XGP

User Guide

For version 1.2
August 28, 2010

by Lindsey Spratt

Contents

1 Sources

The manual for GProlog [3] is *A Native Prolog Compiler with Constraint Solving over Finite Domains* [4]. This can be found online at www.gprolog.org/manual .

Complete documentation of the original MacProlog32 menu, dialog, and control window interfaces is in *The MacProlog32 Programming Guide* [1]. The MacProlog32 graphics interface is documented in *The MacProlog32 Graphics Guide* [2].

The LaTex format for this document is copied from the GProlog manual sources.

2 Introduction

XGP is an integrated development environment that extends gprolog to work with Cocoa under Macintosh OS X. It extends gprolog with builtins predicates for dialog, menu, and graphics facilities.

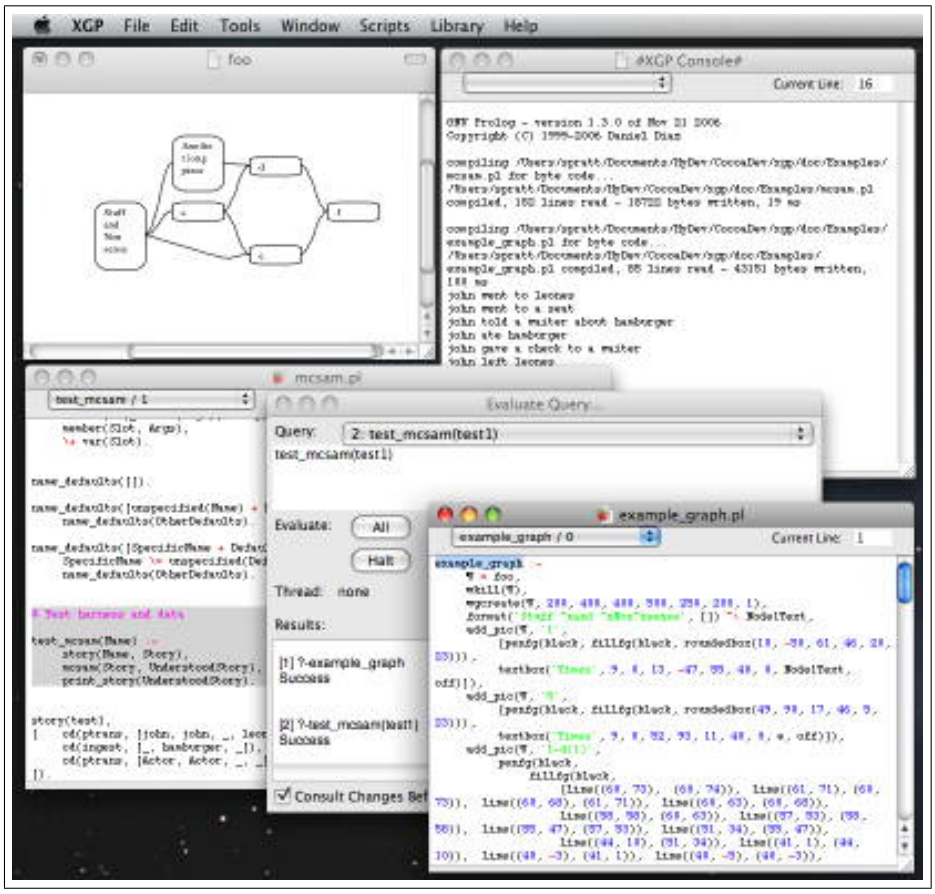
This document describes the XGP application. There is a separate manual for gprolog. GNU Prolog (gprolog) is an open source native-code prolog with a finite domain constraint solver. Its main web site is http://gprolog.org.

XGP uses gprolog version 1.3.2.

