Wlisp Manual

A
Window System
and
User Interface Toolkit

Franz Fabian, Andreas C. Lemke
Technical Report CU-CS-302A-85

July 15, 1985

Translated and Revised by Christopher Morel and Vera M. Patten

Table of Contents

List of Figures

List of Tables

1. Introduction to Wlisp

The following is a short introduction to Wlisp. Wlisp is a window system and user interface toolkit based on Unix 4.2bsd FranzLisp and ObjTalk[1]. Three different aspects of the window system will be described:

1. The perspective of the **user** of the window system and its applications.

2. The perspective of the **programmer** of application systems that are to run within Wlisp.

3. The **implementation** of Wlisp taking the characteristics of the Bitgraph terminal[2] into consideration.

Separation of points 2 and 3 is not completely possible since the Bitgraph terminal has influenced the design of the window system at higher levels.

[1] An object-oriented extension of FranzLisp [Lemke 85; Rathke, Laubsch 83].

[2] Wlisp is currently only implemented on the Bitgraph terminal (produced formerly by Bolt Baranek & Newman (BBN), now by Forward Technology, San Jose, Ca..)

2. The User's Perspective of the System

This chapter will describe Wlisp from the *perspective of the user* of FranzLisp and the application systems embedded in Wlisp. The appearance of a window on the screen, the screen build-up after the start of the system,[3] the interactions with the FranzLisp system, and the use of the mouse for the manipulation of windows and for menu selection will be described.

But first a general remark:

While using Wlisp, the user is interacting with the FranzLisp system. This fact is obvious when viewing the implementation of the application system. The user must be able to communicate with the FranzLisp system especially in case of errors (see also section). For these reasons, a minimal understanding of FranzLisp is desirable for satisfactory use of Wlisp.

2.1. The Appearance of a Window

A window on the screen can be made up of many parts. In general, a window consists of a title bar that contains the name of the window on a black background, a frame, and the window contents in which the information is contained.

Some windows also have a scroll bar (i.e., two borders with arrows inside) located at the left and bottom boundaries. This feature allows the window contents to be moved so that different information inside the window can be made visible. (see also Figure).

Windows on the screen can overlap so that only some windows are completely visible at a given time, some are partially visible, and some are not visible at all (see also Figure).

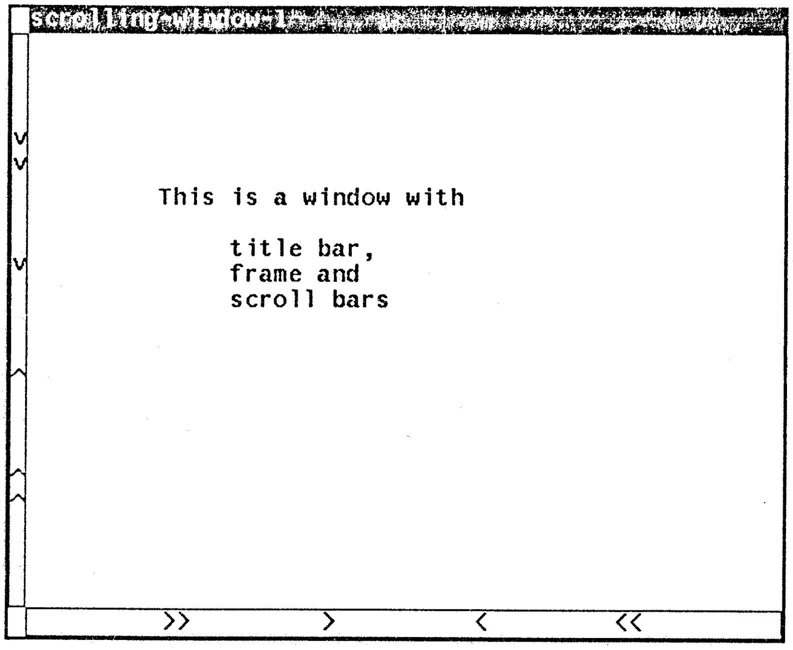
Some windows are only used for the display of information (*passive* windows), i.e., the user can just inspect the information contained in the window. Other windows can be used to edit their contents or to input data for an application associated with them. (*active* window). The way in which the window types behave is in general determined by the specific application.

Every window also has a menu which you can utilize to perform important window operations. This menu is usually not visible until it is requested by the user (`pop-up-menu`). The procedure to display the menu on the screen to choose an operation is described in subsection .

2.2. The Screen

After start-up of Wlisp, a window titled `toplevel` is displayed in the upper left corner of the screen. The cursor is located in the upper left corner of this window to the right of the prompt. By using the `toplevel` window, the user can communicate with the FranzLisp interpreter (see also subsection). Above the `toplevel` window another window can be found that is difficult to detect since it has no title and its background is in *reverse-video* (i.e., normally black). This window is the *prompt window* which is used for messages from application programs or from the window system, and for the entry of parameters (see also Figure).

[3]The start up of the system from a UNIX shell is described in Appendix



This is a window with

title bar,
frame and
scroll bars

Figure 2-1: Example of a window with title, frame, and scrollbar.

2.3. The FranzLisp Environment of the Window System

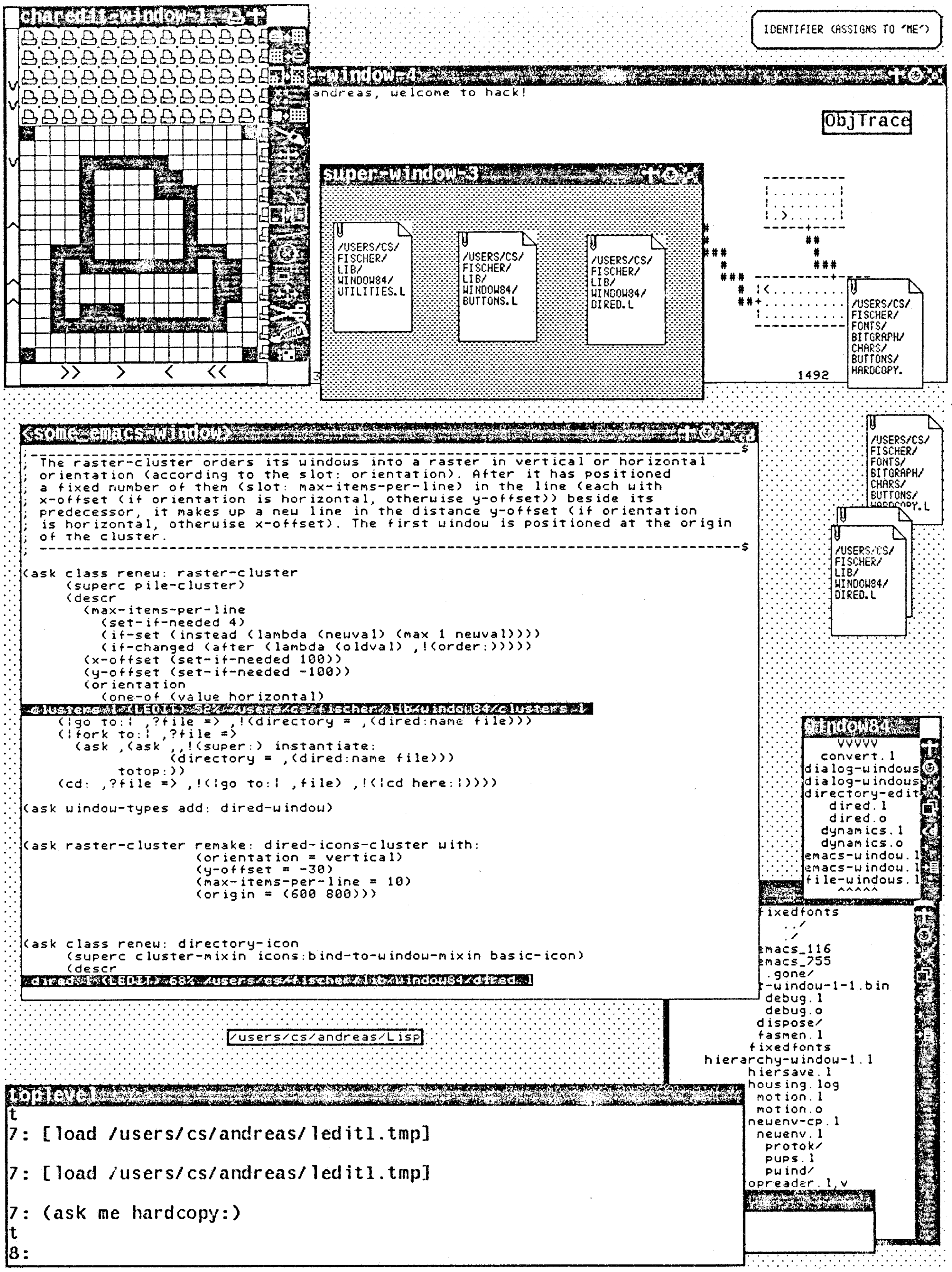
Several components of the FranzLisp programming environment were integrated into the window system to improve its quality. Further information regarding Trace and Break can be found in the FranzLisp manual [Foderaro 82].

2.3.1. Toplevel

The interactions of the user with the FranzLisp Toplevel (the print-eval-read loop of the interpreter) occur in the `toplevel` window. Whenever the system is waiting for user input, the system will return to this window. As a consequence, all windows that overlap with the `toplevel` window will be covered by it.

2.3.2. Break

Debugging of Lisp programs is supported through the BREAK mechanism of FranzLisp that is integrated into the window system. If you enter into the BREAK status through an error in the program or through a user interruption (**ctrl-c** key), then a new window is displayed on the screen with its own `print-eval-read` loop executing in which the internal (Lisp) environment can be examined. The regions outside the BREAK window remain unchanged and available to the user during the error search in BREAK.



charedit-window-1
>>
>
<
<<
IDENTIFIER (ASSIGNS TO 'ME')
-window-4
ObjTrace
super-window-3
/USERS/CS/FISCHER/LIB/WINDOW84/UTILITIES.L
/USERS/CS/FISCHER/LIB/WINDOW84/BUTTONS.L
/USERS/CS/FISCHER/LIB/WINDOW84/DIRED.L
/USERS/CS/FISCHER/FONTS/BITGRAPH/CHARS/BUTTONS/HARDCOPY.
1492
/USERS/CS/FISCHER/FONTS/BITGRAPH/CHARS/BUTTONS/HARDCOPY.L
/USERS/CS/FISCHER/LIB/WINDOW84/DIRED.L
<some_emacs-window>
window84
/users/cs/andreas/Lisp
toplevel

```
andreas, welcome to hack!
```

```
;------------------------------------------------------------------------$
; The raster-cluster orders its windows into a raster in vertical or horizontal
; orientation (according to the slot: orientation). After it has positioned
; a fixed number of them (slot: max-items-per-line) in the line (each with
; x-offset (if orientation is horizontal, otherwise y-offset)) beside its
; predecessor, it makes up a new line in the distance y-offset (if orientation
; is horizontal, otherwise x-offset). The first window is positioned at the origin
; of the cluster.
; ------------------------------------------------------------------------$

(ask class renew: raster-cluster
     (superc pile-cluster)
     (descr
       (max-items-per-line
         (set-if-needed 4)
         (if-set (instead (lambda (newval) (max 1 newval))))
         (if-changed (after (lambda (oldval) ,!(order:)))))
       (x-offset (set-if-needed 100))
       (y-offset (set-if-needed -100))
       (orientation
         (one-of (value horizontal)
 clusters.l (LEDIT) 52% /users/cs/fischer/lib/window84/clusters.l
    (!go to:! ,?file => ,!(directory = ,(dired:name file)))
    (!fork to:! ,?file =>
      (ask ,(ask ,,!(super:) instantiate:
                  (directory = ,(dired:name file)))
           totop:))
    (cd: ,?file => ,!(!go to:! ,file) ,!(!cd here:!))))

(ask window-types add: dired-window)


(ask raster-cluster remake: dired-icons-cluster with:
                    (orientation = vertical)
                    (y-offset = -30)
                    (max-items-per-line = 10)
                    (origin = (600 800)))



(ask class renew: directory-icon
     (superc cluster-mixin icons:bind-to-window-mixin basic-icon)
     (descr
 dired.l (LEDIT) 68% /users/cs/fischer/lib/window84/dired.l
```

```
 VVVVV
 convert.l
dialog-windows
dialog-windows
directory-edit
 dired.l
 dired.o
 dynamics.l
 dynamics.o
emacs-window.l
emacs-window.l
file-windows.l
 ^^^^^
```

```
fixedfonts
   ../
   ./
emacs_116
emacs_755
 .gone/
-window-1-1.bin
  debug.l
  debug.o
 dispose/
  fasmen.l
 fixedfonts
hierarchy-window-1.l
 hiersave.l
 housing.log
  motion.l
  motion.o
 newenv-cp.l
  newenv.l
   protok/
   pups.l
   pwind/
 opreader.l,v
```

```
t
7: [load /users/cs/andreas/leditl.tmp]

7: [load /users/cs/andreas/leditl.tmp]

7: (ask me hardcopy:)
t
8:
```

Figure 2-2: A screen with various overlapping windows.

