GWM Manual

The X11 **G**eneric **W**indow **M**anager

Version 1.7e

Colas Nahaboo
colas@mirsa.inria.fr

KOALA Project — BULL Research

Copyright restrictions[1]

Acknowledgements

Although I am the only designer and implementor of GWM, the project could not have been completed without the help of many individuals, namely:

Vania Joloboff, who created the KOALA BULL team at INRIA, and started the GWM project, and KOALA members Vincent Bouthors and Daniel Dardailler who tested GWM and wrote WOOL profiles.

Gilles Kahn, Francis Montagnac, and the CROAP team at INRIA who intensively used GWM and provided me with machines to develop it. A special thanks goes to Janet Incerpi who corrected the documentation.

All my beta testers: Raphael Bernhard, Didier Poirot, Matthieu Devin, Christophe Muller, Laurent Hill, Anne Marie Bustos, Simon Kaplan ... and last, but certainly not least, Bob Scheifler himself!

And since July 1989, GWM has been enhanced by the growing community of the **gwm-talk** mailing list, whose members are alas too numerous to be all cited here, but are giving me an invaluable feedback.

Colas Nahaboo. INRIA – KOALA Project
BP 109 – 06561 Valbonne Cedex – FRANCE

Support

Although GWM comes without ANY guarantee nor official support, You can mail any questions relative to GWM by electronic mail to **gwm@mirsa.inria.fr**[2], and you can find the latest GWM version by anonymous ftp on export.lcs.mit.edu (contrib/gwm) and on avahi.inria.fr (pub/gwm) where the mailing lists are also archived (pub/gwm/gwm-talk-archive).

---

[1] This is the same copyright as the X Window system distribution

[2] Where you can mail also requests to be added on the two mailing lists: **gwm-talk@mirsa.inria.fr**, for general comments, and **gwm-bugs@mirsa.inria.fr**, for reporting (and be warned of reported) bugs

Contents

1 Usage

```
gwm [-mtDFia] [-x screens] [-f profile] [-p path] [-d display]
```

1.1 Options

The following command-line options[1] are supported:

-f filename Names an alternate file as a GWM startup file. Default is **.gwmrc.gwm** (note that the `.gwm` extension is optional, as for any WOOL file).

For instance, to use the MOTIF emulation package, type `gwm -f mwm`.

-p path Gives the path to be searched for WOOL or bitmap files when loaded, including the startup file. Overrides the setting by the environment variable **GWMPATH**. Defaults to `.:$HOME:$HOME/gwm:GWMDIR`, where `GWMDIR` is the local directory where GWM is installed (normally, `/usr/local/lib/X11/gwm`).

You can append or prepend a path to the current path by preceding the path given as argument to the `-p` option by `+` (for appending) or `-` (for prepending). For instance, if you want to search the directory `/usr/local/gwm` before the standard ones (including your homedir), just say: `-p -/usr/local/gwm`.

-d display Specifies the name of the display whose windows must be managed, such as `unix:0`. The `-d` is optional, you can type `gwm unix:0`.

-x screens Do not manage the screens given in the comma-separated list of numbers, as in: `-x 2,5,3`. Normally, GWM manages **all** the screens of the given display.

-1 Manages only the given screen, e.g. `gwm -1 foo:0.2` manages only the third screen of the display number `0` on the `foo` machine. Same as defining the GWM_MONOSCREEN shell variable.

-D Enables debugging mode for WOOL files. In this version the only effect of debug mode is to continue reading a file after a WOOL error occurred. Default behavior is to abort reading a file after an error. Thus, if you modified your profile and introduced an error, GWM will refuse to complete execution, use `gwm -D` to run it anyway.

[1]The options follow the GETOPTS(3) conventions: they can be in any order, a space is optional between an option and its argument; they can be combined (as in `-Dmt`), but all options must appear **before** any argument, which is the managed display for `gwm`.

-s Synchronize X calls, useful for debugging but slower.

-t Turns tracing on, as if you had done the call `(trace t)` in your profile.

-q (Quiet) Does not print startup banner, and sets the WOOL `gwm-quiet` variable to 1.

-m Maps all toplevel windows already on screen. Useful after unmapping some windows by accident.

-F Does not freeze the server during pop-up menus, move and resize of the windows, which is the default behavior.

-i Disables the setting of input focus by GWM (`set-focus` has no effects, except `(set-focus ())`, which resets the focus to PointerRoot) on a window, keypresses go to the window under the cursor. Very useful to debug profiles with only one screen.

-a Asynchronously handle moves and resizes, do not cancel a move or a resize operation if the user released the button before the grid appeared, which is the default behavior.

-? This, or any invalid option lists the available options and shows the default path defined at compile time by your local installer.

1.2 Shell variables

GWM can make use of the following shell variables:

GWMPATH for the path to search for files. If unset, defaults to `.:$HOME:$HOME/gwm:GWMDIR`, where `GWMDIR` is the local directory in which are installed all the standard GWM files. (Normally `/usr/local/lib/X11/gwm`). Overridden by the `-p` command line option.

GWMPROFILE for the name of the profile file. Defaults to `.gwmrc.gwm`. Overridden by the `-f` command line option.

DISPLAY for the name of the X11 display to use, such as `unix:0.0`. Overridden by the `-d` command line option.

GWM_MONOSCREEN if set will make GWM manage only the given screen.

1.3 Files

GWM needs at least one file for its startup, `.gwmrc.gwm` which must be in GWM's path. New users do not need one, since a default profile should already be present in the default path. The standard profile (see 3.1, p 12) makes use of the `.profile.gwm` file in the home directory.

The value of the default path can be printed by calling GWM with the `-?` command line option.

The standard extensions used for GWM file names are:

`.gwm` for WOOL files.

`.xbm` for **X**11 **B**it**M**ap files, such as those created with BITMAP(1).

`.xpm` for **X**11 **P**ix**M**ap files, which is an ASCII portable format for distributing color images (see the `pixmap-load` function, p 65).

1.4 Description

The GWM command is a window manager client application of the X11 window server specified in the `display` argument (or the `DISPLAY` shell variable if no argument is given), extensible via a built-in *lisp* interpreter, WOOL (Window Object Oriented Lisp) used to build **Wobs** (Window OBjects) which are used to decorate the windows of the other X11 applications running on the display. GWM tries to enforce the ICCCM conventions to communicate between X11 clients and thus should be compatible with any well-behaved X11 application.

On startup, GWM interprets its profile to build wobs describing how to decorate user windows, which we will call **Clients**. Then it creates **Windows** around each client windows on the screens attached to the managed display. A Window is made of 4 (optional) **Bars** on the 4 sides of the window. Each of these bars is made of a variable number of **Plugs**, the most primitive wobs. **Menus** can then be made with a list of bars. To each of these objects is associated a **fsm** (**F**inite **S**tate **M**achine) describing their behavior in terms of WOOL code triggered by X or WOOL events.

When GWM wants to decorate a window, it calls the user-defined WOOL function `describe-window` which must return a list of two window descriptions (one for the window itself, and one for its icon) made by the `window-make` WOOL primitive describing the window. To build these descriptions the user can query the client window for any X11 properties and use the X11 Resource Manager to decide how to decorate it.

The screens must also be described by such a description that GWM will find by calling the user-defined WOOL function `describe-screen` for each managed screen.

2 Overview

GWM like any other window manager is in charge of all that is **exterior** to other application windows on the display. It performs its task by **decorating** the existing windows on the screen with its own windows. The appearance and behavior of these windows are described by the user through programs written in the lisp-like WOOL language, which are interpreted by the built-in WOOL interpreter of GWM.

The GWM Window manager is composed of 2 modules: the window manager itself, GWM, and the lisp interpreter WOOL. The role of the garbage-collected WOOL objects is to build shared descriptors that will be used by GWM to build its (non-shared and non-garbage-collected) objects.

2.1 GWM Window OBjects: the wobs

GWM is built upon the notion of a **wob**, which, not unlike an X *Widget* is an object composed of:

an X11 window used to display the wob on the screen (output)

a WOOL **finite state machine** used to trigger **WOOL** functions in response to X11 events sent to the wob (input)

Wobs are not directly created with WOOL contructors, they are **described**, and GWM uses this description to physically create the wobs when it needs to, when decorating a new client window for instance.

Like any X11 window, the user can choose the width, color(s), and tiling pixmap of the border of the wob (which is considered to be in the wob for input purposes). There are four kinds of wobs:

Window An X11 application, such as XTERM(1) or XCLOCK(1) usually opens one or more windows directly on the screen (in fact the *root window*). We will call these windows, which are not created by GWM, **client windows**. GWM will "decorate" these client windows by *reparenting* them, i.e. by creating a **Window** wob and making the client window a son of this newly created window.

A window is a wob made by creating a new toplevel X window, reparenting the client window as its child, and framing it with four (optional) *bars*, children of the new toplevel window. Note that the inside of a window wob is thus never visible, since it is entirely covered by the bars and the client window.

Bar The only extensible wob, it has a width and an extensible length. It is a line (vertical or horizontal) of *plugs* centered on the axis of the bar with optional stretchable space between them.

Plug The simplest of all wobs, its contents is just a graphic which is displayed in their X window. It acts thus like some kind of button. Current graphics are text and pixmaps.

Menu A bar of bars (horizontal menus consist of a horizontal bar of vertical bars, vertical menus are a vertical bar of horizontal ones).

Menus are the only "stand-alone" wobs, their windows are directly created by GWM on the screen. They can be used to implement pop-up, pull-down or fixed menus, control panels, message windows and icon boxes.

Other GWM objects

Other gwm objects are just X objects (fonts, pixel colors ...) that are referenced by their X id, and are accessed via encapsulating WOOL types, such as Numbers or Pixmaps.

2.2 WOOL objects

WOOL is a lisp interpreter of a special kind, since it has some important design differences from "real" lisps:

incremental garbage collection WOOL has a *reference-count* driven memory management scheme. This means that the load of memory management is evenly distributed on all phases of computing, thus avoiding the dreadful garbage collection pauses. But, since reference count memory management doesn't work with circular lists, no WOOL function allows the user to do physical replacements on lists[1].

no real lists, but arrays WOOL lists are internally stored as arrays, speeding up list scanning. We do not really need the generality of the chained cells model, since we do not want to have circular lists.

monovalued atoms In classical lisp dialects, you can give a variable and a function the same name without conflict. In WOOL, an atom can only have one value.

internally typed objects All of the WOOL interpreter is written in an object oriented way. This is not visible to the user, since we do not offer a way to define user types, but it greatly improves the modularity of the code, and allows us to provide generic functions, such as the `+` function operating on numbers, lists, or strings.

The different WOOL objects are:

atoms associates any WOOL object to any other via a hash table. Only one wool object can be referenced, which implies that a `(setq foo 1)` assignment will remove any function definition made by `(defun foo ...)`. There is **no** limit on the length of atom names. Atom names can be composed of any printable ASCII character except `" ' ( )`, and cannot begin by a digit.

[1] In fact, some do, but are flagged as "for experts only" in the documentation

active values are predefined atoms that can be used as atoms or functions. If `foo` is an active value, then `foo` and `(foo)` give the same result, and the calls `(setq foo obj)` and `(foo obj)` have the same effect.

Active values are just like other atoms, but setting and evaluating them trigger specific functions to allow for greater expressive power. For instance, just setting a wob's `borderwidth` will actually resize the corresponding X window's border, and declaring a local value for it by a `(with (wob-borderwidth 3) ...)` will actually change the borderwidth of the current wob on the screen during the execution of the `with` body, and then revert to the previous value.

numeric variables are atoms that can only store integer values. Like active values, they can be used as variables or functions. Setting them to `()` or `t` equivalent to setting them to `0` or `1`.

namespaces are sets of variable names that have a different value for each state of the namespace. For instance, the most useful namespace is the `screen.` one, having one state for each screen. So each name in this namespace, such as `white`, can hold a screen-dependent value that will always eval to the correct value relative to the screen.

integers are 32-bit signed integer values.

strings are 8-bit strings of characters, with no size limit.

functions may evaluate or not their arguments and have a fixed or variable arity.

fsms the finite state machines. They are WOOL objects shared by wobs and respond to both X events and WOOL-made events, the so-called "user" events.

wobs descriptors: plug, bar, menu, client are used by GWM to build its wobs.

X objects: pixmaps, cursors, events, labels allow X resources to be shared via the WOOL memory management.

internal objects used to improve efficiency (These include collections and quoted expressions).

2.3 Operation

When you start GWM, it:

- connects to the display it must manage to initialize its X11 structures
- reads and interprets the user's WOOL profile (searched for in a user-defined path, see usage). This profile must define at least two WOOL functions:

  describe-window which will be called by GWM to know how to decorate any client window and must return a list of two window descriptors, one for the window itself, and one for the associated icon;

  describe-screen which will be called by GWM for each managed screen and must return a window descriptor;
- looks to see if it is the only window manager running; if not, it aborts.
- decorates the managed screens by calling the user-defined `describe-screen` function for each one, with the `screen` active value being set to the current screen.

- decorates all already mapped client windows, by calling `describe-window` with the current client window being set to each window.
- executes the (user-defined) opening function for each screen.
- enters GWM main loop, which consists in:
  - waiting for an X event
  - examining the received event, and if it is for an existing wob, sends it to the *fsm* of this wob, else if it is a new window which is being mapped for the first time, decorates it (by calling `describe-window`)

When an event is sent to a *fsm*, it is matched against the transitions in the current state of the fsm, and as soon as one matches, the corresponding WOOL expression is evaluated, and the fsm changes state if necessary. If no transition is waiting for the event, it is discarded.

2.4 Lazy evaluation

For sub-wobs of wobs, i.e., bars of a window, plugs of a bar, bars of a menu, and menu of any wob, lazy evaluation is performed. That is, on the realization of the wob the field is re-evaluated if the result is not of the expected type. This allows for constructs such as:

```
(plug-make '(label-make window-name))
```

which creates a plug whose text is fetched as the name of the window on each realization, you do not have to explicitly `eval` a quoted expression.

2.5 Screen-dependent objects

An invocation of GWM can manage all screens attached to a display (there is one keyboard and mouse per display), but in X, screens are very distinct worlds. If you create a pixmap or a color on a screen, you cannot use it on another one. The list of objects created on one screen that cannot be used on another is:

Objects	made by
`colors`	color-make
`pixmaps`	pixmap-make, label-make, active-label-make
`cursor`	cursor-make
`menus`	menu-make

And of course, all the wobs and windows are screen-specific.

2.6 Sending commands to GWM

Any program can make GWM execute any WOOL code by putting the WOOL code to execute as an ASCII string in the **GWM_EXECUTE** property on any root window or client window of the screens managed by GWM, which will parse and evaluate the given string with the current window being the one on which the property was set.

This feature is built in, so that it will work regardless of the profile used.

2.7 Checking that GWM is running

On each screen it manages, GWM will maintain a hidden window whose X Id will be given as a 32-bit Id in the **GWM_RUNNING** property on the root window **and** the hidden window itself. Thus, if this window exists, and has the **GWM_RUNNING** property set, you can be sure that the screen is managed by GWM.

3 The Standard GWM Packages

This chapter describes the Wool packages available on the standard GWM distribution. The name of the involved files are listed in the title of each package.

Gwm does not try to enforce any policy in writing profiles, but for the sake of simplicity and maintainability, all the Wool packages delivered with Gwm will try to be compliant with a set of rules described in section 3.18, page 28, which should assure the compatibility between them.

Note: Distributed code is normally indented under emacs by Alan M. Caroll's `amc-lisp.el` emacs-lisp package, which is now included in the Gwm distribution in the `data/` subdirectory.

You can have a look at my personal profile in the file `data/profile-colas.gwm` if you are looking for actual examples.

3.1 The Standard Profile `.profile.gwm`

The standard profile can be customized to your taste by creating a `.profile.gwm` file in your home directory, or by copying the one in the Gwm distribution directory in your home directory and editing it. [1]

3.1.1 mouse buttons

The default behavior for clicking of the mouse buttons is, in a window decoration or an icon:

left button Moves the window. Releasing the button actually moves the window, pressing another one while still holding down the left button cancels the move operation

middle button in a window Resizes the window. The size of the window will be displayed in the upper-left corner of the screen. Releasing the button actually resizes the window, pressing another one while still holding down the middle button cancels the resize operation

middle button in an icon de-iconifies the icon

right button brings up a pop-up menu for additional functions, such as iconification and destruction

[1] The standard profile is the default one, which you get if you do not have a `.gwmrc.gwm` file in your home directory. It is a real-estate overlapping environment. It is defined in the `.gwmrc.gwm` file in the distribution directory.

These functions are enabled only in the GWM-added decoration around the window, or anywhere in the window if the **alternate** (or **meta** or **left** key on some keyboards) modifier key is depressed when clicking, in the `uwm` style of interaction.

Moreover, if you click in the icon in the upper left of the frame around the xterm windows with the left or right button, the window will be iconified, and with the middle button, it will be iconified and the icon moved just underneath the pointer. You can still move the icon elsewhere by dragging it while the middle button is down.

Whether you want the window to be raised on top of others when performing a move, resize, or (de-)iconify operation can be toggled by setting to `t` or `()` the global values `raise-on-move`, `raise-on-resize`, and `raise-on-iconify`.

3.1.2 customization

Customization is achieved by creating a `.profile.gwm` file in your home directory (or anywhere in your `GWMPATH`). In this file you can set the variables to modify the standard profile to suit your taste. Note that you must set variables used in decorations **before** loading this decoration by a `set-window` or `set-icon-window` call.

Your `.profile.gwm` will be loaded once for each screen managed, and since pixmaps, colors, cursors and menus are screen-dependent objects, try to define them as `names` in the `screen.` namespace (`namespace-make`, p 64).

Mouse bindings

The standard mouse button bindings can be redefined by re-defining the default **states** (`state-make`, p 77) for a click in a window decoration, an icon and the root window, which are respectively the global variables **window-behavior**, **icon-behavior**, **root-behavior**. These states will be used to build the **fsms** of the windows, icons and root window. The state **standard-behavior** is included in both window and icon behaviors, so that you can add transitions to it if you want to have them in both contexts. You may then need to redefine the events grabbed by GWM, in the **grabs** variables; all events in these lists are "stolen" from the application and redirected to GWM. There are three grabs variables: `root-grabs`, `window-grabs`, and `icon-grabs`, pointing to the lists of event to be "grabbed" from all applications, only from windows, and only for icons. That is why you can move a window by clicking anywhere in it with the left button while depressing the `Alt` key: the standard grabs consists of the list:

```
(list (button any with-alt)
      (button select-button (together with-shift with-alt)))
```

If you only want to change button bindings, change the value of `select-button`, `action-button`, and `menu-button`, which are initially bound to 1, 2, and 3 respectively.

You need to call the `reparse-standard-behaviors` function after modifying any of these states to take your changes into account. For instance, to add iconification on the "F1" function key only on windows, you would write in your .profile.gwm file:

```
(setq window-behavior
    (state-make
       (on (keyrelease "F1" alone) (iconify-window))
       standard-behavior))          ; include previous actions

(reparse-standard-behaviors)        ; commit changes
(setq window-grabs                  ; grab F1 from clients
```

```
(+ window-grabs (list (keyrelease "F1" alone)))
```

global switches

The following global variables (which are names in the `screen.` namespace for all pixmaps, colors, cursor and menus) controlling the way the standard profile operates can be set in your `.profile.gwm` file:

`cursor` to the cursor displayed in any decoration or icon

`root-cursor` to the cursor displayed on the root window. The available cursors in the distribution are:

- `arrow` a big arrow
- `arrowhole` same with a hole inside
- `arrow3d` a 3d-looking triangular shape. Especially nice on a clear background.

For instance, to use the "`arrow3d`" cursor, just say:

```
(setq root-cursor (cursor-make "arrow3d"))
```

`screen-tile` to the pixmap used to tile the root window with. Provided bitmaps are `back.xbm` (default) and `grainy.xbm`

`autoraise` if set to `t`, (defaults to `()`), GWM will raise on top of others the window which has the input focus

`xterm-list` the list of machines the user wants to launch a remote xterm on (via the `rxterm`[2] command)

`xload-list` the list of machines the user wants to launch a remote xload on (via the `rxload` command)

`icon-pixmap` the pixmap to be displayed in the upper left corner of window to iconify it

`to-be-done-after-setup` the list (`progn`-prefixed) of things to be executed after all windows already present have been decorated.

`look-3d` to `t` to specify that window decoration packages that support it should adopt a tridimensional look. The default for this variable is `()` on monochrome displays and `t` on color and grayscale ones.

Window and icon decoration

Moreover, you can decide to change the decoration (look and feel) of a client or an associated icon by using the following functions.

For most following functions, when a `window-description` is expected, it means a X resource specification of the form:

`client-class.client.name.window-name.machine-name`

where *-notation, or "any" to means any value for a field is accepted. Note that all fields are optional, except for client-class. Thus, you can say that you want all xterm icons to be xterm-icons, execpt for the one named Console, on machine avahi, for which you want to use a simple icon by:

```
(set-icon-window XTerm xterm-icon)
(set-icon-window XTerm*Console.avahi simple-icon)
```

[2] `rxterm`, `rxload` and `rx` are Bourne shell scripts used to start remote xterms, xloads or any other X command. The rxterm script is included in the distribution, in the `gwm` subdirectory, install it and make rx and rxload as links to it

Note: for all functions, to set defaults for a screen type or a client class, use the `any` keyword.

Note: Since version **1.7**, the values affected to `set-window` et al. functions are evaluated at **decoration time**, not while reading the profile as it was the case before.

`(set-window [screen-type] window-description decoration)`
will tell GWM to use the decoration described in `decoration` for all clients of client description `window-description` in the current screen of type `screen-type`. `decoration` can be either:

- a real decoration made with a `window-make` call
- a function returning a decoration when called without parameters
- the file name (as a string or atom) of one of the standard decorations as listed in the section 3.14. The file will be loaded and should set the `decoration` atom to either a real decoration or a function returning a decoration

For instance, these declarations say that xterms on this screen, if it is a monochrome one will be decorated by the `simple-ed-win` style, but if the screen is a color or grayscale one, will use the `simple-win` decoration, and will use the "no-decoration" style for windows not otherwise describe (The `no-decoration` window description adds no visible decoration to a window).

```
(set-window mono XTerm simple-ed-win)
(set-window any XTerm simple-win)
(set-window any no-decoration)
```

To choose decorations on other criteria than just class, define a function that will return a WOOL expression which will give the good decoration when evaluated. For instance, to put a `simple-ed-win` decoration on all XTerms, except those less than 200 pixels wide, use this in your .profile.gwm:

```
(defun decide-which-xterm-deco ()
    '(if (< window-width 200) (no-frame)
         (simple-ed-win)))
(set-window any XTerm decide-which-xterm-deco)
```

For an exemple of an alternative choosing function, see `match-windowspec`, p 27.