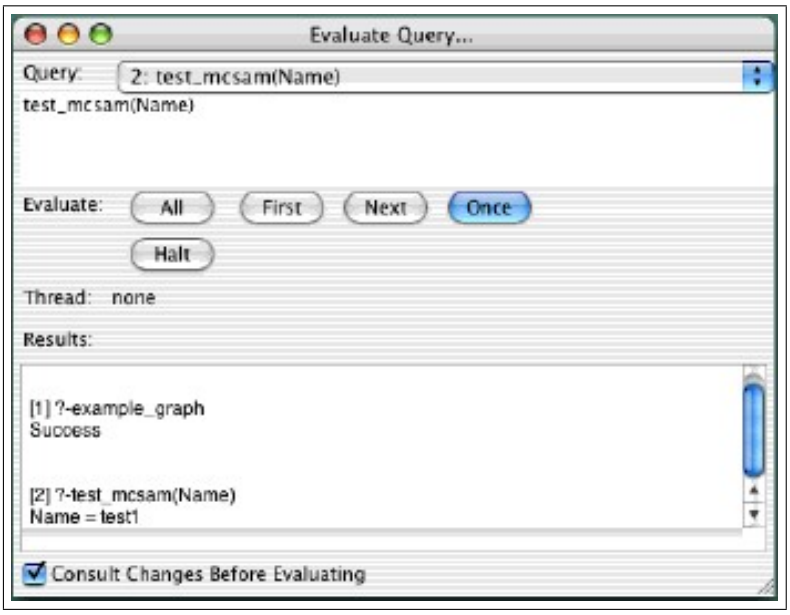
3 A Quick Look

3.1 Introduction

Below is a screen shot of XGP with five windows: a graphics window, the *XGP Console* document window, the *mcsam.pl* document window, the *example_graph.pl* document window, and an *Evaluate Query...* dialog.

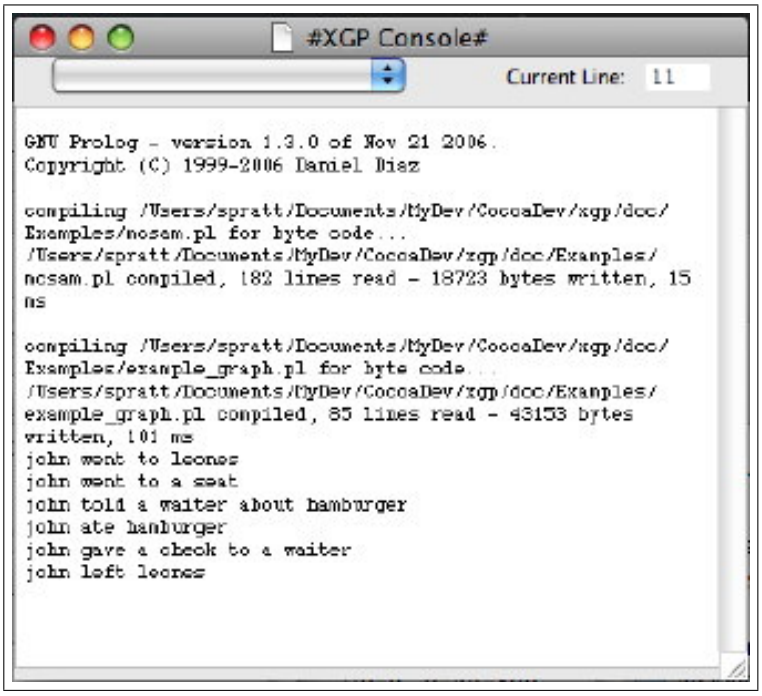


3.2 The Evaluate Query Dialog



This dialog shows the query "test_mcsam(test1)" has just been successfully evaluated (and just prior to that the query "example_graph" was evaluated). The evaluation of this query wrote output to the *#XGP Console#* document window.

3.3 The #XGP Console# Document



This document is the default input and output window for gprolog. The upper right corner indicates number of the line on which the cursor is currently located. The user can type in a line number followed

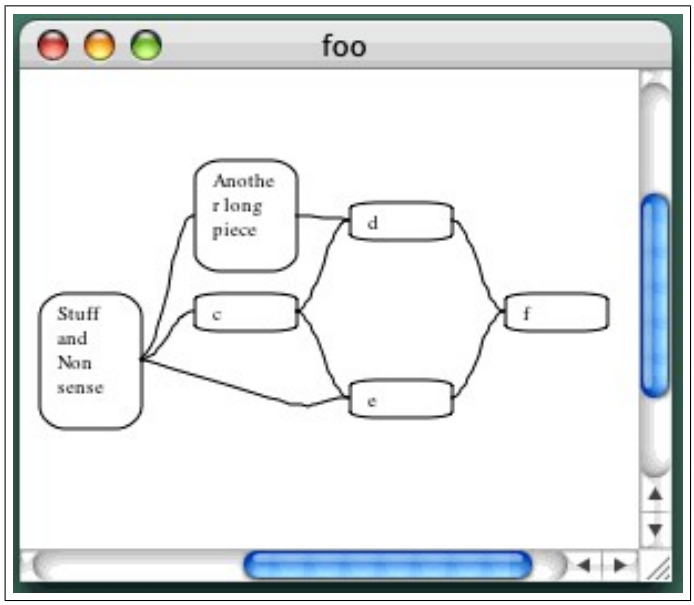
by a Return or Enter and that line is selected and the window is scrolled to display it. The predicate definition popup is empty since this is not a consulted source document window.