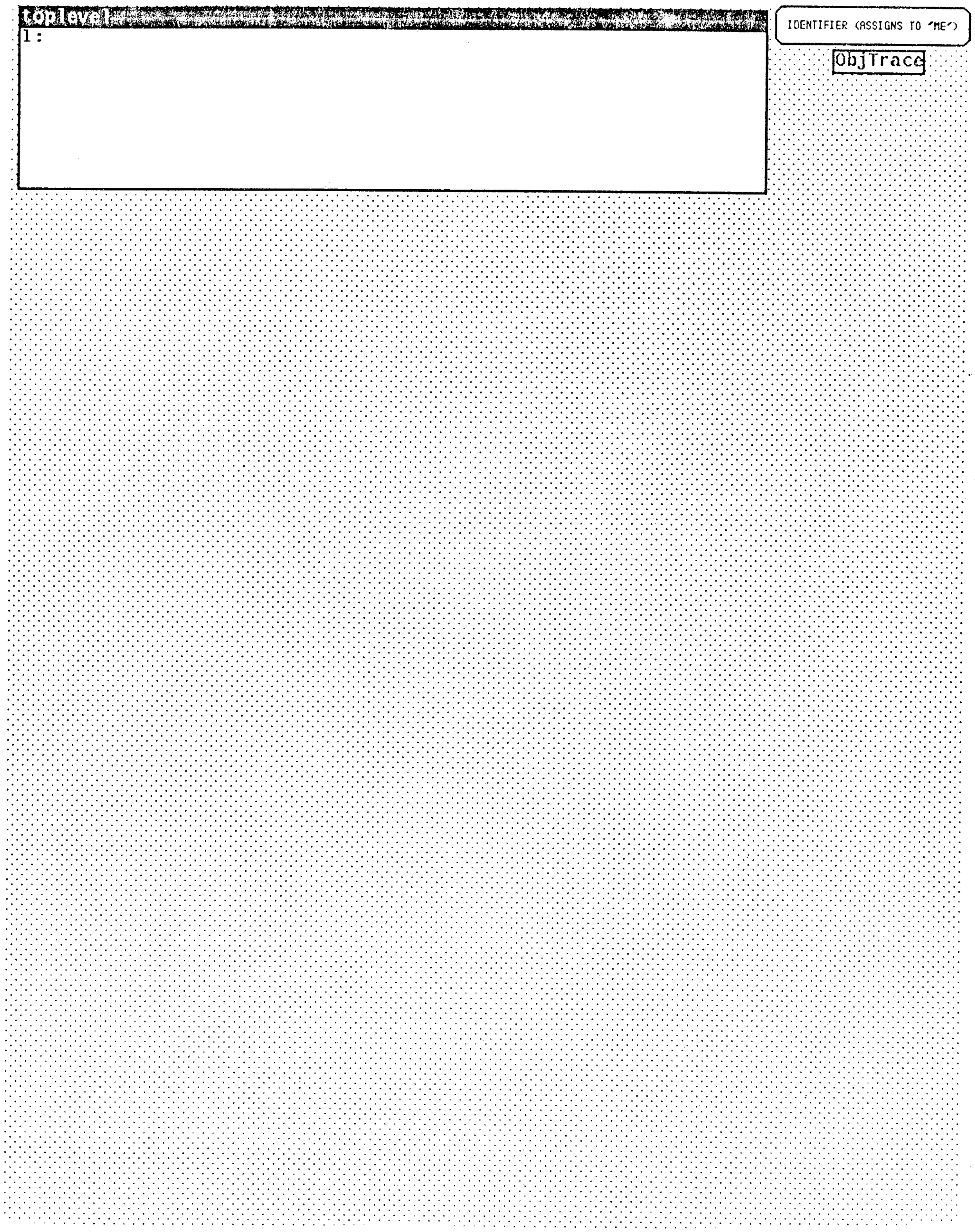


Figure 2-3: The screen after start-up of the window system.

2.3.3. Trace

During the tracing of Lisp functions, the trace output can be redirected into its own window so that it is not confused with the actual output of the program. For this purpose `wtrace` instead of `trace` is to be called. `wtrace` has the same form as `trace`, except `traceenter` and `traceexit` parameters cannot be used because these are utilized by the window trace.

2.4. Resolving Errors

If error messages appear on the screen either from the Lisp interpreter or the window system, there are several ways to deal with these.

1. if only one window is affected, then use the `refresh` command of the window's menu;
2. by selecting `refresh` from the system menu, the entire screen can be cleared and redrawn (see section);
3. if it is necessary to reset the Bitgraph terminal or if it appears that a reset of the terminal and the window system is appropriate, then the `(reset-all)` command, accomplishes this (a terminal-reset must precede it though). Through this, the necessary data for the window system is newly established within the Bitgraph terminal (which takes about 30-60 seconds depending on the number of windows and fonts currently existing).

2.5. Use of the Mouse

The mouse is an important aid for facilitating communication between the user and the window system, and to execute the available application programs. The following, however, only describes the possible interactions with the window system, since the use of the mouse in application programs varies. Yet, general guidelines for the use of the mouse exist for the developer of application systems to maintain consistency between user-window interactions. In general, the following can be assumed:[4]

- with the *left* button, the object pointed to by the mouse (e.g., a window) can be selected, which can result in one of several actions:
  - **marking** of the selected object;
  - **activation** of a program in a window;
  - **scrolling** of the window contents by using the *scrollbar* of a window.
- with the *right* button, a menu is displayed to the user that is related to the object that was chosen (see also subsection).

Normally, the mouse has the form of a small arrow pointing in the direction of the upper right corner. But the mouse can change its appearance to indicate to the user that you are in a mode where the mouse is used in a different way.

The following actions can specifically be carried out by the mouse in the window system:

- if the left button is pressed, then the window in which the mouse is found is made completely

[4] At this point in time, only the left and right buttons of the mouse have standard functions - the middle button does not.

visible and the program (if one exists) within the window is activated;

- with the right button the user receives a menu through which actions such as *move, change size*, etc. can be performed (see Table);
- if the mouse is not pointing to a window, then a global menu (system menu) is displayed by using the right button with which you can, for example, create new windows (see Table).

COMMAND	DESCRIPTION
`refresh`	the screen is cleared and redrawn.
`create`	type of the new window can be selected from a second menu.

Table 2-1: Description of the System Menus.

COMMAND	DESCRIPTION
`refresh`	redraws the window.
`totop`	makes a window completely visible.
`tobottom`	the window is covered by others that overlap with it.
`bury`	takes the window off the screen (it still exists).
`kill`	deletes the window.
`move`	repositions the window to a new location on the screen.
`copy&move`	creates another copy of the window at a new screen location.
`put`	places the window into another super-window.
`newshape`	defines a new window size and position.
`reshape`	repositions one corner.

Table 2-2: Description of Window Operations in the Window Menu.

2.5.1. Window Operations Using the Mouse

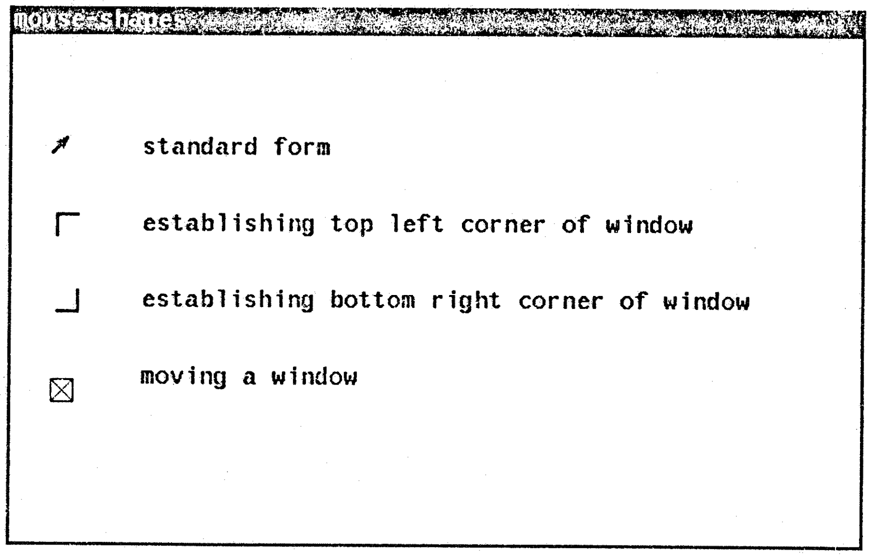
By using the mouse, three different window operations can be performed in conjunction with the window menu (see also Table and Figure):

1. Moving a window: (`move:`) The mouse has the form of a *box with a diagonal cross in it.* After pressing the left button, a frame with the dimensions of the window appears on the screen and

can be moved using the mouse. By releasing the button, the new position of the window is established.

2. Setting the dimensions of a window: (newshape:) The mouse has the form of an *upper left corner* of a window. By pressing the left button, the position of the upper left corner of the window is established. Then the mouse becomes a *bottom right corner* which is used to establish the other corner of the window. During this, the "elastic" boundaries of the new window are displayed. By releasing the left button the second corner is fixed. The user can also perform these steps in the reverse order (setting the right corner first).

3. Reshaping a window: (reshape:) Only one corner can be re-established. The interactions are similar to point 2 above.



↗	standard form
⌜	establishing top left corner of window
⌟	establishing bottom right corner of window
⊠	moving a window

Figure 2-4: The different forms of the mouse.

2.5.2. Menu Selection with the Mouse

If a menu is already displayed on the screen, then you bring the mouse to the menu and press the left button (don't release it immediately!). Following this, a *black rectangle* (or a frame)[5] indicates the selected menu item. By moving the mouse, the selection can be changed and by *releasing* the button the element is selected.

Pop-up-menus (see section 2.1) are displayed on the screen by pressing the right button and the selection is made by pressing and releasing the left button.

[5]Which of the two is used has not yet been established. Additionally, the mouse can take on the form of a cross and the menu selections can be outlined.

By placing the mouse to the extreme right of the menu (the rectangle or frame will disappear), you can exit the menu selection mode by pressing the left button. In this way, no selection is made from the menu.

2.5.3. Scrolling with the Mouse

If a window has a *scrollbar* (see section 2.1), then the contents of the window can be moved. To accomplish this, you move the mouse to the appropriate arrowhead and *click* once with the left button. The double arrows can be used to move the contents one page.

3. Wlisp Programming

In this chapter, another perspective of Wlisp is described. It is important for those who would like to implement an application program that runs within Wlisp. First, the relationship between Wlisp and applications will be examined. Then, the general design principles of the window system and some functions of the window system kernel that might be helpful for the implementation are described.

To understand this chapter, a familiarity with **ObjTalk** [Lemke 85; Rathke, Laubsch 83] is necessary (at least for the basic concepts).

3.1. The Class Hierarchy of Wlisp

3.1.1. ObjTalk and the Window System

An object-oriented representation was used to create the window system. It is especially suited for this system since a wide variety of window types exist having basic similarities. The language that serves this purpose is ObjTalk, an object-oriented extension of FranzLisp. ObjTalk supports an inheritance hierarchy of *classes* which allows multiple superclasses (i.e., a class can have several superclasses from which it inherits slots and methods). The underlying constructs that are the same for every window type, are defined in a class at the root (display-region) of the inheritance hierarchy, so that they are inherited by all windows. Every application can then select from the predefined classes those that are the most suitable for defining and modifying specific subclasses for an application.

There are two types of classes in the window system:

1. Classes from which windows can be instantiated have display-region as their superclass (either directly or indirectly), and therefore, inherit all the characteristics necessary for a window.