`(set-icon [screen-type] window-description bitmap-file)` will
associate to a client a simple icon made of the the X11 bitmap stored in `bitmap-file` (with the current value of `foreground` and `background` at the time of the call to `set-icon`) and the name of the icon underneath it.

If a list is given as the last argument, it is evaluated and the pixmap is taken as the result of the evaluation.

Suppose that you designed a picture of a mailbox named "mail-icon.xbm", saved it somewhere in your GWMPATH, and want to use it for the icon of the client `xmh`. You would add in your `.profile.gwm` one of the two forms:

```
(set-icon XMh mail-icon.xbm)
(set-icon XMh (pixmap-make black "mail-icon.xbm" white))
```

Note: the icon bitmap can only be set for icon decorations supporting it, such as the (default) `simple-icon` decoration style.

`(set-icon-window window-description icon-file)` This call will associate more complex icons to a given client, such as those listed in section 3.15, p 26. For instance to have `xterm` icons look like a mini computer display, add in your `.profile.gwm`:

```
(set-icon-window XTerm term-icon)
```

The `term-icon` argument can be either a client window decoration, a function returning a decoration or a file name, as for the `set-window` function. On startup, GWM does a:

```
(set-icon-window any any simple-icon)
```

Desktop space management `placements.gwm`

The standard profile supports functions to automatically place your windows or icons on the screen. These functions manage only some type of clients (or all of them if affected to the `any` client), and they are called with one argument set to `t` on opening the window, and to `()` on closing (destruction) it. They are associated to clients by the calls:

`(set-placement [screen-type] window-description function-name)` for the main windows of a client.

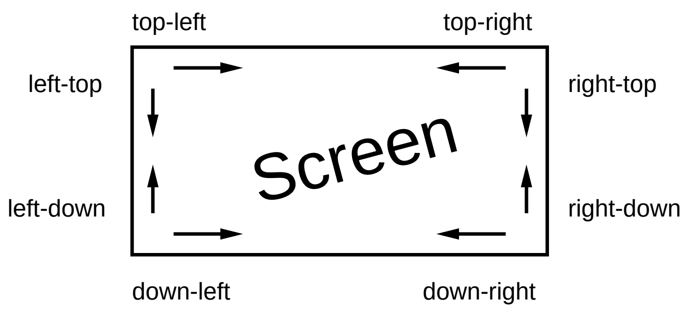
`(set-icon-placement [screen-type] window-description function-name)` for its associated icons.

The currently pre-defined placement functions are:

`()` does nothing, the window justs maps where it was created by the client (this is the default value)

`user-positioning` asks the user to place it interactively

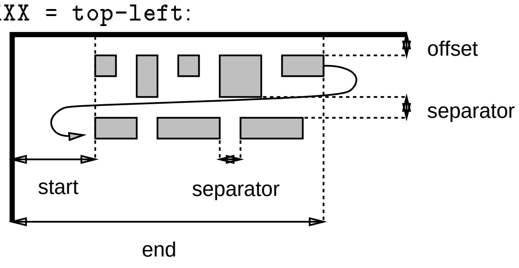
`rows.XXX.placement` automatically aligns the windows or icons on the sides of the screen. Replace `XXX` by one of the eight names in the following figure:



You can set the space where the `XXX` row lives by issuing calls to the control function `rows.limits` whith the syntax:

```
(rows.limits rows.XXX [key value] ...)
```

where `key` is an atom setting a value (in pixels). Key can be either `start`, `end`, `offset`, and `separator` as shown in the following figure for `XXX = top-left`:



`key` can also be `sort`, in which case the icon in this row will always be kept sorted by the funtion given in argument. This function will take two windows as argument, and must return -1, 1, or 0 if the first window must be before, after, or if they have the same precedence. As an exemple, a `sort-icons` function is provided; if you set it as a row sorting function, it will sort windows according to their weight as set in the property list `icon-order` indexed on the class of the application[3]. Windows having the same weight are sorted by window name. If an application is not found in this list, the value of the variable `icon-order-default` is used (default 100). Of course, you are free to redesign other sort functions.

For instance you can manage your `xterm` windows by:

```
(set-placement XTerm user-positioning)
(set-icon-placement any XTerm rows.right-top.placement)
(rows.limits rows.right-top 'start 100 'separator 2)
(rows.limits rows.top-left 'sort sort-icons)
```

You can define your own window placement functions and use them by the `set-placement` call. They will be called with one argument, `t` when the window (or icon) is first created, and `()` when the window is destroyed. This is why we needed an interpretive extension langage! An exemple of another placement routine is given in the user-contributed package `near-mouse.gwm` (see 3.17.1, p 28).

```
(defun do-what-I-mean (flag) ...great code...)
(set-icon-placement any any do-what-I-mean)
```

Menus

The displayed menus can be redefined by setting the following global variables to menus made with the choosen menu package. The default package is the `std-popups` (see 3.11, p 21) package.

`window-pop` for the menu triggered in windows

`icon-pop` for the menu triggered in icons

`root-pop` for the menu triggered in the root window

You can look at their standard implementation in the `def-menus.gwm` distribution file.

3.2 Jay Berkenblit's Virtual Screen `vscreen.gwm`

This little "virtual screen" package, made by Emanuel Jay Berkenbilt, MIT `<qjb@ATHENA.MIT.EDU>` provides a way to use the physical screen of your workstation as a viewport on a larger root window. You move the screen by keys (default is CTRL-ALT arrows), can ask for a map of the screen to be displayed (item `VS Show` in root menu), and to move back to origin (item `VS Restore` in root menu). This quick-and-dirsty package doesn't pretend to compete with VTWM, but it is a good start.

The current upper left of the screen is shown a cross in the map. Clicking in the map will make it disappear, and the map is a snapshot of the current situation which is not automatically updated.

You can customize it by setting the following variables:

[3] for instance, the default order is set by `(setq icon-order '(Xmh 10 XPostit 5 XRn 20 XClock 2 XBiff 1 XLoad 20))`

`vscreen.menupos` position of the `VS Show/Restore` entry in root menu, to draw a map of the screen and to move back the screen to origin.

`vscreen.windowmenupos` position of the `VS UnNail/Nail` Entry in window menu to make window move along the virtual screen.

`vscreen.modifiers` modifiers to press with arrow keys to move the screen around

`vscreen.no-bindings` set to `t` if you do not want to bind arrow keys to `vscreen.move-windows` functions.

`vscreen.right-left` amount the virtual screen is moved by on a horizontal key stroke. defaults to half a screen.

`vscreen.down-up` amount the virtual screen is moved by on a vertical key stroke. defaults to half a screen.

`vscreen.nailed-windows` a list of windows to be carried along with the screen. this list is a list of windowspecs (property lists) (see `match-windowspec`, p 27).

3.3 Duane Voth's rooms `dvrooms.gwm`

Duane Voth (`duanev@mcc.com`) made a mini *rooms* package to manage groups of windows. To use it, put in your .profile.gwm the line

```
(load "dvrooms")
```

before any calls to any `(set-...` call. Then, with the standard profile, you can add new rooms by the root pop-up menu, or by explicitly calling the `new-dvroom-manager` function in your profile, with the name of the room as arguments.

```
(new-dvroom-manager "mail")
(new-dvroom-manager "dbx")
```

The name of the room itself is the same editable plug as the one used for the `simple-ed-win` window decoration, so that you can edit it by double-clicking or control-alt clicking with the left button.

Only one room is "open" (non-iconified) at a time (unless `dvroom.icon-box` is non-nil), and calling the functions `add-to-dvroom` or `remove-from-dvroom` (from the window menu or from WOOL) on a window will add or remove it from the group of windows that will be iconified or de-iconified along with the room manager. Opening a room will close the previously open one, iconifying all its managed windows. New rooms start as icons.

This package will recognize as a room manager any window with the name `rmgr`. You can then create new rooms by other Unix processes. An X property GWM_ROOM is maintained on windows added to rooms containing the name of the room manager, so that rooms are not lost on restarting GWM.

Context used:

`dvroom.font` font of the name of the room

`dvroom.background` background color

`dvroom.foreground` color of the text of the name

`dvroom.borderwidth` borderwidth of the room

`dvroom.x, dvroom.y` initial position of the room

dvroom.name string used to build the name of the room. Defaults to "Room #" (a number will be concatenated to it).

edit-keys.return, edit-keys.backspace, edit-keys.delete keys used for editing, see `simple-ed-win`, p 24

dvroom.auto-add if set to `t` (default `()`), new windows are automatically added to the current active dvroom, if there is one.

dvroom.icon-box if set to `t`, dvrooms are no more exclusive (i.e., opening one do not close the others anymore).

Dwight Shih `<dwight@s1.gov>` added the functions:

roll-rooms-up, roll-rooms-down , to sequentially open the next room. This is very handy to bind to a function key for instance.

magic-dvroom-attach , which look for all the window with name in the form `<Room>::<Name>` and incorporates them into the dvroom `<Room>`, if any exists.

dvroom-remapping unmap all windows belonging to a dvroom.

3.4 Group Iconification icon-groups.gwm

Loading the `icon-groups` package redefines the `iconify-window` function to use only one icon for all windows of the same group. Iconifying the group leader will iconify all the windows in the group, whereas iconifying a non-leader member of the group will only unmap it and map the common icon if it is not already present.

You can specify which groups you do not want iconified this way by setting their class in the list `icon-groups.exclude`. For instance, if you want to iconify your XPostits this way, but not you emacs or xmh windows, add this in your .profile.gwm:

```
(load "icon-groups")
(setq icon-groups.exclude '(Xmh Emacs))
```

It will also add two more items in the menu:

Iconify Group to iconify all the windows belonging to the group of the current window.

Iconify Others to iconify all the windows belonging to the group of the current window, but not the current window.

3.5 Opaque move move-opaque.gwm

Loading the `move-opaque` package redefines the `move-window` function to move the whole window, not just an outline of it. You can control which windows will be moved this way by setting two context variables:

move-opaque.condition will be evaluated, and if the result is non-nil, the window will be moved in an "opaque" way, otherwise the standard outline dragging will be used. Default is to move only the windows whose pixel aera is less than the `move-opaque.cutoff-area` value, which could be specified by:

```
(setq move-opaque.condition
    '(< (* window-width window-height)
        move-opaque.cutoff-area)
```

move-opaque.cutoff-area which defaults to 250000, used when discriminating windows by size.

3.6 Delta deltabutton.gwm

The `deltabutton` function is used to perform two different action on the press of a button, depending on whether the user release the button without moving the mouse more than `deltabutton.delta` pixels (defaults to 4) in any direction. To use deltabutton, you must have loaded the `deltabutton.gwm` file, and in a transition of a fsm triggered by a buttonpress event, this function will wait for the button to be released, and return `t` if the pointer has moved more than `deltabutton.delta` pixels, or `()` if not.

For instance, to raise a window if you click on it, and to move it only if you move the mouse more than 4 pixels, declare in your .profile.gwm:

```
(load 'deltabutton)
(setq standard-behavior
   (state-make
      (on (buttonpress select-button alone)
          (if (deltabutton)
              (progn (raise-window)(move-window))
            (raise-window)))
      standard-behavior))
(reparse-standard-behaviors)
```

3.7 Floating windows float.gwm

Rod Whitby `<rwhitby@adl.austek.oz.au>` made this package to interactively make some windows "float" always on top of others, or always "sink" to the back of the screen. Loading this package will add a multiple menu item to the window menu to make the current window float Up, Down, or to make it a normal window back again (item "No").

3.8 Unconfined-move unconf-move.gwm

Rod Whitby `<rwhitby@adl.austek.oz.au>` made this package to be able to still move and resize windows out of screen boundaries even when you confined them by `confine-windows` (see p 40). With this package loaded, unconfined move is obtained by moving/resizing with Control-Alt mouse buttons, while Alt mouse buttons keep moving/resizing in confined mode.

3.9 Suntools-keyssuntools-keys.gwm

Rod Whitby `<rwhitby@adl.austek.oz.au>` made this package to provide some suntools-like keyboard shortcuts to window management functions:

`L5 or F5` raise window to top, or lower it if it is already on top.

`L7 or F7` iconify/ de-iconify window.

3.10 Mike Newton's Keys mon-keys.gwm

Mike Newton `<newton@gumby.cs.caltech.edu>` is another package to add keyboard shortcuts to window management functions.

`Button 1 (alone or w/ Alt)` raise or move

`Button 3 (w/ Alt-Control)` iconify or raise

F1 (alone) choose next window

F2 (alone) choose previous

F1 (w/ alt) circulate down (no focus change)

F2 (w/ alt) circulate up (no focus change)

F3 (alone) open / close

F4 (various) change window sizes (not Emacs!)

F5 (alone) raise

F11 or F9 (alone, in root) emergency – map everything

F12 or F10 (alone) exec cut buffer, printing results

3.11 Standard pop-up menus std-popups.gwm

This package implements a very simple pop-up menu package. The variables `window-pop-items`, `icon-pop-items`, and `root-pop-items` contains a list of menu item which be used after reading user `.profile.gwm` to build the actual menus[4], which will be named `window-pop`, `icon-pop`, and `root-pop`[5].

You can then insert or delete items in this list at will. Nil entries in this list will just be skipped by the actual menu creation routine. You may want to use the function `insert-at` (see 3.16.3, p 28). Dvrooms and vscreen are example of packages adding menu items in the standard menus.

Menu items can be created with the help of the following functions:

(pop-label-make <Label>) to create an inactive label on top of the menu, where `Label` is the string to be displayed as the title of the menu.

(item-make <Label> <Expr>) to create a label triggering a WOOL function call where `Label` is the string to be displayed as the item of the menu, and `Expr` is WOOL code which will be evaluated when releasing the button in the item.

(multi-item-make <item-desc>) where `item-desc` can be of the form:

- `<Label>` creates an insensitive label with this `<Label>` as text.
- `(<Label> <Expr>)` creates a button with text `Label` triggering the evaluation of the wool expression `Expr`.
- `()` leaves an extensible space.

Note: in fact, in the preceding functions, any `<Label>` can be in fact either a string, an already built pixmap which will be used as-is, or WOOL code that will be evaluated and must return a pixmap which will be used to build the menu item.

For instance, the default window menu is the list:

```
'((item-make "iconify" (iconify-window))
    (item-make "Exec cut"
               (execute-string (+ "(? " cut-buffer ")")))
    (item-make "client info" (print-window-info))
    (item-make "redecorate" (re-decorate-window))
    (item-make "kill" (if (not (delete-window))
                          (kill-window)))
    ))
```

[4] the actual menu will be build by loading the package whose name is defined by the value of the variable `menu.builder`, thus alternative menu packages are thus free to redefine this value.

[5] if the user defines any of this variables in his profile, it overrides the building of the corresponding menu from the lists.

Moreover, you can control the appearance of the label and the items of the menu by the following variables:

pop-item.font font of the items

pop-item.foreground color of their text

pop-item.background color of their background. Due to the simple menu item highlighting code in this package, all items must have the same colors

pop-label.font font of the labels on top of menus

pop-label.foreground color of their text

pop-label.background color of their background

Default action

Menus can have a default action, i.e. wool code which is triggered if the user lets go the mouse button before the menu appears. Default actions should be as harmless as possible, of course. They can be set by

```
(menu-default-action <Menu> <Expr>)
```

where `<Menu>` is the menu (`window-pop`, `icon-pop`, or `root-pop`), and `<Expr>` is the code to be excuted.

```
(menu-default-item <Menu> number)
```

sets the item in which the mouse cursor is when popping up the menu (defaults to first item).

3.12 Gosling Emacs mouse support emacs-mouse.gwm

This package implements a way to use the mouse with the **Gosling** emacs, sold by **Interpress**. You will need to load the emacs macros contained in the `gwm.ml` file included in the distribution in your emacs. Then, in a window decorated with the `simple-ed-win` package, pressing **Control** and:

left mouse button will set the emacs text cursor under the mouse pointer

middle mouse button will set the mark under the mouse pointer

right mouse button will pop a menu of commonly-used emacs functions (execute macro, cut, copy, paste, go to a C function definition, re-do last search)

Clicking in the mode lines will do a full screen recursive edit on the buffer if not in a target and in the targets:

`[EXIT]` will delete the window if it is not the only one on the screen, or do an `exit-emacs`

`[DOWN]` will scroll one page down the file

`[ UP ]` will scroll one page up the file

This package is included rather as an example of things that can be done to work with old non-windowed applications than as a recommended way of developping code.

3.13 The customize function

```
(customize deco screen window-description
           variable1 value1 variable2 value2...)
```

Most following sample window and icon decorations can be tailored in a global way by setting global variables in your `.profile.gwm` before using the decoration, but these variable can be set individually to decorations by use of the customize function. For instance, since the simple-icon documentation tells you that title is added to icons according to the value of the `simple-icon.legend` global variable, you can say that you do not want legends under your icons, except for xclocks by:

```
(set-icon-window XClock simple-icon)
(setq simple-icon.legend ())
(customize simple-icon any XClock
           legend t)
```

Note: In current version, only `simple-win`, `simple-icon`, and `term-icon` decorations support the `customize` function.

The customization arguments can be given as a single list argument. In other words, both following calls are equivalent:

```
       (customize simple-icon any XClock legend t
   background (color-make "green"))
       (customize simple-icon any XClock
                  (legend t
                   background (color-make "green")))
```

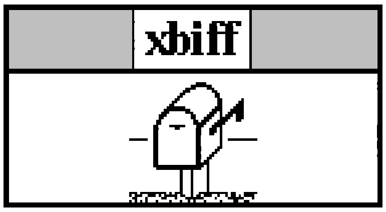
Note: Moreover, customisation values can be also given to decorations which support the customize protocol as arguments (do not forget to quote the variable names, the decoration functions evaluate their arguments). Thus we can define a new decoration `clock-deco`, and use it afterwards just as another decoration with the same results as the preceding examples:

```
(require 'simple-icon)      ; simple-icon must be defined
        (setq clock-deco
              (simple-icon legend t
                           background (color-make "green")))
        (set-icon-window XClock clock-deco)
```

3.14 Sample window decorations `*-win.gwm`

These are standard window decorations which can be used via the `set-window` function of the standard profile. They can be found in files whose names end in `-win.gwm` in the distribution directory of GWM.

3.14.1 Simple window `simple-win.gwm`



This is a really simple window decoration with only a title bar on top of the window. The name of the window is centered in the bar. The title bar and the name of the window can change appearance when they become "active" (i.e. have the keyboard focus).

This style is customizable by setting the following variables at the top of your .profile, before any call to **set-window**:

simple-win.font simple-win.active.font fonts used for printing the title

simple-win.label.borderwidth simple-win.active.label.borderwidth borderwidth of the title plug in the bar

simple-win.background simple-win.active.background background color of the title bar

simple-win.label.background simple-win.active.label.background background colro of the title plug

simple-win.label.foreground simple-win.active.label.foreground pen color to draw window name in the title plug

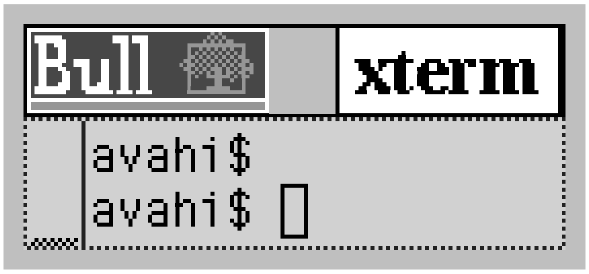
simple-win.label.border simple-win.active.label.border color of the border of the title plug in the bar

As you can see, each variable comes in pair, one for "inactive" state, the other for "active" state. for each of them, the "active" one can bet set to **()**, which means just use the same value as the "inactive" value.

Since this decoration supports the **customize** function, all the above values can also be set via the customize function, or as arguments to the function **simple-win** itself. In these cases, not that you must use the name of the variables without the **simple-win.** prefix, e.g. you could have all simple-win windows with title font written in black, except for XClock by the calls:

```
(setq simple-win.active.label.foreground black)
(customize simple-icon any XClock
        active.label.foreground (color-make "green"))
```

3.14.2 Simple editable window simple-ed-win.gwm



This decoration has a titlebar on top of it, including an iconification plug and an editable name plug. Moreover the whole border changes color to track input focus changes.The look of this frame can be altered by setting the following variables:

icon-pixmap to a pixmap used as an iconification button.

simple-ed-win.borderwidth to the width in pixels of the sensitive border of the window

simple-ed-win.font to the font used for printing the title.

simple-ed-win.active to the color of the top bar and border for the window having the keyboard focus (defaults to darkgrey).

simple-ed-win.inactive to the color of the top bar and border when the window do not have the keyboard focus (defaults to grey).

`simple-ed-win.label.background` background color of the name.

`simple-ed-win.label.foreground` color of text of the name.

When you click in the icon button at the left of the titlebar (whose pixmap can be redefined by setting the global variable `icon-pixmap` to a pixmap) with the left button, the window is iconified. If you click with the middle button, you will be able to drag the ouline of the icon and release it where you want it to be placed.

If you double-click a mouse button, or do a click with the **control** and **alternate** keys depressed, in the editable name plug at the right of the titlebar, you will be able to edit the name of the window (and the associated icon name) by a simple keyboard-driven text editor whose keys are are given as strings in the following variables:

`edit-keys.return` to end edition (defaults to `"Return"`).

`edit-keys.delete` to wipe off all the text (defaults to `"Delete"`).

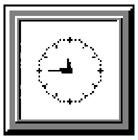
`edit-keys.backspace` to erase last character (defaults to `"Backspace"`).

any key to be appended to the text.

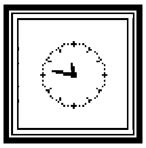
The plug will invert itself during the time where it is editable. You quit editing by pressing **Return**, double-clicking in the plug or exiting the window.

This decoration style also supports the `emacs-mouse` package.

3.14.3 Frame `frame-win.gwm`



look-3d t



look-3d ()

This decoration consists in a frame around the window. The look of this frame can be altered by setting the following variables:

`look-3d` to `t` to have a “3D-looking” frame (left figure) instead of the 2d-looking one (right figure).

`frame.top-text` to an object which will be evaluated to yield the text to be put on top of the frame

`frame.bottom-text` to an object which will be evaluated to yield the text to be put on the bottom of the frame, for instance:

```
(setq frame.bottom-text '(machine-name))
```

`frame.pixmap-file` to the prefix of the 8 bitmaps (or pixmaps) files used to build the frame. The suffixes will be `.tl .t .tr .r .br .b .bl .l`, clockwise from upper left corner.

`frame.pixmap-format` to the format of the files: `'bitmap` (default) or `'pixmap`

`frame.bar-width` to the width of the four bars (should match the pixmap files)

`frame.inner-border-width` to the inner border width

3.15 Sample icons `*-icon.gwm`

These are standard icon windows descriptions which can be used via the `set-icon-window` function of the standard profile. They can be found in files whose names end in `-icon.gwm` in the distribution directory of GWM.