This example shows the version information for GProlog, the compilation information resulting from the consulting of the files *example_graph.pl* and *mcsam.pl*, and finally output created by evaluating "test_mcsam(test1)".

3.4 A Source Document

This is the source document for mcsam.pl. This is essentially the same kind of document as the *#XGP Console#*. The upper right corner indicates the cursor is currently located on line 76. The predicate definition popup is currently on test_mcsam / 1 (the currently selected predicate).

3.5 The Graphics Document



This document displays pictures drawn in it using the XGP graphics API from within a prolog program, example_graph. (This example was originally created by a program that does automated layout and display of graphs. Eventually this program will be distributed with XGP.)

4 How to Use XGP

4.1 Creating a Program

A new program can be created from within XGP or using any external text editor to create a new program file and then consult or open that file in XGP. (Consulting or opening a file is discussed in "Consulting a Program File" 4.4.)

Creating a new program file in XGP is done by selecting the "New" option of the "File" menu. This creates a document window named "untitled". Enter a program in this document window, such as:

```
foo.
```

Save the program ('Save' option of the 'File' menu), then consult it ('Consult *document*' of the 'Tools' menu).

4.2 Document Window Features

The document window used for editing a program has certain features to make editing more convenient.

The tokens in the source code are colored according to their syntax. There are four categories of syntax coloring: comments, string constants (enclosed in double-quote characters), numbers, and 'operator' atoms (e.g. '+' and ':-'). The 'Syntax Coloring' preferences (discussed at 4.6.3) can be used to specify the colors to be used. The syntax coloring is refreshed *only* after every 'consult' of the source document.

The upper right corner of the source document indicates the current line number of the editing cursor. If you type a number in this control followed by 'return' or 'enter', then the editing cursor is moved to that line and the view is scrolled to show the cursor.

The upper left corner of the source document has a popup menu button that lists all of the predicates defined in the document. If a predicate is selected in this menu, then the definition of that predicate is selected in the document and the view is scrolled to that selected text. Clicking on the popup menu shows the predicates in the order that they are defined in the document. Option-clicking on the menu shows the predicates in alphabetical order.

The 'Complete Selected Predicate' item of the 'Scripts:Editing' menu replaces selected text with a "completion" of it based on already defined predicates. The 'Uppercase Selected Text' and 'Lowercase Selected Text' items of that menu replaces the selected text with the uppercased and lowercased versions of it, respectively.

4.3 Evaluating a Query

To evaluate a prolog query within XGP, select the 'Evaluate Query...' option of the 'Tools' menu. This displays a dialog with two fields and several 'Evaluate' buttons. Type your query into the 'Query' field, then click on the 'Once' button. The success, failure, or exception result of the query is indicated in the 'Results' field. If the query was successful and there were unbound variables in it, then the bindings acquired by these variables is also shown in the 'Results' field.

The 'First' and 'Next' buttons can be used instead of the 'Once' button to get the first result followed by each of the next results. The 'All' button can be used to get all of the solutions.

The history popup menu above the Query field lists all of the queries used for that query dialog. Selecting one of these prior queries copies it into the Query field.

Any output created (e.g. by a write/1 predicate) is written to the '#XGP Console#' window. Input may also be read (e.g. by the read/1 predicate) from the '#XGP Console#' window. For instance, when using the trace facility (invoked by the gprolog trace/0 predicate), you must type into the '#XGP Console#' window to control the trace process.

Queries can be evaluated using the 'Evaluate Current Query' and 'Evaluate Selected Text' items of the 'Scripts:Evaluation' menu (discussed below in the "Scripts: Evaluation" section 4.5.1).

4.4 Consulting a Program File

A program file is a text file with the '.pl' extension containing Prolog source. There are several ways to consult such a file in XGP. One way is to use the 'Load Programs...' option of the 'Tools' menu. This uses a standard "open file" dialog to allow you to select a source file to be consulted. The other ways are to use the 'Open...' option of the 'File' menu or to explicitly or implicitly consult an already open document.

When a file is opened by the 'Open...' option of the 'File' menu there is a document window containing the contents of that file ¡I¿and¡/I¿ the file is consulted. You may re-consult that file at any time by using the 'Consult Document' option of the 'Tools' menu to load (consult) the active (document) window's contents.

