance is not a presentation type that you are likely to need for writing end-user interfaces to applications. A number of Common Lisp presentation types are in this category, for example, **structure** and **hash-table.** Like **instance,** all inherit their parser, printer, and type history from **sys:expression.** And, as in the case of **instance,** when **sys:expression**'s type history is accessed to provide, for example, a default value in an **accept** function, the history is "pruned" to objects only of the sought-after type. Thus, in the second **accept** function above, not any Lisp object is offered as a default, but an **instance** object.

All flavors are subtype to the **instance** presentation type. Similarly, all structures are subtype to the **structure** type. The two types are thereby important for links they provide to the presentation-type system for flavors and structures, respectively.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

integer (&optional (*range-low* '*) (*range-high* '*)) &key (*base 10*) *Presentation Type*
Type for accepting or presenting integers.

range-low
Data argument specifying a lower limit for integer objects. The default is no lower limit.

range-high
Data argument specifying an upper limit for integer objects. The default is no upper limit.

:base Presentation option specifying the base used for integer presentations; the default is 10.

Examples:

```
(accept '((integer 0 100))) ==>
Enter an integer greater than or equal to 0
and less than or equal to 100: 0
0
((INTEGER 0 100))

(accept '((integer (0) (100)))) ==>
Enter an integer greater than 0 and less than 100: 1
1
((INTEGER (0) (100)))

(present 10 '((integer) :base 8)) ==>12
#<DISPLAYED-PRESENTATION 445411244>

(accept '((integer 0 100)))
Enter an integer greater than or equal to 0
and less than or equal to 100: 12
10
((INTEGER) :BASE 8)

(accept '((integer 0 100) :base 8)) ==>
Enter an octal integer greater than or equal to 0
and less than or equal to 144: 12
10
((INTEGER) :BASE 8)

(present 50 '((integer 0 100))) ==>50
#<DISPLAYED-PRESENTATION 445430232>

(accept '((integer)))
Enter an integer [default 8]: 50
50
((INTEGER 0 100))
```

```
(accept '((integer))) ==>
Enter an integer [default 5]: 50
50
((INTEGER 0 100))
```

When specifying range limits, if the limits are provided without enclosing parentheses, they are inclusive; with parentheses, exclusive. Contrast the first two **present** functions.

The 12 input to the second and third **accept** functions above was entered by clicking on the output of the first **present** function. Note that, regardless of the base used for the integer presentation, the object returned remains the same.

Note also in the second and third **accept**s that the data type returned is the one entered, an integer, not a range-restricted integer, even though the functions restricted the range of acceptable integers. Contrast this with the final **present-accept** pair: the object presented as a range-restricted integer is entered to a non-restricted integer accepting function; the object's data type (subtype, actually) is retained.

Finally, note that the **integer** presentation type supports a type history (inherited from **sys:expression**), the source of the default value offered in the last **accept** function, but that range-restricted integer types do not.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

inverted-boolean *Presentation Type*

Type for accepting or presenting a yes-or-no answer, where "yes" is **nil** and "no" is **t**. Use it when the sense of the internal action is inverted from the user sense.

Examples:

```
(accept '((inverted-boolean))) ==>
Enter Yes or No: No
T
((INVERTED-BOOLEAN))

(present t '((inverted-boolean))) ==>No
#<DISPLAYED-PRESENTATION 444312267>
```

A type history is not available for the **inverted-boolean** presentation type.

inverted-boolean is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

See also the **boolean** presentation type.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

keyword *Presentation Type*

Type for accepting or presenting keywords.

Examples:

```
(accept '((keyword))) ==>
Enter a keyword: orientation
:ORIENTATION
((KEYWORD))

(accept '((keyword))) ==>
Enter a keyword [default ORIENTATION]: :sojac
:|:SOJAC|
((KEYWORD))

(accept '((keyword)))
Enter a keyword: 1492
:|1492|
((KEYWORD))

(present :orientation '((keyword))) ==>ORIENTATION
#<DISPLAYED-PRESENTATION 454276732>
```

keyword inherits its printer and type history from the **sys:expression** presentation type.

keyword is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

net:local-host *Presentation Type*

Type for accepting or presenting the local host. The local host is accepted and presented as "Local".

Examples:

```
(accept '((si:local-host))) ==>
Enter a local host: Local
#<LISPM-HOST OYSTERCATCHER 13702373>
((SI:LOCAL-HOST))

(present net:*local-host* '((si:local-host))) ==>Local
#<DISPLAYED-PRESENTATION 275456200>

(accept '((si:local-host))) ==>
Enter a local host [default Local]: Local
#<LISPM-HOST OYSTERCATCHER 13702373>
((SI:LOCAL-HOST))
```

The **net:local-host** presentation type is subtype to the **net:host** type, but has its own parser and printer. It inherits a type history from the latter, but prunes it to occurrences of "Local".

net:local-host is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

neti:local-network *Presentation Type*

Type for accepting or presenting local network objects. (A local network is one to which th current machine is connected.)

Examples:

```
(accept '((neti:local-network))) ==>
Enter a local network: HELP
You are being asked to enter a local network.

These are the possible local networks:
  CHAOS
  FBAND
  INTERNET

Enter a local network: INTERNET
#<INTERNET-NETWORK INTERNET 13700021>
((NETI:LOCAL-NETWORK))
```

```
(present (car neti:*local-networks*)
         '((neti:local-network))) ==>FBAND
#<DISPLAYED-PRESENTATION 275517001>

(accept '((neti:local-network)))
Enter a local network [default INTERNET]: FBAND
#<FBAND-NETWORK FBAND 261216753>
((NETI:LOCAL-NETWORK))
```

The **neti:local-network** presentation type supports its own type history.

neti:local-network is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

member (&rest *elements*) *Presentation Type*

Type for accepting or presenting one of a series of objects. The printed representations of the objects must be unique, that is, no two representations can be **string-equal**.

elements The series of objects. These objects are data arguments for this presentation type.

Examples:

```
(accept '((member New York Stock Exchange))) ==>
Enter New, York, Stock, or Exchange: York
YORK
((MEMBER NEW YORK STOCK EXCHANGE))


(accept `((member ,(pathname "y:>pgm>ui-1.lisp")
                  ,(pathname "y:>pgm>ui-2.lisp")
                  ,(pathname "y:>pgm>ui-3.lisp")))) ==>
Enter Y:>pgm>ui-1.lisp, Y:>pgm>ui-2.lisp,
or Y:>pgm>ui-3.lisp: Y:>pgm>ui-2.lisp
#P"Y:>pgm>ui-2.lisp"
((MEMBER #P"Y:>pgm>ui-1.lisp" #P"Y:>pgm>ui-2.lisp"
#P"Y:>pgm>ui-3.lisp"))
```

Because the prompt generated by **accept** for input of **member** objects can sometimes be awkward, you may want to use the meta-presentation argument **:description** to change the prompt. (See the section "Predefined Presentation Types", page 71.)

The **member** presentation type works differently from the **member** function in how it determines group membership. The presentation type merely checks to see if the printed representation of an object is the same as one of its elements. The function bases membership decisions on **eql**.

There is no type history for the **member** presentation type.

The **dw:member-sequence** presentation type is similar to **member**, except that it takes a single argument instead of a series of arguments. See the presentation type **dw:member-sequence**, page 311.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

dw:member-sequence (*sequence*) *Presentation Type*

Type for accepting or presenting one of a series of objects. The printed representations of the objects must be unique, that is, no two representations can be **string-equal**.

sequence Data argument specifying a sequence containing the objects.

Examples:

```
(accept '((dw:member-sequence
            (Kierkegaard Heidegger Bubar Barth)))) ==>
Enter Kierkegaard, Heidegger, Bubar, or Barth: Heidegger
HEIDEGGER
((DW:MEMBER-SEQUENCE (KIERKEGAARD HEIDEGGER BUBAR BARTH)))

(setq adenosine-list '("AMP" "ADP" "ATP"))
("AMP" "ADP" "ATP")

(accept `((dw:member-sequence ,adenosine-list)))
Enter AMP, ADP, or ATP: ATP
"ATP"
((DW:MEMBER-SEQUENCE ("AMP" "ADP" "ATP")))
```

Because the prompt generated by **accept** for input of **dw:member-sequence** objects can sometimes be awkward, you may want to use the meta-presentation argument **:description** to change it. (See the section "Predefined Presentation Types", page 71.)

dw:member-sequence is similar to the **member** presentation type, except that it take a single argument instead of a series of arguments. See the presentation type **member**, page 310.

The **dw:member-sequence** presentation type does not support a type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

neti:namespace *Presentation Type*

Type for accepting or presenting namespace objects.

Examples:

```
(present net:*namespace* '((neti:namespace))) ==>SCRC
#<DISPLAYED-PRESENTATION 275467554>

(accept '((neti:namespace)))
Enter a namespace: SCRC
#<NAMESPACE SCRC 13700207>
((NETI:NAMESPACE))

(accept '((neti:namespace)))
Enter a namespace [default SCRC]: SCRC
#<NAMESPACE SCRC 13700207>
((NETI:NAMESPACE))
```

Through flavor inheritance, the **neti:namespace** presentation type is subtype to the **net:object** type, from which it inherits a type history. The history inherited includes all accepted objects of the **net:object** type; that is, no pruning of the history occurs.

For presentations of namespace classes, as opposed to the namespace objects themselves, use the **net:namespace-class** presentation type.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

net:namespace-class *Presentation Type*

Type for accepting or presenting namespace classes, of which there are currently seven:

:file-system
:user
:printer

:network
:host
:site
:namespace

Examples:

```
(accept '((net:namespace-class))) ==>
Enter a namespace class: printer
:PRINTER
((NET:NAMESPACE-CLASS))

(accept '((net:namespace-class))) ==>
Enter a namespace class: Namespace
:NAMESPACE
((NET:NAMESPACE-CLASS))

(present :site '((net:namespace-class))) ==>Site
#<DISPLAYED-PRESENTATION 275427546>
```

The **net:namespace-class** presentation type is based on the **dw:member-sequence** type. Neither supports a type history.

For presentations of namespace objects, as opposed to namespace classes, use the **net:namespace** presentation type.

net:namespace-class is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

net:network *Presentation Type*

Type for accepting or presenting network objects.

Examples:

```
(present (net:find-object-named
           :network "DNA") '((net:network))) ==>DNA
#<DISPLAYED-PRESENTATION 275510033>

(accept '((net:network))) ==>
Enter a network: DNA
#<DNA-NETWORK DNA 13702517>
((NET:NETWORK))
```

Through flavor inheritance, the **net:network** presentation type is subtype to the **net:object** type, from which it inherits a type history. The history inherited includes all accepted objects of the **net:object** type; that is, no pruning of the history occurs.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

dw:no-type *Presentation Type*

Bogus presentation type for use with mouse handlers. **dw:no-type** is used to ensure that handlers intended to be active only over blank areas of a window are not active over presentations. See the macro "**define-presentation-action**", page 179.

For an overview of **dw:no-type** and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

not *(type)* *Presentation Type*

Type for modifying a **satisfies** presentation type. There is no parser or printer for this type; it can only be used as part of a compound type incorporating **satisfies**.

type Data argument specifying the presentation type to qualify. The only legitimate possibility is a **satisfies** type.

There is no type history for the **not** presentation type.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

null *Presentation Type*

Type for accepting or presenting a null object (**nil**). The **null** type is necessary because no parser or printer can be defined for **nil**.

Null objects are presented as "None". They can be accepted by pressing RETURN to the **accept** function prompt, or clicking on a previously presented "None".

Examples:

```
(present nil '((null))) ==>None
#<DISPLAYED-PRESENTATION 454227454>

(present nil) ==>None
#<DISPLAYED-PRESENTATION 454227707>
```

```
(accept '((null))) ==>
Enter a null value: <RETURN>
NIL
((NULL))

(accept '((null))) ==>
Enter a null value: None
NIL
NULL
```

The most common use of **null** is as part of an **or** compound presentation type. For such a combination, use the **null-or-type** presentation type.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

null-or-type (*presentation-type*) *Presentation Type*

Compound type for accepting or presenting either **nil** or an object of a specified presentation type. **nil** is accepted or presented as "None".

presentation-type
Data argument specifying a presentation type.

Examples:

```
(accept '((null-or-type number))) ==>
Enter a null or type: 2.2
2.2
((NULL-OR-TYPE NUMBER))

(accept '((null-or-type number))
        :prompt "Enter a number or \"None\"") ==>
Enter a number or "None" [default 2.2]: None
NIL
((NULL-OR-TYPE NUMBER))

(present nil '((null-or-type number))) ==>None
#<DISPLAYED-PRESENTATION 444713264>
```

If the type specified in the **null-or-type** presentation type supports a type history, this history is used. This is the source of the default value shown in the second call to **accept** above.

null-or-type is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

number (&optional *range-low range-high*) &key (*base 10*) *Presentation Type*

Type for accepting or presenting numbers.

range-low
: Data argument specifying a lower limit for number objects. The default is no lower limit.

range-high
: Data argument specifying an upper limit for number objects. The default is no upper limit.

:base Presentation option specifying the base used for integer presentations; the default is `10`.

Examples:

```
(accept 'number)
Enter a number: 23
23
(NUMBER)

(accept '(number :base 10)) ==>
Enter a decimal number: 12
12
(NUMBER :BASE 10)

(accept '((number 0 10) :base 2)) ==>
Enter a binary number greater than or equal to 0
and less than or equal to 1010: 111
7
((NUMBER 0 10) :BASE 2)

(accept '((number 0 10) :base 2)) ==>
Enter a binary number greater than or equal to 0
and less than or equal to 1010: 2
2
((NUMBER 0 10) :BASE 2)
```

When specifying range limits, if the limits are provided without enclosing parentheses, they are inclusive; with parentheses, exclusive.

Unlike the **integer** presentation type, number does not check input for violation of the **:base** specification. Thus, in the final **accept** function above, a 2 is entered and returned even though binary numbers are sought.

number is supertype to all other numeric presentation types. See the section "Types of Numbers" in *Symbolics Common Lisp: Language Concepts*. It provides the family with its printer and parser functions. As with other Common Lisp types, **number** is subtype to **sys:expression**, from which it inherits a type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

net:object *Presentation Type*

Type for accepting or presenting network objects.

Examples:

```
(accept '((net:object))) ==>
Enter a namespace object: (Class) HELP==>
You are being asked to enter a namespace object.

These are the possible namespace classes:
  File-System  Printer
  Host         Site
  Namespace    User
  Network

Enter a namespace object: User JO
#<USER JO 13731243>
((NET:OBJECT))

(accept '((net:object))) ==>
Enter a namespace object [default JO]: Host OYSTERCATCHER
#<LISPM-HOST OYSTERCATCHER 13702373>
((NET:OBJECT))

(present (net:find-object-named :network "chaos")
'((net:object))) ==>CHAOS
#<DISPLAYED-PRESENTATION 275037261>

(accept '((net:object))) ==>
Enter a namespace object [default OYSTERCATCHER]: CHAOS
#<CHAOS-NETWORK CHAOS 13700033>
CHAOS:CHAOS-NETWORK
```

When accepting **net:object** input, the user is first prompted for the class of the object. The possible classes, from File-System to User, are listed in the help display shown in the first example above. After entering the class of net object, the user should type a space and then the name of the object itself.

The **net:object** presentation type is built on a flavor of the same name. It inherits its printer and type history from the **sys:expression** presentation type. It is, in turn, supertype to several other network-related types:

net:host
net:local-host
neti:namespace
net:network
neti:site
net:user

When you wish handle a particular class of network object, as opposed to any object, one of the above presentation types might be more suitable than **net:object**.

net:object is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

or (&rest *types*) *Presentation Type*

Compound type for accepting objects as one of two or more possible presentation types. (Presenting objects as **or** types is not useful.)

types Data arguments specifying the possible presentation types.