2. Mixin-classes define a specific aspect of a window. By including a mixin-class as a superclass, a window receives the characteristics of this mixin-class.[6]

3.1.2. The Predefined Window Classes

The class hierarchy of the window system consists of a main component (display-region), which must be the *superclass of every window class*, and several mixin-classes, which are independent of each other and can be used as additional superclasses of a user-defined class to expand the behavior of this class. Also, various predefined window classes exist that can serve as the foundation for application oriented window classes (see Figure).

A more complete description of the individual classes can be found in section .

display-region	A display-region is an arbitrary rectangular section of the screen. In this class, the basic attributes of windows and the methods for most of the messages that can be sent to windows are defined.

[6]The inheritance mechanism is used in two different ways: for the *changing* of characteristics and for the addition of mixin-classes to *extend* the attributes.

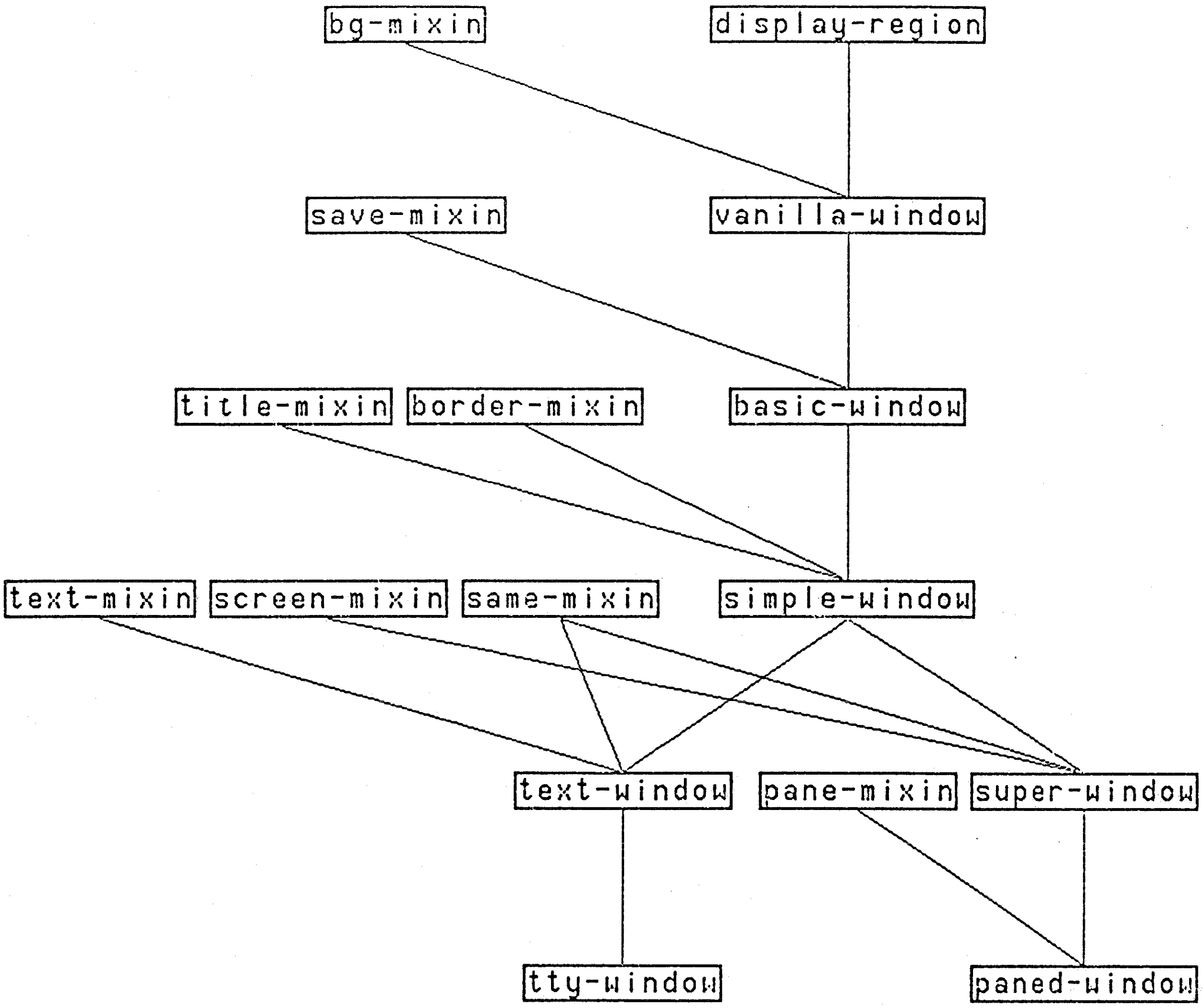


Figure 3-1: The class hierarchy of the window system.

`bg-mixin`	Defines the mapping of various window characteristics on the region-concept of the Bitgraph terminal. Every window can then be treated like an independent Bitgraph terminal for which all the text and graphic features can be supported. Only through the addition of this mixin-class to `display-region` do you create a class from which windows in the true sense can be instantiated.
`save-mixin`	Allows for the saving of the contents of overlapped windows in the Bitgraph terminal memory, so that the redisplay of these windows or window portions is substantially accelerated.
`title-mixin`	With this mixin, every window receives a title bar in which the name of the window is displayed. The default for the `title` is the `pname` of the corresponding ObjTalk instance, but can be modified after an instantiation.
`border-mixin`	Equips a window with a frame. The default for the width of the frame is (currently) `2`, but can be reset by changing the `border-size` slot.
`scroll-mixin`	Supports the scrolling of the window contents by using the mouse by displaying two *scrollbars* at the left and the bottom corners of the window (see also section).
`text-mixin`	Allows the specification of window size in lines and columns (`lns` and `cls` methods) and forces the fitting of the window size (measured in pixels) to a multiple of the character height and width.
`same-mixin`	Makes sure that the output region for an application is always equal to the inner region of the window (see also section). This mixin is only meaningful when used with the `bg-mixin`.
`screen-mixin`	The contents of a window may consist of other windows (`display-regions`). This class contains the necessary methods and slots for the handling of these nested windows.

`pane-mixin (superc screen-mixin)`

Enables a symbolic description of the specifications and size of the nested windows of a window (the actual size and position of the nested window will then be dependent on the size of this window) and therefore, constitutes the foundation for *split-screen* applications.

Next, the predefined window classes are described that can be used as the starting point for individual window types.

`vanilla-window (superc bg-mixin display-region)`

The simplest form of a window.

`basic-window (superc save-mixin vanilla-window)`

A window that can save its representation on the screen in the terminal memory.

`simple-window (superc title-mixin border-mixin basic-window)`

A window type that can be used as a starting point for most of the application oriented window types.

`text-window (superc same-mixin text-mixin simple-window)`
: Starting point for text oriented window types as opposed to the `simple-window`, which is to be used for graphic oriented systems.

`tty-window (superc same-mixin text-window)`
: Can be used for teletype-like interactions (the Lisp-toplevel-window is, for example, a `tty-window`). Since this window type modifies the behavior of the refresh command, it should not be used as the superclass of other window types.

`super-window (superc screen-mixin same-mixin simple-window)`
: A window type that can contain nested windows.

`paned-window (superc pane-mixin super-window)`
: The foundation for *split-screen* applications.

3.2. The Programmer's Model of the System

For the implementation of an application that is to run within the window system, several characteristics of the window system should be noted. A general model of the window system will be described in this section, and from this model the basic factors for the interface between the application and the window system will be derived.

The following will expound on the basic model of the window system. The following are some concept definitions:

`objectregion`	Every application system has a rectangular region where it sends its output. For the application system only the size (i.e., length and width) of this region is relevant. This region constitutes the external world for the system. For character-oriented applications, the *upper left* corner of this region is also the *home-position* of the cursor and the `objectregion` constitutes the boundaries (Bitgraph "margins") for the cursor.[7]
`origin`	This is the coordinate origin for graphic cursor positioning that normally corresponds to the *left bottom* corner of the `objectregion`. It can also be defined to enable a shift of the coordinate system relative to the `objectregion`.
`innerregion`	A *window* displays an `objectregion` or a portion thereof on the screen, i.e., it makes this portion of the external world of an application system visible to the user. The `innerregion` is also rectangular, and its position on the screen with reference to the `objectregion` is important.

The basic operations for windows (*changing window size, changing window position* and *changing the visible area of the window*) which change the relative position of these three regions to one another can be explained by the following model (see also Figure):

- **moving windows** (`move:`): `objectregion` and `innerregion` remain fixed relative to one another and are shifted relative to the screen.

[7] In regions of unspecified size, especially in graphic applications, the `objectregion` should not even exist. Since a value must be defined in the Bitgraph, the `objectregion` can, for example, be defined as (0 0 10000 10000) (the Bitgraph terminal only accepts coordinate values within the range of -16384 through +16383).

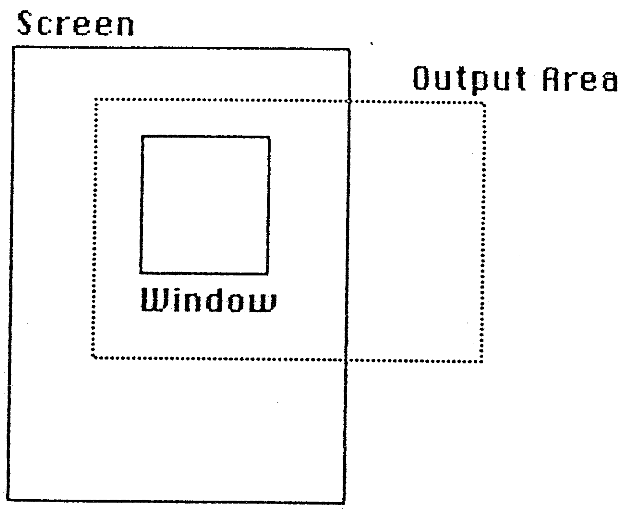


Figure 3-2: The relationship between the output region of an application system and its window.

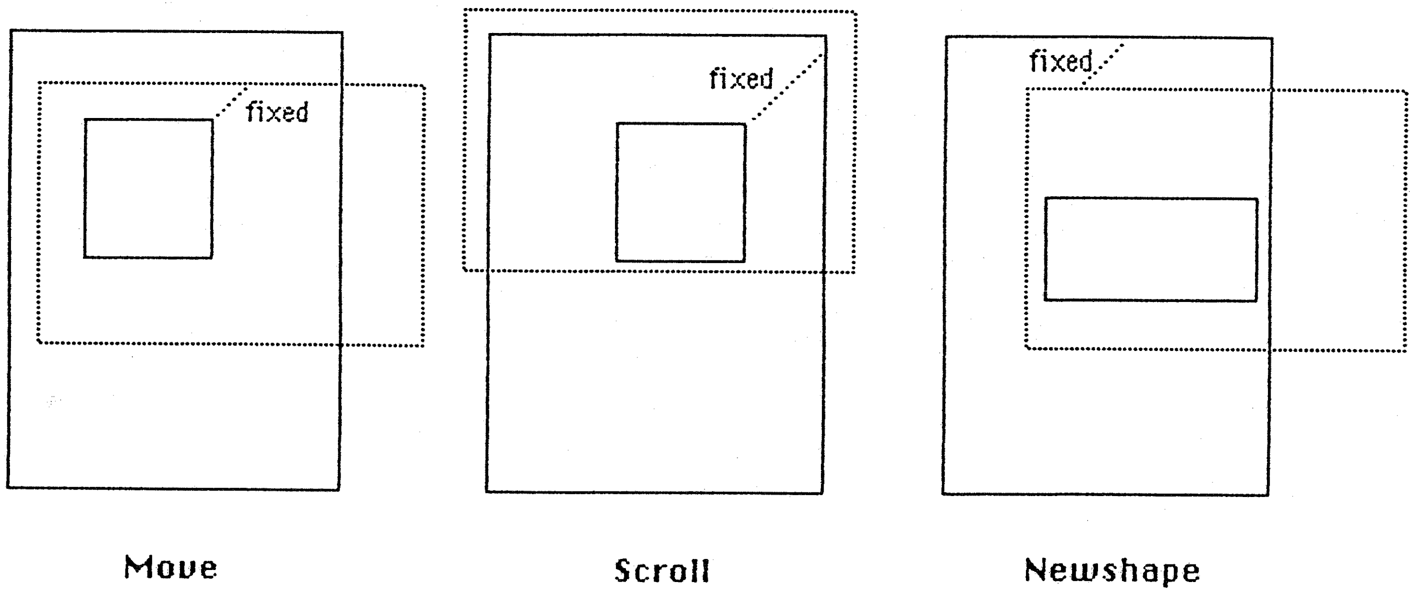


Figure 3-3: The three basic operations on windows.