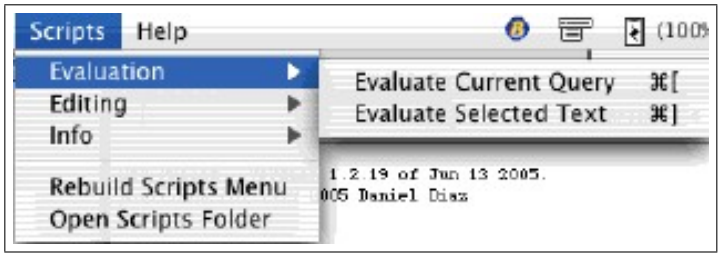
All of the documents that have been changed since the last time they were consulted explicitly or implicitly. They may be consulted explicitly in a single command by selecting the 'Consult All Changed Documents' option of the 'Tools' menu. They may be implicitly consulted by selecting the 'Consult Changed Documents Before Evaluation' option on the 'Evaluate Query...' dialog. With this option selected, whenever a query evaluation is requested XGP first ensures that all changed (source) documents are re-consulted before actually running the evaluation.

The program file from which a predicate was compiled can be found using the 'Source File For Selection' item of the 'Scripts:Info' menu, as discussed below.

4.5 Scripts

The standard 'Scripts' menu contains items that provide additional features for query evaluation and program editing. This menu can be easily modified by the user with her own "scripts" (prolog programs).

4.5.1 Evaluation



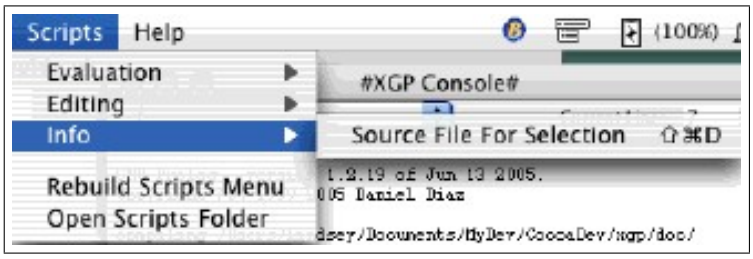
'Evaluate Current Query' re-evaluates whatever was the most recently executed query in the 'Evaluate Query...' dialog. 'Evaluate Selected Text' copies the currently selected text (generally in some program document window) to the 'Evaluate Query...' dialog query box and evaluates it.

4.5.2 Editing

'Complete Selected Predicate' presents a dialog with completions of the selected text from the predicates defined in the current XGP environment. The user selects an item from this list and it replaces the selected text.

'Uppercase Selected Text' and 'Lowercase Selected Text' make the selected text all uppercase or lowercase, respectively.

4.5.3 Info

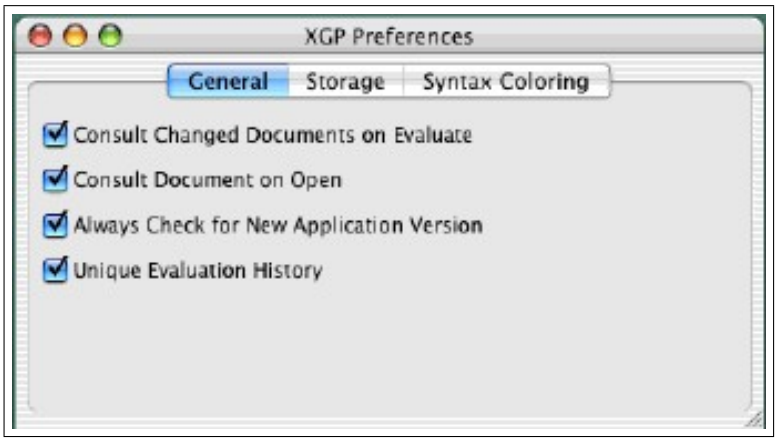


'Source File For Selection' displays a message dialog containing the pathname of the source file from which the selected predicate was compiled.

4.6 Preferences

Preferences for the XGP environment are set through the 'Preferences...' option of the 'XGP' application menu. There are three tabs for the Preferences window: 'General', 'Storage', and 'Syntax Coloring'.