Examples:

```
(present 'some-symbol) ==>SOME-SYMBOL
#<DISPLAYED-PRESENTATION 274336643>

(present "some-string") ==>some-string
#<DISPLAYED-PRESENTATION 274337201>

(accept '((or symbol string))) ==>
Enter a symbol or a string: SOME-SYMBOL
SOME-SYMBOL
SYMBOL
```

```
(accept '((or symbol string))) ==>
Enter a symbol or a string [default SOME-SYMBOL]: some-string
"some-string"
STRING
```

Some tips on the use of **or**: Never give it to **accept** directly or use it in a **cp:define-command**. What **or** is good for is automatically writing token rescanning multiple syntax parsers for your own presentation type. Use it in an **:expander**: See the function "**define-presentation-type**", page 366. The types **null-or-type**, **token-or-type**, and **type-or-string** are provided for the common cases.

The **or** presentation type has access to the **sys:expression** type history. The value provided as a default in an **accept** of an **or** type is the most recently accepted object whose presentation type is one of the possible *types*.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

dw:out-of-band-character (&rest *chars*) *Presentation Type*

Type for accepting characters that would normally be interpreted as input editor commands, such as the shifted characters c-B or c-E.

chars Data arguments specifying the shifted characters.

Examples:

```
(accept '((dw:out-of-band-character #\c-F #\m-Scroll #\m-C))) ==>
Enter one of the characters c-F, m-SCROLL, or m-sh-C: m-SCROLL
#\m-Scroll
((DW:OUT-OF-BAND-CHARACTER #\c-F #\m-Scroll #\m-C))

(accept '((dw:out-of-band-character #\c-F #\m-SCROLL #\m-C))) ==>
Enter one of the characters c-F, m-SCROLL, or m-sh-C
[default Meta-Scroll]: c-F
#\c-F
((DW:OUT-OF-BAND-CHARACTER #\c-F #\m-Scroll #\m-C))
```

dw:out-of-band-character is subtype to the **character** presentation type, from which it inherits its printer and type history. The type history is pruned to include only previously accepted out-of-band characters.

To accept or present ordinary characters, use **character**: See the presentation type **character**, page 290.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

package *Presentation Type*

Type for accepting or presenting packages.

Examples:

```
(present (find-package 'dynamic-windows) '((package))) ==>
DYNAMIC-WINDOWS
#<DISPLAYED-PRESENTATION 274353464>

(accept '((package))) ==>
Enter a package: DYNAMIC-WINDOWS
#<Package DYNAMIC-WINDOWS 45652740>
((PACKAGE))

(accept '((package))) ==>
Enter a package [default DYNAMIC-WINDOWS]: SCL
#<Package SYMBOLICS-COMMON-LISP 46405507>
((PACKAGE))
```

The **package** presentation type supports a type history.

package is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

pathname &key (*default-version :newest*) (*default-type nil*) (*default-name nil*) *dont-merge-default* (*direction :read*) (*format :normal*) *Presentation Type*

Type for accepting or presenting pathnames.

:default-version
Presentation option specifying the default version number of an accepted file. The default value for this option is **:newest**, the newest file version.

:default-type
Presentation option specifying the default file type, for

example, "lisp", "text", "data", and so on. The default value for this option is **nil**.

:default-name
Presentation option specifying the default file name. The default value for this option is **nil**.

:dont-merge-default
Presentation option specifying whether to prevent merging of a partially specified pathname entered by the user against the default pathname. The default value for this option is **nil**, meaning that merging occurs when appropriate; that is, parts of the pathname not entered by the user are supplied from the default.

Suppression of merging against the default and providing a different default (against which merging may or may not be enabled) are different issues. To deal with the latter, use the **:default** option to **accept**: See the function "**accept**", page 167. An example follows:

```
(accept '((pathname) :default-type nil)
        :default (send (fs:default-pathname)
                       :new-pathname :type nil
                       :version :newest))
```

:direction
Presentation option specifying either **:read** (the default) or **:write**. The value supplied is passed through to **fs:complete-pathname** and affects completion behavior. (See the function **fs:complete-pathname** in *Reference Guide to Streams, Files, and I/O.*)

Use the default (**:read**) if the user is likely to enter the pathname of an already existing file when prompted by **accept**, **:write** otherwise.

:format Presentation option specifying the output format of the pathname. There are four choices:

:normal For example, `S:>mb>dw-pgms>fancy-windows.lisp`. This is the default format.

:directory
For example, `>mb>dw-pgms>`. The host, file name, and file type are not displayed.

:dired For example, `fancy-windows.lisp`. Only the file name and type are displayed.

:editor For example, `fancy-windows.lisp >mb>dw-pgms S`. The display format is that used by Zmacs.

Examples:

```
(present #p"y:>yosemite-s>gold.text") ==>Y:>yosemite-s>gold.text
#<DISPLAYED-PRESENTATION 274370245>

(present #p"y:>yosemite-s>gold.text" '((pathname)
                                       :format :editor)) ==>
gold.text >yosemite-s Y:
#<DISPLAYED-PRESENTATION 274370523>

(accept '((pathname))) ==>
Enter the pathname of a file: gold.text >yosemite-s Y:
#P"Y:>yosemite-s>gold.text"
((PATHNAME) :FORMAT :EDITOR)

(accept '((pathname) :default-version 1)) ==>
Enter the pathname of a file
[default Y:>yosemite-s>gold.text]: silver
#P"Y:>yosemite-s>silver.text.1"
FS:LMFS-PATHNAME

(accept '((pathname) :default-type "data"
                     :default-name "the-rabbit")) ==>
Enter the pathname of a file
[default Y:>yosemite-s>silver.text.1]: Y:>yosemite-s>
#P"Y:>yosemite-s>the-rabbit.data.newest"
FS:LMFS-PATHNAME

(accept '((pathname) :dont-merge-default t)) ==>
Enter the pathname of a file
[default Y:>yosemite-s>the-rabbit.data]: other-varmints
#P"Y:other-varmints"
FS:LMFS-PATHNAME
```

```
(accept '((pathname))) ==>
Enter the pathname of a file
[default Y:>other-varmints]: VIXEN:/b-bunny/y-s.data
#P"VIXEN:/b-bunny/y-s.data"
FS:UNIX42-PATHNAME
```

The **pathname** presentation type supports a type history.

pathname is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

Two subtypes to **pathname** are included among the documented predefined presentation types:

- **fs:directory-pathname**
- **fs:wildcard-pathname**

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sys:printer *Presentation Type*

Type for accepting or presenting printers.

Examples:

```
(accept '((sys:printer)))
Enter a printer [default Symbolics Paradigm]: Symbolics Paradigm
#<LGP2-PRINTER PARADIGM 13701250>
SYS:PRINTER

(present (net:find-object-named :printer "Asahi")
         '((sys:printer))) ==>Asahi Shimbun
#<DISPLAYED-PRESENTATION 275641455>
```

The **sys:printer** presentation type supports a type history.

sys:printer is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

neti:protocol-name (&key *service*) *Presentation Type*

Type for accepting or presenting names of network protocols.

Examples:

```
(accept '((neti:protocol-name))) ==>
Enter a network protocol: Domain-Simple
:DOMAIN-SIMPLE
((NETI:PROTOCOL-NAME))

(present :converse '((neti:protocol-name))) ==>CONVERSE
#<DISPLAYED-PRESENTATION 275603433>

(present (car neti:*protocol-list*)
         '((neti:protocol-name))) ==>MANDELBROT
#<DISPLAYED-PRESENTATION 275607026>
```

There is no type history for the **neti:protocol-name** presentation type.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

dw:raw-text *Presentation Type*

Type providing access to the individual characters from which all textual presentations are constructed. This type is for the exclusive use of mouse handlers, usually as the *from-presentation-type* argument. (For more on handlers: See the section "Overview of Mouse Handler Facilities", page 39.) You cannot use it to accept or present text or characters.

The following example is the source code for a translating mouse handler defined on **dw:raw-text**, translating it to an internal presentation type, **dw::character-style-family**:

```
(define-presentation-translator
        si:characters-character-style-family
        (dw:raw-text dw::character-style-family) (bp)
  (when (< (second bp) (string-length (first bp)))
    (let ((char (aref (first bp) (second bp))))
      (si:cs-family (si:char-style char)))))
```

zwei:bp is a presentation type inheriting from **dw:raw-text**, and used for accessing text characters in editor buffers.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

satisfies (*satisfies-function*) *Presentation Type*

satisfies is a Common Lisp type specifier that takes a predicate as an argument and returns **t** or **nil** according to the return from the predicate. The **satisfies** type is used only as part of an **and** presentation type.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sequence (&optional (*type* '*)) &key (*sequence-delimiter* #\,) (*echo-space t*) *Presentation Type*

Type for accepting or presenting one or more objects of a specified presentation type.

type Presentation type for the objects in the sequence. The specified type is a data argument to the **sequence** presentation type.

The *type* argument defaults to the **t** presentation type. Because **t** has no parser and uses **princ** as its printer, not supplying the *type* argument when you use the **sequence** presentation type does not produce useful results.

:sequence-delimiter

Presentation option specifying the character used to delimit items in the sequence; the default is the comma character, #\,.

When accepting objects in an enumerated sequence, the user must enter the sequence-delimiter character between items.

:echo-space

Presentation option specifying whether to echo a space character after the comma (or other **:sequence-delimiter** character) is typed; the default is **t**.

Although not a subtype, **sequence** can be regarded as a specialized version of the **sequence-enumerated** presentation type. Instead of specifying a series of presentation types as in the case of **sequence-enumerated**, you specify only one type for the entire series of objects. In fact, when objects are entered individually to an **accept** of a **sequence**, the types of the objects, although identical, are enumerated. Observe this behavior in the first example below.

Examples:

```
(accept '((sequence package))) ==>
Enter one or more packages
[default SYMBOLICS-COMMON-LISP]: SCL, DW, TV, SCT
(#<Package SYMBOLICS-COMMON-LISP 46405507>
#<Package DYNAMIC-WINDOWS 45652740>
#<Package TV 46031453>
#<Package SYSTEM-CONSTRUCTION-TOOL 46366410>)
((SEQUENCE-ENUMERATED PACKAGE PACKAGE PACKAGE PACKAGE))

(present '(0 16 32 64) '((sequence ((integer) :base 16))))
#<DISPLAYED-PRESENTATION 274631670>

(accept '((integer))) ==>
Enter an integer: 40
64
((INTEGER) :BASE 16)

(accept '((sequence integer))) ==>
Enter one or more integers: 0, 10, 20, 40
(0 16 32 64)
((SEQUENCE ((INTEGER) :BASE 16)))
```

Note that when you have presented a **sequence** of objects, that the objects are subsequently acceptable as input either as individual objects or as the **sequence**. This is shown by the last three examples above. We **present** a series of integers, and subsequently click on one of them (40) to enter it to an **accept** or an **integer**; and then click on the entire sequence to give it to an **accept** of and **integer** sequence.

The **sequence** presentation type has access to the type history supported, if any, by the specified **type**.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sequence-enumerated (&rest *data-types*) &key (*sequence-delimiter* Presentation Type #\,) (*echo-space* **t**)

Compound type for accepting or presenting a sequence of objects, each of a specified presentation type.

data-types

The presentation types of the objects. These are the data arguments to the **sequence-enumerated** presentation type.

:sequence-delimiter

Presentation option specifying the character used to delimit items in the sequence; the default is the comma character, #\,.

When accepting objects in an enumerated sequence, the user must enter the sequence-delimiter character between items.

:echo-space

Presentation option specifying whether to echo a space character after the comma (or other **:sequence-delimiter** character) is typed; the default is **t**.

Examples:

```
(accept '((sequence-enumerated (integer 1 10)
                sys:form string))) ==>
Enter an integer greater than or equal to 1 and less
than or equal to 10, A Lisp expression to be evaluated,
and a string: 5, (setq alpha "bravo"), "Not very useful"
(5 (SETQ ALPHA "bravo") "Not very useful")
((SEQUENCE-ENUMERATED (INTEGER 1 10) SYS:FORM STRING))

(present `(,(pathname "y:>ui.lisp") telson
           ,(find-package "dynamic-windows"))
         '((sequence-enumerated pathname symbol package))) ==>
Y:>ui.lisp, TELSON, and DYNAMIC-WINDOWS
#<DISPLAYED-PRESENTATION 444476230>
```

The **sequence-enumerated** presentation type does not support a type history.

sequence-enumerated is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

neti:site *Presentation Type*

Type for accepting or presenting site objects.

Examples:

```
(present net:*local-site* '((neti:site))) ==>SCRC
#<DISPLAYED-PRESENTATION 275626405>

(accept '((neti:site))) ==>
Enter a site: SCRC
#<SITE SCRC 13700014>
((NETI:SITE))
```

Through flavor inheritance, the **neti:site** presentation type is subtype to the **net:object** type, from which it inherits a type history. The history inherited includes all accepted objects of the **net:object** type; that is, no pruning of the history occurs.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sys:stack-frame *Presentation Type*

Type for accepting or presenting stack frames. This presentation type is intended primarily for use by the debugger and debugging functions.

The following example shows entry into the debugger from an editor typeout window. The debugger was entered because **oddp** was called with no arguments. The frame, **ODDP**, containing the error is at the top of the stack.

```
Command: (oddp) ==>
Trap: The function ODDP was called with too few arguments.

ODDP:
   --Missing args:--
   Arg 0 (INTEGER)
s-A, RESUME:   Supply the missing arguments.
s-B:           Retry the FUNCALL-N-RETURN instruction
s-C, ABORT:    Return to Breakpoint ZMACS in Editor Typeout Window
s-D:           Editor Top Level
s-E:           Restart process ZMACS-WINDOWS
→ Eval (program): (setq stk-frm (accept '((sys:stack-frame)))) ==>
Enter a stack frame: ODDP
(#<DTP-LOCATIVE 52700741> . #<TOO-FEW-ARGUMENTS-TRAP 44070612>)
→ Eval (program): (present stk-frm '((sys:stack-frame))) ==>ODDP
#<DISPLAYED-PRESENTATION 276024201>
→ Abort Abort
Return to Breakpoint ZMACS in Editor Typeout Window
```

sys:stack-frame does not support a type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

string &key *delimiters* *Presentation Type*

Type for accepting or presenting strings.

:delimiters

Presentation option specifying a list of characters serving as string delimiters (terminators) during input of strings to **accept**. The default delimiters are #**return** and #**end**.

Examples:

```
(accept '((string))) ==>
Enter a string: "Morgan the Pirate"
"Morgan the Pirate"
((STRING))

(accept '((string) :delimiters (#\line))) ==>
Enter a string (end with LINE)
[default Morgan the Pirate]: Several species
of small, furry
creatures gathered
together in a cave ...

"Several species
of small, furry
creatures gathered
together in a cave ..."
((STRING) :DELIMITERS (#\Line))

(present "Another whimsical string") ==>Another whimsical string
#<DISPLAYED-PRESENTATION 274760165>

(accept '((string)))
Enter a string: Another whimsical string
"Another whimsical string"
STRING
```

The **string** presentation type supports a type history.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

subset (&rest *keywords*) *Presentation Type*

Type for accepting or presenting zero or more objects from a group of keyword identifiers.

keywords The set of keywords. These are data arguments to the **subset** presentation type.

Examples:

```
 (accept '((subset :mercenaria :mya :mytilus))) ==>
Enter a subset of the identifiers MERCENARIA,
MYA, and MYTILUS: Mercenaria, Mytilus
(:MERCENARIA :MYTILUS)
((SUBSET :MERCENARIA :MYA :MYTILUS))

(present '(:mya) '((subset :mercenaria :mya :mytilus))) ==>MYA
#<DISPLAYED-PRESENTATION 444621057>
```

When accepting input of this type, the user must separate identifiers with commas. If input is terminated without any identifiers having been entered, **accept** returns **nil**.

A type history is not available for the **subset** presentation type.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

symbol *Presentation Type*

Type for accepting or presenting symbols.