3.15.1 Simple icon `simple-icon.gwm`

This icon consists in an (optional) image and the icon-name of the window below it. The image is by priority order:

the user pixmap set by the `set-icon` (see 3.1.2, p 15) call

the window the client has set in its hints to the window manager to use as an icon

the pixmap the client has set in its hints to the window manager to use for its icon

Used context variables:

`simple-icon.font` for the font used to display the icon name.

`simple-icon.legend` boolean flag telling to add the icon name under the icon for the application. Defaults to `()`.

`simple-icon.foreground` pen color of the icon name

`simple-icon.background` background color of the icon

`simple-icon.borderwidth` borderwidths used

`simple-icon.borderpixel` color of the borders

This decoration supports the `customize` (see 3.13, p 23) function.

3.15.2 Terminal display icon `term-icon.gwm`

This icon looks like a small computer display with the window name inside it. Note that the icon resizes itself to adjust to the dimensions of the displayed name.

Used context variables:

`term-icon.font` for the font used to display the icon-name.

`term-icon.foreground` for the color of text and decorations (defaults to black)

`term-icon.background` for the background color (defaults to white)

`term-icon.borderwidth` defaults to 0

`term-icon.borderpixel` defaults to black

This decoration supports the `customize` (see 3.13, p 23) function.

3.16 Utilities `utils.gwm`

This package implements some useful functions for the WOOL programmer. it is automatically loaded by the standard profile. List this file to see the current ones.

3.16.1 Standalone Buttons: `place-3d-button`

labelplace-3d-button

`(place-3d-button text pencolor maincolor wool-expression)`
`(place-button text pencolor ulb normal pressed lrb wool-expression)`

Will create and place as an independent window (of client class `Gwm`, client name `button`, and window name `text`) a 3d-looking window with visual feedback when pressed, which will execute the `wool-expresssion` when pressing any button in it. `pencolor` is the color of the text, and `maincolor` must be one of the shaded colors in the `/usr/lib/X11/rgb.txt` file[6].

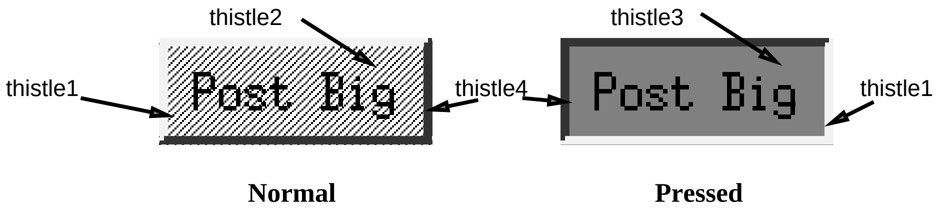
If you want to tailor colors yourself, use the `place-button` form, where you must choose the upper-left and lower-right border colors as well as the background colors of the button when normal and depressed. `place-3d-button` calls in fact `(place-button text pencolor maincolor1 maincolor2 maincolor3 maincolor4 wool-expression)`

To have a demo of this feature, you can execute the `(demo-button)` function, which will make a button cycling through all the colors in the `shaded-colors` list.

You can implement different behaviors depending on button pressed and modifier by looking at the value of `(current-event-modifier)` and `(current-event-code)` in the `wool-expression` body. Look at the end of the file `profile-colas.gwm` for examples.

The following code will create a button that will toggle iconification of all the big postits on my screen, the button being in the thistle range of colors.

```
(place-3d-button "Post Big"
  black 'thistle
  '(for window (list-of-windows)
    (if (= window-name 'PostItNoteBig)
      (if window-is-mapped (iconify-window)
        (progn (map-window)(raise-window))
))))
```



3.16.2 Matching windows by regular expressions: `match-windowspec`

`(match-windowspec windowspec)`

Where `windowspec` is a property list with `'client-class`, `'client-name`, and `'window-name` as possible tags. Windowspecs can themselves contain regular expressions.

```
((list 'client-class "XTerm" 'window-name ".*build"))
```

[6] for instance, `pink` is such a color, since the color table contains the entries `pink1`, `pink2`, `pink3`, and `pink4` in clear-to-dark order. The full list of such colors is in the `shaded-colors` list.

will match (return `t`) all xterms whose window name ends in `build`.

This code was provided by Jay Berkenbilt `<qjb@ATHENA.MIT.EDU>`.

3.16.3 insert-at

```
(insert-at element list position)
```

Utility function to insert an element `element` in a list `list` at position `position`. The list is physically modified in place. useful to insert items in menu lists.

3.17 User-contributed utilities

Some user-provided useful little hacks or programming helps have been included too.

3.17.1 Near-mouse `near-mouse.gwm`

This placement function was provided by Eyvind Ness `<eyvind@hrp.no>`. Saying:

```
(require 'near-mouse)
        (set-placement XPostit near-mouse)
```

will make all newly created XPostit windows pop up near the mouse.

3.18 Programming Styleguide for the standard distribution

The styleguide to write decorations is to be written. Until then, look at existing files such as `.gwmrc.gwm, .profile.gwm, simple-ed-win.gwm` to see what is the current style. We will appreciate all feedback to these conventions, which are not settled yet.

The main idea is that a decoration package `foo` should, when loaded, define a `foo` function which, once executed will return the appropriate decoration, using the pre-defined behaviors.

All persistent variables of the packages should be prefixed by the package name, as in `foo.bar`.

Do not forget to define in fact one decoration per screen or be cautious not to mix colors, pop-ups, pixmaps and cursors from screen to screen. Use `defname screen.` to declare screen-specific variables.

The user should be allowed to customize the decoration by setting global variables in its `.profile.gwm` file, which will be interpreted during the loading of your package.

The fsm you make for your package should be contructed from the behaviors defined in `.gwmrc.gwm`, such as standard-behavior, standard-title-behavior, window-behavior, icon-behavior, root-behavior, or already built fsms such as fsm, window-fsm, icon-fsm, root-fsm.

All values you want to attach to windows or wobs should be put as properties in the property-list of the wobs, by calls to `(## 'key wob value)`

The main idea is, if you must modify `.gwmrc.gwm` to code your window decoration, mail us your desiderata and/or enhancements, so that we should be able to keep the same .gwmrc for all decorations. I maintain mailing lists for people to exchange ideas about GWM. Mail me a request if you are interested at `gwm@mirsa.inria.fr`.

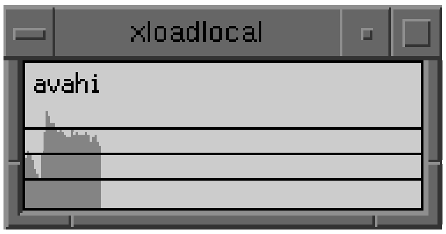
3.18.1 The `simple-win` example

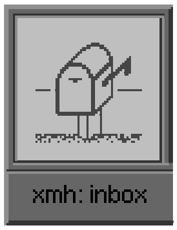
In the distribution, you can look at the `simple-win.gwm` file to see how to define a proper decoration. In this file you will see how to define a decoration that supports multiple screens, and some user customization via the `customize` function. The trick is not to forget to put at build time all necessary information (in the `property` field of the window) for using later during normal operation, where all code is triggered by the decoration `fsms`.

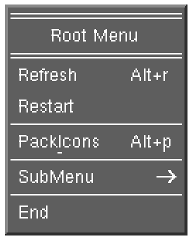
3.19 Other profiles

Other nice profiles have also been developed in parallel to the standard profile, but they have not been integrated yet, i.e. they need their own `.gwmrc.gwm`.

3.19.1 The MWM emulation package `mwm.gwm`







Frederic Charton made this profile emulating the **M**otif **W**indow **M**anager. To use it, you must give the command line option **-f mwm** to GWM.

You can customize it by copying in your **gwm** directory (in you GWMPATH) the files:

`mwmrc.gwm` all the resources settable in ".mwmrc" for MWM are also settable here, except for the menus.

`mwm-menusrc.gwm` description of the menus.

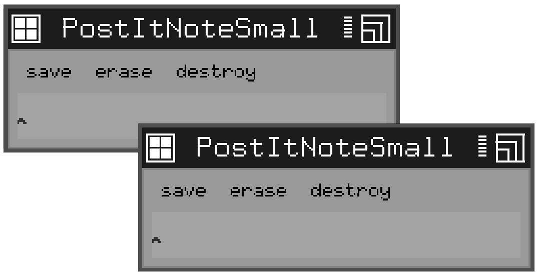
`mwmprofile.gwm` miscellaneous wool customizations (needs WOOL knowledge)

and editing it. You may want to get the MWM manual for the description of all the available functions. For instance, to set the input focus management from "click to type" (default) to "real estate" (input focus is always to the window underneath the pointer), edit `mwmrc.gwm` and change the line `(: keyboardFocusPolicy 'explicit)` to `(: keyboardFocusPolicy 'pointer)`.

Warning: This profile is still mono-screen, i.e. to manage 2 screens on your machine, you must run 2 gwms, for instance by:

```
gwm -1 -f mwm unix:0.0 &
gwm -1 -f mwm unix:0.1 &
```

3.19.2 The TWM emulation package `twm.gwm`



Arup Mukherjee (`arup@grasp.cis.upenn.edu`) made a TWM emulator. To use it, you must give the command line option **-f twm** to GWM.

You can customize it by copying in your `gwm` directory (in you GWMPATH) and editing the files:

`twmrc.gwm` Contains numerous options (mainly colors) that can be set from here. The file is well commented, and most of the color variables have self-explanatory names. You can also specify from here whether or not the icon manager code is to be loaded. It also contains definitions for the three variables `emacs-list`, `xterm-list`, and `xload-list`. The specified hostnames are used to build menus from which you can have gwm execute the respective command on a host via the "rsh" mechanism (note that your .rhosts files must be set up correctly for this to work). Note that unlike with the standard profile, the rxterm and rxload scripts are **NOT** used.

`twm-menus.gwm` The contents of all the menus are specified here. To change more than the xterm, xload, or emacs lists, you should modify this file.

`twm.gwm` The only things that one might wish to customize here are the behaviors (which specify the action of a given button on a given portion of the screen)

3.20 Troubleshooting

To debug a WOOL program, you can:

- use the `trace` function to trace code execution or evaluate an expression at each expression evaluation.
- read, execute, and print WOOL code by selecting it and using the **Exec cut** (for execute cut buffer) menu function.
- use the `-s` command line option to synchronise X calls, if you want to know where you issue a non-legal X call.
- compile GWM with the -DDEBUG compile option, which will turn on many checks (stack overflow, malloc checks, etc...) in case you manage to make GWM crash.
- If gwm appears to freeze, it might be because of a bus error. Running gwm under a debugger such as `dbx` to see where it crashes.

4 WOOL Reference manual

; — WOOL comment

`; any text up to the end of line`

Comments in WOOL begin with a semicolon and end at the end of the line.

! — executes a shell command

`(! command arguments...)`

Executes (forks) the command given in string with its given arguments, and does not wait for termination (So you don't need to add a final "&"). For instance, to open a new "xterm" window on the machine where GWM is running, execute `(! "xterm")`. This is an interface to the **execvp** call: the command is searched for in the **PATH** variable, and executed via **/bin/sh** if it is a command file, directly executed otherwise.

Examples:

```
(! "xterm")                        ; machine-executable code
(! "rxterm" "foobar")              ; or shell script
(! "xterm" "-display" "bar:0.1")  ; with arguments
(! "/bin/sh" "-c" "for i in a b c;do xclock -display $i:0;done")
                                   ; needs a shell for commands
```

#
nth — accesses an element of a list

```
(# n list [object])
(# atom list [object])
```

Gets (or sets to `object` if present) the n-th element of the list `list` (starting with 0). Increases the list size if needed (with nils). When the first argument is an atom, this references the element just after the atom in the list, thus providing the ability to maintain classical Lisp "property lists".

Note that this function does not do a physical replacement, and constructs a new list with the `list` argument. (see **replace-nth**, p 32)

Examples:

```
(# 2 '(a b c d))            ==>    c
(# 2 '(a b c d) 'foo)       ==>    (a b foo d)
(# 'x '(x 4 y 6))           ==>    4
(# 'y '(x 4 y 6) 8)         ==>    (x 4 y 8)
(# 'z '(x 4 y 6) 10)        ==>    (x 4 y 6 z 10)
(# 6 '(a b c d) 'foo)       ==>    (a b c d () () foo)
```

In fact the index can be any Lisp object, but as the list is scanned to find the **same** object (EQ predicate), using atoms is the safest way.

```
(# "foo" '("foo" 1))           ==>     ()
(progn (setq name "foo")
       (# name (list name 1))) ==>     1
```

Note: The second argument can also be a:

wob in which case the contents of the `wob-property` is taken as the list.

symbol which must evaluate to a list, which is then used by the function.

##
replace-nth — physically replaces an element of a list **EXPERT**

`(## n list object)`
`(## atom list object)`

This function physically replaces an element of the list, which is located as with the `#` function. It returns the modified list.

This provides a way to implement variable-sized lists, such as linked lists, which are the real lists in the Lisp world, but should be used with care, as you are able to create circular references with it. To allow the Wool garbage collector to handle correctly circular lists, you should always explicitly set to () the fields of a circular cell before disposing of it.
Example: a circular list with cells of the form `(car cdr)`

```
(setq elt2 '(b ()))            ; second cell of list
(setq elt1 (list a elt2))      ; first one pointing to second
(## 1 elt2 elt1)               ; we make a circular list
```

Now, if we set elt1 and elt2 to `()`, their storage will not be freed!, we must do

```
(## 1 elt2 ())
```

before setting them to `()`.

Note: The second argument can also be a:

wob in which case the contents of the `wob-property` is taken as the list.

symbol which must evaluate to a list, which is then used by the function.

This is the **only** case where lists can be extended, by specifiyng an atom which do not exist in the list. In this case, if the list is pointed to by many objects, the list is duplicated, and only the copy pointed to by the wob or atom argument is modified, e.g.:

```
(setq l '(a 1))
(setq l-bis l)                 ; l and l-bis points to the same list
(## 'b 'l 2)                   ; only l is modified in place
l                              ==> (a 1 b 2)
l-bis                          ==> (a 1)
```

(— list notation

`(elt1 elt2 ... eltN)`

This notation is used to describe a **list** of `N` elements, as in other Lisp languages. Note that Wool is not a true Lisp dialect since lists are represented internally as arrays, allowing for a greater speed and a smaller memory usage in the kind of programs used in a window manager. If you are a die-hard Lisp addict, you can still use CONS, CAR and CDR if you want by defining them as:

```
(defun cons (e l) (+ (list e) l))
(defun car (l) (# 0 l))
(defun cdr (l) (sublist 1 (length l) l))
```

Note: The WOOL parser ignores extraneous closing parentheses, allowing you to close top-level expressions in a file by a handful of closing parentheses, such as:

```
(defun cons (e l) (+ (list e) l)))))))))))))
(setq x 1))))))))
```

`()`
`nil` — the nil value

`()`

The nil value, backbone of every Lisp implementation. In Lisp an object is said to be true if it is not nil.

`""` — string notation

`"string"`

Strings are surrounded by double quotes. C-style escape sequences are allowed, that is:

Sequence	**stands for**
`\n`	newline (control-J)
`\r`	carriage return (control-M)
`\t`	tab (control-I)
`\e`	escape (control-[)
`\\`	the backslash character itself \
`\"`	double quote
`\xNN`	the character of ascii code NN in hexadecimal
`\nnn`	the character of ascii code nnn in octal
`\c`	c if c is any other character.

Moreover, you can ignore end of lines in strings by prefixing them with \, as in:

```
(print "This is a very \
long string")           ==> This is a very long string
```

`'`
`quote` — prevents evaluation

`'object`
`(quote object)`

Returns the object **without evaluating** it. The form `'foo` is not expanded into `(quote foo)` during parsing, but in a "quoted-expression" WOOL type for efficiency.

`%`
`*`
`/` — arithmetic operators

`(% n1 n2)`
`(* n1 n2)`
`(/ n1 n2)`

Returns respectively the modulo, product, and quotient of the integer arguments as a integer.

+ — adds or concatenates

```
(+ n1 n2 ... nN)
(+ string1 string2 ... stringN)
(+ list1 list2 ... listN)
```

Numerically add numbers, concatenate strings or, concatenate lists. Determines the type of the result as being the type of the first argument (`()` is a list, `""` is a string, `0` is a number).

- — arithmetic difference

```
(- n)
(- n1 n2 ...)
```

Returns the arithmetic opposition or difference of numbers. `(- n1 n2 n3 n4)` is equivalent to `(- n1 (+ n2 n3 n4))`.

=
equal — tests equality of any two objects

```
(= obj1 obj2)
(equal obj1 obj2)
```

Returns obj1 if obj1 is the same as obj2, nil otherwise. This is the traditional **equal** predicate of Lisp.

Equality of lists is tested by testing the equality of all their elements.

< — tests for strict inferiority

```
(< n1 n2)
(< string1 string2)
```

Compares two numbers or two strings and returns `t` if argument 1 is inferior and not equal to argument 2 and `nil` otherwise. Strings are compared alphabetically.

> — tests for strict superiority

```
(> n1 n2)
(> string1 string2)
```

Compares two numbers or two strings and returns `t` if argument 1 is superior and not equal to argument 2 and `nil` otherwise. Strings are compared alphabetically.

?
print — prints WOOL objects

```
(? obj1 obj2 ... objN)
```

Prints the objects, without adding spaces or newlines. Output is flushed at the end of every call to this function. For now, output goes to the stdout stream.

active-label-make — makes a label (text in a given font)

```
(active-label-make label [font])
```

Creates a label with string **label** drawn in the font **font**. The text will be redrawn on each expose (The active label can also be used to paint a string on a pixmap, see **pixmap-make**, p 66).

Context used:

Variable	used for
`foreground`	color of the text string
`font`	font of the string if not given

`allow-event-processing` — un-freezes the server after catching a replayable event

`(allow-event-processing)`

When you set passive grabs on "replayable" events (see **replayable-event**, p 70), when such event is caught in a fsm, the grab is activated and the server is frozen, in such a way that pointer motions or button states is not tracked anymore. To be able to use a function such as **current-mouse-position** (see p 42) you must then allow the processing of events by calling this function.

After the call, you will not be able to use **ungrab-server-and-replay-event** (see p 79) for the event.

`alone` — specifies that no modifier key is used

Constant

Specifies that no modifier is pressed for a button or key event for use in event descriptions.

`and` — logical AND of expressions

`(and obj1 obj2 ... objN)`

Returns `()` as soon as an argument is false (`nil`), `t` otherwise.

`any` — matches any modifier or button

Constant

Matches any modifier or button or key in events. It can be also used in many other functions with appropraite semantics.

`atoi` — ascii string to integer conversion

`(atoi string)`

Returns the integer described by string (in base 10).

Example:

```
(atoi "123")                ==>     123
```

`atom` — makes an atom from a string

`(atom string)`

Returns the atom of name string. Useful to create atoms with special imbedded characters, such as `'` or blanks. The inverse function is not necessary since every WOOL function expecting strings in arguments can accept atoms instead, and will take the string of their name.

This should be used when accessing property lists via the `#` function. For instance if you store properties on machine names (which are strings), you would use the following call to retrieve the color of a xterm for a machine:

Examples:

```
(# (atom machine-name) '(Maalea "Green" Honolua "Blue"))
```

`background` — color of the background

Numeric variable — screen-relative (color)

The value of this global variable is used by many functions as the color of the background. It is a pixel value, such as returned by `color-make`, and is initially set to the pixel of the color `White`. Note that a non-nil `tile` context value overrides the background value.

`bar-make` — makes a bar descriptor

`(bar-make plug1 plug2 ... plugN)`

Creates a bar of transversal width[1] bounded by the current value of `bar-min-width` and `bar-max-width`, and containing the `N` plugs, which are centered in the bar. If a plug is `()`, it is considered extensible space, which is expanded when the bar stretches to its actual dimensions. The plugs are clipped on the right if necessary.

Context used:

Variable	used for
`fsm`	the finite state machine of the bar
`borderwidth`	the width of the bar's border in pixels
`borderpixel`	the color of the border
`bordertile`	the pixmap used as the border pattern
`background`	the color of the background
`plug-separator`	the number of pixels between consecutive plugs
`tile`	the pattern of its background (may be ())
`menu`	the default pop-up associated with it
`cursor`	the cursor's shape when the cursor is in it
`property`	the WOOL property associated with the bar
`bar-min-width`	the minimum width in pixels of the bar
`bar-max-width`	the maximun width in pixels of the bar (width for vertical bars, height for horizontal bars)

Note that the plugs are evaluated a second time when the bar is physically created, allowing you to give expressions for plugs (quoted to evade the first evaluation of arguments of the `bar-make` function itself) will evaluate to a plug on the realisation of the wob.

For instance, to have the window name in a plug, the bar can be made by:

[1] Bars have their own width and a length which is adjusted to fit around the client window. For an horizontal bar, the width is their physical height and the length their physical width, and for a vertical one, it is the opposite

```
(bar-make '(plug-make (label-make window-name)))
```

If you don't quote the plug, all the bars will have the same name on the realisation of the bar, the value of `window-name` at the time of the call to bar-make.

`bar-min-width`
`bar-max-width` — limits to the transversal width of a bar

Numeric variable (number)

The values of these global variables are used by the constructors of bars to limit their transversal width, clipping the plugs if necessary. `bar-min-width` defaults to 1 and `bar-max-width` defaults to 100.

`bar-separator` — number of pixels between consecutive bars in menus

Numeric variable (number)

The value of this global variable is used by the constructors of menus to yield the number of pixels separating two contiguous bars. Default is 4 pixels. Used in `menu-make` function.

`bell` — rings the keyboard bell

`(bell [percent])`

Rings the bell on the keyboard, with volume set by the `percent` numeric argument, which can take values ranging from -100 to 100 (defaults to 0). -100 means lowest volume, 0 means base volume, 100 means full volume.

`bitwise-and`
`bitwise-or`
`bitwise-xor` — bitwise operators

`(bitwise-op n1 n2 ... nN)`

Returns the result of the bitwise operator on the N arguments.

Examples:

```
(bitwise-or 1 4)           ==>    5
(bitwise-and 3 5)          ==>    1
(bitwise-xor 3 5)          ==>    6
```

`borderpixel` — color of the border of a wob

Numeric variable — screen relative (color)

The value of this global variable is used by the constructors of all wobs as the color of their border. It is a pixel value, such as returned by `color-make`, and is initially set to the pixel of the color `Black`. It is always overridden by the value of `bordertile` if not nil.

`bordertile` — pixmap to tile the border of a wob

Variable (pixmap)

The value of this global variable is used by the constructors of all wobs as the pixmap to display in their border. It is a pixmap object, such as returned by `pixmap-make`. If set to (), the borderpixel value is used.

`borderwidth` — width in pixels of the border of a wob

Numeric variable (number)

The value of this global variable is used by the constructors of all wobs to set their border width in pixels. A value of 0 means no border.