4.6.1 General

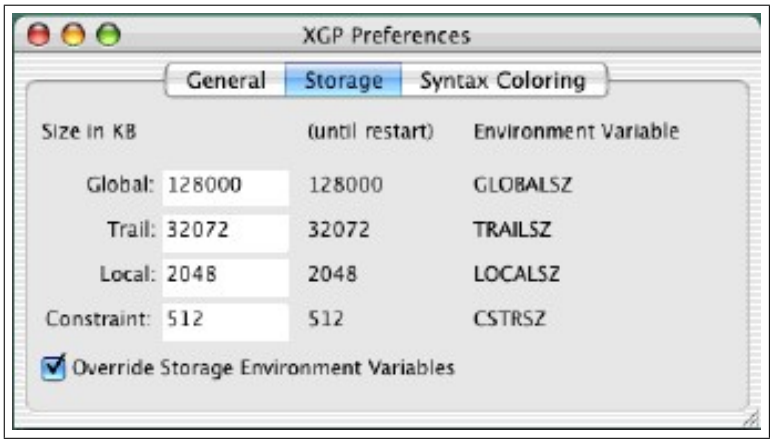


The four preferences of the 'General' tab are 'Consult Document on Open', 'Consult Changed Documents Before Evaluation', 'Always Check for New Application Version', and 'Unique Evaluation History'. If 'Consult Document on Open' is checked, then opening a source document using the 'Open...' option of the 'File' menu will cause that document to be consulted automatically. The state of the 'Consult Changed Documents Before Evaluation' option is used as the default value for the 'Evaluate Query...' dialog's same-named option.

When the 'Always Check...' option is checked XGP compares the version of the currently released XGP (as recorded on the xgp.sourceforge.net site) with the version of XGP being opened. If the version being opened is not up-to-date, then user is queried if she would like to download the current version. This is the same service provided by the 'Check for New Application Version' option of the 'XGP' application menu.

The 'Unique Evaluation History' option controls specifies that the history popup on the 'Evaluation Query...' dialog lists only unique occurrences of queries. Otherwise, all queries are listed separately, even if identical.

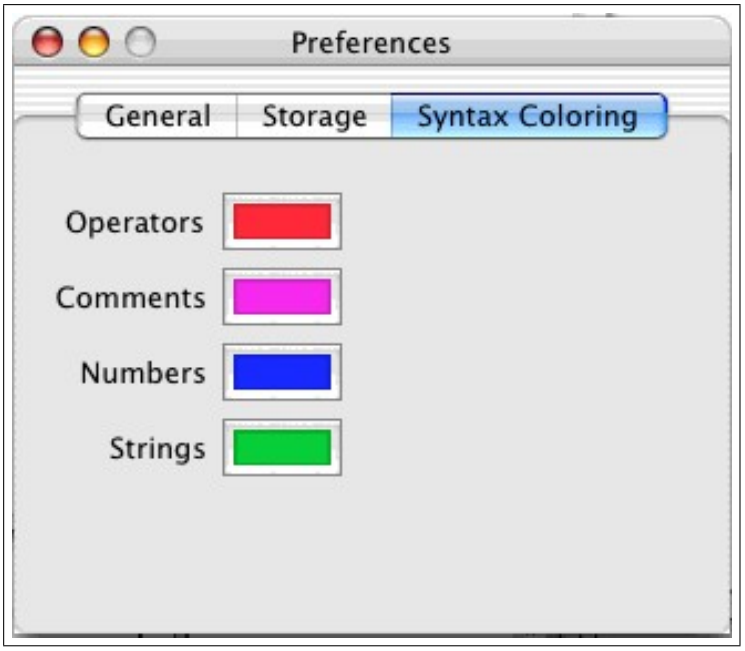
4.6.2 Storage



The five preferences in the 'Storage' tab allow the user to specify the "stack" sizes used by gprolog and to control whether the 'preference' version of the stack size overrides the corresponding environment variable

(if any). Changes to the stack sizes don't take effect until after XGP is restarted. The currently in effect ("until restart") values are shown in the central column.

4.6.3 Syntax Coloring



The 'Syntax Coloring' tab specifies the colors to use for "Operators" (all of the current operators after consulting the file being colored), "Comments" (end-of-line indicated by "%" and multi-line indicated by an opening"/*" and closing "*/"), "Numbers" (any symbols interpreted as numbers by gprolog), and "Strings" (atoms delimited by single-quotes and lists delimited by double quotes).

4.7 Changing XGP

4.7.1 Changing Scripts

The Scripts menu is defined by the script files found in one of the standard locations: '~/Library/Application Support/XGP/Scripts', '/Library/Application Support/XGP/Scripts', and '$XGPINITPATH/../../Scripts', in this order. For a user to change her own XGP Scripts, she should put the desired script files in '~/Library/Application Support/XGP/Scripts'. To change the script files for all users on a system, put the files in '/Library/Application Support/XGP/Scripts'. The script files shipped with XGP are in '$XGPINITPATH/../../Scripts', which translates to '*XGP.app location*/Contents/Resources/Scripts'.