Examples:

```
(accept '((symbol)))
Enter a symbol: RNA
RNA
((SYMBOL))

(accept '((symbol)))
Enter a symbol [default RNA]: DNA
DNA
((SYMBOL))
```

```
(present 't-RNA)
#<DISPLAYED-PRESENTATION 274753204>

(accept '((symbol)))
Enter a symbol [default RNA]: T-RNA
T-RNA
SYMBOL
```

The **symbol** presentation type inherits its parser, printer, and type history from the **sys:expression** presentation type.

To accept or present symbol names as opposed to symbol objects, use the **symbol-name** presentation type.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

symbol-name *Presentation Type*

Type for accepting or presenting a symbol name, that is, the print name of a symbol. (For accepting or presenting symbol objects, use the **symbol** presentation type.)

Examples:

```
(accept '((symbol-name)))
Enter a symbol name: T-M-S
"T-M-S"
((SYMBOL-NAME))

(present "T-M-S" '((symbol-name))) ==>T-M-S
#<DISPLAYED-PRESENTATION 444645436>
```

The **symbol-name** presentation type inherits its printer and type history from the **string** presentation type.

symbol-name is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sct:system (&key (*patchable-only* **nil**) (*true-systems-only* **nil**)) *Presentation Type*

Type for accepting or presenting systems.

:patchable-only

Data option restricting systems to those that are patchable; the default is **nil**.

:true-systems-only

Data option restricting systems to true systems, as opposed to either subsystems or systems that are undefined; the default is **nil**.

Examples:

```
(accept '((sct:system))) ==>
Enter a system or subsystem: Dynamic Windows
#<SUBSYSTEM DYNAMIC-WINDOWS 261254415>
((SCT:SYSTEM))

(accept '((sct:system :true-systems-only t))) ==>
Enter a system: Documentation Database
#<SYSTEM DOC 261374510>
((SCT:SYSTEM :TRUE-SYSTEMS-ONLY T))

(present (sct:find-system-named 'extended-help)
         '((sct:system))) ==>Extended Help
#<DISPLAYED-PRESENTATION 274651506>

(present (car sct:*all-systems*) '((sct:system))) ==>System
#<DISPLAYED-PRESENTATION 274641244>
```

The **sct:system** presentation type supports a type history.

sct:system is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

sct:system-version *Presentation Type*

Type for accepting or presenting a system version designator. Three kinds of designators are permitted:

- a positive, non-zero integer

- one of the special keywords **:released**, **:latest**, or **:newest**
- an arbitrary keyword

Examples:

```
(accept '((sct:system-version))) ==>
Enter a version designator: 2
2
((SCT:SYSTEM-VERSION))

(accept '((sct:system-version))) ==>
Enter a version designator: Released
:RELEASED
((SCT:SYSTEM-VERSION))

(accept '((sct:system-version))) ==>
Enter a version designator: arbitrary
:ARBITRARY
((SCT:SYSTEM-VERSION))

(present :newest '((sct:system-version))) ==>Newest
#<DISPLAYED-PRESENTATION 274677471>
```

The **sct:system-version** presentation type does not support a type history.

sct:system-version is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

t *Presentation Type*

Type that is supertype to all other presentation types.

t occupies a necessary spot (the top) in the type hierarchy, and is important for that reason. However, it has no parser and cannot be used with **accept**. Moreover, objects presented as **t** presentations are not mouse-sensitive in any input context.

One of the key uses for the **t** type is in mouse handlers, as the *from-presentation-type* or *to-presentation-type*. If the former, it means that the handler in question is potentially applicable to any type of presentation; if the latter, it means that the handler is potentially applicable in any input context. See the section "Advanced Mouse Handler Concepts", page 42.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

time:time-interval *Presentation Type*

Type for accepting or presenting intervals of time. Internally, time intervals are in seconds; externally, in seconds, minutes, hours, days, weeks, and years. **nil** is represented as "never".

Examples:

```
(accept '((time:time-interval))) ==>
Enter a time interval: 1 second
1
((TIME:TIME-INTERVAL))

(accept '((time:time-interval))) ==>
Enter a time interval [default 1 second]: 1 minute
60
((TIME:TIME-INTERVAL))

(accept '((time:time-interval))) ==>
Enter a time interval [default 1 minute]: 1 hour
3600
((TIME:TIME-INTERVAL))

(present 3661 '((time:time-interval))) ==>1 hour 1 minute 1 second
#<DISPLAYED-PRESENTATION 276047342>

(present nil '((time:time-interval))) ==>never
#<DISPLAYED-PRESENTATION 276047575>
```

Note that time intervals are specified with integers only.

The **time:time-interval** presentation type supports a type history.

time:time-interval is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

time:time-interval-60ths *Presentation Type*

Type for accepting or presenting intervals of time. Internally, time intervals are in 60ths of a second; externally, in seconds, minutes, hours, days, weeks, and years. **nil** is represented as "never".

Examples:

```
(accept '((time:time-interval-60ths))) ==>
Enter a time interval 60ths: 1 second
60
((TIME:TIME-INTERVAL-60THS))

(accept '((time:time-interval-60ths))) ==>
Enter a time interval 60ths [default 1 second]: 1 minute
3600
((TIME:TIME-INTERVAL-60THS))

(accept '((time:time-interval-60ths))) ==>
Enter a time interval 60ths [default 1 minute]: 1 hour
216000
((TIME:TIME-INTERVAL-60THS))

(present 3661 '((time:time-interval-60ths))) ==>1 minute 1 second
#<DISPLAYED-PRESENTATION 276061445>

(present 30 '((time:time-interval-60ths))) ==>0 seconds
#<DISPLAYED-PRESENTATION 276062366>

(present 31 '((time:time-interval-60ths))) ==>1 second
#<DISPLAYED-PRESENTATION 276062621>

(present nil '((time:time-interval-60ths))) ==>never
#<DISPLAYED-PRESENTATION 276061700>
```

Note that time intervals are specified with integers only; also, that they are rounded to the nearest second when presented.

The **time:time-interval-60ths** presentation type supports a type history.

time:time-interval-60ths is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

time:timezone &key *force-numeric-p* *Presentation Type*

Type for accepting or presenting timezones.

Timezones are represented externally either by commonly accepted

abbreviations, for example, "EST" (for Eastern Standard Time), or by a signed digit string, for example, "-0500". The sign of the digit string indicates the location of the timezone relative to Greenwich; positive means east, negative west.

Internally, timezones are represented by numbers in the form *n.0* or *n.5*. Note that the sign of the externally displayed digit string is opposite to that of the number used internally. The printed digit string "-0530", for example, corresponds to an internal representation of 5.5.

:force-numeric-p
: Presentation option specifying whether a timezone is presented only by a signed digit string. The default is **nil**; this causes the timezone's unique abbreviation, if there is one, to be printed. If a unique abbreviation is not available, the digit string is printed regardless of the value supplied for this option.

Examples:

```
(accept '((time:timezone))) ==>
Enter a defined timezone symbol or an hour offset from GMT
such as +0500 (east of GMT) or -0330 (west of GMT): EST
5
((TIME:TIMEZONE))

(accept '((time:timezone))) ==>
Enter a defined timezone symbol or an hour offset from GMT
such as +0500 (east of GMT) or -0330 (west of GMT): -0500
5
((TIME:TIMEZONE))

(present 5 '((time:timezone))) ==>EDT
#<DISPLAYED-PRESENTATION 274454265>

(present 5 '((time:timezone) :force-numeric-p t)) ==>-0400
#<DISPLAYED-PRESENTATION 274454520>
```

Note in the last two examples, created in July, that the displayed presentations reflect daylight savings time. At sites in timezones for which straightforward rules exist governing the change from standard to daylight-savings time and back again, the timezone utility automatically switches over to the appropriate abbreviation and digit string. For other timezones,

the switch must be made manually. In either case, **time:timezone** presentations display the current setting for daylight savings time. For more information: See the section "Specifying a Time Zone for Your Site" in *Site Operations.*

The **time:timezone** presentation type does not support a type history.

time:timezone is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

token-or-type (*special-tokens otherwise-type*) *Presentation Type*

Compound type for accepting or presenting a special token – for example "None", "Any", "All" – or an object of a specified type.

special-tokens
: Data argument specifying a list of tokens. The list is an alist: each item is a dotted pair of a print string and its object: ((*String-1* . *object-1*) (*string-2* . *object-2*) ... (*string-n object-n*))

otherwise-type
: Data argument specifying the presentation type to use for accepting or presenting objects other than listed tokens.

Examples:

```
(accept '((token-or-type (("either" . :either)
                          ("neither" . :neither)
                          ("both" . :both))
                         ((subset  :fixed-wing :rotary-wing))))
        :prompt "Enter \"fixed-wing\", \"rotary-wing\", \"either\",
         \"neither\", or \"both\"") ==>
Enter "fixed-wing", "rotary-wing", "either", "neither",
or "both": Fixed-Wing
(:FIXED-WING)
((SUBSET :FIXED-WING :ROTARY-WING))
```

```
(accept '((token-or-type (("either" . :either)
                          ("neither" . :neither)
                          ("both" . :both))
                         ((subset  :fixed-wing :rotary-wing))))
        :prompt "Enter \"fixed-wing\", \"rotary-wing\", \"either\",
         \"neither\", or \"both\"") ==>
Enter "fixed-wing", "rotary-wing", "either", "neither", or "both":
neither
:NEITHER
(#<DTP-LOCATIVE ...)

(present '(:fixed-wing) '((token-or-type (("either" . :either)
                                          ("neither" . :neither)
                                          ("both" . :both))
                                         ((subset  :fixed-wing
                                                   :rotary-wing))))) ==>
FIXED-WING
#<DISPLAYED-PRESENTATION 444762334>

(present :both '((token-or-type (("either" . :either)
                                 ("neither" . :neither)
                                 ("both" . :both))
                                ((subset  :fixed-wing :rotary-wing))))) ==>
both
#<DISPLAYED-PRESENTATION 444763221>
```

If the presentation type specified by *otherwise-type* supports a type history, the history is available for objects of that type.

token-or-type is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

type-or-string *(presentation-type)* *Presentation Type*

Compound type for accepting or presenting objects of a specified type or strings.

presentation-type

Data argument specifying the presentation type to use for accepting or presenting objects which are not strings.

Examples:

```
(accept '((type-or-string net:user)))
Enter a user: JWALKER
#<USER JWALKER 6434203>
SI:USER

(accept '((type-or-string net:user))
        :default (dw:presentation-type-default 'net:user)
Enter a user [default JWALKER]: JBIRD
"JBIRD"
STRING

(present 'JWALKER '((type-or-string net:user))) ==>JWALKER
#<DISPLAYED-PRESENTATION 445112577>

(present "JWALKER" '((type-or-string net:user))) ==>JWALKER
#<DISPLAYED-PRESENTATION 445105072>
```

Although the type specified by *presentation-type* might support a type history, accepting a **type-or-string** does not automatically display the default; you have to provide one to **accept** yourself. This is illustrated in the second **accept** form above.

Note in the **present** examples that the objects presented have the same printed representation. The first, however, is an **net:user** object, the second a string object. Each will only be mouse-sensitive in the appropriate input context.

type-or-string is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

time:universal-time &key *base-time past-p must-have timezone long-date brief* *Presentation Type*

Type for accepting or presenting universal times. (Universal time is measured in seconds elapsed since midnight, Jan 1, 1900, GMT.)

When accepting universal times, a large variety of input formats are possible. For more information and examples: See the section "Reading Dates and Times" in *Programming the User Interface, Volume B.*

The following keyword options, all presentation arguments, are available. The first three – **:base-time**, **:past-p**, and **:must-have** – affect the input of

universal times. The second three – **:timezone**, **:long-date**, and **:brief** – affect their output. The discussion of each includes examples.

:base-time
Presentation option specifying a base time from which defaults are taken for unspecified components when accepting a universal time.

The base time is specified as the number of seconds since midnight, January 1, 1900 (that is, 1/01/00 00:00:00). In the following example, the base time is midnight, January 1, 1986.

Example:

```
(accept '((time:universal-time) :base-time 2713928400
          :description "a date in 1986"))) ==>
Enter a date in 1986: 3/2 ==>3/02/86 00:00:00
2719112400
((TIME:UNIVERSAL-TIME) :BASE-TIME 2713928400)
```

:past-p Presentation option specifying whether partially specified times default to the nearest corresponding universal time in the past or future; the default is **nil**.

The following examples were created in 7/86.

Examples:

```
(accept '((time:universal-time))) ==>
Enter a universal time: 3/2 ==>3/02/87 00:00:00
2750648400
((TIME:UNIVERSAL-TIME))

(accept '((time:universal-time))) ==>
Enter a universal time
[default 3/02/87 00:00:00]: 8/2 ==>8/02/86 00:00:00
2732328000
((TIME:UNIVERSAL-TIME))

(accept '((time:universal-time) :past-p t)) ==>
Enter a universal time in the past
[default 8/02/86 00:00:00]: 3/2 ==>3/02/86 00:00:00
2719112400
((TIME:UNIVERSAL-TIME) :PAST-P T)
```

```
(accept '((time:universal-time) :past-p t)) ==>
Enter a universal time in the past
[default 3/02/86 00:00:00]: 8/2 ==>8/02/85 00:00:00
2700792000
((TIME:UNIVERSAL-TIME) :PAST-P T)
```

:must-have

Presentation option specifying that the year field or second field or both must be explicitly entered when accepting a universal time. The required fields are provided as a list.

Example:

```
(accept '((time:universal-time) :must-have (year))) ==>
Enter a universal time, year is required
[default 7/07/86 19:19:00]: 12/12 ==>
no year supplied
Type RUBOUT to correct your input.
Enter a universal time, year is required
[default 7/07/86 19:19:00]: 12/12/47 00:00:00
1512968400
((TIME:UNIVERSAL-TIME) :MUST-HAVE (YEAR))
```

:timezone

Presentation option specifying the timezone used when presenting universal times. **time:*timezone*** provides the default value.

Supply the value as a number (either *n* or *n.5*): 0 specifies Greenwich Mean Time; positive numbers timezones to the west of Greenwich; negative numbers timezones to the east. (For more on timezone representations: See the presentation type **time:timezone**, page 335.)

Examples:

```
(present 123456789 '((time:universal-time)
                     :timezone -5)) ==>12/1/03 02:33:09
#<DISPLAYED-PRESENTATION 274337427>

(present 123456789 '((time:universal-time)
                     :timezone 0)) ==>11/30/03 21:33:09
#<DISPLAYED-PRESENTATION 274340115>
```

```
(present 123456789 '((time:universal-time)
                     :timezone 5)) ==>11/30/03 16:33:09
#<DISPLAYED-PRESENTATION 274337662>

(present 123456789 '((time:universal-time)
                     :timezone 5.5)) ==>11/30/03 16:03:09
#<DISPLAYED-PRESENTATION 274345125>
```

:long-date
Presentation option specifying that the date be presented as in the following example when presenting universal times;

```
(present 123456789 '((time:universal-time)
                     :long-date t)) ==>
Monday the thirtieth of November, 1903; 4:33:09 pm
#<DISPLAYED-PRESENTATION 274353534>
```

:brief Presentation option specifying whether presented times should be printed briefly, that is, without the seconds field. Contrast the following two examples:

```
(present (time:get-universal-time)
  '((time:universal-time))) ==>7/07/86 14:55:35
#<DISPLAYED-PRESENTATION 274421523>

(present (time:get-universal-time)
  '((time:universal-time) :brief t)) ==>7/7/86 14:55
#<DISPLAYED-PRESENTATION 274421756>
```

time:universal-time is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

net:user *Presentation Type*

Type for accepting or presenting user objects.

Examples:

```
(present si:*user* '((si:user))) ==>REG
#<DISPLAYED-PRESENTATION 275633757>

(accept '((si:user))) ==>
Enter a user: REG
#<USER REG 13730364>
((SI:USER))
```

Through flavor inheritance, the **net:user** presentation type is subtype to the **net:object** type, from which it inherits a type history. The history inherited includes all accepted objects of the **net:object** type; that is, no pruning of the history occurs.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

fs:wildcard-pathname &key *(default-version :newest) default-type nil) (default-name nil) dont-merge-default (direction :read) (format :normal)* Presentation Type

Type for accepting or presenting pathnames that include wildcard characters.