`boundp` — tests if an atom has already been defined

`(boundp atom)`

Returns the (evaluated) atom if it has been defined, () otherwise.

Examples:

```
(setq foo 1)
(boundp 'foo)                ==>    foo
(boundp 'bar)                ==>    ()
```

`button` — makes a button event

`(button number modifier)`

Returns a button event matching the X event **ButtonPress** of button number `number`, and will verify that the modifier argument was depressed during the click. This event will wait for the corresponding **ButtonRelease** event before returning.

Number or modifier can take the value `any`, thus matching any of the values of the corresponding argument. These button events can then be used in the matching part of the transitions of fsms or in the `grabs` field of a window-make description.

Examples:

`(button 1 with-alt)`	matches clicking left button of the mouse while keeping the alt key depressed.
`(button any with-control)`	matches control-clicking with any button

`buttonpress` — makes a buttonpress event

`(buttonpress number modifier)`

Same as button, but does not wait for the buttonrelease, meaning that the buttonrelease is available for another transition. This can be used to trigger `move-window`, since this function waits for a button release to terminate.

`buttonrelease` — makes a buttonrelease event

`(buttonrelease number modifier)`

Same as button, but only matches a button release. Useful when you cannot wait for a button event, for instance when iconifying a window, because if the window is iconified on a button press, the correponding release event will be lost (in fact will go to the window underneath)

`check-input-focus-flag` — follow input hint for setting focus

Numeric variable (number)

If this flag is set to **1**, which is the default, GWM will refuse to set the keyboard focus via the **`set-focus`** call to windows having declared that they didn't need the focus. This flag is provided so that you can use clients which set their input hint the wrong way.

`circulate-windows-down`
`circulate-windows-up` — circulates mapped windows

```
(circulate-windows-up)
(circulate-windows-down)
```

Put the topmost window down, or the lowest window up respectively.

`color-components` — gives RGB color decomposition of a pixel

```
(color-components pixel)
```

Given a pixel (integer, normally returned by **`color-make`**), returns a list of 3 integers, the Red, Green, Blue components of the color. Each of this integer can take values between 0 and 65535.

`color-free` — de-allocates a pixel

```
(color-free pixel)
```

Frees the entry **`pixel`** in the default colormap. **`pixel`** must have been the result of a previous **`color-make`** call. This is a very dangerous function, since colors returned by **`color-make`** can be shared with multiple applications.

`color-make` — allocates a pixel color by name

```
(color-make color-name)
```

Takes a string **`color-name`** describing the color by its english name (as found in the **`rgb.txt`** system file), and allocates a new entry in the default colormap with a pixel of this color, and returns it if it wasn't allocated before, otherwise just returns the previously allocated pixel if an entry in the colormap already existed. Two consecutive calls with the same argument should then return the same value. A pixel is a number which is the index of the color in the colormap.

This pixel can later be used as values for foreground and background, or in the **`pixmap-make`** function. If color is not found, prints a warning, and returns the pixel used for the black color.

Colors can also be specified directly by their **RGB**[2] values, with the **#**-convention: if the **`color-name`** string begins with a sharp sign, the following characters are taken as hexadecimal digits specifying the most significant part of the Red, Green and, Blue values, with the following syntax: (each letter stands for a digit)

```
#RGB
#RRGGBB
#RRRGGGBBB
#RRRRGGGGBBBB
```

[2] Red, Green, Blue.

Example:

```
(color-make "DarkSlateBlue")
(color-make "light grey")
(color-make "#f00")            ; pixel for red
(color-make "#300000a076be")
```

color-make-rgb — creates a color from RGB values

`(color-make-rgb red green blue)`

This function takes three numeric arguments giving the values of the red, green and blue components of the color which will be returned. Components are scaled between 0 and 65535 (half brightness is 32767 and no light is 0).

compare — ordering comparison

`(compare n1 n2)`
`(compare string1 string2)`

Compares two numbers or two strings and returns `-1`, `0` or `1` if its first argument is lesser than, equal to or, greater than the second.

cond — conditional test

`(cond (condition then) [(condition then) ...])`

This is the classical "cond" Lisp function, returning the evaluation of the "then" part of the first true condition.

```
(defun fib (n)
    (cond
        ((= n 0) 1)
        ((= n 1) 1)
        (t (+ (fib (- n 1))
              (fib (- n 2))))))
```

confine-grabs — cursor stays confined in grabbing wobs

Numeric variable (number)

If set, during all grabs (either via `pop-menu` or `grab-server`) will confine the pointer inside the grabbing wob during the duration of the grab.

confine-windows — forces windows to stay on-screen

Numeric variable (number)

If set, during all interactive moves or resizes, gwm will ensure that windows stay entirely within screen bounds.

context-save
context-restore — context management

`(context-save context)`
`(context-restore context)`

A context is a kind of property list, it is an even-sized list whose even elements are atoms and whose odd ones are their values (see the **with** function), of the form **(var*1* val*1* var*2* val*2* ... var*N* val*N*)**. **context-save** creates a new context where the **val*i*** are the result of the evaluation of the **var*i*** in the argument context, whereas context-restore does a **(setq var*i* val*i*)** for each of its variable/value pairs.

Note: the provided **val*i*** serves as default for **context-save** in the case where the corresponding **var*i*** is undefined:

```
(setq my-context (list 'a 1 'b 2))
(setq a "foo")                      ; b is undefined
(print (context-save my-context))
                                ==>  (a foo b 2)
```

copy — copies a wool object **EXPERT**

`(copy object)`

Returns a copy of the `object` argument, which can only be a list for now. This function is flagged as "*Expert*" because it is of use to people doing physical replacement functions, which are reserved to experts.

current-event-code — code of the last event

`(current-event-code)`

Returns the code (button or keycode) of the **last** event received (The one which triggered the transition you are in).

current-event-from-grab — tests if last event was generated by a grab

`(current-event-from-grab)`

If the last event was a crossing or focus event consecutive to a grab set or removed, returns **t**.

current-event-modifier — modifier of the last event

`(current-event-modifier)`

Returns the modifier (state of shift, control, alternate,...) keys that were depressed when the **last** event occurred (The one which triggered the transition you are in). Note that the combination of two modifiers is expressed by bitwise-**or**ing the modifiers.

current-event-time — time in milliseconds of the last event

`(current-event-time)`

Returns the time at which occured the last button, key, crossing, or focus event. Time is expressed as a number of milliseconds.

current-event-window-coords — relative position of the last event

`(current-event-window-coords)`

Returns the list of coordinates of the last event, if it was a pointer or a key event, relative to the current client window.

This list has six elements:

Element	Meaning
0	x position in size increments
1	y position in size increments *(character positions for xterm)*
2	same x position in pixels
3	same y position in pixels
4	x position in pixels relative to the decorating window
5	y position in pixels relative to the decorating window

WARNING: The position in size increments do not work in the general case, but only for windows having the text widget flush to the bottom right corner of their main windows, like Xterm. There is no fix for it, but this function is a kind of a hack anyways.

`current-event-x`
`current-event-y`
`current-event-relative-x`
`current-event-relative-y` — position of the last event

```
(current-event-x)
(current-event-y)
(current-event-relative-x)
(current-event-relative-y)
```

Returns the x or y coordinate (number) of the mouse during the **last** event, if it was a key or button event (The one which triggered the transition you are in).

The first coordinates are relative to the root, whereas the last ones are relative to the wob where they occurred.

`current-mouse-position` — queries server for mouse position

```
(current-mouse-position)
```

Queries the server for the mouse state and returns it as a list of four element where the first element is x, second is y, third is state of modifiers and buttons bitwise-**or**ed, and fourth is the number of the screen where the pointer is.

`current-user-event` — name of the last user event

```
(current-user-event)
```

If the last event was an user event, returns the label of the event (the atom that was given as argument to the `send-user-event` call). Triggers an error if the last event was not a user event.

`cursor` — shape of the cursor in a wob

Variable (cursor)

The value of this global variable, which must be a cursor returned by a call to **`cursor-make`**, is used by the constructors of all wobs to set the appearance of the mouse cursor when the pointer is in the wob. If set to (), the cursor will not change when entering the wob.

This is also used by functions like **`grab-server`**, **`move-window`**, and **`resize-window`** to change the shape of the cursor during a function.

`cursor-NW` — cursor shapes for the 8 directions

Variables (cursor)

These values (**`cursor-NW, cursor-NE, cursor-N, cursor-SW, cursor-SE, cursor-S, cursor-E, cursor-W`**) defines cursors to be used on the eight directions by some functions, such as resize-window with the mwm-like style (**`resize-style`**, p 70). The eigth corners are respectively: NorthWest, NorthEast, North, SouthWest, SouthEast, South, East and West.

`cursor-make` — makes a cursor with a bitmap and a mask

```
(cursor-make foreground-bitmap-filename
             mask-bitmap-filename)
(cursor-make cursor-name)
(cursor-make number)
```

Constructs a mouse cursor with two bitmaps (strings containing the file names). This cursor can then be used in the cursor variable. The bitmaps are files searched for in the same path as in the **`pixmap-make`** function.

The convention in use for naming a cursor **`foo`** is to name the foreground bitmap as **`foo-f.xbm`** and the mask bitmap as **`foo-m.xbm`**. This convention is used in the calling of **`cursor-make`** with one argument, so that **`(cursor-make "foo")`** is equivalent to **`(cursor-make "foo-f" "foo-m")`**.

You can also define the cursor as one of the predfined ones in the server cursor font by giving its index in the font as the **`number`** argument to cursor-make. See the Appendix B of the **Xlib** manual for the list of available cursors. For instance, "star trek" cursor can be made by the call **`(cursor-make 142)`**. The symbolic names of these cursors are defined in the **`cursor-names.gwm`** file. Once this file is loaded, you can for instance use the "star trek" cursor by a **`(cursor-make XC_trek)`**.

`cut-buffer` — contents of cut buffer 0

Active value (string)

Its value is the content of the X cut buffer 0, returned as a string. When set, it takes the string argument and stores it in the same cut buffer. This can be used to communicate with clients still using this obsolete way to perform cut-and-paste operations, such as xterm.

`defun`
`defunq`
`lambda`
`lambdaq`
`de`
`df` — defines a WOOL function

```
(defun function-name (arg1 ... argn) instructions ...)
(defunq function-name (arg1 ... argn) instructions ...)
(lambda (arg1 arg2 ... argn) instructions ...)
(lambdaq (arg1 arg2 ... argn) instructions ...)
```

Defines a Lisp function and returns the atom pointing to the function. The list of arguments must be present, even if (). If defined by **`defun`**, the function will evaluate its arguments before executing, and will not evaluate them if defined by **`defunq`**. The return value of an execution of the function is the value returned by the last instruction of its body. **`de`** and **`df`** are just synonyms for **`defun`** and **`defunq`**. The **`lambda`** call, which evaluates its arguments, defines a function (without binding it to a name) while **`lambdaq`** creates a function which does not evaluates its arguments. The two following expressions are equivalent:

```
(defun foo(x) (+ x 1))
(setq foo (lambda (x) (+ x 1)))
```

When lists are evaluated, WOOL applies the result of the evaluation of the CAR of the list to its CDR, like in the **Scheme** language. Thus to apply a lambda function to its arguments, just eval the list constructed with the lambda construct and its arguments. The classic Lisp **`apply`** function could thus be defined by:

```
(defun apply (function list-of-arguments)
    (eval (+ (list function) list-of-arguments)))
```

Since functions are Lisp objects, to define a synonym of a function, you must use **`setq`**. Thus, you can change the meaning of a function easily, for instance to make **`move-window`** always raise the window, you would say in your profile:

```
(setq original-move-window move-window)
(defun move-window ()
        (raise-window)
        (original-move-window))
```

This means also that an atom can only have one value, and **`(setq move-window 1)`** will erase the definition of move-window if you didn't save it in another atom.

Example:

```
(defunq incr (value delta)
        (set value (+ (eval value) (eval delta))))

(setq x 4)
(incr x 2)
(print x)                       ==>    6
```

Functions taking a variable number of arguments can be defined by providing a parameter name instead of a list of parameters. In the body of the function during its execution, this parameter will be set to the list of the parameters given to the function.

```
(defun max l
    (with (max-val 0)
        (for obj l (if (> obj max-val)
                           (setq max-val obj)))
        max-val))
(max)                           ==>    0
(max 34 65 34 12)               ==>    65
```

Note: You are not allowed to redefine active values (such as `window`) or numeric variables (such as `foreground`), or use them as names for the parameters to the function.

```
(defun window (a b) (+ a b))     ==>     ERROR!
(defun my-window (wob)
    (window wob))                ==>     ERROR!
```

`defname` — declares a name in a namespace

`(defname name namespace [value])`

Defines the atom **`name`** to be a name in the namespace. If the value is given, for each state of the namestate, a **`(set name (eval value))`** is done, otherwise the value of name is left undefined in all the states.

Suppose you want to have a variable **`dpi`** giving the density of the screen in dots per inches for each screen:

```
(defname 'dpi screen.
    (/ (* screen-width 254)
        (* screen-widthMM 10)))
```

`delete-nth` — physically removes an element of a list **EXPERT**

`(delete-nth index list)`
`(delete-nth key list)`

This function physically removes an element of a list and returns the list. Elements are specified as for the **`#`** function. For property lists, the key and the value are deleted.

```
(setq l '(a 1 b 2 c 3 d 4))
(delete-nth 5 l)
l                                ==>     (a 1 b 2 c d 4)
(delete-nth 'b l)
l                                ==>     (a 1 c d 4)
```

`delete-read-properties` — flags to delete X properties after reading them

Numeric variable (number)

If non-zero, all calls to **`get-x-property`** deletes the X property after reading it.

`delete-window` — asks client to delete one of its windows

`(delete-window [window])`

This function will ask the client owning the argument **`window`** (or the current window if none is given) to delete it. This is only possible if the client participates in the `WM_DELETE_WINDOW` ICCC protocol, in which case the function returns **`t`**, otherwise `()`.

This is different from the **`kill-window`** call, which destroys the client and every top-level window owned by it, in that other windows should survive the call.

`dimensions`
`width`
`height` — dimensions of a WOOL object

```
(dimensions object)
(width object)
(height object)
```

These functions return the width and height in pixels of any **wob**, **pixmap**, **active label**, **cursor**, or **string**. For wobs, the returned dimensions **include** the borderwidth. For strings, these are the dimensions that such a string would take on the screen, if printed in the current `font`, thus including the `label-horizontal-margin` and `label-vertical-margin`.

The `dimensions` function returns a list of **four** values: x, y, width, heigth. x and y being non-null only for wobs in which case they are the coordinates of the upper-left corner of the wob relative to the parent wob.

`direction` — direction of menus

Numeric variable (number)

This variable controls the main direction of menus created by the `menu-make` constructor. Possible values are `horizontal` or `vertical`.

`describe-screen` — user function called to describe a screen

```
(describe-screen)
```

This function **must** be provided by the user. It must return the description of the screen (cursor, fsm, ...), made with a window-make call. Of the window-make parameters, only the `opening` (expression evaluated when all windows are decorated, just before entering the main loop of GWM) and `closing` (expression evaluated just before ending) parameters are used.

Context used:

Variable	used for
`fsm`	for the fsm of the root window
`menu`	for the default menu of the root window
`cursor`	for the shape of the cursor when in root
`property`	for the initial value of the property field
`tile`	for the pixmap to be tiled on the screen (() means do not touch the exixting screen)

This function is called by GWM for each managed screen once. In the current version, only one screen is managed. Before each invocation of this function, the screen characteristics (visual, depth, ...) are updated to reflect the characteristics of the corresponding screen.

`describe-window` — user function called to decorate a new window

```
(describe-window)
```

When a new window must be decorated by GWM, it calls this user-defined function with `window` and `wob` set to the new window. It must return a list of two window descriptors made with `window-make` describing respectively the window and the icon of the new client window.

A recommended way to use **describe-window** is to store in advance all the window descriptions in the resource manager by **resource-put** calls and use a **resource-get** call in your **describe-window** function to retrieve the appropriate description. But this is not mandatory and you are free to design the way to describe a window freely. For instance with the following describe-window function:

```
(defun describe-window ()
    (resource-get (+ window-client-class "." window-client-name)
          "any-client.any-client"))

(resource-put "any-client.any-client" any-window)
(resource-put "XTerm.xterm" xterm-window)
```

GWM will decorate xterm windows by the description held in **xterm-window**, and will decorate the windows of any other client by the **any-window** description.

display-name — name of the X server

Variable (string)

This variable holds the name of the display on which GWM is running.

double-button
double-buttonpress — makes a double-click button event

(double-button number modifier)
(double-buttonpress number modifier)

These events look for an incoming X **ButtonPress** event and matches it if the **last** key or button event was on the same button, in the same wob, and not a key event, and happened less than **double-click-delay** milliseconds before it.

The button to be pressed is specified by the number **number**, and these events verify that the modifier argument was depressed during the click. **double-button** waits for the corresponding **ButtonRelease** event before returning, while **double-buttonpress** returns immediately. Number or modifier can take the value **any**, thus matching any of the values of the corresponding argument.

Note that handling double-click events this way implies that the action that is done on a simple click event is executed even for a double click event, just before the action associated with it. The recommended way to use double-clicks is to place the **double-button** event **before** the **button** in the state of the fsm as in:

```
(on (double-button 1 any) (do-double-action))
(on (button 1 any) (do-simple-action))
```

Note: The "last" event can be a **ButtonRelease** event only if it is waited for explicitly by a GWM **buttonrelease** event, in which case the **double-click-delay** is measured between the release of the first click and the press of the second. Otherwise (if the first click is awaited by a **button** event) the delay is taken between the two "presses" of the button.

double-click-delay — maximum time between double clicks

Numeric variable

If two buttonpress events are closer in time than **double-click-delay** milliseconds then the second one can be matched by the **double-button** and **double-buttonpress** events.

`draw-line` — draws a line in a pixmap

`(draw-line pixmap x1 y1 x2 y2)`

Draws a line (via the **XDrawLine** X call) in a pixmap from point `x1`, `y1` to point `x2`, `y2` in the pixmap coordinates.

Context used:

Variable	used for
`foreground`	color of the line

`draw-rectangle` — draws a (filled) rectangle in a pixmap

`(draw-rectangle pixmap x1 y1 width height border style)`

Draws a rectangle in the pixmap. upper corner is placed at `x1`, `y1` coordinates in the pixmap, with the rectangle dimensions being `width` and `height`. `border` is the border width. Note that a value of `0` for `border` does not means "no border" but a border drawn with 0-width lines, which in X means the default line width, usually 1. In the X tradition, the upper-left corner is taken inside the border, and the dimension do not include the border.

`style` tells which kind of rectangle to draw, namely:

Value	Style of rectangle
0	nothing drawn
1	only border is drawn
2	only filled rectangle without border
3	filled rectangle + border

Context used:

Variable	used for
`foreground`	color of the border, if any
`background`	color of the inside, if any

`draw-text` — draws a string of characters in a pixmap

`(draw-text pixmap x1 y1 font text)`

Draws a string of characters `text` on font `font` into the pixmap `pixmap`. Baseline of the string will start at coordinates `x1`, `y1` in the pixmap.

Context used:

Variable	used for
`foreground`	pen color

`elapsed-time` — gets running time of GWM

`(elapsed-time)`

Returns the time in milliseconds for which GWM has been running.

`end` — terminates GWM

`(end)`

Terminates GWM, de-iconifying all the windows, un-decorating them, restoring their orignal borderwidth, and closing the display.

enter-window
leave-window — events generated when the pointer crosses the border of a wob

Constant (event)

This event is sent to the wob when the pointer crosses its border. Note that no leave event is generated if the pointer goes over a child window inside it.

enter-window-not-from-grab
leave-window-not-from-grab — pointer actually crosses the border of a wob

Constant (event)

When the server is grabbed (for instance by GWM when popping a menu or moving/resizing a window), the cursor appears to leave the current window it is in, and thus a **`leave-window`** is sent to this window. These events allow you to wait only for real crossing events, and not these grab-provoked ones. This distinguo is needed to code some tricky menu actions.

eq — tests strict equality of any two objects **EXPERT**

`(eq object1 object2)`

Returns true only if the two object are the same, i.e., if they are at the same memory location.

Example:

```
(setq l '(a b c))
(setq lc '(a b c))
(= l lc)                    ==>     t
(eq l lc)                   ==>     ()
(eq (# 1 l) (# 1 lc))       ==>     t
```

error-occurred — traps errors occurring in expressions

`(error-occurred expressions ...)`

Executes the given expressions and returns `()` if everything went fine, but returns `t` as soon as a WOOL error occurred and does not print the resulting error message.

eval — evaluates a WOOL expression

`(eval wool-expression)`

This function evaluates its argument. Note that there is a double evaluation, since the argument is evaluated by Lisp before being evaluated by eval.

Example:

```
(? (eval (+ '( + ) '(1 2))))    ==>    prints "3"
```

execute-string — executes a WOOL string

`(execute-string string)`

Parses the string argument as a WOOL program text and evaluates the read expressions. Returns `()` if an error occurred in the evaluation, `t` otherwise. Very useful in conjunction with **`cut-buffer`**, to evaluate the current selection with:

```
(execute-string cut-buffer)
```

focus-in
focus-out — events received when input focus changes on the client window

Constant (event)

These events are sent to a wob when a child receives or loses the keyboard focus, i.e., the fact that all input from the keyboard goes to that child regardless of the pointer position. Only window wobs receive these events when their client window gets the focus, they must redispatch them to their sons (bar wobs) if they are expected to respond to such events (a title bar might want to invert itself for instance) by using the **send-user-event** function.

font — default font

Numeric variable (number)

This is the font ID used by default for the constructors **label-make** and **active-label-make**. Initialized to "fixed" or your local implementor's choice. It is also the font returned by **font-make** when it fails to find a font.

font-make — loads a font

```
(font-make font-name)
```

This function loads a font in memory, from the ones available on the server. You can list the available fonts by the XLSFONTS(1) UNIX command. The function returns the descriptor (number) of the loaded font. If it fails to find it, it will issue a warning and return the default font found in the **font** global variable.

for
mapfor — iterates through a list of values

```
(for variable list-of-values inst1 inst2 ... instN)
(mapfor variable list-of-values inst1 inst2 ... instN)
```

This Lisp control structure affects successively the variable (not evaluated) to each of the elements of the evaluated list-of-values, and executes the n instructions of the body of the for. The variable is local to the for body and will be reset to its previous value on exit.

for returns the value of the evaluation on **instN** in the last iteration, whereas **mapfor** builds a list having for values the successive values of **instN** for the iterations.

Examples:

```
(for window (list-of-windows)          ; will replace all windows
    (move-window 0 0))                 ; in the upper-left corner

(for i '(a b 2 (1 2))
    (print i ","))             ==>     a,b,2,(1 2),
(mapfor i '(a b 2 (1 2))
    (list i))                  ==>     ((a) (b) (2) ((1 2)))
```

foreground — color of the foreground

Numeric variable — screen relative (color)

The value of this global variable is used by the constructors of graphics (labels and pixmaps) to paint the graphic. It is a pixel value, such as returned by **color-make**, and is initially set to the pixel of the color **Black**.