- **changing size** (newshape:, reshape:): objectregion and the screen remain fixed relative to one another and the innerregion (including the frame) is newly defined.

- **shifting window contents** (scroll:): innerregion and the screen remain fixed relative to one another and the objectregion is moved.

With the above definitions, two distinct models can be developed that require the application system to have different levels of knowledge of the window system:

1. The objectregion is completely independent of the size of the window on the screen (i.e., the innerregion) that displays a portion of the objectregion to the user. In this way, the application program and the window system are loosely connected. The position of a window on the screen and its size, two aspects that can be changed by the user, do not have to be considered by the application. The objectregion remains constant during the modification of these aspects, so that the application relies on a constant sized output region, which considerably simplifies the implementation of the input/output interface.

2. The objectregion and the innerregion are always the same, i.e., when the user changes the size of the window on the screen, then the objectregion also modifies itself (see the description of the same-mixin in Section 3.1.2). In general, the change must be transmitted to the application, since, for example, the layout of its output region must be changed. Additionally, restrictions of the possible values of the size of the objectregion can exist that can have side effects on the window and therefore interfere with the intentions of the user concerning the divisions and form of the window on the screen. Because of this, a considerable restriction of the user-interface can occur.

3.3. Basic Window Characteristics

In this section, the basic window characteristics are described. These are defined in display-region and therefore every window has them. The characteristics of a window are established through messages (methods) that the ObjTalk class (the superclass of window classes) can understand. In the following descriptions, the method filters of the methods that are recognized by a window and the slots of a window instance are explained.

Also, an instance of the class display-region is designated as a window in the following, although several substantial characteristics are missing from this instance. Particularly, this type of window does not yet have automatic *clipping* or restriction of cursor positioning on its screen region. The statements with reference to the objectregion during the description of individual methods are meaningless for instances of display-region.

3.3.1. The Hierarchy of Windows

Every window is contained within a *parent window*. At the peak of this *hierarchy of windows* exists a *screen*, i.e., a window-like object, that in general represents an entire screen.[8] To make a window visible on the screen, the *parent window* must also be completely visible.

c-w ObjTalk Slot

[8]Until now only one *screen* exists, which has the name THE-SCREEN. The possibility of having many adjacent *screens* has not yet been implemented.

The value of the slot `c-w` is the reference to the *parent window*. By modifying this slot value, a window can be nested into another window or to the `screen`. After this change the window is not visible until a `totop:` message is given.

The size and position of a window are restricted to the objectregion of the parent window and are automatically adjusted to achieve this. If, however, the size of this region is changed (either through a size change of the parent window or by resetting the slot `c-w`) then the adjustment of the child window is limited to moving the window. Therefore, errors can conceivably occur in subsequent window operations.

3.3.2. Size and Position on the Screen

Every window has the slots `screenregion` and `innerregion` which define its size and position on the screen. The values of the slots are of the type `region`, i.e., a list of four numbers: the x and y coordinates of the left bottom corner, as well as the width and height (see also section). The numbers are pixel specifications and the position is relative to the `origin` or the bottom left corner of the `innerregion` of the parent window.

The `innerregion` describes the inside of a window; the `screenregion` defines the total area of the window on the screen. If one of the values is modified then the other value is automatically adjusted by adding or subtracting the size of the frame as a constant. A window without a frame and title bar will have the same value for both of these regions.

`innerregion` ObjTalk Slot

Position and size of the *inside* of a window. It specifies how much space is available to display information, and is therefore a relevant value for any given application.

`screenregion` ObjTalk Slot

Position and size of a window including the window frame, i.e., the entire area of the window on the screen. This value is mainly necessary for the internal system, however, if the size and position of a window is modified by using the mouse, the new value for `screenregion` is specified.

For the modification of position and size of the window on the screen, a variety of procedures exist[9] (See also section 2-2 and Table 3.2):

3.3.2.1. Moving a Window

`move:` ObjTalk Method

The new position is specified by the user with the mouse.

`move: ,?x-pos ,?y-pos` ObjTalk Method

`x-pos` and `y-pos` designate the new bottom left corner of the entire window (i.e., of the `screenregion`), not the inside of the window.

Both methods leave the size of the window and its contents unchanged, i.e after moving, the same information in the window is visible as before.

[9]Size specifications are given in *pixel*-units

3.3.2.2. Changing the Size of a Window

When changing the size of a window, you can distinguish between methods that affect the entire window (including frame and title, etc.) and methods that refer to the inside of a window. By specifying a new window size, the position of the window on the screen can also be modified. In contrast to moving, the `objectregion` remains unchanged relative to the screen, i.e., besides a change in the window relative to the screen, you also produce a change in the visible portion of the `objectregion`.

<u>changing the `screenregion`</u>

`reshape:` — ObjTalk Method

The user can redefine a corner of the window.

`newshape:` — ObjTalk Method

The new size of the window is specified by the user with the mouse.

`screenregion = ,?sreg` — ObjTalk Method

The new position and size of the window is specified by `sreg` which is of type `region` (see section).

<u>changing the `innerregion`</u>

`innerregion = ,?ireg` — ObjTalk Method

The new position and size of the window is specified by `ireg`.

`size = ,?width ,?height` — ObjTalk Method

These parameters specify the new height and width of the window. The window position remains fixed.

`width = ,?width` — ObjTalk Method

New window width; the *right* border of the window is moved in the x-coordinate direction based on the new width.

`height = ,?height` — ObjTalk Method

New window height; the *top* border of the window is moved in the y-coordinate direction based on the new height.

3.3.3. Coordinate Systems

There are two coordinate systems that are used at different places in the window system. The coordinates that are located in the slots `innerregion`, `screenregion`, `objectregion` and `origin` as well as the coordinates received from a *mouse report* are all relative to the left bottom corner of the screen. In contrast, every application has its own coordinate system whose origin is specified by `origin`, and additionally, if using the character-oriented cursor commands, by the upper left corner (home position) defined by the `objectregion` (margins). The conversion of coordinates between the two systems is achieved through two Lisp functions:

`make-local nr-list`

`nr-list` is a list of numbers whose first two elements are interpreted as the x- or y-values on the screen, and returns a list of numbers whose first two elements contain the transformed values for the coordinate system of the application.

Example:

```
      (make-local ,!innerregion) --> (x y width height)
```

returns the currently visible portion of the application program.

```
      (make-local mouse-report) --> (mouse-x mouse-y mouse-buttons)
```

if `mouse-report` is bound to a mouse report, you receive the report in coordinates of the coordinate system of the application.

`make-global nr-list`

converts application coordinates into screen coordinates.

3.3.4. Covering Windows

Of course only one window of several that occupy the same position on the screen can be visible at any one time.[10] Overlap between windows can only occur between those with the same parent window. All the windows underneath a window are ranked in an order that defines which window covers another. Windows that are located at the end of this ranking lie at the "bottom" of the others on the screen. To change the order of overlapping windows, there are three messages available to send to a window.

`totop:` ObjTalk Method

The referred to window is made entirely visible, i.e., it is brought to the top.

`tobottom:` ObjTalk Method

Covers the window with all the other windows that are located in the same place, i.e., it is brought to the bottom.

`bury:` ObjTalk Method

Removes the window from the screen and uncovers any windows below it as much as possible.

A window can be in three possible conditions in an overlapping situation.

1. It is not at all visible on the screen, i.e., it has not received `totop:`-messages.

2. It is on the screen, but may be totally, partially, or not covered by other windows.

3. The window is on the screen and is *guaranteed* to be totally visible (for example, by having received a `totop:`-message.

To recognize which of these conditions apply to a particular window, two methods exist:

`onscreenp:` ObjTalk Method

Returns `non-nil`, if neither itself nor one of its *ancestry windows* (i.e., its parent window and its ancestry windows) is buried.

`exposed?` ObjTalk Method

[10] Cyclical covering of many windows so that none are completely visible is impossible due to the implementation.

Returns `t`, when the window and all its ancestry windows are completely *visible*.

Since a window does not know anything about other windows also located on the screen, a management component in the parent window is necessary (see `screen-mixin`, section) to recognize overlappings and to manage the available space in addition to the methods and slots of a window.

3.3.5. Processing Mouse Reports

When no application program is active, then the window system recognizes `button-down` events from the mouse. It determines the window that the mouse was in as the button was pressed, and sends an appropriate message to this window. The parameter of this message (`rep`) is a mouse report that describes the mouse position and conditions of the mouse buttons at the time of the mouse occurrence. A mouse report is implemented as a list of three numbers: `x-pos`, `y-pos` and `button-state`. These describe the coordinates of the position of the mouse as well as the conditions of the three buttons (for a detailed definition see the Bitgraph manual). The button related messages and slots that are understood by a window are the following:

`left-button-down: ,?rep` ObjTalk Method

The window is activated and made completely visible[11].

`right-button-down: ,?rep` ObjTalk Method

Displays the appropriate menu (a pop-up menu that is internally bound to the *slot* `top-menu`) and the user can then select a window operation.

`top-menu` ObjTalk Slot

An appropriate menu (usually the `window-menu`) is bound to this slot that displays all the window operations for selection by the user. It is activated by pressing the right mouse button when the mouse is found inside the window.

3.3.6. Appearance of Windows

The appearance of a window on a screen, in general, consists of two components: the *window content* and the *window frame*.

- The *window frame* itself can consist of various parts or may not be present at all. This depends on which of the mixin classes (e.g. `title-mixin`, `border-mixin`, `scroll-mixin`, see section 3.1.2) are given as superclasses of a window.

- The *window content* displays the information of an application system. The background is by default white, but can be set to an arbitrary pattern by modifying the `background` slot.

`background` ObjTalk Slot

The value of the `background` is either `nil` or a list of the ascii number of a character and a font number that designates the character that is to be used for the filling of the

[11] Actually this is enacted by the `left-button-up:` method, which is called by `left-button-down:`. The window is activated after releasing the button.

background.[12]

If a window is to be made totally visible on the screen, then the frame and the window contents have to be created. The frame is created through internal system methods depending upon the various mixin classes, which are superclasses of the referred to window. For the establishment of the window contents, the window system can only offer slight support. Since the window system itself has no knowledge about the contents of a window, the *default*-action of clearing (filling with background) the corresponding region of the window contents is executed.

update-contents: ,*ar ObjTalk Method

ar is a list of regions (see section) given in *local* coordinates that define the display region of the window.

Normally, the programmer defines his/her own update-contents: method, in order to create the contents of a window (more information is given in Section). At the time of execution of this method, the Bitgraph terminal is already in a suitable state (the affected window is already *activated*, see section), and only the output routines have to be executed. A user-defined update-contents: method must not activate the window again.

For the representation of title bars and frames, the two mixin classes, title-mixin and border-mixin, exist which have the class **margins-mixin** as their superclass.

title-mixin (superc margins-mixin) ObjTalk Class

Endows a window with a title bar that contains its name. The height of the title bar is determined globally through the LISP variable title-height which is the same for all windows (16 pixels).

title ObjTalk Slot

The value of the title is a string or a Lisp symbol. The pname of the window instance is used as the default value.

border-mixin (superc margins-mixin) ObjTalk Class

Provides a window with a frame of the same width on all sides. The width is defined through the value of the slot border-size.

border-size ObjTalk Slot

Specifies the width of the window frame in pixel units (the default value is 2).

In addition to these two classes, the mixin class scroll-mixin influences the form and size of the window frame (see section).

3.3.7. Saving Windows in the Bitgraph Terminal

The Bitgraph terminal offers the possibility of copying portions of the screen into the internal memory of the terminal and redisplaying these on the screen at a given time. This fact is taken advantage of in the window system to temporarily store windows that are partially or totally covered so that quick redisplay

[12] An example of this is the value of the global Lisp variable **backg**. Its symbol achieves "gray" background as in the window system for the XEROX Lisp Machines.

of the window can be achieved. However, after modification of the size of the window, the stored copy can no longer be used for redisplay (this is why after each size modification the copy in the terminal is "thrown away").

If a portion of a window or the entire window is to be made visible on the screen, then two ways of achieving this are available depending on whether the window has already been stored in the Bitgraph terminal:

1. If the window is stored in the Bitgraph terminal, then the window system can display the stored picture of the window on the screen.

2. If the window has not been stored, then the window must be newly established by the window system and the corresponding application program.