This presentation type can be useful if you need to distinguish unequivocally between pathname presentations that include wildcard characters (asterisks) and other file pathname presentations. For example, if you can arrange for the availability to your users of some **fs:wildcard-pathname** presentations, then mouse handlers performing functions specifically on pathnames containing wildcards can be defined that do not have to test whether a given **pathname** presentation includes a wildcard character.

fs:wildcard-pathname is a subtype of the **pathname** presentation type, from which it inherits a printer, parser, and type history. It also takes the same keyword arguments, as follows:

:default-version
Presentation option specifying the default version number of an accepted file. The default value for this option is **:newest**, the newest file version.

:default-type
Presentation option specifying the default file type, for example, "lisp", "text", "data", and so on. The default value for this option is **nil**.

:default-name
Presentation option specifying the default file name. The default value for this option is **nil**.

:dont-merge-default
Presentation option specifying whether to prevent merging of a partially specified pathname entered by the user against the default pathname. The default value for this option is **nil**, meaning that merging occurs when appropriate; that is, parts of the pathname not entered by the user are supplied from the default.

Suppression of merging against the default and providing a different default (against which merging may or may not be enabled) are different issues. To deal with the latter, use the **:default** option to **accept**: See the function "**accept**", page 167. An example follows:

```
(accept '((pathname) :default-type nil)
        :default (send (fs:default-pathname)
                       :new-pathname :type nil
                       :version :newest))
```

:direction
Presentation option specifying either **:read** (the default) or **:write**. The value supplied is passed through to **fs:complete-pathname** and affects completion behavior. (See the function **fs:complete-pathname** in *Reference Guide to Streams, Files, and I/O.*)

Use the default (**:read**) if the user is likely to enter the pathname of an already existing file when prompted by **accept**, **:write** otherwise.

:format Presentation option specifying the output format of the pathname. There are four choices:

:normal For example, `S:>mb>dw-pgms>fancy-windows.lisp`. This is the default format.

:directory
For example, `>mb>dw-pgms>`. The host, file name, and file type are not displayed.

:dired For example, `fancy-windows.lisp`. Only the file name and type are displayed.

:editor For example, `fancy-windows.lisp >mb>dw-pgms S`. The display format is that used by Zmacs.

For examples illustrating the use of these keywords in pathname presentations: See the presentation type **pathname**, page 320.

fs:wildcard-pathname is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

tv:window *Presentation Type*

Type for accepting or presenting window objects.

Examples:

```
(accept '((tv:window))) ==>
Enter a window [default Editor Typeout Window 3]: HELP ==>
You are being asked to enter a window.

These are the possible windows:
  Background Dynamic Lisp Interactor 1  Main ... (2)
  Command ... (2)                       Mode ... (2)
  Converse                              Peek ... (2)
  Converse Frame 1                      Typein ... (4)
  Dex ... (7)                           Who ... (8)
  Dynamic ... (2)                       Zmacs ... (3)
  Editor Typeout Window 3               Zmail ... (4)
  Fsmaint ... (2)                       Zwei ... (7)

Enter a window [default Editor Typeout Window 3]: Converse Frame 1
#<CONVERSE-FRAME Converse Frame 1 1107255 deexposed>
((TV:WINDOW))

(present (tv:make-window 'dw:dynamic-window)
         '((tv:window))) ==>Dynamic Window 1
#<DISPLAYED-PRESENTATION 274625374>
```

The **tv:window** presentation type supports a type history.

tv:window is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Overview of Presentation Substrate Facilities", page 69.

PART VII.

Dictionary of Presentation Substrate Facilities

20. Dictionary Notes

This dictionary includes reference documentation for the presentation substrate facilities, excluding predefined presentation types. These facilities are:

Table of Presentation Substrate Facilities

Presentation-Type Definition Facilities
- **define-presentation-type**
- **dw:read-char-for-accept**
- **dw:peek-char-for-accept**
- **dw:unread-char-for-accept**
- **dw:compare-char-for-accept**
- **dw:read-standard-token**
- **dw:with-accept-activation-chars**
- **dw:with-accept-blip-chars**
- **dw:with-accept-help**
- **dw:with-accept-help-if**
- **dw:completing-from-suggestions**
- **dw:suggest**
- **dw:complete-input**
- **dw:complete-from-sequence**

Presentation Input Context Facilities
- **dw:with-presentation-input-context**
- **dw:clear-presentation-input-context**
- **dw:presentation-input-context-option**
- **dw:with-presentation-input-editor-context**
- **dw:*presentation-input-context***

Presentation Input Blip Facilities
- **dw:echo-presentation-blip**
- **dw:presentation-blip-object**
- **dw:presentation-blip-options**
- **dw:presentation-blip-presentation-type**
- **dw::presentation-blip-mouse-char**
- **dw:presentation-blip-typep**
- **dw:presentation-blip-p**
- **dw:presentation-blip-case**
- **dw:presentation-blip-ecase**

Other Presentation Facilities
dw:presentation-type-p
dw:presentation-subtypep
dw:presentation-object
dw:presentation-type
dw:presentation-equal
dw:describe-presentation-type
dw:check-presentation-type-argument
dw:with-presentation-type-arguments
dw:with-type-decoded
dw:presentation-type-name
dw:presentation-type-default
dw:displayed-presentation-set-highlighting
dw:displayed-presentation-clear-highlighting

In the dictionary, the facilities are arranged in alphabetical order (package prefixes excluded).

For conceptual documentation: See the section "Overview of Presentation Substrate Facilities", page 69.

For documentation of predefined presentation types: See the section "Dictionary of Predefined Presentation Types", page 281.

21. The Facilities

dw:check-presentation-type-argument *type-arg* &key (*evaluated* **t**) *Function*
(*function* **compiler:default-warning-function**)
(*definition-type*
compiler:default-warning-definition-type)

Checks an argument that is expected to be a presentation type for validity.

type-arg A form evaluating to a presentation type.

:evaluated
Boolean option specifying whether *type-arg* is expected to be quoted; the default is **t**.

:function
Specifies a symbol naming the function for which the compiler warning is issued. This name is displayed in the warning instead of the name of the function in which the error occurred; the latter behavior is the default.

:definition-type
Specifies the definition type (`'defun`, `'defvar`, etc.) of the Lisp object that caused the compiler warning. The name for objects of this type ("Function", "Variable", etc.) is displayed in the warning instead of the name for the type of object in which the error occurred; the latter behavior is the default.

This function should be used in macros that take presentation types as arguments and in style-checkers for functions that take presentation types.

Here is an example of the use of **dw:check-presentation-type-argument** in a macro:

```
(defmacro with-value ((variable-name presentation-type) &body body)
  (dw:check-presentation-type-argument presentation-type :evaluated nil)
  `(let ((,variable-name (accept ',presentation-type)))
     ,@body))
```

If you try to compile the following function, which contains an invalid specification of the **integer** presentation type inside an invocation of with-value, you get a compiler error diagnosing the problem:

```
(defun check-type-test ()
  (with-value (x ((integer 3 5 extra-argument)))
    (format t "~&Value is ~S" x)))
```

The **:evaluated** keyword is used to control whether **dw:check-presentation-type-argument** expects the presentation type to be quoted or not. In the macro example above, the presentation type is inserted unquoted into the invocation of the `with-value` macro. If you wanted `with-value` to evaluate its presentation-type argument (for instance, so that a variable that was bound to a presentation type could be used), then you would supply `:evaluated t` (the default). The rewritten example follows:

```
(defmacro with-value ((variable-name presentation-type) &body body)
  (dw:check-presentation-type-argument presentation-type :evaluated t)
  `(let ((,variable-name (accept ,presentation-type)))
     ,@body))

(defun check-type-test ()
  (with-value (x '((integer 3 5 extra-argument)))
    (format t "~&Value is ~S" x)))
```

(In both of the above examples, multiple error messages result because **accept** itself uses **dw:check-presentation-type-argument** to validate its arguments.)

For an overview of **dw:check-presentation-type-argument** and related facilities: See the section "Overview of Other Presentation Facilities", page 79.

dw:clear-presentation-input-context *Function*

Clears the current input context. This is useful for eliminating the input context established by a function's callers in order to establish a new input context that doesn't inherit from the callers.

For an overview of **dw:clear-presentation-input-context** and related facilities: See the section "Overview of Presentation Input Context Facilities", page 78.

dw:compare-char-for-accept *char-from-accept comparandum* *Function*

Compares an input character with a specified character. Use this function instead of **char-equal** when manipulating characters read with **dw:read-char-for-accept.**

char-from-accept
: The input character (returned by **dw:read-char-for-accept**).

comparandum
The comparison character. This may be any standard character.

For an overview of **dw:compare-char-for-accept** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:complete-from-sequence *sequence stream* &key *type* (*name-key* #'**string**) (*value-key* #'**identity**) (*delimiters* **dw::*standard-completion-delimiters***) (*allow-any-input* **nil**) (*enable-forced-return* **nil**) (*initially-display-possibilities* **nil**) (*partial-completers* **nil**) (*complete-activates* **nil**) (*compress-choices* **20**) (*compression-delimiter*) *Function*

Provides input completion from a sequence of possible completions for input to **accept**. Returned values are the object associated with the completion string; **t** or **nil** depending on whether or not the completion was the only one possible; and the completion string.

sequence The sequence of possible completions.

stream The input stream.

:type Specifies the presentation type to use when displaying help information for possible completions. This makes the displayed possibilities mouse-sensitive.

If the completion utility is being called from the parser of a presentation type, that type should be supplied as the value of this option.

:name-key
Specifies the function called on each element in the sequence for extracting the completion string. The default function is **string**. Another useful function is **string-capitalize-words**.

:value-key
Specifies the function called on each element in the sequence for extracting the value to be associated with the element's completion string. The default function is **identity**, which extracts the element itself.

:delimiters
Specifies a list of characters used by the standard completion mechanism to tokenize completion strings. The default value

is the binding of **dw::*standard-completion-delimiters***; this variable is preset to "- " (hyphen and space).

:allow-any-input
Boolean option specifying whether the completer accepts keyboard input from the user that does not match any of the possible completion strings; the default is **nil**.

Most parsers should specify `:allow-any-input nil`. In a call to **accept** for which you want to allow input that does not match any of the completions, use the **type-or-string** presentation type.

Note that the completion facilities always signal the error **dw:input-not-of-required-type** when a user types `RETURN` at blank input. This is intended to allow **accept** to fill in the default in the blank case. It means, however, that a caller of a completion facility that passes `:allow-any-input t` must also **condition-bind** for **dw:input-not-of-required-type**, if you want a null line to be treated the same as any other input.

:enable-forced-return
Boolean option specifying whether the user can force a response that is not a member of the completion set; the default is **nil**.

If this option is **t**, the user can terminate input with `c-RETURN`, causing the completion utility to return to the caller whatever input the user supplied. This is useful in situations where you expect the user to specify a member of a set of possibilities, but want to provide a way for supplying a new name to be added to the set. (The Zmacs Select Buffer (`c-X B`) command uses this feature to allow the user to create new buffers.)

:initially-display-possibilities
Boolean option specifying whether to display the entire set of completion possibilities before prompting for input; the default is **nil**. If **t**, the behavior is as if the user typed `Help` before any other input.

Most parsers should supply to this option the same value that was supplied to them by **accept**. **accept**, in turn, has an **:initially-display-possibilities** option controlled by its caller: See the function "**accept**", page 167.

:partial-completers

Specifies a list of characters that trigger partial completion when entered by the user.

Partial completion restricts completion to only one token of the completion set possibilities, even if enough characters have been supplied to uniquely identify one of the members of the completion set. For example, the Command Processor uses #**space** as a partial completer.

The syntax of a token is defined by the **:delimiters** option: See the function "**dw:complete-input**", page 356.

:complete-activates

Boolean option specifying whether the COMPLETE key causes activation, that is, whether the completion utility returns if a unique completion was found. The default is **nil**.

This option is used to control completion behavior in a multi-field input context, such as in the command processor. Normally, the END key performs completion and then returns if the resulting completion is unique.

:compress-choices

Specifies whether to compress the display of completion possibilities that have a common left token (as defined by the **:compression-delimiter** option: For more information: See the function "**dw:complete-input**", page 356. Three values are possible:

An integer
: When the possibilities exceed this number, the display is compressed. The default value is 20.

:always
: Whenever more than one possibility exists, the display is compressed.

:never
: The display is never compressed, regardless of the number of possibilities.

Compressed displays have the form "*token ... (n)*", where *token* is the shared left token and *n* is the number of possible completions.

To see an example of choice compression, press HELP to the command processor prompt in a Dynamic Lisp Listener. You get the following display (abbreviated for this example):

```
You are being asked to enter a command or form.
Use the Help :Format Detailed command to see a full
list of command names.

These are the possible command names:
  Add Paging File
  Append
  Clean File
  Clear ... (3)
  Close File
  Compare Directories
  Compile ... (2)
  Copy ... (5)
  Create ... (4)
  Debug Process
```

"Add Paging File", "Append", and "Clean File" are full command names. "Clear" is a left token shared by three commands, Clear All Breakpoints, Clear Breakpoint, and Clear Output History. These three completion choices have been compressed to "Clear ... (3)". The user can expand this and other compressed choices by clicking on them with the mouse.

:compression-delimiter
Specifies a list of characters used for delimiting the shared left tokens in a display of completion possibilities. The default value is '(#**space**).

For more information: See the function "**dw:complete-input**", page 356.

For an overview of **dw:complete-from-sequence** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:complete-input *stream function* &key (*allow-any-input* **nil**) *Function*
enable-forced-return partial-completers (*type* **nil**)
parser (*compress-choices* **20**)
(*compression-delimiter*) (*help-offers-possibilities* **t**) (*initially-display-possibilities* **nil**)
(*complete-activates* **nil**) (*documenter* **nil**)
(*document* **(not (null dw::documenter))**)

Provides input completion for input to **accept**. Returned values are the object associated with the completion string; **t** or **nil** depending on whether or not the completion was the only one possible; and the completion string.

stream The input stream.

function The completion function. The function receives two arguments, the input supplied by the user and a keyword specifying an operation.

Operations are divided into two categories, completion operations and possibility operations. The former attempt to complete and return the completion; the latter return either a list of possible completions or the number of possible completions. Available keywords for each type are described below:

Completion Operations

:complete
Complete and return as much as possible based on the input so far.

:complete-limited
Complete and return the current input "chunk" only, even if the input uniquely identifies a full completion possibility. The meaning of "chunk" depends on the type of input. For example, in the case of command processor commands, a chunk is a word in the command name.

:complete-maximal
Complete and return as much as possible based on the input so far, even if that means adding empty tokens between delimiters.

Regardless of the completion operation, the completion function must return the following five values:

1. The string resulting from completing the input string.

2. A boolean indicating if the completion is unique, that is, if it identifies one and only one of the completion possibilities.

3. The object associated with the completion if it is unique.

4. The index in the completion string of the first point of ambiguity if the string is not unique, that

is, the leftmost place in the string where a difference arises between two or more completion possibilities. The completer generally tries to position the input cursor at that point so that the user can resolve the ambiguity.

5. The number of possible input completions; this may be `0`.

Possibility Operations

:possibilities
Return a list of completion possibilities that begin with the input string.

:apropos-possibilities
Return a list of the completion possibilities that contain the input string anywhere in the completion string.

The function may split the input into tokens and search for possibilities that contain all the tokens somewhere in the completion string. In this case, it should return as a second value the list of tokens extracted from the original input string.

:possibilities-quick-length
Returns the number of completion possibilities that begin with the input string.

:apropos-possibilities-quick-length
Return the number completion possibilities that contain the input string anywhere in the completion string.

The completion function can return **nil** to indicate that it does not support the "quick-length" operations. In this case, the completer utility asks for a full **:possibilities** or **:apropos-possibilities** list and counts the number of elements to return.