`freeze-server` — stops processing other clients during grabs

Numeric variable (number)

If set to `t` or 1, the X server is **frozen**, i.e. it doesn't process the requests for other clients during the `move-window`, `resize-window` and `grab-server` operations, so that you cannot have your pop-up menu masked by a newly mapped window, or you display mangled when moving or resizing a window.

If set to () or 0, the processing of other clients requests are allowed so that you can print in an xterm while popping a menu, for instance. (If the server is frozen, GWM tries to write on an xterm whose X output is blocked and GWM is deadlocked, forcing you to log on another system to kill it!). This should happen only when debugging, though, so that the default for this variable is `t` (Menus are normally coded so that user actions are triggered **after** de-popping).

`fsm` — Finite State Machine of the wob

Variable (fsm)

This global variable is used by all wob constructors to determine their future behavior. Holds a fsm made with `fsm-make`, or `()` (no behaviour).

`fsm-make` — compiles an automaton

```
(fsm-make state1 state2 ... stateN)
```

This function creates a finite state machine with `N` states. Each wob has an associated fsm, and a current state, which is the first state at creation time. Each time a wob receives an event, its current state of its fsm is searched for a transition which matches the event. If found, the corresponding action is executed and the current state of the fsm becomes the destination state of the transition. If no destination state was specified for the transition (see the `on` function), the current state of the fsm does not change. See `on` and `state-make` functions.

`geometry-change` — event generated when window changes size

Constant (event)

This event is sent to the window when its geometry changes by a resize operation either from the client or from GWM.

`get-wm-command` — gets the WM_COMMAND property

```
(get-wm-command)
```

Returns the command line that can be used to restart the application which created the current window in its current state as a list of strings. (For instance, `("xterm" "-bg" "Blue")`). See `save-yourself`, p 72.

`get-x-default` — gets a server default

```
(get-x-default program-name option-name)
```

Calls the "`XGetDefault`" Xlib function to query the server defaults about an option for a program. This function should be used to retrieve defaults set either by the obsolete `.Xdefaults` files or the XRDB client. It returns `()` if no such option was defined, or the option as a WOOL string.

get-x-property — gets an X property on the client

(get-x-property property-name)

Returns the value of the X11 property of name **property-name** of the current client window. It only knows how to handle the types STRING and INTEGER. For instance, **(window-name)** is equivalent to **(get-x-property "WM_NAME")**.

The value of the global variable **delete-read-properties** determines whether the read property is deleted afterwards.

getenv — gets the value of a shell variable

(getenv variable-name)

If variable-name (string) is the name of an exported shell variable, getenv returns its value as a string, otherwise returns the nil string.

grab-keyboard-also — grab-server grabs also keyboard events

Numeric variable (number)

If set, all grabs will also grab the keyboard, i.e. all keyboard events will also be redirected to the grabbing wob, whereas if unset only the pointer events get redirected.

grab-server — grabs the server

(grab-server wob ['nochild])

Grab-server freezes the server, so that only requests from GWM are processed, and every event received by GWM is sent to the wob (called the "grabbing" wob) passed as argument, or to a child of it. Re-grabbing the server afterwards just change the grabbing wob to the new one if it is not a child of the wob currently grabbing the server, in which case the function has no effect. Grabbing the server by a wob is a way to say that all events happening for wobs which are not sons of the grabbing wob are to be processed by it.

For instance, the pop-menu function uses **grab-server** with the menu as argument in order for the sons to be able to receive enter and leave window events and to catch the release of the button even outside the pop-up.

Grabbing the server sets the mouse cursor to the value of **cursor** if not ().

If the atom **nochild** is given as optional argument, if an event was sent to a child of the grabbing wob, it is redirected to the grabbing wob as well (default not to redirect it).

grid-color — color to draw (xor) the grids with

Numeric variable — screen relative (color)

This is the color (as returned by color-make) that will be used to draw the grid with (in XOR mode) on the display. Initially set to the "Black" color.

grabs — passive grabs on a window

Variable (list)

The events specified in this list are used by the **`window-make`** constructor to specify on which events GWM establishes passive grabs on its window to which the client window gets reparented. Passive grabbing means redirecting some events from a window (bars, plugs, client) to the main window, "stealing" them from the application.

`gwm-quiet` — silent startup

Numeric variable (0/1)

This variable evals to 1 if the users specified (by the `-q` option) a quiet startup. Your code should check for this variable before printing information messages.

`hack` — raw access to GWM internal structures **EXPERT**

`(hack type memory-location)`

WARNING: Internal debug function: returns the WOOL object constructed from C data supposed present at memory-location (number). The type of the type argument is used to know what to find:

type of type	interpretation of pointer	returns
number	pointer on integer	number
string	C string	string
()	object	object
atom	pointer to an WOOL object	number

This function is **not** intended for the normal user, it should be used only as a temporary way to implement missing functionalities or for debugging purposes.

`hashinfo` — statistics on atom storage

`(hashinfo)`

Prints statistics on atom storage.

`horizontal`
`vertical` — directions of menus

Constant (number)

Used to specify orientation of menus in menu-make.

`hostname` — name of the machine on which GWM is running

Active value (string — not settable)

This string holds the name of the host GWM is currently running on.

`iconify-window` — iconifies or de-iconifies a window

`(iconify-window)`

Iconify the current window (or de-iconify it if it is already an icon). The current window is then set to the new window.

WARNING: Never trigger this function with a **`button`** event, but with a **`buttonpress`** or **`buttonrelease`** event. If you trigger it with a button event, the window being unmapped cannot receive the corresponding release event and GWM acts weird!

WARNING: Due to the grab management of X, an unmapped window cannot receive events, so a grab on this window is automatically lost.

`if` — conditional test

`(if condition then [condition then] ... [condition then] [else])`

This is similar to `cond` but with a level of parentheses suppressed. It executes the "then" part of the first true condition, or the else condition (if present) if no previous condition is true.

```
(defun fib (n)
    (if (= n 0)
          1
        (= n 1)
          1
          (+ (fib (- n 1) (- n 2)))))
```

`inner-borderwidth` — borderwidth of the client window

Numeric variable (number)

When GWM decorates a new window, it sets the width of the border of the client window to the value of the `inner-borderwidth` variable at the time of the evaluation of the `window-make` call.

If `inner-borderwidth` has the value `any`, which is the default value, the client window borderwidth is not changed.

`invert-color` — color to invert (xor) the wobs with

Numeric variable — screen relative (color)

This is the color (as returned by color-make) that will be used by `wob-invert` (see 4, p 90) to quickly XOR the wob surface. Initially set to the bitwise-xoring of Black and White colors for each screen.

`invert-cursors` — inverts the bitmaps used for making cursors

Numeric variable (number)

Due to many problems on different servers, you might try to set to 1 this numerical global variable if your loaded cursors appear to be video inverted. Defaults to 0 (no inversion).

`itoa` — integer to ascii string conversion

`(itoa number)`

Integer to ASCII yields the base 10 representation of a number in a WOOL string.

```
(itoa 10)                    ==>     "10"
```

`key`
`keypress`
`keyrelease` — keyboard events

`(key keysym modifier)`
`(keypress keysym modifier)`
`(keyrelease keysym modifier)`

Returns an event matching the press on a key of keysym code **`keysym`**[3], with the modifier[4] key **`modifier`** depressed. The **`key`** event will wait for the corresponding keyrelease event before returning, while the **`keypress`** event will return immediately. Number or modifier can take the value **`any`**, thus matching any of the values of the corresponding argument.

Keysyms can be given as numbers or as symbolic names.

Examples:

```
(key 0xff08 with-alt)   ; matches typing Alternate-Backspace
(key "BackSpace" with-alt)       ; same effect
```

Note: These functions convert the keysym to the appropriate keycode for the keyboard, so you should do any re-mapping of keys via the **`set-key-binding`** (see 4, p 75) function **before** using any of them.

`key-make` — makes a key symbol out of a descriptive name

`(key-make keyname)`

Returns a **keysym** (number) associated with the symbolic name **`keyname`** (string), to be used with the **`key`**, **`keypress`**, **`keyrelease`** and **`send-key-to-window`** functions. The list of symbolic names for keys can be found in the include file "**`/usr/include/X11/keysymdef.h`**".

For instance, the backspace key is listed in **`keysymdef.h`** as:

```
#define XK_BackSpace            0xFF08  /* back space, back char */
```

and you can specify the keysym for backspace with:

```
(key-make "BackSpace")          ==> 65288 (0xFF08)
```

`keycode-to-keysym` — converts a key code to its symbolic code

`(keycode-to-keysym code modifier)`

This function returns the server-independent numeric code (the KeySym) used to describe the key whose keyboard code is **`code`** while the modifier **`modifier`** is down.

Note: Only know the **`alone`** and **`with-shift`** modifiers are meaningful in this **X** function.

`keysym-to-keycode` — converts a symbolic code to a key code

`(keysym-to-keycode keysym)`

[3] A keysym is a number representing the symbolic meaning of a key. This can be a letter such as "A", or function keys, or "Backspace", ... The list of keysyms can be found in the **keysymdef.h** file in the **`/usr/include/X11`** directory.

[4] See the list of modifiers at the **`with-alt`** entry, page 89

This function returns the key code, i.e., the raw code sent by the displays keyboard when the user depresses the key whose server-independent numeric code is listed in the "`/usr/include/X11/keysymdef.h`" include file and given as a number as the **`keysym`** argument.

`kill-window` — destroys a client

`(kill-window [window])`

Calls XKillClient on the current window or **`window`** if specified. This is really a forceful way to kill a client, since all X resources opened by the client owning the window are freed by the X server. If only you want to remove a window of a client, use **`delete-window`**, p 45.

`label-horizontal-margin`
`label-vertical-margin` — margins around labels

Numeric variables (number)

The value of these global variables are used by the **`label-make`** functions to add margins (given in pixels) around the string displayed. **`label-horizontal-margin`** defaults to 4 and **`label-vertical-margin`** to 2.

`label-make` — makes a pixmap by drawing a string

`(label-make label [font])`

Creates a label with string **`label`** drawn with **`foreground`** color in the font **`font`** on a **`background`** background. This function builds a pixmap which is returned.

Note that when used in a plug, the resulting pixmap is directly painted on the background pixmap of the wob, speeding up the redisplay since re-exposure of the wob will be done directly by the server, but consuming a small chunk of server memory to store the pixmap.

Context used:

Variable	used for
`foreground`	color of the text string
`background`	color of the background
`font`	font of the string if not given
`label-horizontal-margin` `label-vertical-margin`	margins around the string

`last-key` — last key pressed

`(last-key)`

If the last event processed by a fsm was a key event, returns the string generated by it. (Pressing on A yields the string `"A"`). It uses **XLookupString** to do the translation.

`length` — length of list or string

```
(length list)
(length string)
```

Returns the number of elements of the list or the number of characters of the string.

`list` — makes a list

`(list elt1 elt2 ... eltN)`

Returns the list of its N evaluated arguments.

```
(list (+ 1 2) (+ "foo" "bar"))  ==>     (3 "foobar")
```

`list-make` — makes a list of a given size

`(list-make size elt1 elt2 ... eltN)`

Returns the list of size `size` (number) elements. The element of rank `i` is initialized to `elt`j, where $j = i$ modulo N, if elements are provided, `()` otherwise.

```
(list-make 8 'a 'b 'c)          ==>     (a b c a b c a b)
(list-make 3)                   ==>     (() () ())
```

`list-of-screens` — list of managed screens

`(list-of-screens)`

Returns the list of screens actually managed by GWM.

`list-of-windows` — returns the list of managed windows

`(list-of-windows ['window] ['icon] ['mapped]`
`    ['stacking-order])`

Returns the list of all windows and icons managed by GWM, mapped or not. Called without arguments, this function returns the list of all windows (not icons), mapped or not. It can take the following atoms as optional arguments:

window lists only main windows

icon lists only realized icons

mapped lists only currently mapped (visible) windows or icons

stacking-order lists windows in stacking order, from bottommost (first) to topmost (last). This option is slower than the default which is to list them by the order in which they were managed by GWM, from the the newest to the oldest.

The `window` and `icon` arguments are mutually exclusive.

Note: The two following expressions do not return the same list:

```
(list-of-windows 'icon)
(mapfor w (list-of-windows) window-icon)
```

The first one will return the list of all realized icons, that is only the icons of windows that have already been iconified at least once, but the second one will trigger the realization of the icons of **all** the managed windows, by the evaluation of `window-icon`, on all the windows.

Note: The main window of an application is always realized, even if the application started as iconic.

`load` — loads and executes a WOOL file

    (load filename)

Loads and executes the WOOL file given in the filename string argument, searching through the path specified by the **GWMPATH** variable or the **-p** command line switch. Defaults to `.:$HOME:$HOME/gwm:INSTDIR`, where `INSTDIR` is your local GWM library directory, which is normally `/usr/local/X11/gwm`, but can be changed by your local installer.

On startup, GWM does a `(load ".gwmrc")`.

Returns the complete pathname of the file as a string if it was found, () otherwise but does not generate an error, only a warning message.

Searches first for `filename.gwm` then `filename` in each directory in the path. If the filename includes a `/` character, the file is not searched through the path. If any error occurs while the file is being read, WOOL displays the error message and aborts the reading of this file. This implies that you can't expect GWM to run normally if there has been an error in the .gwmrc. (You can turn off this behavior and make GWM continue reading a file after an error at your own risk, with the **-D** command line option).

`lower-window` — lowers window below another window

    (lower-window [window])

Lowers the current window below all other top-level windows or, if the `window` argument is present, just below the window given as argument.

`make-string-usable-for-resource-key` — strips string from dots and stars

    (make-string-usable-for-resource-key string)

Replaces all characters `.*&␣` (dots, stars, ampersand and spaces) in the string `string` by underscores `'_'`. This function should be used before using names for keys in the resource manager functions, as X can go weird if you do not handle the good number of classes and names to the resource manager (see `resource-get`, p 71).

The string argument is not modified; if remplacement is done, a new string is returned.

`map-notify` — event sent when window is mapped

Constant (event)

This event is sent to a window or icon just afterwards being actually mapped.

`map-window` — maps window

    (map-window [window])

Maps (makes visible) the window (or current window). Does not raise it.

`match` — general regular expression matching package

    (match regular-expression string [number] ...)

This is the general function to match and extract substrings out of a WOOL string. This string is matched against the pattern in regular expression, and returns () if the pattern could not be found in the string, and the string otherwise.

If `number` is given, match returns the sub-string matching the part of the regular-expression enclosed in between the number-th open parenthesis and the matching closing parenthesis (first parenthesis is numbered 1). Do not forget to escape twice the parentheses, once for GWM parsing of strings, and once for the regular expression, e.g., to extract "bar" from "foo:bar" you must use: `(match ":\\(.*\\)" "foo:bar" 1)`. If a match cannot be found, returns the nil string `""`.

If more than one `number` argument is given, returns a list made of of all the specified sub-strings in the same way as for a single `number` argument. If a sub-string wasn't matched, it is the null string. For instance, to parse a X geometry such as `80x24+100+150` into a list `(x y width height)`, you can define the following function:

```
(defun parse-x-geometry (string)
  (mapfor dim
    (match "=*\\([0-9]*\\)x\\([0-9]*\\)\\([-+][0-9]*\\)\\([-+][0-9]*\\)"
           string 3 4 1 2)
    (atoi dim))

(parse-x-geometry "80x24+100+150")  ==>  (100 150 80 24)
```

Note that this function returns an error if there is a syntax error in the given regular expression.

The accepted regular-expressions are the same as for ED or GREP, viz:

1. Any character except a special character matches itself. Special characters are the regular expression delimiter plus `\ [ .` and sometimes `^ * $`.
2. A `.` matches any character.
3. A \ followed by any character except a digit or `( )` matches that character.
4. A nonempty string `s` bracketed `[s]` (or `[^s]`) matches any character in (or not in) `s`. In `s`, \ has no special meaning, and `]` may only appear as the first letter. A substring `a-b`, with `a` and `b` in ascending ASCII order, stands for the inclusive range of ASCII characters.
5. A regular expression of form **1-4** followed by `*` matches a sequence of 0 or more matches of the regular expression.
6. A regular expression, `x`, of form **1-8**, bracketed `\(x\)` matches what `x` matches.
7. A \ followed by a digit `n` matches a copy of the string that the bracketed regular expression beginning with the `n`th `\(` matched.
8. A regular expression of form **1-8**, `x`, followed by a regular expression of form `1-7`, `y` matches a match for `x` followed by a match for `y`, with the `x` match being as long as possible while still permitting a `y` match.
9. A regular expression of form **1-8** preceded by `^` (or followed by `$`), is constrained to matches that begin at the left (or end at the right) end of a line.
10. A regular expression of form **1-9** picks out the longest among the leftmost matches in a line.
11. An empty regular expression stands for a copy of the last regular expression encountered.

`member` — position of element in list or in string

```
(member element list)
(member substring string)
```

In the first form, scans the list (with the **`equal`** predicate) to find the object **`element`**. If found, returns its index in the list (starting at 0), **`()`** otherwise.

In the second form, looks for the first occurrence of the string **`substring`** in the string **`string`**, and returns the character position at which it starts, or **`()`** if not found.

`meminfo` — prints memory used

```
(meminfo)
```

Prints the state of the malloc allocator of all dynamic memory used, by WOOL or by the Xlib. You will note that reloading your profile consumes memory. This is unavoidable.

`menu` — menu associated with wob

Variable (menu)

The value of this global variable is used by the constructors of all wobs to set the associated menu of the wob. It should be set to a menu constructed with **`menu-make`** or (). This menu is the default one used by the **`pop-menu`** and **`unpop-menu`** functions.

`menu-make` — makes a menu

```
(menu-make bar1 bar2 ... barN)
```

Creates a menu, vertical or horizontal depending on the value of the context variable **`direction`** (**`horizontal`** or **`vertical`**). This function returns a menu descriptor and at the same time realize the corresponding menu wob (to get the wob, use the **`menu-wob`** function below). Unlike the plug-make or bar-make functions that return the WOOL description to be used to create different wob instances, this is the only WOOL function which really creates a wob (as an unmapped X window) on execution.

A vertical menu is composed of horizontal bars stacked on top of each other. A horizontal menu is composed of vertical bars aligned from left to right. The menu takes the width (or height) of the largest bar, then adjusts the others accordingly in the limits defined by the values of **`menu-max-width`** and **`menu-min-width`**.

If a bar argument is **`()`**, it is just skipped.

Context used:

Variable	used for
`fsm`	its finite state machine
`direction`	for the direction of the menu
`borderwidth`	the width of the menu's border in pixels
`borderpixel`	its color
`bordertile`	if it has a pixmap as border pattern
`background`	the color of the background
`bar-separator`	the number of pixels between consecutive bars
`tile`	the pattern of its background (may be ())
`cursor`	the cursor's shape when in it
`property`	the property associated with the menu
`menu-min-width`	minimum width of the menu, its bars are stretched to fit in it
`menu-max-width`	maximum width of the menu, clipping its bars

`menu-wob` — returns wob associated with menu

`(menu-wob menu-descriptor)`

Returns the wob representing the menu as given by the descriptor returned by **`menu-make`** and used in the context variable **`menu`**.

`meter` — sets meter attributes

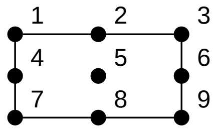
`(meter [key value]...)`
`(meter list)`

Sets the attributes of the current screen meter. The key must be an atom, and the corresponding attribute is set to the value. If a **`list`** is given, it must be a list of keys and values.

Key	Value
`font`	the font of the text displayed in the meter
`background`	the background color of the meter, defaults to black
`foreground`	the color the text will be drawn with, defaults to white
`horizontal-margin`	in pixels between text and sides of meter
`vertical-margin`	in pixels between text and top and bottom of meter
`x`	x position of "anchor"
`y`	y position of "anchor"
`gravity`	which corner of the meter is put at the "anchor". Gravity is a number from 1 to 9, default being 1 (North-West)
`borderwidth`	the width in pixels of the border, defaults to 0 (no border)
`borderpixel`	the color of the border, defaults to white

`meter` returns a list of keys and values describing the previous values of attributes that has been changed. This list can then be given as argument to meter to restore the previous values.

Gravity is expressed as in X11 calls, i.e. by the numbers:



For instance, to set temporarily the meter on the lower right corner of the screen:

```
(setq previous-meter-config
            (meter 'x screen-width 'y screen-height
                  'gravity 9))
                                       ; ==> (x 0 y 0 gravity 1)
       (my-code...)
       (meter previous-meter-config)
```

meter-close — unmaps the meter

`(meter-close)`

Closes the meter (makes it disappear).

meter-open — displays the meter

`(meter-open x y string)`

Displays the meter at location x,y. The string argument is only used to set the minimum width of the meter, call meter-update to display a string in it.

meter-update — writes a string in the meter

`(meter-update string)`

Change the string displayed in the meter to `string`. Updates the width of the meter accordingly. Since the meter is generally used in a context where speed is important, it is never shrunk, only expanded.

move-grid-style
resize-grid-style — style of grid for move and resize

Numeric variables (number)

These variables control the kind of grid which is displayed on move or resize operations. Currently available grids are:

Value	Style of grid
0	outer rectangle of the window (default)
1	outer rectangle divided in 9 (uwm-style)
2	outer rectangle with center cross (X) inside
3	outer rectangle + inner (client) outline
4	styles 1 and 3 combined
5	2 pixel wide outer rectangle

move-meter
resize-meter — shows meter for move and resize

Numeric variables (number)

If set to 1, the meter is shown during an interactive move, displaying the coordinates of the moved window.

`move-window` — moves window interactively or not

```
(move-window)
(move-window window)
(move-window x y)
(move-window window x y)
```

Moves the window. If coordinates are specified, moves directly the current window (or **`window`** argument if specified) to **`x,y`** (upper-left corner coordinates in pixels). If coordinates are not specified, moves the window interactively, i.e., display the grid specified with the current value of **`move-grid-style`**, and track the grid, waiting for a buttonrelease for confirmation or a buttonpress for abort. If confirmed, the window is moved to the latest grid position. The cursor will take the shape defined by the variable **`cursor`** if not (). When move-window is called on another event than buttonpress, there is no way to abort the move.

If you have a strange behavior on move or resize window, check if you didn't trigger them with a **`button`** event instead of a **`buttonpress`**, since GWM will wait for a release already eaten by move-window!

When used interactively (first two forms), the return value can be one of:

Return	Case
()	Ok
0	X problem (window disappeared suddenly, other client grabbing display, etc.)
1	pointer was not in the window screen
2	user aborted by pressing another button
3	user released button before the function started

Also, when used interactively, if the context variable **`confine-windows`** is set, the window will stay confined to the screen.