The hierarchy of folders defines the hierarchy of the Scripts menu. Each Prolog source or byte-code file ('*.pl' or '*.wbc') defines a menu item for the menu corresponding to the source files' parent folder. (This organization of the Scripts folder is modeled on XCode.) The menu item's name, key shortcut, and invoked predicate are defined by an initialization directive in the script source file:

```
:- initialization( xgp_register_script(ItemNameAndKey, InvokedPredicate)).
```

The *ItemNameAndKey* atom has the structure:

```
Item[/Key[/Modifier1[/Modifier2...]]]
```

(where a portion identified by [...] is optional)

Example *ItemNameAndKey*:

- 'Foo' - defines a menu item 'Foo' with no key shortcut.
- 'Bar/b' - defines a menu item 'Bar' with shortcut command-b. (The 'command' key modifier is always implied for a shortcut.)
- 'Baz/z/option/shift' - defines a menu item 'Baz' with shortcut option-command-Z.

The xgp__setup_scripts_menu/[0, 1] predicates load the script source files, maintaining context about the location in the Scripts folder hierarchy of each source file as it is loaded. This context information is used by xgp_script_registration/2 to extend the appropriate menu. Script items are added in the order that their defining files are processed. Script files are processed in alphabetical order of the names of the files; a numeric prefix (e.g. '20-') can be used to force a particular alphabetical ordering. A dummy script file with a numeric prefix and three dashes defines a menu separator.

4.7.2 Initializing the Environment

XGP uses a file called either 'initialize_environment.pl' or 'intialize_environment.wbc' in the Contents/Resources/English.lproj folder of the XGP application bundle to initialize the menus for the XGP application. If *any* '.pl' file in the XGP.app bundle has been modified more recently than the 'initialize_environment.wbc' file, then XGP uses the 'initialize_environment.pl' file.

The 'initialize_environment.pl' file (and the files on which it depends) may be modified to alter the behavior of XGP. It uses the 'environment_basics.pl', 'standard_application_menus.pl', 'evaluate_dialog.pl', 'preference_dialog.pl' and 'scripts.pl' files (in the same folder). The 'evaluate_dialog.pl', 'preference_dialog.pl', and 'scripts.pl' files implement the 'Evaluate Query' dialog, 'Preferences' dialog, and 'Scripts' menu, respectively.

The pathname given to the ':-include()' directive may use a special variable XGPINITPATH. This variable expands to the path of the 'initialize_environment.pl' file.

4.8 Creating a Stand-alone Application

Creating a stand-alone application using XGP requires a number of steps.

An application is implemented with XGP by making a copy of XGP.app, call it NewApp.app and modifying it.

The Contents/Resources/English.lproj/Credits.rtf is modified to be the text to display when the 'About' option is selected. The Contents/MacOS/XGP file is renamed 'NewApp'. The Contents/Resources/English.lproj/initialize_envi file is rewritten to create the menus and generally provide the services of NewApp. In simple cases, NewApp can use the default menus. Generally, the Prolog code for NewApp will be in one or more new files, call them New1.pl and New2.pl. These files are in the same folder with 'initialize_environment.pl' and there should be include directives in 'initialize_environment.pl' to include 'New1.pl' and 'New2.pl'.

Several values should be modified in Contents/Info.plist:

- net.sourceforge.xgp.CheckForUpdateOnApplicationOpen is "off",
- CFBundleExecutable is "*New App*",

- CFBundleGetInfoString is "*New App* Version *1.0.0*, Copyright (C) *Year* by *Author*",
- CFBundleHelpBookFolder is "none",
- CFBundleHelpBookName is "none",
- CFBundleIdentifier is "com.*AuthorCompany.New App*",
- CFBundleName is "*New App*",
- CFBundleShortVersionString is "1.0.0",
- NSHumanReadableCopyright is "Copyright (C) *Year*, *Author*. All Rights Reserved."

Several items should be removed from NewApp.app since they are only used by XGP.app. Items to remove from Contents/Resources/English.lproj: DrawWindow.nib, FindPanel.nib, GridPanel.nib, Inspector.nib, ToolPalette.nib, XGP Help, and xgp.icns. Items to remove from Contents/Resources: XGPSuite.scriptSuite, XGPSuite.scriptTerminology.

5 Windows and Documents

There are three kinds of windows in XGP: text documents, graphics documents, and control windows. There are some predicates for working with text documents and windows in general.

5.1 wcreate/6, wcreate/7

Templates

```
wcreate(+Window, +Top, +Left, +Depth, +Width, +Visibility)
wcreate(+Window, _unused, +Top, +Left, +Depth, +Width, +Visibility)
```

Description

`wcreate(Window, Top, Left, Depth, Width, Visibility)` succeeds if a text document can be created as named by *Window* The new document window's top left corner is at (*Top*,*Left*) and its size is (*Depth*,*Width*). It is visible after creation if Visibility is 1.