:allow-any-input
Boolean option specifying whether the completer accepts keyboard input from the user that does not match any of the possible completion strings; the default is **nil**.

Most parsers should specify `:allow-any-input nil`. In a call to **accept** for which you want to allow input that does not match any of the completions, use the **type-or-string** presentation type.

Note that the completion facilities always signal the error **dw:input-not-of-required-type** when a user types RETURN at blank input. This is intended to allow **accept** to fill in the default in the blank case. It means, however, that a caller of a completion facility that passes `:allow-any-input t` must also **condition-bind** for **dw:input-not-of-required-type**, if you want a null line to be treated the same as any other input.

:enable-forced-return

Boolean option specifying whether the user can force a response that is not a member of the completion set; the default is **nil**.

If this option is **t**, the user can terminate input with c-RETURN, causing the completion utility to return to the caller whatever input the user supplied. This is useful in situations where you expect the user to specify a member of a set of possibilities, but want to provide a way for supplying a new name to be added to the set. (The Zmacs Select Buffer (c-X B) command uses this feature to allow the user to create new buffers.)

:partial-completers

Specifies a list of characters that trigger partial completion when entered by the user.

Partial completion restricts completion to only one token of the completion set possibilities, even if enough characters have been supplied to uniquely identify one of the members of the completion set. For example, the Command Processor uses #**space** as a partial completer.

The syntax of a token is defined by the **:delimiters** option: See the macro "**dw:completing-from-suggestions**", page 362.

:type Specifies the presentation type to use when displaying help information for possible completions. This makes the displayed possibilities mouse-sensitive.

If the completion utility is being called from the parser of a presentation type, that type should be supplied as the value of this option.

:parser Specifies the function called to translate input strings into objects of the desired type. The function is called with one argument, the string entered by the user.

This option is typically used when the set of possible completions is not known in advance, and can therefore not be enumerated. If they can be enumerated, use **dw:complete-from-sequence** or **dw:completing-from-suggestions** instead.

The parser function is called on each possible completion string when a list of possibilities is generated, and on the user-supplied input when the completion utility is about to return a value.

:compress-choices

Specifies whether to compress the display of completion possibilities that have a common left token (as defined by the **:compression-delimiter** option: See the function "**dw:complete-input**", page 356. Three values are possible:

An integer

When the possibilities exceed this number, the display is compressed. The default value is 20.

:always Whenever more than one possibility exists, the display is compressed.

:never The display is never compressed, regardless of the number of possibilities.

Compressed displays have the form "*token* ... (*n*)", where *token* is the shared left token and *n* is the number of possible completions.

To see an example of choice compression, press HELP to the command processor prompt in a Dynamic Lisp Listener. You get the following display (abbreviated for this example):

```
You are being asked to enter a command or form.
Use the Help :Format Detailed command to see a full
list of command names.
```

```
These are the possible command names:
  Add Paging File
  Append
  Clean File
  Clear ... (3)
  Close File
  Compare Directories
  Compile ... (2)
  Copy ... (5)
  Create ... (4)
  Debug Process
```

"Add Paging File", "Append", and "Clean File" are full command names. "Clear" is a left token shared by three commands, Clear All Breakpoints, Clear Breakpoint, and Clear Output History. These three completion choices have been compressed to "Clear ... (3)". The user can expand this and other compressed choices by clicking on them with the mouse.

:compression-delimiter

Specifies a list of characters used for delimiting the shared left tokens in a display of completion possibilities. The default value is `'(#\space)`.

For more information: See the function "**dw:complete-input**", page 356.

:help-offers-possibilities

Boolean option specifying whether the full list of completion possibilities is displayed when the user presses the HELP key; the default is **t**.

:initially-display-possibilities

Boolean option specifying whether to display the entire set of completion possibilities before prompting for input; the default is **nil**. If **t**, the behavior is as if the user typed `Help` before any other input.

Most parsers should supply to this option the same value that was supplied to them by **accept**. **accept**, in turn, has an **:initially-display-possibilities** option controlled by its caller: See the function "**accept**", page 167.

:complete-activates
Boolean option specifying whether the COMPLETE key causes activation, that is, whether the completion utility returns if a unique completion was found. The default is **nil**.

This option is used to control completion behavior in a multi-field input context, such as in the command processor. Normally, the END key performs completion and then returns if the resulting completion is unique.

:documenter
Specifies a function called to generate documentation for the elements of a possibilities display. The function receives two arguments, a completion possibility and the output stream for displaying the documentation.

:document
Specifies how each possibility displayed as a result of a HELP request is documented. Three values are possible:

t Display the documentation. If a documentation function is specified by the **:documenter** option, it is called on each possibility; otherwise, the Common Lisp function **documentation** is called.

nil Do not display any documentation.

:if-unique
Display documentation only if there is a unique completion of the input supplied by the user.

The default for this option is **t** if a **:documenter** function is supplied, **nil** otherwise. (See the function "**dw:complete-input**", page 356.)

For an overview of **dw:complete-input** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:completing-from-suggestions (*stream* &key (*allow-any-input* **t**) (*delimiters* **dw::*standard-completion-delimiters***) (*enable-forced-return* **nil**) (*partial-completers* **nil**) (*type* **nil**) (*parser* **nil**) (*complete-activates* **nil**) (*compress-choices* **20**) (*compression-delimiter* **nil**) (*initially-display-possibilities* **nil**)) &body *body* *Macro*

Binds local environment to provide input completion for input to **accept**.

Returned values are the object associated with the completion string; **t** or **nil** depending on whether or not the completion was the only one possible; and the completion string.

stream The input stream.

:allow-any-input
Boolean option specifying whether the completer accepts keyboard input from the user that does not match any of the possible completion strings; the default is **nil**.

Most parsers should specify `:allow-any-input nil`. In a call to **accept** for which you want to allow input that does not match any of the completions, use the **type-or-string** presentation type.

Note that the completion facilities always signal the error **dw:input-not-of-required-type** when a user types RETURN at blank input. This is intended to allow **accept** to fill in the default in the blank case. It means, however, that a caller of a completion facility that passes `:allow-any-input t` must also **condition-bind** for **dw:input-not-of-required-type**, if you want a null line to be treated the same as any other input.

:delimiters
Specifies a list of characters used by the standard completion mechanism to tokenize completion strings. The default value is the binding of **dw::*standard-completion-delimiters***; this variable is preset to "- " (hyphen and space).

:enable-forced-return
Boolean option specifying whether the user can force a response that is not a member of the completion set; the default is **nil**.

If this option is **t**, the user can terminate input with c-RETURN, causing the completion utility to return to the caller whatever input the user supplied. This is useful in situations where you expect the user to specify a member of a set of possibilities, but want to provide a way for supplying a new name to be added to the set. (The Zmacs Select Buffer (c-X B) command uses this feature to allow the user to create new buffers.)

:partial-completers

Specifies a list of characters that trigger partial completion when entered by the user.

Partial completion restricts completion to only one token of the completion set possibilities, even if enough characters have been supplied to uniquely identify one of the members of the completion set. For example, the Command Processor uses **#\space** as a partial completer.

The syntax of a token is defined by the **:delimiters** option: See the function "**dw:complete-from-sequence**", page 353.

:type Specifies the presentation type to use when displaying help information for possible completions. This makes the displayed possibilities mouse-sensitive.

If the completion utility is being called from the parser of a presentation type, that type should be supplied as the value of this option.

:parser Specifies the function called to translate input strings into objects of the desired type. The function is called with one argument, the string entered by the user.

This option is typically used when the set of possible completions is not known in advance, and can therefore not be enumerated. If they can be enumerated, use **dw:complete-from-sequence** or **dw:completing-from-suggestions** instead.

The parser function is called on each possible completion string when a list of possibilities is generated, and on the user-supplied input when the completion utility is about to return a value.

:complete-activates

Boolean option specifying whether the COMPLETE key causes activation, that is, whether the completion utility returns if a unique completion was found. The default is **nil**.

This option is used to control completion behavior in a multi-field input context, such as in the command processor. Normally, the END key performs completion and then returns if the resulting completion is unique.

:compress-choices

Specifies whether to compress the display of completion

possibilities that have a common left token (as defined by the **:compression-delimiter** option: See the function **dw:complete-input**, page 356. Three values are possible:

An integer
: When the possibilities exceed this number, the display is compressed. The default value is 20.

:always
: Whenever more than one possibility exists, the display is compressed.

:never
: The display is never compressed, regardless of the number of possibilities.

Compressed displays have the form "*token* ... (*n*)", where *token* is the shared left token and *n* is the number of possible completions.

To see an example of choice compression, press HELP to the command processor prompt in a Dynamic Lisp Listener. You get the following display (abbreviated for this example):

```
You are being asked to enter a command or form.
Use the Help :Format Detailed command to see a full
list of command names.

These are the possible command names:
  Add Paging File
  Append
  Clean File
  Clear ... (3)
  Close File
  Compare Directories
  Compile ... (2)
  Copy ... (5)
  Create ... (4)
  Debug Process
```

"Add Paging File", "Append", and "Clean File" are full command names. "Clear" is a left token shared by three commands, Clear All Breakpoints, Clear Breakpoint, and Clear Output History. These three completion choices have been compressed to "Clear ... (3)". The user can expand this and other compressed choices by clicking on them with the mouse.

:compression-delimiter
Specifies a list of characters used for delimiting the shared left tokens in a display of completion possibilities. The default value is '(#\space).

For more information: See the function "**dw:complete-input**", page 356.

:initially-display-possibilities
Boolean option specifying whether to display the entire set of completion possibilities before prompting for input; the default is **nil**. If **t**, the behavior is as if the user typed `Help` before any other input.

Most parsers should supply to this option the same value that was supplied to them by **accept**. **accept**, in turn, has an **:initially-display-possibilities** option controlled by its caller: See the function "**accept**", page 167.

For an overview of **dw:completing-from-suggestions** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

define-presentation-type *type-name* (*data-arglist* . *pr-arglist*) &key *parser printer viewspec-choices description describer no-deftype* (*history* **nil**) *expander abbreviation-for choose-displayer multiple-accept-displayer menu-displayer presentation-type-arguments presentation-subtypep do-compiler-warnings typep* (*data-arguments-are-disjoint* **t**) *Macro*

Defines a new presentation type.

type-name
Specifies the name for the new type.

data-arglist
Specifies arguments describing an object of this type; *data-arglist* may be any permissible **defun**-style argument list.

Data arguments are used to determine the sensitivity of an object in any given input context established by **accept**, and the applicability of defined mouse handlers. They also participate in determining the subtype and supertype relationships of the type. (For more information and

examples: See the section "Overview of Predefined Presentation Types", page 71.)

pr-arglist Specifies keyword arguments that affect the accepting or presenting of an object of this type; such keywords are handled in the body of the presentation type's **:parser**, or **:printer** respectively (see below).

Unlike data arguments, presentation arguments are not relevant to determining mouse sensitivity or subtype and supertype relationships. (For more information and examples: See the section "Overview of Predefined Presentation Types", page 71.)

(Certain predefined keywords are meta-presentation arguments. They can be used when calling any type and are understood directly by **accept** or **present**, rather than used by the type's parser or printer. At present, such arguments are limited to **:description**. For more information: See the section "Predefined Presentation Types", page 71.

:parser Specifies a function for parsing a presentation object of the defined type. This is what **accept** calls for inputting objects entered as a series of characters.

Arguments passed to the parser function include the input stream and a set of optional keywords. These arguments must be declared in the argument list for the parser function. The parser keyword options are:

:data-type-args
Specifies the arguments to the Common Lisp (CL) type specification. For example, for CL type `(integer 0 5)`, the value of this option would be `(0 5)`.

This option is useful only for presentation types that are based on CL types with data arguments.

:presentation-args
Specifies the presentation arguments to the presentation type. This option is not usually needed, as the presentation arguments are available lexically in the bodies of this and other presentation-type functions (that is, the **:printer** and **:describer** functions).

:original-type
Specifies the presentation type originally supplied in the call to **accept**. The **:parser** function that gets invoked is found via the presentation-type inheritance mechanism.

Note that when **accept** is called recursively, as part of an expansion, via the **and** or **or** presentation type, or in similar situations, the original type is not the "top-level" presentation type. Rather, it is the head of the last chain of inheritance tracking.

:type Not used by the parser.

:default Value is supplied by **accept**. You only need to use this for merging; actual defaulting is handled at a higher level.

:initially-display-possibilities
Boolean option specifying whether to display the objects that could be used as input in the current context; the default is **nil**. If **t**, the possibilities are presented before the input prompt appears.

Additionally, there are two other sources of keyword arguments; 1) keywords declared in the type's *data-arglist* and *pr-arglist*; and 2) keywords to the **accept** function that uses the parser.

Keywords originating in the *data-arglist* or *pr-arglist* are available lexically in the body of the parser function, and do not have to be explicitly declared in the argument list to the parser function.

Keywords originating in an **accept** function call, on the other hand, do need to be explicitly declared in the parser's argument list.

The syntax for the parser function is as follows:

:parser ((stream &key *<parser keywords>* **<accept** *keywords>*) *body*)

:printer Specifies a function for printing a presentation object of the defined type. This is what **present** calls for outputting objects.

Arguments passed to the printer function include the object,

the output stream, and a set of optional keywords. These arguments must be declared in the argument list for the printer function. The printer keyword options are:

:data-type-args
Specifies the arguments to the Common Lisp (CL) type specification. For example, for CL type `(integer 0 5)`, the value of this option would be `(0 5)`.

This option is useful only for presentation types that are based on CL types with data arguments.

:presentation-args
Specifies the presentation arguments to the presentation type. This option is not usually needed, as the presentation arguments are available lexically in the bodies of this and other presentation-type functions (that is, the **:parser** and **:describer** functions).

:original-type
Specifies the presentation type originally supplied in the call to **present**. The **:printer** function that gets invoked is found via the presentation-type inheritance mechanism.

:type Not used by the printer.

:acceptably
Boolean option specifying whether to print the presentation in such way that it can be parsed by **accept** as the specified presentation type.

Additionally, there are two other sources of keyword arguments; 1) keywords declared in the type's *data-arglist* and *pr-arglist*; and 2) keywords to the **present** function that uses the printer.

Keywords originating in the *data-arglist* or *pr-arglist* are available lexically in the body of the printer function, and do not have to be explicitly declared in the argument list to the printer function.

Keywords originating in an **present** function call, on the other hand, do need to be explicitly declared in the printer's argument list.

The syntax for the printer function is as follows:

:printer ((object stream &key *<parser keywords>*
<present *keywords>*) *body*)

:viewspec-choices

Specifies form that returns a list of locatives, presentation types, and prompts to slots in the presentation type. This provides the ability to do in-place modification of presentation printing.

Example:

```
(defflavor employee ((first-name)
                     (last-name)
                     (status))
          ()
  :readable-instance-variables
  :writable-instance-variables
  :initable-instance-variables)
```

```
(define-presentation-type employee (()
                ;; keywords for different printed representations
                &key (format :last-name-first)
                     (include-status nil))
   :no-deftype t
   :printer ((employee stream)
             (ecase format
               (:last-name-first
                 (format stream "~A, ~A"
                         (employee-last-name employee)
                         (employee-first-name employee)))
               (:first-name-first
                 (format stream "~A ~A"
                         (employee-first-name employee)
                         (employee-last-name employee)))
               (:last-name-only
                 (write-string
                         (employee-last-name employee) stream)))
             (when include-status
               (format stream " (~(~A~))"
                         (employee-status employee))))
   :viewspec-choices ((&key type)
                ;; a necessary internal function
                (dw::presentation-type-keyword-options-into-cvv
                  type
                    ;; Choice 1:  keyword, pres type (member),
                    ;; selected choice (optional), and prompt
                  '((:format ((member :last-name-first
                                      :first-name-first
                                      :last-name-only))
                     :last-name-first "Format of name")
                    ;; Choice 2:  keyword, pres type (boolean),
                    ;; selected choice (optional), and prompt
                    (:include-status boolean nil "Include status")))))

(present (make-instance 'employee :last-name "Jones"
                        :first-name "Fred" :status :retired))
```

Compile the two definitions; then evaluate the **present** function. You can either click s-sh-Middle on the presentation to invoke the Edit Viewspecs mouse handler or click right on the presentation to get a menu of options, one of which is "Edit viewspecs". Clicking s-sh-Middle or selecting the

"Edit viewspecs" option brings up a **dw:accept-variable-values** menu. Using this, you can specify how the presentation is displayed.