`name-change` — event generated when window changes its name

Constant (event)

This event is sent to the window when the client changes its name. The window should then warn the appropriate wobs, by a **`send-user-event`** if needed. It is equivalent to the event made by the call **`(property-change "WM_NAME")`**.

`namespace` — sets current state of a namespace

```
(namespace namespace current-state-index)
```

Sets the current state of the namespace argument to the index returned from the **`namespace-add`** function at the creation of the state. If the index is out of bounds (like "-1"), returns the current index.

`namespace-add` — adds a state to a namespace

```
(namespace-add namespace)
```

Adds a new state to the argument namespace and returns the index (numeric offset starting at 0) of the newly created state. This index should then be used to set the current state of the namespace by the **`namespace`** function.

`namespace-make` — creates a namespace

`(namespace-make)`

Creates a new **namespace** and returns it. A namespace is a set of variable names, called **names**, that have different values for each state in which the namespace can be.

The special namespace `screen.` has as many spaces as there are screens, and its current state is always updated to match the current screen by GWM. Other namespaces must switch states by the `namespace` function.

For instance, to define a namespace `application.` with a name `application.border` which will depend on the current application:

```
(setq application. (namespace-make))
(setq application.clock (namespace-add application.))
(setq application.load (namespace-add application.))
(defname 'application.border application.)
(namespace application. application.clock)
(setq application.border 1)
(namespace application. application.load)
(setq application.border 2)

(namespace application. application.clock)
application.border                              ==>  1
(namespace application. application.load)
application.border                              ==>  2
```

`namespace-of` — returns namespace of the name

`(namespace-of name)`

Returns the namespace where the name `name` is declared in, otherwise (if it is a plain atom or an active value) returns `()`.

`namespace-remove` — removes a state from a namespace

`(namespace-remove namespace state-index)`

Destroys a state (made with `namespace-add` of a namespace. The number `state-index` is the index returned by namespace-add at the creation of the state.

`namespace-size` — number of states in the namespace

`(namespace-size namespace)`

Returns the number of the states of the namespace, i.e., the number of possible values for each name in this namespace. The legal values for the index to be given to the `namespace` function are in the range from 0 to the return value - 1.

`not` — logical negation

`(not object)`

Logical negation. Returns `t` if `object` is `()`, `()` otherwise.

`oblist` — prints all defined objects

`(oblist)`

Prints the name and value of all currently defined WOOL atoms.

`on`
`on-eval` — triggers a transition on an event in a state of a fsm

`(on event [action [state]])`
`(on-eval event [action [state]])`

This is the function used to build transitions in the states of the fsms. The **`on`** evaluates only its **`event`** argument, so that it is not necessary to quote the **`action`** or **`state`** arguments, whereas **`on-eval`** evaluates all its arguments.

When an event is handled by an fsm, it checks sequentially all the transitions of the current state for one whose **`event`** field matches the incoming event. If found, it calls the WOOL function found in the **`action`** field, which must be a function call, and makes the state argument the current state of the fsm. If no state is given, it is taken as the same state of the fsm. If no action is given, no action is performed.

The destination state is an atom which should have a state value defined in the current fsm (i.e., set by setq of an atom with the result of a **`state-make`**) at the time when the fsm is evaluated, which means that you can make forward references to states, as in the following example.

Example: To define a behavior of a wob cycling through 3 states on the click of any button, use this fsm:

```
(fsm-make
        (setq state1 (state-make
                (on (button any any) (? 1) state2))
        (setq state2 (state-make
                (on (button any any) (? 2) state3)))
        (setq state3 (state-make
                (on (button any any) (? "3.\n") state1))))
```

`opening`
`closing` — WOOL hooks on creation and deletion of windows

Variables (Wool)

The values of these global variables are used by the **`window-make`** constructor to define WOOL expressions which are evaluated just before creating or just after destroying a window or an icon.

Note: The **`opening`** field of a window is always evaluated on the creation of this window, thus the **`opening`** of an icon is only executed on the first iconification of the icon.

`or` — logical OR of expressions

`(or object1 object2 ... objectN)`

Logical OR. Returns the first non-() object or ().

`pixmap-load` — loads an XPM X pixmap from an ascii file

`(pixmap-load filename [symbolic-color-name color-value]...)`

This function builds a pixmap by reading its description in the file **filename.xpm** or **filename**, searched in the directory path held by the **GWMPATH** shell variable (see **GWMPATH**, p 4). In case of error the default pixmap returned is the same as the one returned by the **pixmap-make** function.

The pixmap is expected to be described in the **XPM** (for X PixMap) format. XPM is a de-facto standard X pixmap format which can be found on export or avahi, and whose full distribution (including docs) is included in the **xpm** subdirectory of the GWM distribution

XPM allows you to associate symbolic color names to pixels in the file, which may be overrided by actual color to be used at load time. For instance, to use an icon **undo** defining the symbolic colors **lit dark pen** in a "blue" context, you may want to load it by a

```
(pixmap-load 'undo 'lit (color-make "LightBlue")
                   'dark (color-make "SteelBlue")
                   'pen black)
```

Whereas if you want to give it a pinkish look:

```
(pixmap-load 'undo 'lit (color-make "pink1")
                   'dark (color-make "pink4")
                   'pen (color-make "DeepPink4"))
```

Note: In the current implementation, the 3.0a version of the XPM library is used, which is able to read xpm v2 (but only in C syntax) and xpm v3 formats. Converters to/from other formats are to be found in directory **xpm/converters**.

pixmap-make — builds a pixmap (color image)

```
(pixmap-make filename)
(pixmap-make width height)
(pixmap-make background-color file1 color1
             file2 color2 ... )
```

With a single argument, loads the bitmap file given in the filename string argument, searching through the bitmap path specified by the **GWMPATH** variable or the **-p** command line switch (see **GWMPATH**, p 4). Searches first for **filename.xbm** then **filename** in each directory in the path. If the filename includes a **/** character, the file is not searched through the path. The pixmap is built by drawing the "unset" bits of the bitmap with the color held by the **background** variable and the "set" bits with the color held by the **foreground** variable.

When used with two arguments, **pixmap-make** returns a newly created pixmap of width **width** and height **height** filled with the current **foreground** color.

When used with three or more arguments, returns the pixmap constructed by using the bitmaps given as arguments to paint only the "set" bits of bitmap in file file*i* (searched along the same rules as for the first form) to the color*i*. The "unset" bits remain untouched. The pixmap is first painted with the color given in the **background-color** argument.

If the specified bitmap cannot be loaded, either because the file cannot be found or does not contain a bitmap, a default built-in bitmap is used and a warning is issued. The default is the MIT X Consortium logo as a 20x15 bitmap. Bitmaps can be of any size, they will be centered in the resulting pixmap which will be of the maximum size of its components.

The filename arguments can also be replaced by:

active-labels The string is centered on the bitmap, drawn in the following color.

pixmaps The pixmap is painted in the center, with its own background color (note that `label-make` returns a pixmap!). The following color is then ignored, but **must** be specified for consistency purposes.

`place-menu` — maps a menu as a normal client window

`(place-menu name menu [x y])`

Places a menu (made with `menu-make`) on the screen. This window is then managed like any other client window. Its client class and, client name are given by the context variables `class-name` and `client-name`, by default `Gwm` and `menu`, and its window name is given by the parameter `name`. The window is placed at position `x, y` if specified and at 0,0 otherwise (NB: It is placed at last popped position if it was a popped menu). Returns the created wob.

WARNING: Be careful not to mix popup menus and placed ones, calling `pop-menu` on an already "placed-menu" will result in unpredictable behavior. Context used:

Variable	used for
`class-name`	the client class of the window. Default `Gwm`.
`client-name`	the client name of the window. Default `menu`.
`icon-name`	the name of the icon for the window. If () (default), uses `name`.
`starts-iconic`	if the menu will first appear as an icon

`plug-make` — makes a plug

`(plug-make pixmap)`
`(plug-make active-label)`

This function builds a plug, the atomic wob object. It is composed of a graphic object (either a pixmap, created with pixmap-make or label-make, or another type of object such as an "active-label"). The size of the graphic object determines the size of the plug. Thus if you change the pixmap of the plug via `wob-tile`, the plug is resized accordingly.

Context used:

Variable	used for
`fsm`	its finite state machine
`borderwidth`	the width of the plug's border in pixels
`borderpixel`	its color
`bordertile`	the pixmap to paint the border with
`background`	the color of the background
`menu`	the pop-up associated with it
`cursor`	the cursor's shape when in it
`property`	the property associated with the plug

`plug-separator` — inter-plug space in bars

Numeric variable (number)

The value of this global variable is used by the `bar-make` function to give the space in pixels between 2 consecutive plugs not separated by extensible space.

`pop-menu` — pops a menu

`(pop-menu [menu] [position] ['here])`

This function grabs the server (using the **grab-server** function), thus preventing other client requests to be processed by the server, and maps the given menu (or the menu associated with the current wob if menu is not given or is set to ()). The menu is guaranteed to be on the screen when this function returns. The behavior of the menu is then determined by the menu's fsm. The cursor will take the shape defined by the variable **cursor** if it is non-nil.

The **position** number is the item in which you want the cursor to appear (centered), the first item is numbered 0.

If the atom **fixed** is given, the menu is not moved under the current position of the pointer, it stays at the last position it was. So to pop a menu **menu** at (67, 48), do:

```
(move-window (menu-wob menu) 67 48)
(pop-menu menu 'here)
```

The current wob at the time of the call to **pop-menu** becomes the parent of the menu.

Note: A menu is not a GWM window, so when a menu is popped, the current window becomes the window parent of the wob which has popped the menu.

Note: The arguments to **pop-menu** can be given in any order and are all optional.

`print-errors-flag` — controls printing of error messages

Numeric variable (number)

If nonzero, WOOL error messages are printed on GWM output device, which is the default. **error-occurred** sets this variable to 0 to prevent printing errors.

`print-level` — controls printing depth of lists

Numeric variable (number)

This variable controls the maximum depth at which the lists will be printed.

```
(setq print-level 2)
'(1 (2 (3 (4 rest))))          ==>      (1 (2 (...)))
```

`process-events` — recursively process all pending events **EXPERT**

`(process-events [sync])`

This function reads the event queue and recursivly processes all pending events by re-entering the main GWM loop. It returns when there are no more pending events on the X event queue.

This functions allows a pseudo-multitasking capability for GWM. You can thus implement "background jobs" such as a general desktop space cleaning routine, to be called after each move, opening, or closing function that pauses by calling **process-events** at regular intervals to let GWM do its other tasks. You should then take into account the fact that the current function could be recursively called by another move operation triggered by an event processed by **process-events**, however and thus be re-entrant.

If a non-nil argument **sync** is given, GWM does a "Sync", i.e. requests the server to send all its pending events, before processing the events on the queue.

`process-exposes` — treats all pending expose events

```
(process-exposes)
```

This function reads the event queue and processes all expose events it finds for GWM objects on the screen. It is used by pop-menu to be sure that the menu is completely drawn before letting it react to user events.

`rotate-cut-buffers` — rotate server cut buffers

```
(rotate-cut-buffers number)
```

Exchange the contents of the **8** cut buffers on the X server so that buffer n becomes buffer $n+$ **number** modulo 8.

`send-current-event` — re-sends X event to the client of a window

```
(send-current-event window)
```

Re-sends the last key or button event to the client of the **window** argument. If **window** is **()**, it is taken as the current window.

{}
`progn` — sequence of instructions

```
(progn inst1 inst2 ... instN)
{inst1 inst2 ... instN}
```

The classical Lisp **progn** function, evaluating all its arguments and returning the result of the last evaluation. Useful in places where more then one Lisp instruction is expected, for instance in **then** fields of the **if** instruction, or in the **action** field of the **on** function. The brace notation is just a shortcut for writing **progn** sequences.

`property-change` — event generated when a client window changes a property

```
(property-change property-name)
```

This event is sent when the property of name **property-name** is changed on the client window.

`raise-window` — raises window on top of other windows

```
(raise-window [window])
```

Raises the current window on top of all other top-level windows or, if the **window** argument if present, above the window given as argument.

`re-decorate-window` — re-decorates the client window by GWM

```
(re-decorate-window [window])
```

Un-decorate the client window and re-decorate it as if it had appeared on the screen. Useful after a **(load ".gwmrc")** to test quickly any modifications to your profile.

`reenter-on-opening` — process events on the queue just before mapping a new window

Numeric variable (number)

If nonzero (default), GWM will process all events in the queue before mapping a new window. This may pose re-entrancy problems (WOOL code may be called during a call to `place-menu` for instance), and thus can be turned off.

`refresh` — refreshes the screen

```
(refresh [window])
```

If a window argument is provided, refreshes only this window, otherwise forces a redraw of the screen like XREFRESH(1). Refresing is done by mapping and unmapping a new window over the aera to refresh.

`replayable-event` — makes a replayable event from a normal event

```
(replayable-event event)
```

Makes the given button or key event replayable (see `ungrab-server-and-replay-event`, p 79). This is not the default for events because replayable events freeze the server state for GWM when a passive grab is activated with them, which might not be desirable.

```
(set-grabs (replayable-event (button 1 alone)))
```

WARNING: It is recommended to un-freeze the server if you are to do some processing before the button or key is released, either by replaying the event by `ungrab-server-and-replay-event` or `allow-event-processing` (see p 35).

`resize-style` — style of interactive resize

Numeric variable (number)

This variable controls the way the interaction with the user is handled during resizes, and can take the numeric values:

Value	Style of Resize
0	is a `Uwm`-like resize, i.e. the window is divided in nine regions, and you are allowed to drag which side or corner of the window you were in when the resize began.
1	is a **Mwm**-like resize, i.e. ou resize the window by its side or corner, the width of the corners being determined (in pixels) by the value of the global variable `mwm-resize-style-corner-size`. If you are dragging the side of the window and you go at less than `mwm-resize-style-corner-size` of a corner, you then drag this corner, if the global variable `mwm-resize-style-catch-corners` is nonzero. The corner or side dragged is reflected by the shape of the cursor you have put in the global variables cursor-NW, cursor-NE, cursor-N, cursor-SW, cursor-SE, cursor-S, cursor-E, cursor-W (see p 43). While you are not dragging anything, the cursor is still determined by the value of `cursor`

`resize-window` — resizes the window interactively or not

```
(resize-window)
(resize-window window)
(resize-window width height)
(resize-window window width height)
```

Resizes the window. If the dimensions are specified, resizes directly the current window (or the given **`window`** argument) to the given size, specified in pixels, rounded down to the allowed size increments (if you want to resize by increments, use the **`window-size`** active value). If the dimensions are not specified, resizes the window interactively, i.e., display the grid specified with the current value of **`resize-grid-style`**, and track the grid, waiting for a buttonrelease for confirmation or a buttonpress for abort. If confirmed, the window is resized to the latest grid position. The cursor will take the shape defined by the variable **`cursor`**, if it is non-nil.

NOTE: The dimensions given to resize-window are those of the outer window, in pixels, including the GWM decoration. Use the **`window-size`** active value if you want to specify the client dimensions.

WARNING: When resize-window is called on an event other than buttonpress, there is no way to abort the resize.

The interactive resize interaction is set by the value of the global variable **`resize-style`**

When used interactively (first two forms), the return value can be one of:

Returns	Case
()	Ok
0	X problem (window disappeared suddenly, other client grabbing display, etc.)
1	pointer was not in the window screen
2	user aborted by pressing another button
3	user released button before the function started

Also, when used interactively, if the context variable **`confine-windows`** is set, the window will stay confined to the screen.

`resource-get` — searches GWM database for a resource

```
(resource-get name class)
```

Calls the X11 resource manager for an object of name **`name`** and class **`class`** (strings of dot-separated names) previously stored by a call to **`resource-put`** matching the description (See the X11 documentation for precise description of the rule-matching algorithm of the resource manager).

This is the recommended way to retrieve the window descriptions associated with a client in the **`describe-window`** function.

WARNING: The **`name`** and **`class`** **must** have the same number of dots in them! (see **`make-string-usable-for-resource-key`**, p 58).

`resource-put` — puts a resource in GWM database

```
(resource-put name value)
```

Puts the value in the GWM resource database so that it can be retrieved by a future call to resource-get. Names can use the same conventions as in the .Xdefaults file for normal X11 clients.

This is the recommended way to store the window descriptions associated with a client so that they can be retrieved later in the `describe-window` function.

`restart` — restarts GWM

`(restart)`
`(restart "prog" "arg1" "arg2" ... "argn")`

Without arguments, restarts GWM, more precisely, it does an EXEC(2) of GWM with the same arguments that were given on startup.

With arguments, terminates GWM and starts a new process with the given command-line arguments. This useful to restart GWM with another profile, as in:

```
(restart "gwm" "-f" "another-profile")
```

`root-window` — the root window

Active value (window id)

Holds the wob describing the root window of the current screen. For instance, to set the background pixmap of the screen to the bitmap in file "ingrid", say:

```
(with (wob root-window) (setq wob-tile (pixmap-make "ingrid")))
```

When set to a root window wob, sets the current wob and window to it, and screen to the screen it belongs to.

`save-yourself` — asks client to update its WM_COMMAND property

`(save-yourself [window])`

Sends to the current window's (or to given window's) client window the ICCC message `WM_SAVE_YOURSELF`, telling the application to update its `WM_COMMAND` X property to a command line which should restart it in its current state.

This function returns `t` if the window supported this protocol, and hence is supposed to update its `WM_COMMAND` property; `()` otherwise (see 4, p 51).

`screen` — current screen

Active value (screen number)

Returns or sets the current screen as the screen number (the same number as `X` in `DISPLAY=unix:0.X`). Setting the screen also sets the active values `wob` and `window` to the root window of the screen. See the warning about using `screen` in `with` constructs under the `window` entry, p 81.

`screen-count` — number of screens attached to the display

Constant (number)

The number of screens attached to the display. Not equal to `(length (list-of-screens))` if you excluded screens by the `-x` command line option.

screen-depth
screen-height
screen-width — screen dimensions

Constants (number)

These numerical variables hold the size of the current screen in bitplanes (`screen-depth`) and pixels (`screen-width` and `screen-height`). A screen-depth of 1 means that you are on a monochrome screen.

screen-heightMM
screen-widthMM — actual screen size in millimeters

Constants (number)

Hold the dimensions of the screen in millimeters.

screen-type — visual type of screen

Active value (atom — not settable)

Returns the screen visual type as an atom, which can be either `color`, `gray`, or `mono`, if the screen is a color, gray scale, or monochrome device.

send-button-to-window — sends button event to a client

```
(send-button-to-window button modifier x y)
```

Sends a buttonpress and a buttonrelease X event to the client application of the window. The event is generated as if the user pressed on the button number `button`, with the `modifier` keys down and at location `x,y` in the client window coordinates (in pixels). In particular, this means that the event is sent to the smallest subwindow of the application containing `x,y` and having selected to receive button events.

send-key-to-window
send-keycode-to-window — sends key event to a client

```
(send-key-to-window keysym modifier)
(send-key-to-window string modifier)
(send-keycode-to-window keycode modifier)
```

These three functions are used to send key events to the client of the current window, to define function keys in GWM, or to implement mouse positioning of the cursor in Emacs for instance. The third form sends the keycode (as returned by current-event-code), the first the keysym as defined in the include file "keysymdef.h". Both keysyms and keycodes must be **numbers**, not names. You can however specify the symbolic name of the key with the `key-make` function. The `modifier` parameter indicates which modifier keys or which buttons were supposed to be down for the event.

The second form sends each character of the string (this works for ASCII characters only) with the modifier `modifier` to the window. A Shift modifier is automatically added to all uppercase characters.

send-user-event — sends a GWM "user" event

```
(send-user-event atom [wob [do-not-propagate?]])
```

This function sends a "user" event, to the current window, if no argument is present, or to the wob specified as argument. A user event is a GWM concept and is **NOT** an X event, i.e., it is not seen by the server and is immediately processed before `send-user-event` terminates. The peculiarity of an user-event is that it recursively propagates downwards in the wob tree from the destination wob, the event being sent first to the child, then to the wob itself. That is, if you send an user-event to the window, all its decorations will receive it. The atom, which is evaluated by the function, and thus needs to be quoted, is used merely as a kind of label for the event.

If you provide a non-nil third argument, the event is sent to the given wob, but is not propagated to its sons.

Example:

```
(send-user-event 'get-focus)
```

Note: `send-user-event` saves and restores the current wob, window, screen, and event. Thus if a piece of code triggered in another fsm sets the current wob to another one, it will be restored to its previous value on exit of the calling `send-user-event`.

`set-acceleration` — sets mouse speed

```
(set-acceleration numerator denominator)
```

Accelerates the mouse cursor movement by a ratio of numerator/denominator (two numbers).

`set-colormap-focus` — sets the window whose colormap is installed

```
(set-colormap-focus [window])
```

Installs the colormap of the current window (or the given `window`). Once a window has the *colormap focus*, if the client changes its colormap, the new colormap is automatically installed by GWM.

If the window has no declared colormap, the default colormap (the colormap of the root window) is installed instead.

If the argument is `()`, the default colormap for the screen is re-installed.

`set-focus` — sets input focus on a window

```
(set-focus [window])
```

Sets the focus to the current window, or to the given `window`. The keyboard input will then go to the client of this window, regardless of the pointer position. If window is (), the focus is reset to **PointerRoot** mode, i.e., the focus is always on the window under the pointer.

If the client of the current window follows the ICCC `WM_TAKE_FOCUS` protocol, GWM does not try to set the focus to the window, it just sends the `WM_TAKE_FOCUS` message to the client which should then set the focus itself.

NOTE: `set-focus` will not set the focus on a client window that does not need it, so that `set-focus` on XCLOCK for instance has no effect. This is the only case where `set-focus` will return `()`; it returns `t` otherwise.

set-grabs
unset-grabs — grabs events occurring in the window

```
(set-grabs event1 event2 ... eventN)
(unset-grabs event1 event2 ... eventN)
```

`set-grabs` establishes what is called a **passive grab** on a button or a key on the current window, i.e.; all events matching the given events will be transmitted to the window itself, even if they have occurred on a bar, plug, or client window of the window.

`unset-grabs` removes events from the list of grabbed events. They do not exit an existing active grab.

The `set-grabs` call is used when decorating a window on the `grabs` list.

Example:

```
(set-grabs (button 3 alone)
           (buttonpress any with-alt)
           (key (key-make "Delete") any))
(unset-grabs (key any any))
```

set-icon-sizes — sets desired icon sizes

```
(set-icon-sizes min-width min-height
                max-width max-height
                width-inc height-inc)
```

This function sets a property of name `WM_ICON_SIZE` on the root window which tells the clients which are the desirable size for their icon pixmaps or icon windows, if they define any, as returned by `window-icon-pixmap` and `window-icon-window`.

set-key-binding — redefines keyboard for all applications

```
(set-key-binding keycode keysym
                 [keysym1 [keysym2 ... [keysymN]]])
```

This rebinds the keys on the server's keyboard. When the key of keycode `keycode` is pressed, it will be decoded as the keysym `keysym` if pressed alone, `keysym1` if pressed with `modifier1`,... `keysymN` if pressed with modifier `N` (Modifiers are shift, control, lock, meta, etc.)