`wcreate(Window, _unused, Top, Left, Depth, Width, Visibility)` succeeds if `wcreate(Window, Top, Left, Depth, Width, Visibility)` succeeds. This is present for compatibility with LPA MacProlog32.

Window	atom: name of a control window, graphics, or text document
Top	number: specifies the top of the window.
Left	number: specifies the left of the window.
Depth	number: specifies the depth of the window.
Width	number: specifies the width of the window.

5.2 wkill/1, whide/1, wshow/1, wfront/1

Templates

```
wkill(+Window)
whide(+Window)
wshow(+Window)
wfront(+Window)
```

Description

`wkill(Window)` always succeeds. As a side-effect its evaluation will kill, or close, any control window, graphics, or text document named *Window*.

`whide(Window)` always succeeds. As a side-effect its evaluation will hide any control window, graphics, or text document named by *Window*. The window or document still exists, but is not displayed to the user.

`wshow(Window)` always succeeds. As a side-effect its evaluation will show any control window, graphics, or text document named by *Window*. The window or document must already exist.

`wfront(Window)` always succeeds. As a side-effect its evaluation will bring any control window, graphics, or text document named by *Window* to the 'front' of the windows. The window or document must already exist.

Window	atom: name of a control window, graphics, or text document

5.3 is_win/2

Templates

```
is_win(+Window, ?Type)
```

Description

`is_win(Window, Type)` succeeds if there exists a control window, graphics document, or text document named by *Window* with *Type*.

Window	atom: name of a control window, graphics, or text document
Type	atom or variable: unifies with `document`, `control_window`, or `graphics`.

5.4 centered/4, centred/4

Templates

```
centered(-CenterTop, -CenterLeft, +Depth, +Width)
centred(-CenterTop, -CenterLeft, +Depth, +Width)
```

Description

`centered(CenterTop, CenterLeft, Depth, Width)` succeeds if the *CenterTop* and *CenterLeft* values place a window of size *Depth* and *Width* in the center of the screen.

CenterTop	variable: unifies with number for the center top of a window of size *Depth* and *Width*.
CenterLeft	variable: unifies with number for the center left of a window of size *Depth* and *Width*.
Depth	number: specifies the depth of the window to be centered.
Width	number: specifies the width of the window to be centered..

5.5 screen/2

Templates

```
screen(-Depth, -Width)
```

Description

`screen(Depth, Width)` succeeds if the main screen display has size *Depth* and *Width*.

Depth	variable or number: unifies with the depth of the screen in pixels.
Width	variable or number: unifies with the width of the screen in pixels.

5.6 cursor/3

Templates

```
cursor(+Window, +Start, +End)
cursor(+Window, -Start, -End)
```

Description

`cursor(Window, Start, End)` always succeeds. If *Start* and *End* are ground, then evaluation of this query has a side-effect of setting the text cursor for *Window* to (*Start*, *End*). If Start and End are unbound, then they unify with the current cursor position for *Window*.

Window	atom: name of a control window, graphics, or text document
Start	number or variable: number specifies the start of the selection range, variable unifies with start of the selection range
Width	variable or number: unifies with the width of the screen in pixels.

5.7 scroll_to_cursor/1

Templates

```
scroll_to_cursor(+Window)
```

Description

`scroll_to_cursor(Window)` always succeeds. Evaluation of this query has a side-effect of scrolling *Window* to its current selection range.

Window	atom: name of a control window, graphics, or text document

5.8 actdeact/2

Templates

```
actdeact(+Window, +Functor)
actdeact(+Window, -Functor)
actdeact(-Window, +Functor)
actdeact(-Window, -Functor)
```

Description

`actdeact(Window, Functor)` always succeeds. It has different side-effect semantics for each possibility of *Window* and *Functor* being ground or variable.

The activation handler predicate for text or graphics *Window* is *Functor*/2. This procedure is used to set or get an activation handler.

For `actdeact(+Window, +Functor)` where *Window* is not a variable and *Functor* is not a variable, this predicate sets the activation for *Window* to *Functor* . If the *Functor* is 'true', then the activation handler is ignored.

For `actdeact(+Window, -Functor)` where *Window* is not a variable and *Functor* *is* a variable, this predicate gets the activation handler for *Window* and unifies its functor with *Functor*.

For `actdeact(-Window, +Functor)` where *Window* *is* a variable and *Functor* is not a variable, this predicate sets the default activation handler for all windows to *Functor*.

For `actdeact(-Window, -Functor)` where *Window* is a variable and *Functor* is a variable, then this predicate gets the default activation handler for all windows and unifies its functor with *Functor* (the default default activation handler is 'true').