With the **:viewspec-choices** option, you give your users the ability to modify at runtime all displayed presentations of the defined type. To provide same capability with respect to arbitrary program output, you can use **dw:with-replayable-output**: See the macro **dw:with-replayable-output**, page 274.

:description

Specifies a string describing the presentation type, for example, "an integer". This string is used in the prompt displayed by **accept** when inputting an object of this type.

This option and the **:describer** option are mutually exclusive. If neither option is supplied, a description is created based on inheritance from a Common Lisp type; if that is not possible, then the description defaults to the string "anything".

Do not confuse this option with the **:description** meta-presentation argument: See the section "Predefined Presentation Types", page 71.

:describer

Specifies a function for returning a string to be used as the description of the presentation type. This string is used in the prompt displayed by **accept** when inputting an object of this type. The describer function is generally used only for complex presentation types, such as compound and aggregate types.

Arguments passed to the describer function include the input stream and a set of optional keywords. The describer function keywords are:

:data-type-args

Specifies the arguments to the Common Lisp (CL) type specification. For example, for CL type `(integer 0 5)`, the value of this option would be `(0 5)`.

This option is useful only for presentation types that are based on CL types with data arguments.

:presentation-args

Specifies the presentation arguments to the presentation type. This option is not usually needed,

as the presentation arguments are available lexically in the bodies of this and other presentation-type functions (that is, the **:parser** and **:printer** functions).

:type Specifies the presentation type from which the describer function is inherited.

:plural-count
Boolean option specifying whether the type description is pluralized.

The syntax for the describer function is:

:describer ((stream &key *<describer keywords>*) *body*)

This option and the **:description** option are mutually exclusive. If neither option is supplied, a description is created based on inheritance from a Common Lisp type; if that is not possible, then the description defaults to the string "anything".

:no-deftype
Boolean option specifying whether this definition only defines a presentation type and not also a new data type. The default (**nil**) results in the generation of a **deftype**.

`:no-deftype t` must be supplied if a **deftype** is provided elsewhere for the symbol used as the *type-name* argument in the presentation type definition. This also applies to presentation types being defined for flavors and structures previously defined by **defflavor** and **defstruct**, respectively. For more information: See the section "User-Defined Data Types as Presentation Types", page 82.

:history Boolean option specifying whether a separate history is created for this presentation type. The default is **nil**, meaning that the history will be found via inheritance.

:expander
Specifies a form that is invoked to generate the "expansion" of the presentation type, for example, `((or` *pres-type1 pres-type2*`))`. Expansions allow for presentation types to inherit presentation functions (that is, parsers, printers, describers) from other presentation types. The presentation arguments are available lexically.

If you do not specify an **:expander**, then you must either specify the **:abbreviation-for** option or supply a parser and printer. If you do specify an expander, you can still supply the presentation type with its own parser or printer, and just inherit the function not supplied; however, you may not specify the **:abbreviation-for** option.

:abbreviation-for

Specifies the form for which this presentation type serves as an abbreviation. The form defines a new presentation type by combining or in other ways qualifying existing presentation types, for example, `((and` *pres-type* `(satisfies` *a-predicate*`)))`.

:choose-displayer

Specifies a form that does output showing the choice or choices that can be made for a presentation of this type in a menu or multiple-accept context. This output is in place of the default value normally used, and is useful in cases when you want a sequence or enumeration of choices displayed.

Use the internal function **dw::accept-values-choose-from-sequence** to write this form. The following example is extracted from the definition for the **alist-member** presentation type. The full definition is included in the file sys:dynamic-windows;standard-presentation-types.lisp.

Example:

```
(define-presentation-type alist-member ((&key alist)
        &key (convert-spaces-to-dashes nil))
   :choose-displayer ((stream object query-identifier
                       &key original-type)
                      (accept-values-choose-from-sequence
                       stream alist object query-identifier
                       :type original-type
                       :key #'tv:menu-execute-no-side-effects))
   ...)
```

:multiple-accept-displayer

Specifies a form that does output showing the choice or choices that can be made for a presentation of this type in a multiple-accept context. This output is in place of the default value normally used, and is useful in cases when you want a sequence or enumeration of choices displayed.

Use the internal function **dw::accept-values-choose-from-sequence** to write this form. An example is shown under the **:choose-displayer** keyword: See the function "**define-presentation-type**", page 366.

:menu-displayer

Specifies a form that does output showing the choice or choices that can be made for a presentation of this type in a menu context. This output is in place of the default value normally used, and is useful in cases when you want a sequence or enumeration of choices displayed.

Use the internal function **dw::accept-values-choose-from-sequence** to write this form. An example is shown under the **:choose-displayer** keyword: See the macro "**define-presentation-type**", page 366.

:presentation-type-arguments

Specifies a list of type arguments appearing in the presentation type's *data-arglist* which are themselves presentation types.

define-presentation-type uses this list in writing the appropriate **:do-compiler-warnings** option to the macro if this option is not supplied explicitly.

:presentation-subtypep

Specifies a comparison function for deciding whether this presentation type is a subtype of some other presentation type, that is, for determining its equivalence-class membership.

The function receives two arguments, both lists. The first is a list of the type-name and data arguments of this presentation type; the second is a list of the type-name and data arguments of the putative supertype, that is, of the presentation type with which this one is being compared.

Such decisions are ordinarily made by **dw:presentation-subtypep**. They determine the applicability of mouse handlers to displayed presentations in a given input context. By writing the comparison function yourself, you can control the mouse sensitivity of presentations of the defined type relative to available mouse handlers.

Because you likely want to use arguments in the *data-arglist*

for writing the comparison function, you should not use the default value (t) for the **:data-arguments-are-disjoint** option to **define-presentation-type**. Rather, supply the latter option with a value of **nil**.

:do-compiler-warnings
Specifies a function for checking that presentation-type arguments appearing in the *data-arglist* and available at compile time are of the correct type. If you specify such arguments in the **:presentation-type-arguments** option, you do not need to write a **:do-compiler-warnings** function.

:typep Specifies a function that determines whether a given presentation object is of the type specified by the data arguments in the presentation type. It takes one argument, the presentation object.

The **:typep** function is used to determine, for example, whether a displayed integer presentation in an input context established by `(accept '((integer 1 10)))` can be used as input, that is, whether the displayed integer is in fact between 1 and 10. In the general case, the **:typep** function must consider all of the positional and keyword data arguments to a presentation type in determining if the presentation object at hand is of the type sought. The data arguments are made lexically available to the **:typep** function when it is invoked. (The presentation arguments are not available.)

:data-arguments-are-disjoint
Boolean option specifying whether the arguments included in the *data-arglist* are to be used as keys for determining the equivalence class of the presentation type. The default is **t**; this results in the data arguments to this presentation type being compared by **eql** with those of other presentation types when determining equivalence-class membership.

If you use the **:presentation-subtypep** option to **define-presentation-type** for writing the comparison function controlling equivalence-class membership, then you should supply the **:data-arguments-are-disjoint** option with a value of **nil**. Also supply a value of **nil** if the data arguments to this type are not appropriate for comparison by **eql**.

For an overview of **define-presentation-type** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

For information on writing parsers for presentation types, including examples: See the section "Writing a Presentation Type Parser", page 80. For more examples, see the file sys:dynamic-windows;standard-presentation-types.lisp.

dw:describe-presentation-type *type* &optional (*stream* ***standard-output***) *plural-count* *Function*

Outputs the description of a presentation type provided by the type's definition (**define-presentation-type** macro).

type The presentation type to be described.

stream The output stream; the default is ***standard-output***.

plural-count
Controls whether the description is pluralized. Three values are possible:

nil Do not pluralize the description.

t Pluralize the description.

number Include this number in the pluralization.

Examples:

```
(dw:describe-presentation-type 'integer) ==>an integer

(dw:describe-presentation-type 'integer t t) ==>integers

(dw:describe-presentation-type 'integer t 12) ==>twelve integers

(dw:describe-presentation-type 'integer t 12.2) ==>12.2 integers
```

For an overview of **dw:describe-presentation-type** and related facilities: See the section "Overview of Other Presentation Facilities", page 79.

dw:echo-presentation-blip *stream blip* &optional (*start-bp* **(send stream :read-location)**) *for-context-type* *Function*

Echos a presentation blip from the input buffer.

stream The input stream.

blip The presentation blip.

start-bp The position in the input buffer where the presentation blip begins.

for-context-type
The input context on whose behalf the presentation blip is echoed. This affects the printing of the blip. For example, the Command Processor uses this option to ensure that echoed command names are preceded by colons when in the `'command-or-form` context.

For an overview of **dw:echo-presentation-blip** and related facilities: See the section "Overview of Presentation Input Blip Facilities", page 78.

dw:peek-char-for-accept *stream* &optional *hang* *Function*
Returns the next character in the input stream without removing it from the stream. This is equivalent to calling **dw:read-char-for-accept** followed by **dw:unread-char-for-accept**.

stream The input stream.

hang Boolean option specifying whether, if no character is available in the input stream, the function waits until a character is available or returns **nil**. The default is **nil**.

For an overview of **dw:peek-char-for-accept** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:presentation-blip-case *blip* &body *clauses* *Macro*
Dispatches to clauses based on the presentation-type field of a presentation blip.

blip The presentation blip.

clauses The **case** clauses.

This macro is similar to the **case** special form, and could be written as

```
(case (dw:presentation-blip-presentation-type blip)
  <clauses>)
```

but with one exception: comparison of the extracted presentation type with the types used as keys to the *clauses* is based on **dw:presentation-subtypep**, not **eql**.

Normally, you would not use this macro directly. See the macro **dw:with-presentation-input-context**, page 388.

For an overview of **dw:presentation-blip-case** and related facilities: See the section "Overview of Presentation Input Blip Facilities", page 78.

dw:presentation-blip-ecase *blip* &body *clauses* *Macro*

Dispatches to clauses based on the presentation-type field of a presentation blip.

blip The presentation blip.

clauses The **ecase** clauses.

This macro is similar to the **ecase** special form, and could be written as

```
(ecase (dw:presentation-blip-presentation-type blip)
  <clauses>)
```

but with one exception: comparison of the extracted presentation type with the types used as keys to the *clauses* is based on **dw:presentation-subtypep**, not **eql**.

Normally, you would not use this macro directly. See the macro **dw:with-presentation-input-context**, page 388.

For an overview of **dw:presentation-blip-ecase** and related facilities: See the section "Overview of Presentation Input Blip Facilities", page 78.

dw:presentation-blip-object *presentation-blip* *Function*

Returns the presentation object from a presentation blip.

presentation-blip
The presentation blip.

For an overview of **dw:presentation-blip-object** and related facilities: See the section "Overview of Presentation Input Blip Facilities", page 78.

dw:presentation-blip-options *presentation-blip* *Function*

Returns the options field (a list of keyword-value pairs) of a presentation blip.

presentation-blip
The presentation blip.

The options inserted in a presentation blip are obtained from the values returned by translating mouse handlers. A standard blip option is **:activate**, which can be used by a translator to promote or prevent activation of the current field, that is, a return from the current call to **accept**. (See the macro **define-presentation-translator**, page 185.)

For an overview of **dw:presentation-blip-options** and related facilities: See the section "Overview of Presentation Input Blip Facilities", page 78.

dw:presentation-blip-p *blip* *Function*

Determines whether a blip is a presentation blip.

blip The blip.

For an overview of **dw:presentation-blip-p** and related facilities: See the section "Overview of Presentation Input Blip Facilities", page 78.

dw::presentation-blip-mouse-char *presentation-blip* *Function*

Returns the mouse character from a presentation blip.

presentation-blip
The presentation blip.

For an overview of **dw::presentation-blip-mouse-char** and related facilities: See the section "Overview of Presentation Input Blip Facilities", page 78.

dw:presentation-blip-presentation-type *presentation-blip* *Function*

Returns the presentation type from a presentation blip.

presentation-blip
The presentation blip.

For an overview of **dw:presentation-blip-presentation-type** and related facilities: See the section "Overview of Presentation Input Blip Facilities", page 78.

dw:presentation-blip-typep *blip type* *Function*

Determines whether the presentation type of a presentation blip is of a specified type. (The comparison is based on **dw:presentation-subtypep**).

blip The presentation blip.

type The presentation type with which the type of the blip is compared.

For an overview of **dw:presentation-blip-typep** and related facilities: See the section "Overview of Presentation Input Blip Facilities", page 78.

dw:presentation-equal *presentation-1 presentation-2* *Function*

Determines whether two presentations are "equal", that is, whether they are presenting the same object in the same manner.

presentation-1
The first presentation.

presentation-2
The second presentation.

For an overview of **dw:presentation-equal** and related facilities: See the section "Overview of Other Presentation Facilities", page 79.

dw:*presentation-input-context* *Variable*

Bound to the current presentation input context.

An input context is a list of the form `(presentation-type superior-context throw-flag . options)`. Each time a new input context is established, it becomes the new top-level context, enclosing the previous top-level context. (See the macro **dw:with-presentation-input-context**, page 388. Thus, there may be a hierarchy of contexts.

For example, if you have a presentation type **microcode-version** whose parser is defined as follows

```
(define-presentation-type microcode-version ()
  :parser ((stream)
           (accept 'integer :stream stream))
  :printer ((object stream)
            (princ object stream)))
```

the call `(accept '((microcode-version)))` results in the following input context:

```
(INTEGER (MICROCODE-VERSION NIL T :INHERIT T) T :INHERIT T)
```

The initial call to **accept** establishes the `MICROCODE-VERSION` context and calls the parser for **microcode-version.** The parser calls **accept** with the presentation type **integer**, and **accept** establishes a new context for `INTEGER`; the new context contains the old context for `MICROCODE-VERSION`.

For an overview of **dw:*presentation-input-context*** and related facilities: See the section "Overview of Presentation Input Context Facilities", page 78.

dw:presentation-input-context-option *presentation-input-context indicator* *Function*

Extracts the value of the specified option from an input context. The input context options are supplied in the *options* clause to **dw:with-presentation-input-context.**

presentation-input-context
Specifies the input context.

indicator Specifies the name of the option to be extracted from the input context.

For an overview of **dw:presentation-input-context-option** and related facilities: See the section "Overview of Presentation Input Context Facilities", page 78.

dw:presentation-subtypep *subtype supertype* *Function*

Determines whether one presentation type is a subtype of another presentation type.

subtype The putative subtype presentation type.

supertype The putative supertype presentation type.

This function is the presentation system equivalent of the Common Lisp function **subtypep**. As does the latter, it returns two values: the first indicates whether the first type is a subtype of the second; the second whether the first result is certain. Three combinations are possible:

t	t	*subtype* is definitely a subtype of *supertype*
nil	t	*subtype* is definitely not a subtype of *supertype*
nil	nil	the relationship could not be determined with certainty

For an overview of **dw:presentation-subtypep** and related facilities: See the section "Overview of Other Presentation Facilities", page 79.

dw:presentation-type-default *presentation-type* *Function*

Returns the current default – the object at the top of the type history – for a presentation type, if the type supports a history; otherwise, it returns **nil**.

presentation-type
The presentation type.

Example:

```
(dw:presentation-type-default 'pathname)
==>#P"Y:>reg>saved-mail>ui>defpgm.babyl.newest"
FS:LMFS-PATHNAME
T
```

For an overview of **dw:presentation-type-default** and related facilities: See the section "Overview of Other Presentation Facilities", page 79.

dw:presentation-type-name *type* *Function*

Returns the name of the presentation type from a presentation-type specification.

type The type specification.