WARNING: The storage used by this function is never released.

Note: To obtain the list of current bindings of your server, use the **xprkbd** X11 utility.

set-screen-saver — sets screen-saver parameters

```
(set-screen-saver timeout interval
                  prefer-blanking allow-exposures)
```

Sets the way the screen-saver operates:

Timeout is the time in seconds of no input after which the screen blanks (0 means no screen saver, -1 = restore default)

Interval is the interval in seconds between random motion of the background pattern (0 disables motion)

Prefer-blanking makes the screen go blank if 1

Allow-exposures if 0 means that the screen saver operation should not generate exposures, even if this implies it cannot operate.

`set-subwindow-colormap-focus` — installs the colormap of a subwindow

`(set-subwindow-colormap-focus [number])`

If the current window currently has the colormap focus, as set by the `set-colormap-focus` function, and the client has set a `WM_COLORMAP_WINDOWS` property on its window which tells GWM to install the colormaps of its subwindows, calling `set-subwindow-colormap-focus` without an argument installs the next different colormap in the list of subwindows whose colormaps must be installed.

If a numeric argument is given, it is taken as an offset in the list of subwindows (modulo the size of the list), and the corresponding window's colormap is installed. The main window is always at offset zero.

Calling `set-colormap-focus` on the window resets the current offset to zero, for susequent calls to `set-subwindow-colormap-focus`.

`set-threshold` — sets mouse acceleration threshold

`(set-threshold number)`

Sets the minimum pointer movement in pixels before acceleration takes place (See `mouse-acceleration`). The `number` argument must not be 0.

`set`
`setq`
`:` — variable assignement

`(setq atom value)`
`(set object value)`

This is the assignment function of all Lisp dialects. In the first form, the first argument is not evaluated, in the second it is. (Thus allowing non-standard atom names to be set via the atom constructor `atom`, as in `(set (atom "foo bar") 1)`). Both forms evaluate their second argument and sets the value of the first argument to the resulting value. Setting active values doesn't modify their value, but calls a predefined function on the value.

: is just a synonym for `setq`.

Example:

```
(setq b 'c)
(setq a (+ 1 2))
(set  b 4)
```

yields a = 3 and c = 4.

`set-x-property` — sets an X property on a client window

`(set-x-property property-name value)`

Sets the X11 property of name `property-name` on the current client window. The value currently must be a STRING (or atom) or an INTEGER. For instance, `(window-name "foo")` is equivalent to `(set-x-property "WM_NAME" "foo")`

This is the recommended way for communicating between GWM and an X application.

`sort` — sorts a list in place **EXPERT**

`(sort list comparison-function)`

This function sorts (puts in ascending order) in place the `list` argument using the "quicksort" algorithm with the user-provided comparison function. This function is called on pairs of elements and should return `-1`, `0` or `1` if its first argument is lower than, equal to, or greater than the second.

This function is flagged as "*Expert*" as it physically modifies the list. For instance, to obtain a list of windows sorted by names, do:

```
(sort (list-of-windows) (lambda (w1 w2)
        (compare (with (window w1) window-name)
                 (with (window w2) window-name))))
```

`stack-print-level` — number of stack frames printed on error

Numeric variable (number)

On error, Wool prints a stack dump. The number of stack frames printed is given by the value of this variable. Setting it to a negative number puts no limits on the depth of the dump.

`state-make` — makes a state of a fsm

`(state-make transition1 transition2 ... transitionN)`

Makes a state of a fsm (see `fsm-make` and `on` functions) composed of the transitions given in arguments. Returns the constructed state which can be affected to a name via `setq` to be used as the destination state in transitions.

If an argument is itself a state, the new state will **include** all the transitions in the argument state.

`sublist` — extracts a sub-list out of a list

`(sublist from to list)`

Returns the sublist starting at the from-th element and ending at the to-th element of list `list` (both from and to are numbers). `from` is inclusive and `to` is exclusive, and if they are greater than the size of the list, the elements are set to (). Elements are numbered starting at 0.

Examples:

```
(sublist 2 4 '(1 2 3 4))      ==> (3 4)
(sublist 5 9 ())              ==> (() () () ())
(sublist 10 -8 '(1 2))        ==> ()
```

`t` — the logical "true" value

`t`

The "true" value, evaluates to itself.

`tag`
`exit` — non-local goto

`(tag tag-name inst1 inst2 ... instN)`
`(exit tag-name inst1 inst2 ... instN)`

The pair of functions tag/exit implements a non-local goto. When `tag` is called, the non-evaluated tag-name becomes the label of the goto. The instructions are then evaluated in `progn` fashion. If a call to exit with the same tag-name (non-evaluated) is made during these instructions, the evaluation of the tag instructions is aborted and tag returns the evaluation of the instructions of the exit call, evaluated like progn.

If `exit` makes the flow of control exit from a `with` local variable declarations construct, the previous variable values are restored.

WARNING: Do not, when in a file being loaded by `load` do an `exit` with a `tag` set outside the load call — This breaks GWM in the current version.

`tile` — background pixmap

Variable (pixmap)

The value (pixmap) of this global variable is used by the constructors of all wobs to set their background pixmap. For now, it is only used in bars and screens.

`together` — combines keyboard modifiers

`(together modifier1 modifier2 ... modifierN)`

Used to indicate that the modifiers must be pressed simultaneously,

```
(button 1 (together with-shift with-alt))
```

`trace`
`trace-level` — traces WOOL function calls

Active values ( )

This is a primitive debugging tool. When the trace value is set to a non-null number (or `t`) every call to any WOOL function is printed, with the arguments and return value. If set to an expression, this expression will be evaluated before and after each list evaluation (Setting `trace` to `1` instead of `t` re-enables the evaluation of the previous expression).

The trace level variable holds the current indentation (stack depth) of the calls. You might reset it to 0 if you re-enable the tracing after disabling it at a non-0 level.

NOTE: This is a primitive debugging tool, others will be added in the future.

`trigger-error` — triggers a WOOL error

`(trigger-error exprs...)`

This will trigger an error, returning instantly to the toplevel (unless trapped by a `error-occurred` (see 4, p 49)). It will issue an error message, print all the given arguments `exprs...`, and append a newline.

`type` — type of a WOOL object

`(type object)`

Returns the WOOL type of the object as an atom. Current types are:

Atom	Description
active	WOOL active value atom
atom	WOOL normal atom
bar	bar descriptor
client	window descriptor
collection	syntax tree node
cursor	X cursor
event	X event
fsm	finite state machine
fsm-state	state of a fsm
subr	built-in function
fsubr	non-evaluating built-in function
expr	user-made function (defun)
fexpr	non-evaluating user function (defunq)
label	active-label
list	WOOL list
menu	menu used in pop-ups
number	number (used for fonts, wobs, colors,...)
pixmap	X pixmap
plug	plug descriptor
pointer	atom pointing to a memory location
quoted-expr	quoted expression
state-arc	transition arc of a fsm state
string	character strings

`unbind` — undefines a symbol

`(unbind atom)`

Undefine the (evaluated) atom in argument, so that a **`boundp`** on it will return nil.

`ungrab-server` — releases grab on the X server

`(ungrab-server [wob])`

Ungrabs the server, allowing other client requests to be processed. If an argument is given, ungrabs the server only if the argument was the last wob to grab the server, otherwise does nothing. With no argument, unconditionally ungrab the server (keyboard and pointer).

`ungrab-server-and-replay-event` — releases grab on the X server and replay grabbing event

`(ungrab-server-and-replay-event flag)`

When the X server has been grabbed by a passive grab (a grab set on a window by the **`grabs`** context variable or by the **`set-grabs`** function), you can release the grab and make the X server replay the event as if the grab didn't occur by calling this function.

You must set the **`flag`** parameter to `()` if the grab was on a mouse button, and to a non-nil object if the grab was on a key.

This call is useful in *click to type* window managers, to re-send the event which changed the current active window to the client window.

Note: An event can only be replayed if it has been grabbed on a **replayable** event (see `replayable-event`, p 70).

`unmap-window` — unmaps (make invisible) a window

`(unmap-window [window])`

Unmaps (makes invisible) the window (or the current one).

`unpop-menu` — makes a popped menu disappear

`(unpop-menu [menu])`

Removes the grab set by the menu (or menu affected to the current wob if no argument is given) and unmaps it. The `menu` argument can be either a menu (as returned by the `menu-make` function) or a wob (as provided by the `wob` active-value). This function synchronize GWM with the server, so the menu is guaranteed to be unmapped when it returns.

Warning: The current-wob is not changed by this function. The wob which triggered the menu can be accessed as the parent of the menu.

`user-event` — events internal to GWM

`(user-event atom)`

This function is used in transitions to match user-events of the given (evaluated) atom (see `send-user-event`).

Example:

```
(on (user-event 'get-focus) (set-pixmap focus-pattern))
```

`visibility-fully-obscured`
`visibility-partially-obscured`
`visibility-unobscured` — events sent when window visibility changes

Constants (event)

This events are sent to a window when its visibility changes, e.g. when it gets obscured by moving another window on top of it.

`warp-pointer` — warps the mouse pointer to a location **EXPERT**

`(warp-pointer x y [window-relative-to])`

Warps (sets) the mouse pointer to the position `(x,y)`, relative to its current position if no third argument is given, or in the coordinates of the given `window-relative-to` if given. For instance, to warp the pointer to the next screen at the same relative location say:

```
(setq coordinates (current-mouse-position))
(setq screen (% (+ (# 3 coordinates) 1) screen-count))
(warp-pointer (# 0 coordinates) (# 1 coordinates) root-window)
```

WARNING: Use this function just like you would use "goto" in normal programming: never. `warp-pointer` will ruin a window management policy just as easily as "goto"s will ruin a program.

`while` — while loop

`(while condition inst1 inst2 ... instN)`

Executes the N instructions in sequence until condition becomes (). Returns always ().

`window` — current window id

Active value (window id)

Returns or sets the descriptor of the current window as a WOOL number. The current window is the default for all window functions. Setting window to a value sets also the current wob to this window.

WARNING: A `(with (window w)...)` call will modify the value of the current wob:

```
                                  ; window is A, wob is B
    (with (window C) (foo))       ; in foo, window is C, wob is C
                                  ; window is A, wob is A
```

So, if you want the previous call not to modify `wob`, do a `(with (wob w)...)` call instead. The same remark holds for the `screen` active value: a `(with (screen s)...)` will set the value of `window` and `wob` to the value of `screen` before the `with` call.

`window-client-class` — client application class

Active value (string)

Returns a string containing the class of the client owning the window, e.g., **XTerm** for an xterm window.

`window-client-height`
`window-client-width`
`window-client-x`
`window-client-y` — inner window geometry in pixels

Active value (number)

Returns the dimensions in pixels of the client window and its position inside the GWM window frame.

`window-client-name` — client application name

Active value (string)

Returns a string containing the name of the client owning the window, e.g., **xterm** for an xterm window.

`window-group` — manages groups of windows

Active value (window id)

In X11, windows can be grouped with each window in the group bearing a reference to a distinguished window, the *group leader*. GWM maintains such groups as a list of windows, the group leader being the first one in the list. This active value returns () if the window does not belong to a group, otherwise it returns the list of windows in the group.

The user can himself define groups of windows by setting the **window-group** active value to a window id, which is the group leader to which the window should be grouped. The entire group (list) can also be passed as argument, in which case the first element is taken as the window to be grouped to.

To remove a window from a group, just assign `()` to **window-group** for this window, it is removed from the group it was in. If the window was a group leader, the group is broken and all the windows in it are ungrouped.

Note: A window can only belong to one group.

`window-icon` — icon associated to window

Active value (window id — not settable)

Returns the icon associated with the current window. If the current window is already an icon, returns the current window itself.

Note: When GWM decorates a window, it caches the WOOL description given for the icon, but does not create it. The icon is physically created, (and its **opening** field evaluated) on the first access to the **window-icon** active value or the first call to the **iconify-window** function. To just check that the window has an associated icon, without creating it if it didn't exist, use **window-icon?**.

`window-icon?` — tests if icon has already been created

```
(window-icon? [window])
```

Returns `t` if the icon has already been created, `()` if not, without creating it.

`window-icon-name` — name of the icon

Active value (string — not settable)

Returns the name that the client of the current window has given to its icon.

`window-icon-pixmap` — pixmap to be used in icon

Active value (pixmap — not settable)

Returns the pixmap given as a hint by the current client to be used in its icon, or () if no pixmap was specified. The bitmap specified by the application is used to construct the returned pixmap by painting the unset pixels with the **background** color and the set pixels with the **foreground** color on the invocation of this function for each window. Thus each time it is called a new pixmap is created. It is highly recommended that you store the returned value instead of re-calling the function another time.

Use it as the **plug** argument of the **window-make** function after making a plug via **plug-make**.

`window-icon-pixmap-change` — pixmap to be used in icon has changed

Constant (event)

When an application changes its pixmap to be used as its icon, this event is generated. You should then use the `window-icon-pixmap` function to retrieve it if your icon style supports it.

`window-icon-pixmap-id` — X id of pixmap to be used in icon

Active value (number — not settable)

This id is used to know which bitmap the client provided, and if it has changed since last time.

`window-icon-window` — window to be used as icon

Active value (window id — not settable)

Returns a descriptor (number) of the window provided by the current client to be used as its icon, or () otherwise. Should *only* be used as the `plug` argument of the `window-make` function.

`window-is-mapped` — tells if window is visible

Active value (boolean — not settable)

If the current window is mapped (visible) returns it, () otherwise.

`window-is-shaped` — tells if window has a non-rectangular shape

Active value (boolean — not settable)

If the current client window has a non-rectangular outline (on servers supporting the **shape** X11 extension), returns `t`, `()` otherwise.

`window-is-transient-for` — tells if window is transient

Active value (window — not settable)

If not (), the window is transient for another window, and thus you might decide not to decorate it too much. (The window it is transient for is returned.)

`window-is-valid`
`wob-is-valid` — tests if gwm window ID is still valid

```
(window-is-valid window)
(wob-is-valid wob)
```

Since in GWM, windows and wobs are represented by reference, i.e. by numbers meaning a pointer to some data, there is the risk of "dangling pointers", i.e. acessing a memory containing a no longer valid window. To test for these cases, two functions are provided. `wob-is-valid` will test if `wob` is any valid (non closed) plug, bar, menu , window, icon, or root window, whereas `window-is-valid` will verify that `window` is actually only a window or icon. These functions are not strictly necessary, but are useful for debugging purposes.

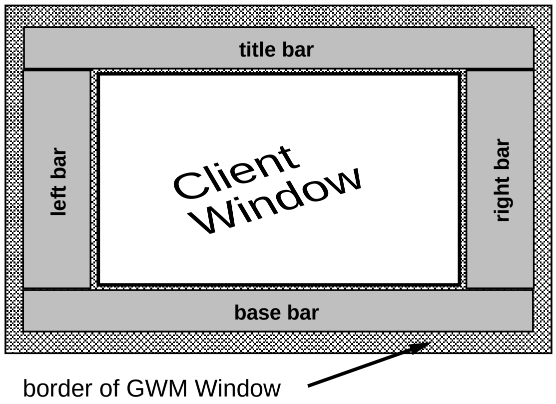
`window-machine-name` — name of host on which the client is running

Active value (string — not settable)

Returns the string containing the name of the machine on which the client owning the window executes (Defaults to `"machine"` if not set.)

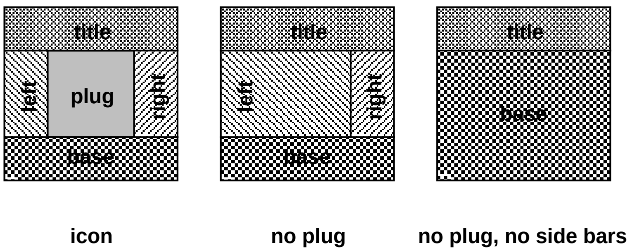
`window-make` — makes a template to decorate a window with

`(window-make titlebar leftbar rightbar basebar plug)`



Returns a description of a GWM window to decorate a newly created X window. This is also used to describe the associated icon and the screen. The four bars are the ones that frame the client and are evaluated another time when the window is physically created. This allows you to give expressions for bars (quoted to evade the first evaluation of arguments of the `window-make` function itself) which evaluates to a bar on the realisation of the wob. Any bar can be set to (), indicating that that no corresponding bar should be created.

The fifth argument `plug` is only used when describing an icon, to be used as the central window around which the bars will be framed. You can give a plug (or an expression which when evaluated gives a plug) or the value of `window-icon-window` for the window. If set to (), the icon has the dimension of the longest side bar, or if they are also set to (), the dimension of the longest top or base bar.



Context used:

Variable	**used for**
`fsm`	the fsm of the window
`borderwidth`	the width of its border
`borderpixel`	the color of its border
`bordertile`	the pixmap tiling its border
`inner-borderwidth`	the border width of the client window
`menu`	the default menu associated to the window
`cursor`	the shape of the cursor when in the window (in fact in the border, which is the only visible part)
`property`	the initial value of the property field
`grabs`	events grabbed from all sons of the window
`opening`	WOOL code evaluated on the creation of the window
`closing`	WOOL code evaluated on the destruction of the window

`grabs` is a list of `button`, `buttonpress`, `key` or `keypress` events which will be "grabbed" from the client window to be sent to the GWM window. This means that the event will be sent directly to the GWM window and **not** to the client window. For instance, to implement a "uwm" move style (moving a window on Alternate/right button anywhere in the window), the grab list should include a `(buttonpress 3 with-alt)`, and the fsm of the window should have a `(on (buttonpress 3 with-alt) (move-window))` transition.

Events declared in the `grabs` list are trapped on the whole surface of the window, including the bars and the client window, and redirected to the main window's fsm.

`opening` and `closing` are two Lisp expressions that are evaluated when the window (or icon) is created and destroyed respectively, just before being mapped or unmapped. This is the right place to position or resize the window before it appears. For the screen, opening is evaluated once all windows already on screen have been framed and closing is evaluated when leaving GWM.

When used for screen description, no arguments are used, only the context values for grabs, opening, closing, fsm, menu, cursor, and property context variables, with the additional context value:

Context used:

Variable	**used for**
`tile`	tiling the screen with a pixmap (if set to a pixmap) or defining the screen color (if set to a color). () means do not change the screen background.

`window-name` — name of the window

Active value (string)

Returns the string containing the name of the window, as set by the client owning it. Note that it is a transient property, and the event `name-change` is issued when it is changed by the client.

When `window-name` is set, the `WM_NAME` X property on the client window is updated accordingly, resulting in the X `name-change` event to be sent to the window by the X server.

Note: All dots in the name are converted to underscores, so that you can use the value of `window-name` safely as an name for the X resource manager.

`window-property`
`wob-property` — WOOL property associated to a wob

Active value

To each wob is associated a property, which is any Lisp object given by the user. These active values return or set the stored property for the current wob or window. When creating a wob, the property is taken as the current value of the global variable `property`.

`window-size` — client window size specified in resize increments

Active value (list of two numbers)

Returns the size of the window expressed as a list of two numbers, multiples of the minimal size (e.g. character positions for xterm). When set, the window is resized accordingly. This is the size of the inner client window, to specify the outer dimensions, use `resize-window`.

`window-starts-iconic` — state in which window must first appear

Active value (boolean)

If not (), the window appears as an icon on its first mapping or the first time it is decorated by GWM. This "hint" can thus be set either by the application or the window manager.

`window-status`
`wob-status` — state of the window

Active value (atom — not settable)

Returns the type of the current wob or window as an ATOM. Possible types are:

Atom	type of wob
window	main window around a client
icon	icon of a window
menu	menu created with `menu-make`
root	the root window
bar	a bar
plug	a plug

Thus, to check if the current window is an icon, just say:

```
(if (= window-status 'icon) ...)
```

`window-user-set-position`
`window-user-set-size` — tells if user explicitly specified the geometry

Active value (boolean — not settable)

Return `t` if the position or size of the current window was set explicitly by the user at creation time via command line switches, () otherwise. If called on an icon, tells if the user defined the initial icon position.

`window-program-set-position`
`window-program-set-size` — tells if program explicitly specified the geometry

Active value (boolean — not settable)

Return `t` if the position or size of the current window was set by default by the program.

`window-to-client`
`client-to-window` — client window Xid to Gwm wob conversions

`(window-to-client gwm-window)`
`(client-to-window X-window-id)`

Converts GWM windows **`gwm-window`** to and from actual decorated client window Ids **`X-window-id`** (numbers used by "**`xwininfo`**" for instance).

`window-was-on-screen` — tells if window was already on screen

Active value (boolean — not settable)

Returns `t` if the window was already on the screen before GWM. You may test its value to do certain actions only on newly created windows (if ()), such as interactive placement.

`window-width`
`window-height`
`wob-height`
`wob-width` — window dimensions in pixels

Active value (number — not settable)

These functions return the size of the current window (with decoration) or wob in pixels. These do not include the width of the border, following the X11 conventions.

`window-window` — main window associated with an icon

Active value (window id — not settable)

Returns the window associated with the current window if it is an icon. If the current window is not an icon, returns the current window itself.

`window-wm-state-icon` — declares user icon for WM_STATE

Active value (window id)

GWM manages automatically the **`WM_STATE`** property on client windows. However, if you implement in a WOOL package your own icons which are not the GWM ones, which are managed by the **`window-icon`** and **`iconify-window`** primitives, for instance by creating a window via the **`place-menu`** primitive, you need to declare which window must logically be considered as the icon for your window. This is done by setting the **`window-wm-state-icon`** on the window to the icon.

Once you have declared that a window has a user-managed icon, GWM no longer updates the **`WM_STATE`** property, so you should call **`window-wm-state-update`** each time you change the state of the window

The state of the window can be:

window / icon	mapped	unmapped
mapped	**normal**	**iconic**
unmapped	**normal**	**withdrawn**

This state information is used by session managers and other window managers, for instance GWM itself will re-iconify the windows that were previouly iconified on a restart.

`window-wm-state-update` — updates WM_STATE property for windows with user icon

`(window-wm-state-update)`

Once you have declared that a window has a user-managed icon, you should update the `WM_STATE` property by calling this function with the `window` global variable set to the managed window each time the state of the window changes.

`window-x`
`window-y` — position of upper-left corner of window in root coordinates

Active value (number — not settable)

These functions return the coordinates of the upper-left corner of the current window, including its decoration, in the root window.

`with`
`with-eval` — local variable declaration

```
(with (var1 value1 ... varN valueN) instructions ...)
(with context instructions ...)
(with-eval expression instructions ...)
```

This is the construct used to declare and initialize variables local to a group of instructions. Active-values and functions are handled as expected, resetting their initial value after the execution of the body of the with. The values are evaluated sequentially.