This behavior of a window is achieved by including the **save-mixin** as a superclass of the corresponding window class.

save-mixin ObjTalk Class

Allows for the temporary storage of the appearance of a window on the screen (window contents and frame) in a portion of the Bitgraph terminal memory. Through this the redisplay of partially or totally covered windows is substantially accelerated.

save: ObjTalk Method

Stores a window in the terminal if it is currently visible.

unsave: ObjTalk Method

Frees the picture of the saved window.

In general, both of these methods are called by internal system methods, but in special circumstances it may be necessary to give the programmer control over when the window should be stored or unstored.

If you wish to newly display a window, even though it has already been stored in the Bitgraph terminal, then either the message **refresh:**, **redisplay:**, or **display-contents:** can be utilized:

refresh: ,*flag ObjTalk Method

Performs a redisplay of the window (window contents and frame) from the host computer. If **flag** (an optional parameter) is available and equal to **nil**, then the redisplay occurs only if the window is on the screen.

redisplay: ObjTalk Method

In case the window is already completely visible, only the window content is newly established. Otherwise, the message **refresh: nil** is sent to the window. This message can be used, for example, when the layout of the active window was modified and the content must be newly established.

display-contents: ,*ar ObjTalk Method

Newly displays the content of the window (the portions specified through the parameter **ar**, a list of **regions**). Before sending this message, the window must already be totally visible (the message **exposed?** must return the value t) or the programmer must explicitly *activate* the window (see section).

3.3.8. Sending Output to a Window

A window in the Bitgraph window system does not have a representation in pixels or characters of its own contents, i.e., no corresponding bitmap or other data structure exists that stores the window contents of the window.[13] Instead various characteristics of the Bitgraph terminals are employed for support to output information to a window.

The Bitgraph terminal knows the concept of so called `Bitgraph regions`. Every *Bitgraph region* has its own series of data that describes the condition of the terminal. The main state variables of the Bitgraph terminal for the window system are described as follows:

Cursor position The actual position of the cursor in a `Bitgraph Region`.

`clipping-region`	The area of the screen outside of which no modification through output operations is possible.
`margins`	Restricts the movement of the cursor within the region specified through `margins`.
`origin`	The origin for graphic output operations.

Various other condition variables also exist that can have different values for every Bitgraph region (these are described in more detail in the Bitgraph manual).

If a window type has the class `bg-mixin` as its superclass, then every window of this type will be associated with a Bitgraph region. Several characteristics of the window are then assigned to the above described condition variables of the corresponding Bitgraph region. The following mapping then occurs: `innerregion` of the window maps to the `clipping-region`, the `objectregion` to the `margins`, and the `origin` of the window to the `origin` of the Bitgraph region.

`bg-mixin` ObjTalk Class

Associates a `Bitgraph region` in the Bitgraph terminal to the window and maps the `innerregion`, the `objectregion` and the `origin` of the window to the `clipping-region`, the `margins` and the `origin` of the Bitgraph region, respectively.

`objectregion` ObjTalk Slot

The output region of an application. The value of this slot is used in the Bitgraph terminal as the value of the `margins` attribute of the corresponding Bitgraph region.

`origin` ObjTalk Slot

The reference point for graphic output operations. The value is mapped to the `origin` attribute of the corresponding Bitgraph region of the window.

Before an application system can output information in a window, this window must be *activated* (i.e., the window must be totally visible and its corresponding Bitgraph region must be made the current active Bitgraph region) to ensure that the output really appears in the window and not somewhere else on the screen. At any point in time only one window can be *active*. There are four possibilities to activate a window:

[13] The transfer of the window contents from the host computer to the terminal in the form of individual pixels takes too much time (at a transfer rate of up to 9600 baud), to be carried out realistically.

`activate:` ObjTalk Method

The window is made totally visible and all of the subsequent output to the Bitgraph terminal is displayed inside the window.

`do: ,*body` ObjTalk Method

The window is activated and the expressions to which *body* is bound to are evaluated. During the evaluation, all the output goes to the window.

NOTE: During the evaluation, `self` is bound to the window itself.

NOTE: The expressions to which *body* is bound to are not compiled during compilation, but are interpreted at run-time. Therefore, you should use `with window` when possible.

`tmp-do: ,*body` ObjTalk Method

behaves like `do:`; but after execution of *body*, the previous active window is reactivated. `tmp-do:`, therefore, serves for temporary output of information to another window while output before and afterwards are sent to the present active window.

NOTE: The same remarks as for the `do:` method apply.

`with (window <window-name>) &rest <body>` Lisp Macro-Function

The `with` function (`with` is a macro) can also be used to activate a window. It works like the `tmp-do:` method with the exception that during the evaluation of `<body>`, `self` is not bound to the active window, instead it remains unchanged. In addition, `<body>` can be compiled (a thorough description of the `with` function is given in section).

3.3.9. Scrolling Window Contents

In order to *scroll* window contents through the use of the *scrollbar* and *mouse*, the corresponding window class must have `scroll-mixin` as one of its superclasses.

`scroll-mixin (superc margins-mixin)` ObjTalk Class

Allows the scrolling of window contents horizontally as well as vertically through the mouse and scrollbar. The window receives a scrollbar at both the left and bottom border. By "clicking" one of the symbols in this scrollbar with the mouse, the scrolling of window contents can be performed.

By "clicking" a symbol, an appropriate message is sent to the window. Altogether, eight different methods exist:

`scroll-up:` ObjTalk Method

Scrolls the distance of the slot `ver-amount` to the top.

`scroll-down:` ObjTalk Method

Scrolls the distance of the slot `ver-amount` to the bottom.

`scroll-left:` ObjTalk Method

Scrolls the distance of the slot `hor-amount` to the left.

`scroll-right:` ObjTalk Method

Scrolls the distance of the slot `hor-amount` to the right.

page-up: ObjTalk Method

Scrolls one window length to the top.

page-down: ObjTalk Method

Scrolls one window length to the bottom.

page-left: ObjTalk Method

Scrolls one window width to the left.

page-right: ObjTalk Method

Scrolls one window width to the right.

An implementation of scrolling is the scroll-mixin which requires the window to contain `bg-mixin` as a (indirect) superclass.

The eight different display methods call a central method that executes the scrolling. The parameters for the `scroll`: method are determined depending on the values of the two slots `hor-amount` and `vert-amount`.

`hor-amount` ObjTalk Slot

Number of pixels the window contents should be moved in a horizontal direction after the reception of the message `scroll-left:` or `scroll-right:`.

`vert-amount` ObjTalk Slot

Number of pixels the window contents should be moved in a vertical direction after the reception of the message `scroll-up:` or `scroll-down:`.

scroll: ,?dx ,?dy ObjTalk Method

`dx` and `dy` specify how far the contents should be shifted in the x and y directions.

Shifts the `objectregion` of the window for the specified region in the indicated direction and sends a `display-contents:` message for the not yet visible new portion to the window. Caution is exercised so that the window (`innerregion`) does not extend over the border of the region for the application (`objectregion`).

If you need a different implementation of scrolling for an application system, then there are two possible modifications of scrolling behavior:

1. Defining your own **scroll: ,?dx,?dy** method, where `dx` and `dy` are specified as explained above.

   In this case, the refresh of the window content must be done in this method (and possibly the `objectregion` must be shifted).

2. Replacing the eight display methods with your own implementation.

   `hor-amount` and `vert-amount` are no longer considered in this instance and only the execution of the scrolling through the mouse is utilized.

3.3.10. Windows in Windows

As described in section 3.3.1, a window consisting of other windows is possible in Wlisp. The mixin class `screen-mixin` manages these contained windows (i.e., representing the covering structure and recognizing modifications to it).

`screen-mixin` ObjTalk Class

Manages the contained windows of a window and sends them messages to partially or totally redisplay their contents by modifying the overlay of the windows.

The only method of this class meaningful for the applications programmer is the `inside-region:` method.

`inside-region:` ObjTalk Method

This method returns as its results a `region` (see section) to which the position and size of all contained windows is restricted. This, in general, is the `objectregion` of the parent window. However, if a contained window has the mixin class `save-mixin` for a superclass, then this contained window must be inside the `innerregion` of the parent window, since the saving of the window in the memory of the Bitgraph terminal (see section 3.3.7) does not function perfectly. In this case, `inside-region:` must return the `innerregion` (i.e., the method `inside-region:` must be redefined).

The use of the `screen-mixin` lends itself especially to applications for which the contents of a window consists of freely positionable graphic objects that have a rectangular form and can possibly cover each other. These objects should be instances of the class `display-region` (or a subclass thereof), in order to allow the coordination between windows and the contained objects.

Should the contained windows of a window split the available space amongst themselves dependent on the size of the parent window (split-screen application), then the window class should be made an instance of `Wpane` instead of `class`. Example:

```
(ask Wpane renew: example-window
     (descr (contents: (pane-of simple-window (title = CONTENTS)))
            (survey:   (pane-of simple-window (title = SURVEY)))
            (menu:     (pane-of menu (items = (add delete show)))))
     (pane-description
       (dummy 0.4 h: (menu: 100) (survey: rest:))
       (contents: rest:)))
```

Example 3-1: Use of the `pane-mixin`

A window of the type `example-window` contains three windows. The bottom window (`contents:`) is as wide as the parent window and the other two are adjacent to one another and occupy 40% of the height of the parent window. The upper left window (`menu:`) has the constant width of 100 pixels; the other (`survey:`) occupies the remaining space up to the right border of the parent window (see Figure).

The value of the slot `pane-description` must adhere to the following syntactic form:

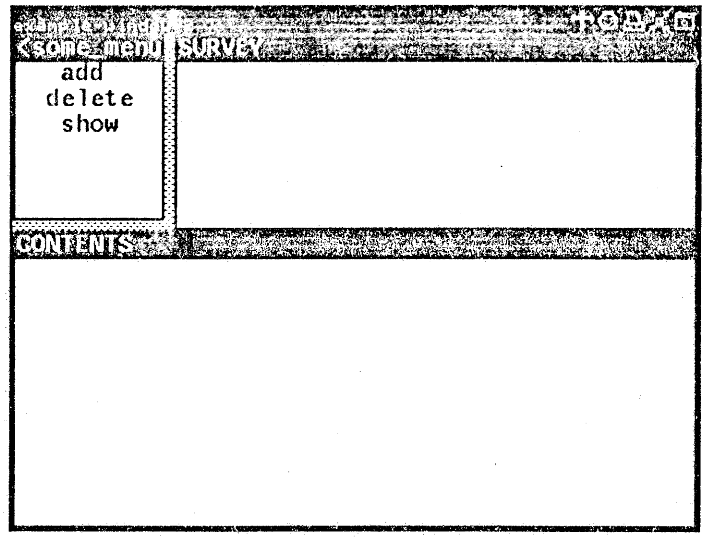


Figure 3-4: An example of windows within another window.

```
<pdescr> ::= ((<name> <size> [<direction> . <pdescr>])
             ...)

<name> ::= <symbol>

<size> ::= <pixels> | <percent> | rest:

<pixels> ::= <fixnum>
<percent> ::= <flonum>

<direction> ::= v: | h:
```

The value of `<size>` specifies the size of the subwindow in the current direction being divided (the first division of the window is a horizontal cut).

If the size specification of the description is a whole number, then it represents the *actual* size of a subwindow in pixel units; if the size specification is a decimal number (between 0 and 1), then it describes the relative size of the subwindow with reference to the total available space. The symbol `rest:`, if used for size specification, means that the associated window should use up the remaining space available.

The `<direction>` (`h:` for dividing in the horizontal direction, `v:` for dividing in the vertical direction) specifies in which direction the space allocated to the dummy pane should be subdivided.

3.4. The Use of Menus

In the window system a menu is a window type with additional features that allow the mouse to be used for menu selection. The different types of menus themselves build a small class hierarchy. A menu contains several keywords or symbols, and by using the mouse, one of these menu items can be selected, which is then returned as the result.[14] The menu items are listed vertically in the window; the width of the menu will automatically be fitted to the length of the longest individual item and the height will correspond to the number of items.

3.4.1. The Basic Menu Class

The basic characteristics of a menu, which are defined in the class `menu` are described as follows:

`menu (superc text-window)` ObjTalk Class

This is the root class for menus, in which the characteristics for selection using the mouse as well as the representation of the items are defined.

The actual slots and methods of `menu` will now be described:

`items` ObjTalk Slot

Contains a list of menu items. Every menu item is either a symbol or a list whose first element is used for the display of the item on the screen.