Example:

```
(dw:presentation-type-name '((pathname) :dont-merge-default nil))
PATHNAME
```

For an overview of **dw:presentation-type-name** and related facilities: See the section "Overview of Other Presentation Facilities", page 79.

dw:presentation-type-p *type* *Function*

Returns **t** if its argument is a presentation type, **nil** otherwise.

type An object.

Example:

```
(defun pres-type-p-test (x type)
  (if (dw:presentation-type-p type)
      (present x type)))

(pres-type-p-test 6 '((integer 1 10))) ==>6
```

For an overview of **dw:presentation-type-p** and related facilities: See the section "Overview of Other Presentation Facilities", page 79.

dw:read-char-for-accept *stream* *Function*

Returns the next character in the input stream and removes this character from the stream.

stream The input stream.

The character returned may be a presentation blip character containing information specific to the **accept** input mechanism. Therefore, characters read via **dw:read-char-for-accept** should only be manipulated by the related Dynamic Window input functions. For example, you cannot use **char-equal** to compare a character returned by **dw:read-char-for-accept** with a standard character; you must use **dw:compare-char-for-accept** instead.

For an overview of **dw:read-char-for-accept** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:read-standard-token *stream* *Function*

Parses string as delimited by activation and blip characters established by **dw:with-accept-activation-chars** and **dw:with-accept-blip-chars**, respectively.

stream The input stream.

For an overview of **dw:read-standard-token** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:suggest *completion-string object* *Function*

Adds an element to a completion table being constructed inside a **dw:completing-from-suggestions** macro. **dw:suggest** is not used independently of this macro.

completion-string
The completion string.

object The object associated with the completion string (and to be returned by **dw:completing-from-suggestions**).

For an overview of **dw:suggest** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:unread-char-for-accept *char stream* *Function*

Puts a character back into the input stream. This character will be the next one read by a subsequent call to **dw:read-char-for-accept**.

char The character.

stream The input stream.

For an overview of **dw:unread-char-for-accept** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:with-accept-activation-chars (*additional-characters* &key *override*) &body *body* *Macro*

Binds local environment to establish additional characters to be used as delimiters of input strings. Predefined activation characters are #**return**, and #**end**.

additional-characters
A list of characters to be used as additional delimiters.

:override
Boolean option specifying whether the characters provided in the *additional-characters* argument are the only delimiters used within the body of the macro. If **t**, the provided characters replace the existing set for the dynamic extent of the macro. The default is **nil**, meaning that the supplied characters are added to the existing set of delimiters.

For an overview of **dw:with-accept-activation-chars** and related facilities:

See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:with-accept-blip-chars (*additional-characters* &key *override*) *Macro*
&body *body*

Binds local environment to establish additional characters to be used as delimiters of input blips. The characters are additional only if a previous, higher-level call to this macro in a nested structure has established an existing set of delimiters; no predefined set exists.

additional-characters
A list of characters to be used as additional delimiters.

:override
Boolean option specifying whether the characters provided in the *additional-characters* argument are the only delimiters used within the body of the macro. If **t**, the provided characters replace the existing set for the dynamic extent of the macro. The default is **nil**, meaning that the supplied characters are added to the existing set of delimiters.

For an overview of **dw:with-accept-blip-chars** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:with-accept-help *options* &body *body* *Macro*

Binds local environment to control HELP-key documentation for input to **accept**.

options A list of option specifications. Each specification is itself a list of the form (*<help-option>* *<help-string>*).

help-option
The *help-type* or a list of the form (*<help-type>* *<mode-flag>*). Help types are:

:top-level-help
Specifies that *help-string* be used instead of the default help documentation provided by **accept**.

:subhelp Specifies that *help-string* be used in addition to the default help documentation provided by **accept**.

Available modes include:

:append Specifies that the current help string be appended to any previous help strings of this type (top-level help or subhelp). This is the default mode.

:override
Specifies that the current help string is the help for this help type; no lower-level calls to **dw:with-accept-help** can override this. (**:override** works from the outside in.)

:establish-unless-overridden
Specifies that the current help string be the help text for this help unless a higher-level call to **dw:with-accept-help** has already established a help string for this help type in the **:override** mode.

help-string
A string or a function returning a string. If a function, it receives two arguments, the stream and the string-so-far.

Examples:

```
(dw:with-accept-help ((:subhelp "This is a test."))
  (accept 'pathname))

==> You are being asked to enter a pathname.   [ACCEPT did this]
    This is a test.                            [You did this]
    Use c-? or c-/ for a list of possibilities.[Completer did this]

(dw:with-accept-help ((:top-level-help "This is a test."))
  (accept 'pathname))

==> This is a test.                            [You did this]
    Use c-? or c-/ for a list of possibilities.[Completer did this]

(dw:with-accept-help (((:subhelp :override) "This is a test."))
  (accept 'pathname))
```

```
==> You are being asked to enter a pathname.   [ACCEPT did this]
    This is a test.                            [You did this]
                                               [Completer did
                                                nothing because
                                                you overrode it]

(define-presentation-type test ()
   :parser ((stream)
            (dw:with-accept-help
                ((:subhelp "A test is made up of three things:"))
              (dw:completing-from-suggestions ...))))

(accept 'test) ==> You are being asked to enter a test.
                   A test is made up of three things:

;;;use function to provide help string
(dw:with-accept-help (((:top-level-help :override)
                       (lambda (stream string-so-far)
                          (format stream "You are typing
                                          a pathname"))))
   ....)
```

For an overview of **dw:with-accept-help** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:with-accept-help-if *cond options* &body *body* *Macro*

Conditionally binds local environment to control HELP-key documentation for input to **accept**. Similar to **dw:with-accept-help**, but conditional.

cond The condition.

options A list of option specifications. Each specification is itself a list of the form (*<help-option>* *<help-string>*).

help-option

The *help-type* or a list of the form (*<help-type>* *<mode-flag>*). Help types are:

:top-level-help

Specifies that *help-string* be used instead of the default help documentation provided by **accept**.

:subhelp Specifies that *help-string* be used in

addition to the default help documentation provided by **accept**.

Available modes include:

:append Specifies that the current help string be appended to any previous help strings of this type (top-level help or subhelp). This is the default mode.

:override
Specifies that the current help string is the help for this help type; no lower-level calls to **dw:with-accept-help** can override this. (**:override** works from the outside in.)

:establish-unless-overridden
Specifies that the current help string be the help text for this help unless a higher-level call to **dw:with-accept-help** has already established a help string for this help type in the **:override** mode.

help-string
A string or a function returning a string. If a function, it receives two arguments, the stream and the string-so-far.

This macro is equivalent to the following form:

```
(if <cond>
    (dw:with-accept-help <> body)
    body)
```

For examples, see the dictionary entry for **dw:with-accept-help**.

For an overview of **dw:with-accept-help-if** and related facilities: See the section "Overview of Presentation-Type Definition Facilities", page 76.

dw:with-presentation-input-context (*presentation-type* &rest *options*) (&optional (*blip-var* **'dw::.blip.**)) *non-blip-form* &body *blip-cases* *Macro*

Binds local environment to the input context of a specified presentation type. (This essentially establishes mouse sensitivity for that type, and is one of the building blocks for **accept**.) The body (*non-blip-form*) is executed. If no mouse gestures are made by the user during execution of the body, this form returns the value of the *non-blip-form*. If the user

clicks on a presentation of an appropriate type, the corresponding *blip-cases* form is executed, with the resulting presentation blip bound as the value of *blip-var*.

presentation-type
The presentation type establishing the new input context. This may be a compound type incorporating more than one primitive type.

options Two predefined keyword options are available:

:stream Specifies the input stream; the default is ***standard-input***.

:inherit Boolean option specifying whether to inherit an existing input context or to establish a new root node; the default is **t**.

You may use any additional keywords you want.

blip-var The symbol to bind to the blip generated by clicking on an object of the specified type while in the context.

non-blip-form
The body form to execute inside the established input context.

blip-cases A case statement clause list. The keys are presentation types. The clause whose key matches the presentation type of the blip is executed, with the *blip-var* bound to the blip.

The presentation types available for use as keys are limited to the type specified by the *presentation-type* argument or, in the case of a compound presentation type (for example, **or**), the types specified; and the type or types inherited in the case of a nested use of this macro.

For an overview of **dw:with-presentation-input-context** and related facilities: See the section "Overview of Presentation Input Context Facilities", page 78.

dw:with-presentation-input-editor-context (*stream presentation-type . options*) (&optional (*blip-var* **'dw::.blip.**) *start-loc-var*) *non-blip-form* &body *blip-cases* *Macro*

Establishes an input context around a call to the input editor to read keyboard input from the user. The body (*non-blip-form*) is executed. If no mouse gestures are made by the user during execution of the body, this

form returns the value of the *non-blip-form*. If the user clicks on a presentation of an appropriate type, the resulting presentation blip is bound as the value of *blip-var*; the current location in the input buffer is bound as the value of *start-loc-var*; and the corresponding *blip-cases* form is executed.

accept uses this mechanism to establish an input context for the presentation type being read. This is one of the substrate functions used to build **accept**. Most programs simply want to call **accept**, instead of working at this low level.

stream The input stream; the default is ***standard-input***.

presentation-type
The presentation type establishing the new input context. This may be a compound type incorporating more than one primitive type.

options One predefined keyword option is available:

:inherit Boolean option specifying whether to inherit an existing input context or to establish a new root node; the default is **t**.

You may use any additional keywords you want.

blip-var The symbol to bind to the blip generated by clicking on an object of the specified type while in the context.

start-loc-var
The symbol to bind to the input buffer location at the time the presentation blip is received.

non-blip-form
The body form to execute inside the established input context.

blip-cases A case statement clause list. The keys are presentation types. The clause whose key matches the presentation type of the blip is executed, with the *blip-var* bound to the blip.

The presentation types available for use as keys are limited to the type specified by the *presentation-type* argument or, in the case of a compound presentation type (for example, **or**), the types specified; and the type or types inherited in the case of a nested use of this macro.

This macro is built on **dw:with-presentation-input-context**, to which it is similar:

```
(dw:with-presentation-input-editor-context (stream type)
                                           (blip-var)
    body-form
  blip-clauses)
```

is the same as

```
(with-input-editing (stream)
  (dw:with-presentation-input-context
     (type :stream stream)
     (blip-var)
       body-form
     blip-clauses))
```

For an overview of **dw:with-presentation-input-editor-context** and related facilities: See the section "Overview of Presentation Input Context Facilities", page 78.

dw:with-presentation-type-arguments (*type-name type*) &body *body* *Macro*

Binds local environment such that the arguments in a presentation-type specification are lexically available within the body of the macro.

type-name
: The name of the presentation type whose arguments are to be used, for example, `pathname`.

type
: The type specification, for example, `'((pathname) :format :directory :direction :write)`.

The *type-name* argument is known at compile time. It fixes the template for decoding the arguments of the particular *type* specification passed to the macro at runtime.

Example:

```
(define-presentation-type wood ((&key tree grade)
                                &key show-price)
  :printer ((wood stream &key type)
            (format stream "~A [wood ~A, ~A~:[~; ~2D cents~]]"
                    wood tree grade show-price
                                (compute-wood-price type))))
```

```
(defun compute-wood-price (presentation-type)
  (dw:with-presentation-type-arguments (wood presentation-type)
    (let ((base-price
            (ecase tree
             (mahogany 69)
             (pine 12)
             (teak 75)))
          (grade-multiplier
            (ecase grade
             (firsts-and-seconds 1.3)
             (firewood .2))))
      (* base-price grade-multiplier))))

(compute-wood-price '((wood :tree teak :grade firewood)
                      :show-price t)) ==>
15.0
```

For an overview of **dw:with-presentation-type-arguments** and related facilities: See the section "Overview of Other Presentation Facilities", page 79. See also: **dw:with-type-decoded**.

dw:with-type-decoded (*type-name-var* &optional *data-args-var* *presentation-args-var*) *type* &body *body* *Macro*

Binds local environment such that the type-name and, optionally, arguments in a presentation-type specification are bound to variables lexically available within the body of the macro.

type-name-var
: Symbol to bind the type-name of the presentation type.

data-args-var
: Symbol to bind to a list of the data arguments of the presentation type.

presentation-args-var
: Symbol to bind to a list of the presentation arguments of the presentation type.

Example:

```
(defun with-type-decoded-test ()
   (dw:with-type-decoded (type-name data-args pres-args)
                         '((integer 1 10) :base 8
                           :description "Integer between 1 and 10")
     (format t "~2%Type:  ~A
                ~%Data Arguments:  ~A
                ~%Presentation Arguments:  ~A"
                type-name data-args pres-args)))


(with-type-args-test) ==>

Type:  INTEGER

Data Arguments:  (1 10)

Presentation Arguments:  (BASE 8 DESCRIPTION Integer between 1 and 10)
```

For an overview of **dw:with-type-decoded** and related facilities: See the section "Overview of Other Presentation Facilities", page 79. See also: **dw:with-presentation-type-arguments**.

PART VIII.

Dictionary of Window Substrate Facilities

22. Dictionary Notes

This dictionary includes reference documentation for the following facilities:

Table of Window Substrate Facilities

Dynamic Window Facilities
- **dw:dynamic-window**
- **dw:margin-borders**
- **dw:margin-white-borders**
- **dw:margin-whitespace**
- **dw:margin-drop-shadow-borders**
- **dw:margin-ragged-borders**
- **dw:margin-label**
- **dw:margin-scroll-bar**
- **(flavor:method :set-margin-components dw:margin-mixin)**
- **(flavor:method :set-borders dw:margin-mixin)**
- **(flavor:method :set-label dw:margin-mixin)**
- **(flavor:method :delayed-set-label dw:margin-mixin)**
- **(flavor:method :update-label dw:margin-mixin)**
- **dw:set-default-end-of-page-mode**

Dynamic Frame Facilities
- **dw:program-frame**

In the dictionary, the facilities are arranged in alphabetical order (package prefixes excluded).

For conceptual documentation: See the section "Overview of Window Substrate Facilities", page 87.

23. The Facilities

:delayed-set-label *new-label* of **dw:margin-mixin** *Method*

Provides a new label for a Dynamic Window, but delays the writing of the new label until the **:update-label** message is sent: See the method **(flavor:method :update-label dw:margin-mixin)**, page 411.

new-label The label string.

The *new-label* can be specified with any of the options acceptable to **dw:margin-mixin**.

For an overview of **(flavor:method :delayed-set-label dw:margin-mixin)** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

dw:dynamic-window *Flavor*

The basic Dynamic Window flavor. It provides output-history recording (of displayed presentations) as well as vertical and horizontal scrolling. Dynamic Windows are created in the same manner as static windows, with the **tv:make-window** function.

dw:dynamic-window is built on several component flavors, from which it inherits a large number of init options. These include all init options (about 40) to the basic, non-Dynamic Window flavor, **tv:window**. Below we provide references to these inherited options, but first discuss four that are specific to Dynamic Windows.

:end-of-page-mode

Specifies what happens when queued output exceeds the space available in the current viewport of the window. There are four possibilities:

:default Uses the global default for Dynamic Windows set by the Set Screen Options command or the **dw:set-default-end-of-page-mode** function on which the command is based.

:scroll Causes the window to scroll automatically to accommodate the output. The amount by which the window is scrolled is set by the **:scroll-factor** init option to Dynamic Windows.

:truncate

Causes scrolling to be the responsibility of the user, who must press the SCROLL key to see more output.

:wrap Causes new output to appear at the top of the window, rather than at the bottom as in the case of **:scroll** or **:truncate**.

:scroll-factor
Specifies the amount by which a Dynamic Window is scrolled when the value of its **:end-of-page-mode** init option is **:scroll**. Possible values include an integer (number of lines), ratio (fraction of the screen), or **nil** (use the global default set by the Set Screen Options command or the function **dw:set-end-of-page-mode**).

:mouse-blinker-character
Specifies the shape of the mouse cursor when it is over the window, for example, **#\mouse:fat-circle**. The default is **#\nw-arrow**. For a full listing of all the possibilities: See the section "Mouse-Blinker Characters", page 89.