A context is a list of variables and associated values that can be re-used in many `with` functions (see `context-save`).

`with-eval` first evaluates the expression and then uses it as a context, so that the two following calls are equivalent:

```
(with (a 1 b 2) c)
(with (+ '(a 1) '(b 2)) c)
```

Due to the structure of the WOOL interpreter, `with` works also with `active-values` and functions. For example the following call can be made to move the window "my-window" in the upper left corner of the screen.

```
(with (window my-window move-ul (lambda () (move-window 0 0)))
      (move-ul))
```

Example:

```
(setq bluegreen '(foreground green background blue))
(with bluegreen (pixmap-make "Bull"))
(with (a 2 b (+ a 1)) (print b))        ==>     3
```

`with-shift`
`with-control`
`with-alt`
`with-lock`
`with-modifier-N`
`with-button-N` — modifier states

Constants (number)

These numerical values describe what was in a "down" state (i.e., pressed) when a key or button event was sent. You can combine them with the `together` function, for instance if you want shift **and** control pressed.

N can takes values 2 to 5 for modifiers and 1 to 5 for buttons, e.g., the `with-modifier-5` and `with-button-1` variables are defined.

`wob` — current wob

Active value (wob id)

Returns the descriptor of the current wob (i.e., the wob which received the event being processed) as a WOOL number. The current wob is the default for all wob functions. When set, the current wob is now the wob argument.

`wob-at-coords` — wob containing coordinates

`(wob-at-coords x y)`

Returns the wob including the coordinates relative to the root. Returns () if coordinates were off-screen.

`wob-background` — (solid) background color of wob

Active value (color)

Returns or sets the (solid) background color of the wob. If set, discards the tile of the wob (if there was one) to paint its background with a solid color. Works only with bars and screens.

`wob-borderpixel` — color of border

Active value (color or pixmap)

Get or set the solid color or pixmap of the border of the current wob.

`wob-borderwidth` — width of the border of a wob

Active value (number)

Gets or sets the width in pixels of the border of the current wob. If it is in a composite wob, the father is then resized.

`wob-cursor` — cursor displayed when pointer is in a wob

Active value (cursor)

This active value allows the user to get or modify the mouse cursor which is displayed when the pointer is in the current wob.

`wob-fsm` — gets or sets the fsm associated with current wob

Active value (fsm)

This active value allows the user to get or modify the fsm associated with a wob. If you set the fsm of a wob, it is placed in the initial state. This function is intended for debugging purposes, and its use should be avoided in normal operation. Try using multiple-state fsms instead of changing the fsm.

`wob-invert` — quick and dirty inversion of a wob

```
(wob-invert)
```

Inverts the colors over the surface of the current wob. This is just drawn on top of everything else, and does not affect permanently the wob. This should only be used after calling `grab-server` and then `process-exposes` (Note that `pop-menu` does that for you). This is a "lightweight" function to be used on transient objects. To invert a wob in a cleaner way, use `wob-tile`.

The wob is inverted by using the `invert-color` color (screen relative). This color should be choosen to be the **bitwise-xor**ing of two colors that will be inverted by this functions, the other colors being affected in an unpredictable way.

`wob-menu` — gets or sets the menu associated with current wob

Active value (menu)

This active value allows the user to get or modify the menu associated with a wob.

`wob-parent` — finds the parent of a wob

Active value (wob id — not settable)

Returns the parent of the current wob. Note that the parent of a pop-up is the wob which called the `pop-menu` function. The parent of windows and icons is the root window itself, and the parent of a root window is **nil**.

`wob-tile`
`wob-pixmap` — graphic displayed in a wob

Active value (pixmap)

Returns or sets what is the current wob's pixmap background. For the screen itself or a bar, this is its background tile; for a plug this is the pixmap around which the plug is built. If you change the size of the `wob-tile` for a plug, it is automatically resized (but not a bar or a screen). `wob-pixmap` is just a synonym for `wob-tile` for backward compatibilty purposes.

`wob-x`
`wob-y` — absolute screen position in pixels of current wob

Active values (wob id — not settable)

Returns the position of the top left corner of the wob, including its border as absolute pixel coordinates in its screen.

`xid-to-wob` — translates X ID to wob object

`(xid-to-wob id)`

Returns the wob whose associated X window has the X ID `id`. If no wob if found, returns `()`. The X ID is the identification of windows used by all standard X tools such as XWININFO.

5 Quick Reference

This chapter lists all WOOL objects (functions, variables, and active values) grouped by topics.

5.1 WOOL constructs

Object	Description	p
`;`	WOOL comment	31
`(), nil`	the nil value	33
`t`	the logical "true" value	77
`', quote`	prevents evaluation of argument	33
`eval`	evaluates a WOOL expression	49
`load`	loads and executes a WOOL file	58
`set`	variable assignment	76
`:, setq`	variable assignment	76
`=, equal`	tests equality of any two objects	34
`eq`	tests strict equality of any two objects	49
`execute-string`	executes (parses and evaluates) a WOOL string	49
`hack`	raw access to GWM internal structures	53
`type`	type of a WOOL object	78
`with`	local variable declaration	88
`with-eval`	local variable declaration (with evaluation)	88

5.2 Flow control

Object	Description	p
`progn`, {}	sequence of instructions	69
`if`	conditional test	54
`cond`	conditional test	40
`for`	iterates through a list of values	50
`mapfor`	constructs a list by iterating through a list of values	50
`while`	while loop	81
`tag`	non-local goto: label	77
`exit`	non-local goto: branching	77
`end`	terminates GWM	48
`error-occurred`	traps errors occuring in expressions	49
`trigger-error`	triggers a wool error	78
`process-events`	recursively process all pending events	68

5.3 I/O

Object	Description	p
`?, print`	prints WOOL objects	34
`bell`	rings the keyboard bell	37

5.4 Atoms

Object	Description	p
`atom`	makes an atom from a string	35
`boundp`	tests if an atom has already been defined	38
`unbind`	Un-defines an atom	79

5.5 Namespaces

Object	Description	p
`namespace-make`	creates a namespace	64
`namespace-add`	adds a state to a namespace	63
`namespace-remove`	removes a state from a namespace	64
`defname`	declares a name in a namespace	45
`namespace`	sets current state of a namespace	63
`namespace-of`	returns namespace of a symbol	64
`namespace-size`	number of states in the namespace	64

5.6 Functions

Object	Description	p
`defun, de`	defines a Wool function evaluating its arguments	44
`defunq, df`	defines a Wool function quoting its arguments	44
`lambda`	defines an unnamed Wool function evaluating its arguments	44
`lambdaq`	defines an unnamed Wool function quoting its arguments	44

5.7 Lists

Object	Description	p
`#, nth`	accesses or modifies an element of a list or a property list	31
`##, replace-nth`	physically modifies an element of a list or a property list	32
`( ... )`	list notation	32
`+`	appends lists	34
`length`	length (number of elements) of a list	56
`list`	creates a list	57
`member`	position of element in a list	60
`sublist`	extracts a sub-list out of a list	77
`delete-nth`	physically removes an element of a list	45
`copy`	copies a list	41
`list-make`	makes a list of a given size	57
`sort`	sorts a list in place	77

5.8 Strings

Object	Description	p
`" ... "`	string notation	33
`+`	concatenates strings	34
`<`	tests for strict lexicographic inferiority	34
`>`	tests for strict lexicographic superiority	34
`compare`	compare two strings for lexicographic order	40
`member`	position of substring in string	60
`atoi`	ASCII string to Integer conversion	35
`itoa`	Integer to ASCII string conversion	54
`atom`	makes an atom from a string	35
`length`	length (number of characters) of a string	56
`match`	general regular expression matching and extracting package	58

5.9 Logical functions

Object	Description	p
`()`	the nil value	33
`t`	the logical "true" value	77
`and`	logical AND of expressions	35
`not`	logical negation	64
`or`	logical OR of expressions	65

5.10 Numbers

Object	Description	p
`+`	adds numbers	34
`*, %, /`	arithmetic binary operators	33
`-`	arithmetic difference or sign inversion	34
`<`	tests for strict numerical inferiority	34
`>`	tests for strict numerical superiority	34
`compare`	compare two numbers	40
`bitwise-and`	bitwise and operator	37
`bitwise-or`	bitwise or operator	37
`bitwise-xor`	bitwise exclusive-or operator	37
`atoi`	ASCII string to Integer conversion	35
`itoa`	Integer to ASCII string conversion	54

5.11 Graphical primitives

Object	Description	p
`active-label-make`	makes a label (text in a given font)	34
`bordertile`	pixmap to tile the border of a wob	37
`foreground`	color of the foreground	50
`dimensions`	position and dimensions of a WOOL object	46
`height`	height of a WOOL object	46
`width`	width of a WOOL object	46
`label-horizontal-margin`	margins around labels	56
`label-vertical-margin`	margins around labels	56
`active-label-make`	makes a graphic string object refreshing itself on exposures	34
`label-make`	makes a pixmap by drawing a string	56
`pixmap-make`	builds a pixmap (color image)	66
`pixmap-load`	builds a pixmap from an XPM file description	65
`tile`	background pixmap for creating a wob	78
`wob-tile, wob-pixmap`	gets/sets the graphic displayed in a wob	90
`draw-line`	draws a line in a pixmap	48
`draw-rectangle`	draws a (filled) rectangle in a pixmap	48
`draw-text`	draws a string of characters in a pixmap	48

5.12 System interface

Object	Description	p
`!`	executes a shell command	31
`getenv`	gets the value of a shell variable	52
`restart`	restarts GWM	72
`elapsed-time`	gets time used by GWM in milliseconds	48
`hostname`	name of the machine on which GWM is running	53

5.13 Events

Object	Description	p
`any`	matches any modifier or button	35
`button`	makes a button event	38
`buttonpress`	makes a buttonpress event	38
`buttonrelease`	makes a buttonrelease event	38
`current-event-code`	code of the last event	41
`current-event-modifier`	modifier of the last event	41
`current-event-from-grab`	tests if last event was generated by a grab	41
`current-event-window-coords`	relative position of the last event	41
`current-event-x`	absolute position of the last event	42
`current-event-y`	absolute position of the last event	42
`current-event-relative-x`	relative position of the last event	42
`current-event-relative-y`	relative position of the last event	42
`current-event-time`	time in milliseconds of the last event	41
`current-user-event`	name of the last user event	42
`double-button`	makes a double-click button event	47
`double-buttonpress`	makes a double-click buttonpress event	47
`double-click-delay`	maximum time between double clicks	47
`enter-window`	event generated when the pointer crosses the border of a wob	49
`leave-window`	event generated when the pointer crosses the border of a wob	49
`enter-window-not-from-grab`	event generated when the pointer actually crosses the border of a wob	49
`leave-window-not-from-grab`	event generated when the pointer actually crosses the border of a wob	49
`focus-in`	event received when input focus changes on the client window	50
`focus-out`	event received when input focus changes on the client window	50
`geometry-change`	event generated when window changes size	51
`key`	makes a key event	55
`keypress`	makes a keypress event	55
`keyrelease`	makes a keyrelease event	55
`key-make`	makes a key symbol out of a descriptive name	55
`last-key`	last key pressed	56
`name-change`	event generated when window changes its name	63
`property-change`	event generated when a client window changes a property	69
`window-icon-pixmap-change`	pixmap to be used in icon has changed	82
`send-user-event`	sends a GWM "user" event to another wob	73
`user-event`	event internal to GWM	80
`set-grabs`	grabs events occurring in the window	75
`unset-grabs`	removes grabs set by set-grabs	75
`map-notify`	event sent when window is mapped	58
`visibilty...`	events sent when window visibility changes	80

5.14 Keyboard modifiers

Object	Description	p
`alone`	specifies that no modifier key is used	35
`any`	matches any modifier or button	35
`together`	combines keyboard modifiers	78
`with-shift`	shift key was depressed for the event	89
`with-control`	control key was depressed for the event	89
`with-alt`	alt key was depressed for the event	89
`with-lock`	lock key was depressed for the event	89
`with-modifier-N`	modifier N key was depressed for the event	89
`with-button-N`	button N was depressed for the event	89

5.15 Access to X11 primitives

Object	Description	p
`screen-type`	returns visual type of screen	73
`current-mouse-position`	queries server for current mouse position	42
`wob-at-coords`	wob containing coordinates	89
`display-name`	name of the X server	47
`get-x-property`	gets an X property on the client window	52
`set-x-property`	sets an X property on a client window	76
`grab-server`	grabs the server	52
`ungrab-server`	releases grab on the X server	79
`replayable-event`	makes a replayable event from a normal event	70
`allow-event-processing`	un-freezes the server after catching a replayable event	35
`ungrab-server-and-replay-event`	releases grab on the X server and replay grabbing event	79
`refresh`	refreshes the screen	70
`resource-get`	searches GWM database for a resource	71
`resource-put`	puts a resource in GWM database	71
`set-focus`	sets input focus on a window	74
`set-colormap-focus`	sets the window whose colormap is installed	74
`set-subwindow-colormap-focus`	installs the colormap of a subwindow	76
`set-key-binding`	redefines keyboard for all applications	75
`keycode-to-keysym`	converts a key code to its symbolic code	55
`keysym-to-keycode`	converts a symbolic code to a key code	55
`set-screen-saver`	sets screen-saver parameters	75
`set-acceleration`	sets mouse speed	74
`set-threshold`	sets mouse acceleration threshold	76
`process-exposes`	treats all pending expose events	69
`warp-pointer`	warps the mouse pointer to a location	80
`xid-to-wob`	translates X ID to wob object	91

5.16 Global variables controlling GWM behavior

Object	Description	p
`describe-screen`	user function called to describe a screen	46
`describe-window`	user function called to decorate a new window	46
`freeze-server`	stops processing other clients during grabs	51
`grab-keyboard-also`	grab-server grabs also keyboard events	52
`confine-grabs`	cursor stays confined in grabbing wobs	40
`confine-windows`	forces windows to stay on-screen	40
`reenter-on-opening`	process events on the queue just before mapping a new window	70
`invert-cursors`	inverts the bitmaps used for making cursors	54
`move-grid-style`	style of grid for move	62
`resize-grid-style`	style of grid for resize	62
`resize-style`	style of interactive resize	70
`check-input-focus-flag`	follows input hint for setting focus	39
`print-errors-flag`	controls printing of error messages	68
`print-level`	controls printing depth of lists	**68**
`gwm-quiet`	silent startup	53

5.17 Colors

Object	Description	p
`color-make`	allocates a pixel color by name	39
`color-make-rgb`	creates a color from RGB values	40
`color-free`	de-allocates a pixel	39
`color-components`	gives RGB color decomposition of a pixel	39
`background`	color of the background	36
`borderpixel`	color of the border of a wob	37
`foreground`	color of the foreground	50
`grid-color`	color to draw (xor) the grids with	52
`invert-color`	color to invert (xor) the wobs with	54

5.18 Wobs

Object	Description	p
`borderpixel`	color of the border of a wob	37
`bordertile`	pixmap to tile the border of a wob	37
`borderwidth`	width in pixels of the border of a wob	38
`dimensions`	position and size of a Wool object	46
`height`	height of a Wool object	46
`width`	width of a Wool object	46
`menu`	menu associated with wob	60
`root-window`	the root window	72
`wob-property`	Wool property associated to a wob	86
`wob-status`	state of the window	86
`wob-x`	x position in pixels	90
`wob-y`	y position in pixels	90
`wob-height`	wob height in pixels	87
`wob-width`	wob width in pixels	87
`wob`	current wob	89
`wob-at-coords`	wob containing screen coordinates	89
`wob-borderpixel`	color of border	89
`wob-borderwidth`	width of the border of a wob	89
`wob-fsm`	gets or sets the fsm associated with current wob	90
`wob-invert`	quick and dirty inversion of a wob	90
`wob-menu`	gets or sets the menu associated with current wob	90
`menu-wob`	returns wob associated with menu	61
`wob-parent`	finds the parent of a wob	90
`wob-tile`	graphic displayed in a wob	90
`wob-pixmap`	graphic displayed in a wob	90
`wob-cursor`	cursor displayed when pointer is in a wob	89

5.19 Plugs

Object	Description	p
`plug-make`	makes a plug	67
`wob-tile`	graphic displayed in a wob	90
`wob-pixmap`	graphic displayed in a wob	90

5.20 Bars

Object	Description	p
`bar-make`	makes a bar descriptor	36
`bar-min-width`	minimum transversal width of a bar	37
`bar-max-width`	maximum transversal width of a bar	37
`background`	color of the background	36
`plug-separator`	inter-plug space within bars	67
`wob-background`	(solid) background color of wob	89
`wob-tile`	graphic displayed in a wob	90
`wob-pixmap`	graphic displayed in a wob	90

5.21 Menus

Object	Description	p
`direction`	direction of menus	46
`bar-separator`	number of pixels between consecutive bars in menus	37
`horizontal`	direction of menus	53
`vertical`	direction of menus	53
`menu`	menu associated with wob	60
`menu-make`	makes a menu	60
`place-menu`	maps a menu as a normal client window	67
`pop-menu`	pops a menu	68
`unpop-menu`	makes a popped menu disappear	80
`wob-menu`	gets or sets the menu associated with current wob	90

5.22 Windows

Object	Description	p
`describe-window`	user function called to decorate a new window	46
`circulate-windows-down`	circulates mapped windows	39
`circulate-windows-up`	circulates mapped windows	39
`grid-color`	color to draw (xor) the grids with	52
`iconify-window`	iconifies or de-iconifies a window	53
`inner-borderwidth`	borderwidth of the client window	54
`kill-window`	destroys a client	56
`list-of-windows`	returns the list of all windows	57
`lower-window`	lowers a window below another window	58
`map-window`	maps window	58
`move-window`	moves window	63
`raise-window`	raises a window on top of other windows	69
`re-decorate-window`	re-decorates the client window by GWM	69
`resize-window`	resizes the window	71
`root-window`	the root window	72
`unmap-window`	unmaps (make invisible) a window	80
`window`	current window id	81
`window-make`	makes a template to decorate a window with	84
`window-property`	WOOL property associated to a wob	86
`window-size`	client window size specified in resize increments	86

5.23 Window characteristics

Object	Description	p
`window-client-class`	client application class	81
`window-client-name`	client application name	81
`window-client-height`	inner window height in pixels	81
`window-client-width`	inner window width in pixels	81
`window-client-x`	inner window x in pixels	81
`window-client-y`	inner window x in pixels	81
`window-group`	manages groups of windows	81
`window-icon`	icon associated to window	82
`window-window`	window associated to icon	87
`window-icon-name`	name of the icon	82
`window-icon-pixmap`	pixmap to be used in an icon	82
`window-icon-window`	window to be used as icon	83
`window-is-mapped`	tells if window is visible	83
`window-is-shaped`	tells if window has non-rectangular shape	83
`window-is-transient-for`	tells if window is transient	83
`window-machine-name`	name of host on which the client is running	84
`window-name`	name of the window	85
`window-size`	client window size specified in resize increments	86
`window-starts-iconic`	state in which window must first appear	86
`window-status`	state of the window	86
`window-user-set-position`	tells if user explicitly specifed the position	86
`window-user-set-size`	tells if user explicitly specifed the size	86
`window-program-set-position`	tells if program explicitly specifed the position	87
`window-program-set-size`	tells if program explicitly specifed the size	87
`window-was-on-screen`	tells if window was already on screen	87
`window-width`	window width in pixels	87
`window-height`	window height in pixels	87
`window-x`	x position of upper-left corner of window in root	88
`window-y`	y position of upper-left corner of window in root	88
`grabs`	events grabbed on the whole window	52

5.24 Screen

Object	Description	p
`screen`	current screen	72
`describe-screen`	user function called to describe a screen	46
`root-window`	the root window	72
`screen-depth`	screen depth	73
`screen-height`	screen height	73
`screen-width`	screen width	73
`list-of-screens`	list of managed screens	57
`screen-count`	number of screens attached to the display	72
`tile`	background pixmap	78
`wob-background`	(solid) background color of wob	89

5.25 Context

Object	Description	p
`context-save`	context management	40
`context-restore`	context management	40
`with`	local variable declarations	88
`with-eval`	local variable declarations	88

5.26 Cursors

Object	Description	p
`cursor`	shape of the cursor in a wob	42
`cursor-make`	makes a cursor with a bitmap and a mask	43
`invert-cursors`	inverts the bitmaps used for making a cursor	54

5.27 Communication with other X11 clients

Object	Description	p
`GWM_EXECUTE`	Sending commands to GWM	10
`cut-buffer`	contents of cut buffer 0	43
`rotate-cut-buffers`	rotate server cut buffers	69
`get-x-property`	gets an X property on the client	52
`set-x-property`	sets an X property on a client window	76
`delete-read-properties`	flags to delete X properties after reading them	45
`resource-get`	searches Gwm database for a resource	71
`resource-put`	puts a resource in Gwm database	71
`get-x-default`	gets a server default	51
`get-wm-command`	gets the WM_COMMAND property	51
`send-key-to-window`	sends key event to a client	73
`send-keycode-to-window`	sends key event to a client	73
`send-current-event`	re-sends X event to the client of a window	69
`window-wm-state-icon`	declares user icon for WM_STATE	87
`window-wm-state-update`	updates WM_STATE property for windows with user icon	88
`set-icon-sizes`	sets desired icon sizes	75
`delete-window`	asks client to delete one of its windows	45
`window-to-client`	client window Xid to Gwm wob conversions	87
`client-to-window`	client window Xid to Gwm wob conversions	87

5.28 Session manager functions

Object	Description	p
`save-yourself`	asks client to update its WM_COMMAND property	72

5.29 Fonts

Object	Description	p
`font`	default font	50
`font-make`	loads a font	50

5.30 Fsms

Object	Description	p
`fsm`	Finite State Machine of the wob	51
`fsm-make`	compiles an automaton	51
`on`	triggers a transition on an event in a state of a fsm	65
`on-eval`	triggers a transition on an event in a state of a fsm	65
`state-make`	makes a state of a fsm	77
`wob-fsm`	gets or sets the fsm associated with current wob	90

5.31 Meter

Object	Description	p
`meter-close`	unmaps the meter	62
`meter-open`	displays the meter	62
`meter-update`	writes a string in the meter	62
`meter`	sets meter attributes	61
`move-meter`	shows meter during moves	62
`resize-meter`	shows meter during resizes	62

5.32 Hooks

Object	Description	p
`opening`	WOOL expression evaluated at the creation of a window or icon	65
`closing`	WOOL expression evaluated at the destruction of a window or icon	65

5.33 Debugging tools

Object	Description	p
`trace`	traces WOOL function calls	78
`trace-level`	traces WOOL function calls	78
`stack-print-level`	number of stack frames printed on error	77
`hashinfo`	statistics on atom storage	53
`meminfo`	prints memory used	60
`oblist`	prints all defined objects	65
`window-is-valid`	tests if window is still valid	83
`wob-is-valid`	tests if wob is still valid	83

Index