`pname-selector` ObjTalk Slot

The value of this slot is a Lisp function that uses a menu item as an argument and returns a symbol or a string, which can then be used for the representation of the item on the screen. By setting this slot value, other structures can be used to represent an item (e.g., ObjTalk objects could also be used as menu items with an appropriate `pname-selector`, see Example 4.

`add: ,?item` ObjTalk Method

Adding an item to a menu. If the item is already contained in the menu (comparison with `eq`, not with `equal`), then it is not added.

`delete: ,?item` ObjTalk Method

Deleting an item from the list of items. The argument `item` must be `eq` (*not only* `equal`) to the deleted element of `items`.

Methods for menu selection with the mouse:

`select-with-mouse: ,*pos` ObjTalk Method

This method allows menu selection with the mouse. This is actually an internal method that should not be sent to the menu externally. If you want to modify its behavior during menu selection (e.g., to carry- out a side effect dependent on the selected item, see Example 2), then this method must be extended. The parameter `pos` should be passed on, not changed, during modification.

[14] Since menu selection can be interrupted at any point, the possibility exists that `nil` will be returned as the result of the selection.

```
(ask class renew: active-menu
     (superc menu)
     (methods
       (select-with-mouse: ,*pos =>=>
         (let ((sel ,!(select-with-mouse: ,@pos)))
           (and sel
                (eval (cond ((symbolp sel) (list sel))
                            (t (cadr sel)))))))))
```

Example 3-2: Extending the **select-with-mouse:** method

`select:` ObjTalk Method

Allows menu selection with the mouse at the position on the screen on which it is currently located.

`select: at: ,?x ,?y` ObjTalk Method

Shifts the menu to the coordinates (x,y) and carries out the menu selection.

`select-at-mouse-pos:` ObjTalk Method

Brings the menu to the position at which the mouse is located and carries out the menu selection.

3.4.2. Other Predefined Menu Classes

Three other menu classes which are subclasses of `menu` have been defined:

`pop-up-menu (superc menu)` ObjTalk Class

Menus that appear on the screen at the time of menu selection and disappear again after completion of the selection (before the return of the selected item, however).

`active-menu (superc menu)` ObjTalk Class

If an item was selected, then it is immediately evaluated and the result of the evaluation instead of the item itself is returned. An item must either be the name of an argumentless Lisp function, or a list whose first element is used for the external representation and whose second element can be evaluated with `eval` (see Example 2).

`active-pop-up-menu (superc active-menu pop-up-menu)` ObjTalk Class

A combination class of `pop-up-menu` and `active-menu`.

3.4.3. An Example of Menu Use

This example of menu use is based on the definitions given in Example 3.[15] This right-button-down method is useful when a menu is associated with a specific type of object, i.e., when the menu contains messages or commands to be sent or applied to these objects, respectively.

[15] This message is sent to a window if the right mouse button was pressed while in this window.

```
(ask class new: ...
     (descr (top-menu (default some-menu)))
     (methods
       ...
       (right-button-down: ,?rep =>
         (let ((oper (ask ,,!top-menu select: at:
                          ,(car rep) ,(cadr rep))))
           (and oper (ask self ,(concat oper ":")))))
       ...
       ))
```

Example 3-3: Use of a menu.

The slot `top-menu` has a specific menu as its value. This menu is moved to the actual mouse position during selection. By adding a colon to the item (a symbol) returned from the menu, the name of a method is generated and this message is then sent to the window.

The second example (Example 4), which is similar to Example 3-2 on page 28, shows how you can easily modify the characteristics of a class of menus through the definition of a subclass of `menu`.

```
(ask class renew: object-menu
  (superc menu)
  (descr (pname-selector (default 'get-object-pname))
         (message-to-send (default 'totop:)))
  (methods
    (select-with-mouse: ,*rest =>=>
       (let ((s ,!(select-with-mouse: ,@rest)))
         (and s
              (ask ,s ,,!message-to-send))))))

(defun get-object-pname (obj)
  (ask ,obj pname))
```

Example 3-4: Extending the class `menu`

The `items` slot of an `object-menu` is a list of ObjTalk objects (in this case, objects of the type `window`). The `pname-selector` is a function that when applied to an object returns its name. If an object is selected, then a message is sent to it which is contained in `message-to-send`.

3.5. Defining Window Classes for an Application

To implement an application system that is to run in the window system, two different methods exist. In one case, the application is relatively isolated from the window system, and the only connection between them is the fact that the print function of the application returns its output to the associated window (and the echo of the user input is also sent to this window). The associated window of an application then corresponds to (in UNIX terminology) the standard input and standard output device. Examples of this extension in the window system itself are the `prompt-window` and the implementation of the Lisp Trace for the window system.

In the other case, the application and window system are related much more closely. The application can be thought of as a *screen and structure-oriented editor* for specific objects or for their external representation. The application program, in a sense, "lives" in its window. By *creating* a window, editing of the contained objects is *begun*; by *leaving* the window, the application running within is *interrupted*; by *activating* a window, it is *continued*; and by *deleting* (`kill:`) the window, it is *terminated*.

In both cases, several considerations must be taken into account during implementation.

3.5.1. Defining an Application Specific Window Type

Although several predefined window classes are available in the window system (e.g., **simple-window**, **tty-window**, **text-window**, see section 3.1.2), these usually need to be modified (a reason for this is given in section . They are to be regarded as the basis for application oriented window types. For this reason, you should define an object class that receives at least **display-region** or one of the predefined window classes as its superclass (these should always appear *after* all mixin classes in the list of superclasses).

3.5.2. Redisplay of the Window Contents

This subsection describes how the window contents are newly displayed through support from the application and how this output is postponed in case of several modifications of the window.

3.5.2.1. To Output the Window Contents

The window system or the individual predefined window types have no knowledge about the information represented in a window. To be able to redisplay the window contents, e.g., after a modification of the window size or after shifting of the window contents, support from the application system is necessary. For this purpose, every window must understand the message **update-contents:** which is always sent to a window from the window system when a portion of the window contents must be newly displayed.[16] Since most of the applications have a representation of the information to be displayed in the window for their own purposes, the implementation of this method is usually relatively easy.

Example 5 shows a simple implementation of **update-contents:** for menus. This implementation, however, always redisplays the entire output region of the application, since the argument of **update-contents:**, a list of the regions to be redisplayed, is ignored.

```
(ask class new: simple-menu
  (superc simple-window)
  (descr (items (default nil)))
  (methods
     (update-contents: ,*regions =>=>
        (ask self update-contents: ,@regions)
        (cursor-position 0 0)
        (mapc '(lambda (item) (princ item) (terpri))
              ,!items))
  ))
```

Example 3-5: Simple Implementation of the Method **update-contents:**

In Example 6, the argument of the message is considered and only the necessary portions are newly displayed.

[16] Every window inherits a simple definition of this method (the portion to be newly represented is filled with a background) from **display-region**.

```
(ask class new: my-window
  (superc simple-window)
  (descr (items (default nil))
  (methods
     (update-contents: ,*regions =>=>
        (ask self update-contents: ,@regions)
        (mapc '(lambda (region)
                  (ask self update-region: ,region))
              regions))
     (update-region: ,?region =>
        (cursor-position 0 (beginning-of-region region))
        (mapc '(lambda (item)
                  (princ item)
                  (terpri))
              (items-in-region region ,!items)))
  ))
```

Example 3-6: A Better Implementation of the Method **update-contents:**

Whether or not the argument of `update-contents:` is considered, is not important to the window system. This decision is based on the quickness or the ease of implementation. During scrolling, above all, the redisplay of a section of the window contents should occur quickly so that the user does not perceive it as disadvantageous.

3.5.2.2. Delaying the Redisplay

The redisplay of the complete or partial window after, for example, changing of the window size, only occurs if the slot `immediate-refresh?` does *not* have the value `nil`. By setting the slot to `nil`, such a modification can be suppressed. The application program itself is then responsible for later resetting the slot to `t` and for redisplaying the window with the modified values. A convenient way to do this is by using the macro **with (immediate-refresh nil) <body>** (see section).

```
(ask class renew: mywindow
  (superc text-window)
  (methods
     (new-lns&cls: ,?lns ,?cls =>
         (with (immediate-refresh nil)
               (ask self lns = ,lns)
               (ask self cls = ,cls)))))
```

Example 3-7: Delaying the Redisplay through `immediate-refresh?`

In Example 7, the automatic redisplay after reception of the `lns`-message is suppressed and only accomplished after dealing with the `cls`-message, since the slot `immediate-refresh?` has the value `nil` during processing of the two messages. The redisplay of the window after changing its size (as well as setting `immediate-refresh?` back to `t`) is automatically accomplished by the `with` function.

If the slot `immediate-refresh?` has the value `t` during the processing of the messages `lns` and `cls`, then the window would be redisplayed twice.

3.5.3. Changing the Reaction to Mouse Reports

Many application systems, in addition to a `pop-up-menu` for window specific operations, make available to the user a `pop-up-menu` for application specific operations (i.e., operations to the window contents). Since the user should receive both menus by pressing the right button of the mouse, which one you receive depends on the mouse position. If the mouse is in the inside window, then the user receives the menu with application specific operations; if the mouse is in the frame of the window (generally in the title bar of the window), then you receive the menu with the window operations.[17]

```
(ask class new: mywindow
   ...
   (descr
      (application-menu (default mywindow-application-menu)))
   (methods
      (right-button-down: ,?rep =>=>
          (cond ((region:containp (report:x rep) (report:y rep)
                                   ,!innerregion)
                 (ask self application-selection: ,rep))
                (t (ask self right-button-down: ,rep))))
      (application-selection: ,?rep =>
         (let ((oper (ask ,,!application-menu
                          select: at: ,(report:x rep)
                                      ,(report:y rep))))
           (and oper (ask self ,oper))))
))
```

Example 3-8: Extending the method `right-button-down:`

Example 8 demonstrates how the method `right-button-down:` can be extended to give the user the option of receiving both the window specific and application specific menus.

3.6. Meddling with the Window System

Several possibilities exist for application systems to interact with the internal mechanisms of the window system, primarily to allow the user better access to the Bitgraph terminal and to take advantage of its features. This mainly concerns being able to store or load options (e.g., loading further fonts) locally in the Bitgraph terminal.

3.6.1. Global Adjustments

During the (re)initialization of the Bitgraph terminal, global adjustments of the terminal are done by the window system. Additionally, the list of forms that is bound to the Lisp variable `reset-all-hook` is evaluated. Using the Lisp function `ws:eval-when-reset-all` (nlambda) arbitrary evaluable forms can be specified to be executed during start-up of the window system and by the `reset-all` command (if the window system has already been started up then the forms are executed anyway). With this, for example, additional fonts can be loaded into the Bitgraph that are also reloaded with every `reset-all`.

3.6.2. Window Specific Adjustments

In order to specify window specific commands for the Bitgraph (e.g., another symbol for the cursor), two possibilities exist. Both adjustments are automatically executed during the initialization of the window and after `reset-all`.[18]

[17]Keeping this convention consistent through all application systems leads to a uniform user-interface with respect to the activation of `pop-up-menus` and therefore, should be considered by every programmer of an application system.

[18]These options, like output in a window (see section 3.3.8), are implemented using the `region` concept of the Bitgraph terminal, and therefore, are only available if the mixin class `bg-mixin` is a superclass of its window class.

1. Adjustments that should hold for every window of a window class can be specified by extending the *method* **bginits:**.

2. Adjustments that are to be different for specific windows of the same type can be specified by setting the *slot* `bg-inits` (whose value must be a list of evaluable forms).

Adjustments for the entire class are done first, followed by the adjustments defined for every individual window.

3.6.3. Start of the System

With the function **ws:eval-when-winit** (nlambda) arbitrary forms can be specified that should be evaluated at startup of the window system (i.e., immediately after calling **winit**). Additionally, user specific files or commands for the window system that can only be executed after initialization can be directly entered in the source file of the application.[19]

3.7. Helpful Functions

In this chapter, several helpful functions will be described that are used for the implementation of the window system, and that also can be used in an application system program.

3.7.1. Temporary Modifications of Values

The problem of needing to temporarily specify a new value for a parameter that subsequently should be replaced by the original value comes up during the utilization of the features of the Bitgraph terminal as well as at other times. This applies, for example, to the temporary use of a different font, to adjustments of the parameters of the Bitgraph mouse, and also to the short-lived use of a window for output.