:margin-components
Specifies a list of the form ((*component-1* [*keys*]) (*component-2* [*keys*]) ... (*component-n* [*keys*])), where *component-x* is one of a set of margin-component flavors and *keys* are zero or more keywords or keyword-value pairs appropriate for the given flavor.

Available margin-component flavors include the following:

dw:margin-borders
Provides a four-sided, black (normal video) border of a specified thickness.

dw:margin-white-borders
Provides a four-sided, white border of a specified thickness.

dw:margin-whitespace
Provides whitespace of a specified thickness on a specified margin.

dw:margin-drop-shadow-borders
Provides a three-pixel-wide black border shadowed on its right and bottom margins by an eight-pixel-wide gray border.

dw:margin-ragged-borders
Provides a ragged (wavy) border of a specified thickness.

dw:margin-label
Provides a label on the upper or lower margin. By default, the label string is created from the name of the window flavor.

dw:margin-scroll-bar
Provides the standard elevator scroll bar on the specified margin.

For more detailed information on these flavors, including allowable keywords, see the respective dictionary entry for each.

The following example illustrates the use of margin-component flavors. Note that the margin is built from the outside in.

```
(defun dynamic-window-margin-example ()
  (let ((test (tv:make-window 'dw:dynamic-window
        :edges-from :mouse
        :margin-components
        '((dw:margin-borders :thickness 1)
          (dw:margin-white-borders :thickness 3)
          (dw:margin-borders :thickness 10)
          (dw:margin-white-borders :thickness 8)
          (dw:margin-borders :thickness 3)
          (dw:margin-whitespace :margin :left :thickness 10)
          (dw:margin-scroll-bar)
          (dw:margin-whitespace :margin :bottom :thickness 7)
          (dw:margin-scroll-bar :margin :bottom)
          (dw:margin-whitespace :margin :left :thickness 10)
          (dw:margin-label :margin :bottom
                           :style (:sans-serif :italic :normal))
          (dw:margin-whitespace :margin :top :thickness 10)
          (dw:margin-whitespace :margin :right :thickness 13))
        :expose-p t)))
    (send test :set-label "Margin Test Window")))
```

The remaining init options to **dw:dynamic-window** are those it shares with **tv:window**. These are documented elsewhere. Below are references to the relevant sections followed in each case by a list of the init options covered:

Reference: See the section "Creating a Window" in *Programming the User Interface, Volume B.*

:blinker-p
:default-character-style
:save-bits
:superior
:activate-p
:expose-p

Reference: See the section "Window Attributes for Character Output" in *Programming the User Interface, Volume B.*

:more-p
:vsp
:reverse-video-p
:deexposed-typeout-action
:deexposed-typein-action
:right-margin-character-flag
:backspace-not-overprinting-flag
:cr-not-newline-flag
:tab-nchars

Reference: See the section "Initializing Window Size and Position" in *Programming the User Interface, Volume B.*

:left
:x
:top
:y
:position
:right
:bottom
:width
:height
:size
:inside-width
:inside-height
:inside-size
:edges
:character-width
:character-height
:integral-p
:edges-from
:minimum-height
:minimum-width

Reference: See the section "Window Borders" in *Programming the User Interface, Volume B.*

:borders
:border-margin-width

Reference: See the section "Window Labels" in *Programming the User Interface, Volume B.*

:name
:label

Reference: See the section "Flavors for Panes and Frames" in *Programming the User Interface, Volume B.*

:io-buffer

In addition to the large overlap in init options between static and Dynamic Windows, virtually all of the window methods, messages, and functions documented in *Programming the User Interface, II* for static windows can also be used with Dynamic Windows. These are too numerous to list individually as we did for the init options; we refer you to the following sections for more information:

Reference: See the section "Window Graying" in *Programming the User Interface, Volume B.*

Reference: See the section "Window Status" in *Programming the User Interface, Volume B.*

Reference: See the section "Activities and Window Selection" in *Programming the User Interface, Volume B.*

Reference: See the section "Creating a Window" in *Programming the User Interface, Volume B.*

Reference: See the section "Character Output to Windows" in *Programming the User Interface, Volume B.*

Reference: See the section "Line-Truncating Windows" in *Programming the User Interface, Volume B.*

Reference: See the section "Graphic Output to Windows" in *Programming the User Interface, Volume B.* (Also: See the section "Overview of Graphic Output Facilities", page 57.)

Reference: See the section "Notifications" in *Programming the User Interface, Volume B.*

Reference: See the section "Using TV Fonts" in *Programming the User Interface, Volume B.*

Reference: See the section "Handling the Mouse" in *Programming the User Interface, Volume B.*

Reference: See the section "Window Sizes and Positions" in *Programming the User Interface, Volume B.*

Reference: See the section "Window Labels" in *Programming the User Interface, Volume B.* (Only the **:name** method.)

Finally, a number of methods intended exclusively for Dynamic Windows are available. These are included among both Basic Program Output Facilities (See the section "Overview of Other Facilities for Program

Output", page 60.) and window substrate facilities (See the section "Overview of Window Substrate Facilities", page 87.)

dw:dynamic-window *Resource*

A resource of Dynamic Windows. The resource is created via **tv:defwindow-resource** with the **:initial-copies** option set to 1 and the **:reuseable-when** option set to **:deactivated**. (For more information on resources generally: See the section "Resources" in *Internals, Processes, and Storage Management.*)

The following keyword options are available when allocating from or using the Dynamic Window resource:

:momentary-p
Boolean option specifying whether the window provided is momentary, that is, whether it is deactivated if the mouse cursor is moved off the window. The default is **nil**.

:temporary-p
Boolean option specifying whether the window provided is temporary, that is, whether it locks the superior window until it is deactivated. The default is the value of the **:momentary-p** option. If **nil**,

:hysteresis
If the **:momentary-p** option is **t**, specifies the distance, in pixels, that the mouse cursor must be from the edge of the window before it is deactivated. The default value is 25.

Note that in order to use these keywords, you must also supply an optional positional argument for the window's superior. In the following example, the superior is **tv:main-screen**, which is also the default if no arguments are supplied.

Example:

```
(defun dw-resource ()
  (using-resource (my-dw dw:dynamic-window tv:main-screen
                         :momentary-p t :hysteresis 15)
    (send my-dw :set-size 500 300)
    (send my-dw :expose)))
```

dw:margin-borders *Flavor*

Flavor for providing Dynamic Windows with a four-sided, black (or draw-alu color) border.

dw:margin-borders accepts the following init option:

:thickness
Specifies the thickness, in pixels, of the border; the default is 1.

For an overview of **dw:margin-borders** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

dw:margin-drop-shadow-borders *Flavor*

Flavor for providing Dynamic Windows with a black (normal video) border shadowed on its right and bottom margins by a gray border.

dw:margin-drop-shadow-borders accepts the following init options:

:non-shadow-thickness
Specifies the thickness, in pixels, of the black border; the default is 3.

:outside-margin
Specifies the thickness, in pixels, of whitespace surrounding the shadowed and non-shadowed borders of the box; the default is 0.

:shadow-thickness
Specifies the thickness, in pixels, of the gray margins on the right and bottom edges of the window; the default is 8.

For an overview of **dw:margin-drop-shadow-borders** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

dw:margin-label *Flavor*

Flavor for specifying Dynamic Window labels.

dw:margin-label accepts the following init options:

:background-gray
Specifies a binary array to use as a background pattern for the label.

You can provide your own array via the **tv:make-binary-array** function – for an example, see the dictionary entry for **graphics:draw-pattern** – or use one of the standard, background-gray patterns: **tv:25%-gray**, **tv:33%-gray**, **tv:50%-gray**, or **tv:75%-gray**.

Note that the specification is for an array object, not its symbol.

:box Specifies whether to enclose the label in a box; the default is **nil**. Other permissible values are **:inside** and **:outside**. If you wish to box the label within a just-specified border, use **:inside**; if you wish to box the label outside of a border about to be specified, use **:outside**.

:box-thickness
Specifies the thickness, in pixels, of the line used to draw a box around the label when the **:box** init option is non-**nil**.

:centered-p
Boolean option specifying whether the label is left-right centered. The default is **nil**, causing the label to appear on the left side of the margin.

:extend-box-p
Boolean option specifying whether the box drawn (when the **:box** option is non-**nil**) extends the full length of the margin or is limited to the length of the label; the default is **t**.

:margin Specifies the margin, **:top** or **:bottom**, on which the label appears; the default is **:bottom**.

:string Specifies the label string. The default string is derived from the name of the window flavor used to make the window instance.

:character-style
Specifies the character style used for writing the label string. The default value is the character-style default for the screen.

After a window instance is created, you can change its label by using **(flavor:method :set-label dw:margin-mixin)**.

For an overview of **dw:margin-label** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

dw:margin-ragged-borders *Flavor*

Flavor for providing Dynamic Windows with a ragged (wavy) border to indicate that more output can be viewed by scrolling in the direction indicated. The border is only ragged when there is in fact more output to be viewed; otherwise, it is straight.

dw:margin-ragged-borders accepts the following init options:

:horizontal-too
Boolean option specifying whether to provide ragged left and right margins in addition to ragged top and bottom margins; the default is **t**.

:thickness
Specifies the thickness, in pixels, of the border; the default is 1.

For an overview of **dw:margin-ragged-borders** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

dw:margin-scroll-bar *Flavor*

Flavor for providing an "elevator" scroll bar to a Dynamic Window.

dw:margin-scroll-bar accepts the following init options:

:elevator-thickness
Specifies the overall width, in pixels, of the scroll bar; the default is 10.

:margin Specifies the margin – **:left**, **:right**, **:top**, or **:bottom** – on which the scroll bar appears; the default is **:left**.

:shaft-whitespace-thickness
Specifies the thickness, in pixels, of additional whitespace (normal video) inserted on each side of the scroll bar between it and the neighboring component. The default is 0, causing the whitespace to be one-pixel-wide on both sides.

:visibility
Specifies when the scroll bar is visible. Three values are permitted:

:normal The scroll bar appears when the flavor is instantiated and remains visible regardless of whether it is needed. This is the default.

:if-requested
An empty elevator shaft appears when the flavor is instantiated and after each new output operation to the window. If the user moves the mouse cursor into the scroll bar area, the standard cross-hatched pattern in drawn in the shaft and the scroll bar becomes normally active.

:if-needed
The scroll bar does not appear until the output exceeds the window space available for displaying it, that is, until the need for scrolling arises; thereafter it remains visible and normally active. The space needed for drawing the scroll bar is reserved by whitespace (normal video) until the scroll bar appears.

For an overview of **dw:margin-scroll-bar** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

dw:margin-white-borders *Flavor*

Flavor for providing Dynamic Windows with a four-sided, white (or erase-alu color) border.

dw:margin-white-borders accepts the following init option:

:thickness
Specifies the thickness, in pixels, of the border; the default is 1.

For an overview of **dw:margin-white-borders** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

dw:margin-whitespace *Flavor*

Flavor for providing Dynamic Windows with whitespace (or erase-alu color) on a margin.

dw:margin-whitespace accepts the following init options:

:margin Specifies the margin, one of **:left**, **:right**, **:top**, or **:bottom**.

:thickness
Specifies the thickness, in pixels, of the border; the default is 1.

For an overview of **dw:margin-whitespace** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

dw:program-frame *Flavor*

The flavor used by **dw:define-program-framework** for the program frames it creates. **dw:program-frame** is the Dynamic Window equivalent of **tv:constraint-frame-with-shared-io-buffer**, which it incorporates as one of its component flavors; another component flavor is **tv:process-mixin**. Generally, you do not make direct use of this flavor; that you leave up to **dw:define-program-framework**.

Init options, methods, and messages for this flavor include all of those for **tv:constraint-frame-with-shared-io-buffer**: See the section "Frames" in *Programming the User Interface, Volume B.* The following are additional init options:

:label See the section "Window Labels" in *Programming the User Interface, Volume B.*

:margin-components
See the flavor **dw:dynamic-window**, page 399.

:process See the section "Windows and Processes" in *Programming the User Interface, Volume B.*

:program
The name of the program for which this is the program frame.

:query-io-pane
Specifies the pane to which ***query-io*** is bound when an instance of the program frame is active.

:size-from-pane
Specifies the pane on which to base the size of the program frame.

:terminal-io-pane
Specifies the pane to which ***terminal-io*** is bound when an instance of the program frame is active.

For an overview of **dw:program-frame** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

dw:program-frame *Resource*

A resource of program frames (of the kind used by **dw:define-program-framework**). The resource is created via **tv:defwindow-resource** with the **:initial-copies** option set to **nil** and the **:reuseable-when** option set to **:deactivated**. (For more information on resources generally: See the section "Resources" in *Internals, Processes, and Storage Management.*)

The following keyword options are available when allocating from or using the program frame resource:

:temporary-p
Boolean option specifying whether the frame provided is

temporary, that is, whether it locks the superior window until it is deactivated.

:process The process associated with the frame or **nil**, for no associated process. The default process is that of the program for which this frame was created (by **dw:define-program-framework**).

When using this resource, you must supply the name of the program whose frame is to be provided. In the following example, a Frame-Up Layout Designer frame is specified.

Example:

```
(defun pf-resource ()
  (using-resource (my-pf dw:program-frame 'dw::layout-designer)
    (send my-pf :expose)))
```

:set-borders *borders* of **dw:margin-mixin** *Method*

Replaces the current borders of a Dynamic Window with simple borders (like those provided by **dw:margin-borders**).

borders The thickness, in pixels, of the new borders; the default is 1.

For an overview of **(flavor:method :set-borders dw:margin-mixin)** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

dw:set-default-end-of-page-mode *new-end-of-page-mode* &optional (*new-scroll-factor* **nil**) *Function*

Sets global default for what happens when queued output exceeds the space available in the current viewport of a Dynamic Window.

new-end-of-page-mode
The new mode. There are three possibilities:

:scroll Causes the window to scroll automatically to accommodate the output. If you supply this argument, make sure you also supply a numeric value for the *new-scroll-factor* argument.

:truncate
Causes scrolling to be the responsibility of the user, who must press the SCROLL key to see more output.

:wrap Causes new output to appear at the top of the window, rather than at the bottom as in the case of **:scroll** or **:truncate**.

new-scroll-factor
The amount by which the window is scrolled when the value of the *new-end-of-page-mode* argument is **:scroll**. Permissible values include integers (number of lines) and ratios (fraction of the screen). Do not use the default value (**nil**), or else an error results.

For an overview of **dw:set-default-end-of-page-mode** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

:set-label *label* of **dw:margin-mixin** *Method*
Provides a new label for a Dynamic Window.

label The label string.

The *label* can be specified with any of the options acceptable to **dw:margin-mixin**.

For an overview of **(flavor:method :set-label dw:margin-mixin)** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

:set-margin-components *new-components* of **dw:margin-mixin** *Method*
Replaces the current margin components of a Dynamic Window with a new set of components.

new-components
Specifies a list of the form ((*component-1* [*keys*]) (*component-2* [*keys*]) ... (*component-n* [*keys*])), where *component-x* is one of a set of margin-component flavors and *keys* are zero or more keywords or keyword-value pairs appropriate for the given flavor.

For a list of available margin-component flavors and an example: See the flavor **dw:dynamic-window**, page 399.

For an overview of **(flavor:method :set-margin-components dw:margin-mixin)** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

:update-label of **dw:margin-mixin** *Method*
Causes a new label to be written for a Dynamic Window. The label must have previously been created via the **:delayed-set-label** method: See the method **(flavor:method :delayed-set-label dw:margin-mixin)**, page 399.

For an overview of **(flavor:method :update-label dw:margin-mixin)** and related facilities: See the section "Overview of Window Substrate Facilities", page 87.

Index

B

C C C

D

D D

F F F

G

G

G

H

H H

K

L

M

N **N** **N**

O O O

Q

Q Q

R

R R

S S S

T

U

V

W W W

X

Y

Z Z Z