For this purpose, the Lisp macro **with** is available that allows a series of Lisp expressions to be evaluated in an environment in which these temporary adjustments are valid through the specification of keywords and their appropriate temporary parameter. After processing of the instructions and even at the event of an error or a nonlocal jump[20], these adjustments are undone.

with (keyword$_1$ val$_1$) ... (keyword$_n$ val$_n$) &rest <body> Lisp Macro

The argument of **with** is a list of *keyword-value pairs* followed by a list of Lisp s-expressions. The value of the individual keywords can also be modified (many times) within the **with** function. If a *keyword-value pair* does not have a value, then the adjustment valid before the call of **with** is valid inside the **with** function.

The keywords have the following meaning:

tyimode (value is 0 or 1)

[19]If these instructions themselves would be included in the source files, then they would be executed during the loading of the files. This could lead to errors if at the time of loading of the files the window system was not already initialized.

[20]More on this can be found in the FranzLisp manual under the function **unwind-protect**.

	Adjusting the `tyimode` for reading a symbol from the terminal without echoing[21] (value = 1) or with echoing[22] (value = 0).
`window`	An arbitrary window object can be specified for displaying the output that is generated during processing of the *body* (see also section 3.3.8).
`mouse`	The value is a list of the mouse adjustments as stated in the Bitgraph manual (page 49, *Set Pointer Parameters*). Through this, for example, a temporary restriction of the reaction of the mouse to specific buttons is possible.
`context`	Executes `push-context` before processing and `pop-context` afterwards in the Bitgraph terminal (see Bitgraph manual page 42 and 43) and serves for the simple utilization of other fonts or mouse forms. .
`immediate-refresh`	Allows the temporary prevention of the redisplay of a window for, for example, a series of size modifications of a window (see section 3.5.2.2).
`in-region`	The parameter is the number of a `bitgraph-region`. This keyword is necessary for the implementation of the window system. Its use for applications is only possible sometimes, since knowledge of the state of the Bitgraph terminal will temporarily be invalid for the window system and this may lead to consistency problems.[23]

Good examples of the use of the `with` function are the implementations of the menu selection with the mouse, the functions that follow movement of the mouse and those which change the size of windows.

3.7.2. The Lisp Data Type Region

For the internal representation of rectangular areas, a data type `region` is available that is implemented using the FranzLisp Record Package. A `region` consists of a list of four numbers that specify the x and y coordinates of the bottom left corner as well as the height and width. Besides the actual interpretation functions for portions of the `region`, various functions for determining the intersection or difference between `regions` as well as a predicate for the containment of a point in a `region` are available.

`region x y width height`
creates a `region` with the associated values.

`region:x region`, `region:y region`, `region:w region`, `region:h region`
these functions return one of the four components of a `region`.

`region:endx region`, `region:endy region`
calculates the coordinate of the right or upper border of a `region`.

`region:containp x y region`

[21] The *ECHO* and *CRMOD* modes of the tty driver are turned off, the *CBREAK* mode is turned on.

[22] The *ECHO* and *CRMOD* modes of the tty driver are turned on, the *CBREAK* mode is turned off.

[23] No window switch may occur, especially in the instructions of the corresponding `with`.

returns t if the point (x, y) lies within the region; otherwise returns nil.

`region:intersectp reg1 reg2`
returns t if the two regions overlap; otherwise returns nil.

`region:intersect reg1 reg2`
calculates the intersection of two regions which defines another region.

`region:difference reg1 reg2`
returns a list of up to four section-regions that describe the range of reg2 that is *not* included in reg1 (e.g., reg2 *minus* reg1).

Since during the determination of the difference between two regions a list of regions can result, and must be processed, a data type area is available. This consists of a list of regions and can, for example, be used to describe the currently visible portion of a window. For area, intersection and difference are also defined and each returns a list of regions, i.e., an area, or nil.

`area:intersection area1 area2`
calculates the intersection area of two areas.

`area:difference area1 area2`
returns the portion of area1 different from area2 (caution: a different parameter order than above).

3.8. Using the Mouse

There are two methods of using the mouse in an application program that differ in their programming complexity. In the simple case, mouse reports are read by the window system itself, interpreted, and then sent to the appropriate window as a button-down message. This, however, functions only if the window system is waiting for input at the time the mouse report is made (e.g., the toplevel window is waiting for the next symbol from the terminal). How the button-down messages can be appropriately processed by the window or the associated application is described in section 3.5.3.

The complex case occurs if the application system itself reads input from the user from the terminal. Then a mouse report can happen at any point during the reading of the user input, and the application must then either ignore the report (it must at least be read) or the application must interpret it itself in order to use it as input from the user. In this case, a check must be made to see if the mouse is contained in the window, since the mouse report should otherwise be processed by a different window.

3.8.1. Interpreting Simple Mouse Reports

Simple mouse reports should be interpreted as independent occurrences from each other. In contrast to these are reports that are continuously created (dependent on the settings of the Bitgraph mouse), and that continuously monitor the mouse by an application program (see also the following section).

To recognize mouse reports in the input stream, a character by character interpretation of the input is required.[24] A mouse report in the input stream is denoted with the ESC symbol and is followed by a coded series of symbols representing the mouse status.

To read this series of symbols, the following function exists:

`read-pointer-report` Lisp Function

reads a mouse report from an input stream and returns a list of three numbers that represent the x and y coordinates of the mouse and the status of the mouse buttons. Before calling this function, the ESC symbol that has already been read should be returned to the input stream using the function **untyi**.

During processing of a mouse report received in this manner, it should be checked whether it was meant for the application or for another window. To recognize in which window the mouse was contained at the time the mouse report occurred, the `which-window?` function exists.

`which-window? x-y-list` Lisp Function

the result is a list of the window and the x-y-list that contains the point (x, y). If the point is on the background the object **THE-SCREEN** is used instead of a window.

If the application decides that the received mouse report is meaningful to it, then it can proceed with the processing thereof. If the application does not want to handle the mouse report, because the user wants to do something else, then it should return this report to the window system, interrupt or end its processing (for this, several cleanup tasks may need to be completed), and return the control to the window system.

Returning the mouse report to the window system can be achieved using the Lisp function `enqueue-report`.

`enqueue-report rep` Lisp Function

returns the mouse report to the window system for further processing.

If the window system receives control from an application system, then it checks to see if unprocessed mouse reports need to be handled. If there are some, then these are sent to the windows using the appropriate messages which (the windows) then react according to their own methods (i.e., by activating a specified menu of the window or application system).

3.8.2. Interpreting Continuous Mouse Reports

For some applications (for example, drawing with the mouse), the main task could be monitoring the mouse to react immediately to every change in its position. For this, the mouse can be adjusted so that at every modification of its position by a certain distance, it will create a mouse report (see Bitgraph manual, page 49, *Set Pointer Parameters*). The Lisp function `read-pointer-report` mentioned above can be used to read these reports. However, the case where reports are created faster than they can be processed by the application system can occur. This can result in the overflow of the input buffer of the operating system or considerable delay in the reactions of the application to the input of the user. To prevent this from happening, the Lisp function `read-last-pointer-report` should be used for reading.

`read-last-pointer-report` Lisp Function

returns the *last* mouse report contained in the input buffer of the host computer at the time of the call of the function. The mouse reports created sinced the last call of the

[24] e.g., use the Lisp function **tyi** and set the *tyimode* to *1* (using the function **tyimode** or the **with** macro with the keyword (**tyimode** 1) for reading characters without echo).

function are ignored.

By using this function to read mouse reports, not all movements of the mouse are processed, but the user has the option to match their transmission rate to the processing speed of the system and thereby, avoid possible errors[25].

Similar to the interpretation of simple mouse reports, the application system must recognize when the user wants to end the communication with the currently active application (for example, when the mouse leaves the application window or a mouse button is pushed). The special adjustments of the mouse must then be canceled before returning control to the window system (see the function **with**, section 3.7.1).

[25]The necessity of this unsatisfactory processing of continuously created mouse reports results from the configuration terminal - host computer with a relatively slow connection in between and the limited processing speed on the basis of the multiple user operating system of the host computer.

Appendix I. Start-up of the System

I.1. Call from the UNIX Shell

Two options are available for calling Wlisp:

1. Call of `wlisp` at the *shell* level:

   A new Lisp system is started that contains the window system. `wlisp` loads the Lisp-Init-File .lisprc.

2. Loading of the file `window.l` in an already running Lisp system:

   This way, the code and the objects for Wlisp are loaded. The window system must be initialized by calling `(winit)`.

I.2. Compiling

If you want to compile Lisp code that should run in the window system, then

```
(include-file 'window window:dir)
```

must be at the beginning of the source file. During loading of this file, the window system is automatically loaded if it has not already been loaded.

Appendix II. The Use of Icons

As implemented and summarized by Michael Herzceg.

Warning: this appendix describes an outdated version!!!

II.1. The Meaning of Icons

Icons are representations of objects.

In the following, an icon is defined as a representation of one or more objects on the screen. Represented objects can be arbitrary objects of the application system, for example, documents, objects of the dialogue system, or the list of previously executed actions. The concept "object" until now has not been truly defined and consisted of an intuitive idea of an object that "you can touch" as we are used to doing in our environment. This metaphor will be retained. Yet, actions (functions, operations) should also be thought of as objects that can be represented by icons. To prevent misunderstandings, these will be called **function icons**.

The objects to be represented on the screen are often large and complex. You should not expect to see every object completely and in detail on the screen. Two procedures are available to solve this:

Filters: Only the portion of the object is shown that is currently relevant or interesting. In a CAD-system, you see the portion of the display that is currently being processed. Within this section, you do not want to see every detail, so another type of filter that suppresses visible portions of the object is used. This display method, therefore, only partially represents the object.

Coding: Instead of suppressing complex portions of an object, you can represent these in simplified (coded) form. Through this abstraction, the representation remains complete in a sense, without becoming unnecessarily complicated. By using decoding rules, the representation can be expanded again.

The advantage of coding over the described filtering lies in the fact that completeness is retained. The object or portions of an object do not disappear; they are represented in coded form.

An application for icons can now be inferred. Whenever an object should remain visible in its normal representation, but is too big or complex and an abstraction thereof is acceptable, then it should be represented as an icon.

II.2. Icons in the Bitgraph Window System

In the Bitgraph Window System, a special type of icon was implemented. These icons have window characteristics, i.e., they can be moved like windows; they can overlap; they can be brought to the top; etc. However, they have a fixed size and content. The content is an arbitrary "printable" character of the Bitgraph.

An additional specialized class of icons consists of those icons that can be bound to functions (`function-bound-icon`). These function icons present themselves as "keys" on the screen (Softkeys, Lightbuttons) with text indicating their functionality. By activating the icon with the mouse, the function bound to it will be executed. During the execution of the function, the user receives feedback in the form of an inverted icon (reverse color).

Function icons are a generalization of the normal menu concept in which the function is not only represented as text but also in the form of a picture on the screen. Also, they are a specialization of menus since only one function can be selected. More functions can only be offered through more function icons.

Another type of icon can be bound to a window of an application system (`window-bound-icon`). These icons, called "window icons" in the following, communicate with a specified window (partner window). They are visible when their partner window is not and vice versa. These are useful for windows that, for example, require a large amount of space and need to be replaced from time to time with a small icon. By activating the icon, the window is redisplayed and the icon disappears. The standard icon is a small stylized window. Document icons, form icons, program icons, among others are available for special applications.

II.3. Implementing Icons

Icons are implemented in Objtalk. A specific icon is an instance of a specific icon class. The hierarchy of icons has the following structure:

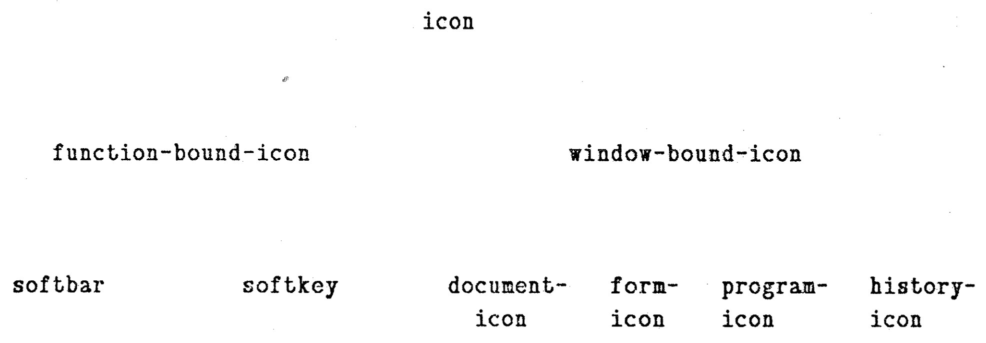


Figure II-1: The Objtalk hierachy of icons.

II.3.1. Class: icon

The window characteristics of `icon` are based on the superclasses of `same-mixin` and `basic-window`.

Slots:

`icon-font`	Number of the font from which the icon originated (default is 1 = PICTURES-FONT).
`icon-char`	Number of the character that should serve as the icon; no default value.
`display-text?`	A boolean value that specifies if text should be written into the icon (default = t).
`icon-text`	Text written into the icon if display-text? is `non-nil`.

`text-font`	Font for `icon-text` (default is 13 = SMALL-FONT).
`text-pos`	Position of the text in the icon given as a list (x,y) (default = (5 30)).
`max-text-length`	The `icon-text` is only printed in the icon up to this length; the rest is truncated (default 10).

Besides setting these slots during instantiation of an icon, the `screenregion` must be set to give the window a position as well as a height and width.

The default values are arbitrary, since without knowledge of the usage of `icon-char` appropriate values cannot be foreseen.

Methods:

`change-icon: ,?i-char ,?i-font ,?i-width ,?i-height`
Replaces the `icon-char` with another character (parameters: character number, character font, width of the icon, height of the icon).

II.3.2. Class: function-bound-icon

Function icons of this class evaluate LISP expressions when activated.

Slots:

All slots from `icon` are inherited (see inheritance hierarchy and description of `icon`). Default values are newly defined.

`list-of-sexprs`	List of LISP s-expressions to be evaluated.
`screenregion`	Is set to the default value (0 0 160 32).
`icon-char`	A default icon is specified (note in `icon` there is no default value).
`text-font`	Is set to the default value MINI-FONT.
`max-text-length`	Default value is 22 characters when using MINI-FONT.

Methods:

`execute:` Execution of the LISP s-expressions

In the following, two subclasses of `function-bound-icon` are described that differ only in their representation on the screen.

`softbar`	An icon of this type represents itself as a short wide rectangle. The text can be up to 30 characters long (if using the default SMALL-FONT).
`softkey`	Icons of this type look like the normal keys of the terminal keyboard. In standard font (SMALL-FONT), only five characters of text can be displayed.

II.3.3. Class: window-bound-icon

For most applications of the window system, these icons are the most interesting because they serve to visualize windows that are buried. The concept of partnership between window and icon, where only one is visible at a time, applies here.

By clicking the icon with the mouse or by using the method `totop:` of the window, the icon is buried and the window is made completely visible; by using the method `shrink-to-icon:` of the window, or the method `totop:` of the icon, the window disappears and the icon is displayed.

If the `kill:` message is sent to the window or the icon, then it is also automatically sent to the partner.

The icon recognizes the methods `make-sticky:` and `anchor:`, where the former ensures the placement of the icon at the location of the window (reference point is the left upper corner of the icon and the window) or the placement of the window in the location of the icon. In contrast, the method `anchor:` anchors the icon on a specific place on the screen, independent of the partner window. By employing the `move:` method of the icon, this anchor point can be modified.

By associating an icon to a window, the window must receive various attributes for communication and synchronization with the icon. The associated new window class must, therefore, contain the so-called *icon-mixin* as a superclass. This mixin *must* be included in the list of superclasses *before* the window mixins and the window class, for example:

```
(ask class new: icontextwindow
    (superc icon-mixin text-mixin basic-window))
```

Slots:

`partner-window`	Reference to the partner window: this slot is automatically filled during the instantiation of a window icon when the slot `partner-icon` is defined in the window class.
`sticky?`	`sticky?` = `t`: Icon follows the window and vice versa; `sticky?` = `nil` : Positions of the window and the icon are independent of each other.
`icon-char`	The default value is a small stylized window.
`icon-text`	If no text is specified, the icon receives the `title` of the window; this is **not** a default value, but a value that should be acquired from the partner window when necessary.

All other slots are the same as those of the class `icon` except for the described default values.

Methods:

`make-sticky:`	The slot `sticky?` is set to `t`.
`anchor:`	`sticky?` is set to `nil`.
`expand-to-window:`	The icon is buried and the partner window receives the message `totop:`
`kill:`	Icon **and** partner window receive the message `kill:`

kill-icon: Only the icon receives the kill: message; this method serves to rid windows of unwanted icons without removing the window.

totop: Dependent on its "sticky-condition" the icon is positioned and displayed while the partner window is buried.

left-button-down:
By clicking the left button with the mouse, the method expand-to-window: is executed by the icon.

The following subclasses of window-bound-icon differ only in their representation on the screen.

document-icon Creates icons that appear like manuscripts. Maximum text length using SMALL-FONT is 11 characters.

form-icon Creates icons that look like forms. Maximum text length using SMALL-FONT is 11 characters.

program-icon Creates icons that look like a program listing. Maximum text length using SMALL-FONT is 17 characters.

history-icon Creates icons that look like a scroll. Maximum text length using SMALL-FONT is 7 characters. The default text is "history".

II.3.4. Class: icon-mixin

The icon-mixin provides a window with the characteristics needed to be associated with an icon.

Slots:

partner-icon Reference to icon. By setting this slot, the inverse relationship partner-window is automatically set in the icon. The creation of a window icon should be achieved through instantiation in this slot, as for example:

```
(ask icontextwindow make: window-x with:
     (partner-icon =
        ,(ask window-bound-icon instantiate:)))
```

Methods:

partner-icon = ,?icon
Attaching window icon to the window; the inverse relationship partner-icon and partner window is automatically propagated.

shrink-to-icon:
The window disappears (bury:) and the partner icon appears (totop:).

totop: The partner icon disappears (bury:) and the window appears (totop:).

kill: The window **and** the associated icon are removed.

`init:` By extending the `init:` method, the `top-menu` of the window receives the additional entry "shrink-to-icon".

II.4. Several Remarks About Menus When Using Icons

The icon-mixin ensures that the `top-menu` of the window receives the additional entry "shrink-to-icon". If a global menu is used as the `top-menu` in the window, i.e., this menu is also used by other windows, then this entry appears there also and may be undesirable. Therefore, windows that have the class `icon-mixin` as their superclass should have their own local menus.

Icons of the following class^{es} and their subclasses use global menus:

class	**menu**
`icon`	`icon-menu`
`function-bound-icon`	`function-bound-icon-menu`
`window-bound-icon`	`window-bound-icon-menu`

The option of using context specific local menus for window icons is available. The context sensitivity is currently limited to the menu entries "make-sticky" and "anchor" where only one is meaningful. These menus can be acquired by using (load 'lm-icons). All window icons that are instantiated thereafter own these menus. The already existing icons continue to have the global menu.

Appendix III. TOPREADER

```
|                    TOPREADER                  |
|                                               |
|       A Real-Time Indenting LISP Reader       |
|_______________________________________________|
```

First written by Franz Fabian and Christian Rathke.
Modified by Andreas Lemke.
Extended and reimplemented by Franz Fabian.

TOPREADER is a real-time indenting lisp-reader that supports a history-list of inputs. TOPREADER replaces the standard FranzLisp toplevel print-eval-read loop. It interacts well with the break-function of FranzLisp.

III.1. Invoking TOPREADER

TOPREADER can be used in two ways:

1. by calling it at the FranzLisp toplevel;

2. by binding the variable `user-top-level` to the atom `topreader`.

III.2. Leaving TOPREADER

This depends on the way TOPREADER was invoked: if it was called explicitly by the user, you have to call *(reset)* either by typing this at TOPREADER-toplevel or in a break-level. If you bound the variable *user-top-level*, you have to rebind it to `nil` and then do the reset.

III.3. Interaction with the Break-Package

If you use `^D` to leave a break, you automatically will be prompted by TOPREADER for the next input. If you do a reset to leave the break and `user-top-level` is bound to `topreader` then you will be back in the TOPREADER-loop again, otherwise, you will be on the standard FranzLisp toplevel and have to re-invoke TOPREADER yourself.

III.4. Using TOPREADER

After invocation of TOPREADER, you are prompted with *n:* (n is an integer, which is increased at each new prompt). Then you can type in your commands as in the standard FranzLisp toplevel loop. When you finish your input, it will be evaluated. The time needed for the evaluation (in msec) and the result is printed to the terminal and you get the next prompt for input.

When typing input, several characters have a special meaning to TOPREADER (ε means *control*):

`<linefeed>`	the cursor jumps to the next line and indents properly to an appropriate column-position.
`<return>`	If you didn't type any left parenthesis or left bracket typing `<RETURN>` will finish input and start evaluation, otherwise the cursor will go to the beginning of the next line.
`^U`	cancels all the input typed in so far and prompts you again.
`^N`	when input is finished all the typed characters are stored in an internal variable. By typing `^N` these characters can be reused in order to do minor changes to the last input.
`^R`	causes TOPREADER to reprint the remaining characters (those not taken away by `^N`) of the last input.
) or]	if this is the matching parenthesis to the first left parenthesis, TOPREADER stops taking input and evaluates the given command.
`<del>`	you can use `<del>` to delete mistyped characters in the same way as when using standard read functions. In addition, it allows you to delete characters typed at previous lines until you reach the prompt.

If you type an illegal character (ascii-code 0 to 31, with the exception of the characters mentioned above), the bell signals this to you and the character is ignored.

III.5. User-Extensibility

Each time a character is typed, TOPREADER funcalls a function that is stored in `topr:readtable`[26] under a index unique to the character. To change the meaning of a character you only have to restore the s-expr in the appropriate place in this readtable. This can easily be done by:

```
    (topr:change-read-table '<ascii-code> '<s-expr>)
```

To get the current s-expr associated with a character do:

```
    (topr:readtable '<ascii-code>)
```

III.6. History Mechanism

TOPREADER supports a history mechanism similar to those of INTERLISP or the CMU-FranzLisp toplevel. Each input of the user is saved together with the input-number (i.e the prompt-number). There exist several commands to re-use previous input (brackets are used to denote optional parameters):

`(redo [<nr>])`	`<nr>` is either a positive or negative integer (default: -1). A positive argument is the absolute number of an input, a negative one is used to select the `<nr>`-th last input.

[26] NOTE: `topr:readtable` is not a special version of a FranzLisp readtable which is used when scanning over the input to detect symbols or read macros. It is only used to control the process of typing in characters and positioning the cursor after `<Linefeed>` or `<Rubout>`. In some sense, TOPREADER is a special terminal driver for Lisp.

This causes TOPREADER to print the input determined by number and evaluate it again. Redo without an argument is the same as typing ^R.

`(ed [<nr>])` <nr> same as above

Ed invokes the FranzLisp editor on the denoted input. When leaving the editor, the changed expression is evaluated.[27]

NOTE: in `ed` and `redo`, <nr> may also be a symbol. Then the most recent command at toplevel that mentions <nr> is used.

`(use <x> for <y> [in <nr>])`

<nr>: same as above, but when omitted, the first expression found by going backwards through the list of inputs that contains <y> is taken.

<y>: expression to be substituted

<x>: replaces <y> in the selected input

`use` substitutes <x> for <y> in previous input and evaluates the resulting expression.

`(show [<start> [<number>]])`

<start>: same as <nr> above (default -1)

<number>: a positive integer (default 1)

shows the last <number> inputs starting at <start>.

III.7. Topreader-Hook

Each time TOPREADER has evaluated an input, it looks whether the variable `topreader-hook` is bound. If `topreader-hook` has a binding, this is evaluated. This allows some additional work to be done (e.g. `'(checkmail)'`) before typing in each new command. When loading TOPREADER, `topreader-hook` is bound to `nil`, but may be rebound to another value by the user.

[27] Bug: there should be the possibility to return to TOPREADER without automatically evaluating the edited expression.

References

[Foderaro 82] J.K. Foderaro.
The FranzLisp Manual.
Technical Report, University of California, Berkeley, 1982.

[Lemke 85] A. Lemke.
ObjTalk84 Reference Manual.
Technical Report CU-CS-291-85, University of Colorado, Boulder, 1985.

[Rathke, Laubsch 83]
C. Rathke, J.H. Laubsch.
ObjTalk - eine Erweiterung von Lisp zum objekt-orientierten Programmieren.
Technical Report, Institut fuer Informatik, Universitaet Stuttgart, Februar, 1983.

Index