The activation handler is invoked by:

```
Functor(Activation, Window)
```

where Activation is: 'activate', 'deactivate', or 'close'. `Functor(activate, Window)` is invoked when *`Window`* becomes the main window. `Functor(deactivate, Window)` is invoked when *`Window`* resigns being the main window. `Functor(close, Window)` is invoked when *`Window`* is closed.

Caution: do not have the evaluation of *`Functor`* cause a text or graphics window to activate, deactivate, or close. This could lead to an infinite recursion.

`Window`	atom or variable: name of a control window, graphics, or text document
`Functor`	atom or variable: atom specifies the functor of the arity-2 predicate to evaluate when *`Window`* activates, deactivates, or closes.

5.9 date/3

Templates

```
date(?Day, ?Month, ?Year)
```

Description

`date(Day, Month, Year)` succeeds if *`Day`* unifies with the day of the time of evaluation, *`Month`* with the month, and *`Year`* with the year.

`Day`	variable or number: day of the current date.
`Month`	variable or number: month of the current date.
`Year`	variable or number: year of the current date.

5.10 time/3

Templates

```
time(?Hour, ?Minute, ?Second)
```

Description

`time(Hour, Minute, Second)` succeeds if *`Hour`* unifies with the hour of the time of evaluation, *`Minute`* with the minute, and *`Second`* with the second.

`Hour`	variable or number: hour of the current time.
`Minute`	variable or number: minute of the current time.
`Second`	variable or number: second of the current time.

6 Menus

There are several predicates in XGP that provide support for menus. (These predicates are modeled after the LPA MacProlog32 system's menu predicates.)

There is always a menu named 'XGP' in the main menu bar. This is the application or "apple" menu.

6.1 `install_menu/2`, `install_menu/3`

Templates

```
install_menu(+Title, +Items)
install_menu(+Title, +Items, +Attachment)
```

Description

`install_menu(Title, Items)` installs a menu named *`Title`* into the main menu of the XGP application with items named by *`Items`*. When the user selects one of the items (`Item`) of the menu, then the predicate `Title(Item)` is evaluated. (`install_menu(Title, Items)` is equivalent to `install_menu(Title, Items, pulldown)`.)

`install_menu(Title, Items, Attachment)` installs a menu *`Title`* with items named by *`Items`* *attaching* it as specified by *`Attachment`*. There are three kinds of attachment:

- pulldown: a pulldown menu is attached the main menu of the XGP application. This is the default attachment.
- popup: a popup menu is not attached to another menu; it may be displayed in response to a user interaction such as control-click.
- submenu: a submenu is attached to another menu as specified by `Menu(Item)`, where *`Item`* is the name to be used in *`Menu`* to display the submenu.

An element of the *`Items`* list is either an atom that specifies a menu item or it is an arity 1 term that defines a submenu where the functor of the term is the name of the menu item that invokes the submenu and the argument of the term is a list of items that define the submenu.

An atom definition for an item specifies the name of the item, a key shortcut (optional), and an alternative value (optional). The possible formats of this atom are: $nameAndKey => value$ or $nameAndKey$. The $nameAndKey$ format is: $name[/key[/mod_1]...]$, where key is any single character keystroke and mod_i is one of 'shift', 'option' (or 'alternate'), and 'control'. (The 'Command' key modifier is always assumed.)

For example, to specify a menu item that is named "foo", has a shortcut of "option-command-f" and has 'foo' as its value, use: `'foo/f/option'`.

For a menu item that is named "Quit", has a key shortcut of "command-q" , and value of "terminate" use: `'Quit/q=>terminate'`.

`Title`	:atom, name of the menu.
`Items`	:list of $nameAndKey$ terms, as discussed above. Each term defines an item in the menu being installed.
`Attachment`	:atom or structure term, as discussed above. This term specifies to where the menu being installed is to be attached.

6.2 extend_menu/2

Templates

```
extend_menu(+Title, +Items)
```

Description

`extend_menu(Title, Items)` succeeds if there exists a menu named *Title*. The intended side-effect of evaluation extends an existing menu named *Title* with additional *Items*.

Title	:atom, name of the menu.
Items	:list of *nameAndKey* terms, as discussed above. Each term defines an item in the menu being extended.

6.3 kill_menu/1, is_menu/1, disable_menu/1, enable_menu/1

Templates

```
kill_menu(+Title)
is_menu(+Title)
disable_menu(+Title)
enable_menu(+Title)
```

Description

`kill_menu(Title)` always succeeds. The intended side-effect kills (or removes) the definition of an existing menu named *Title* and removes its parent item from its parent menu (if any).

`is_menu(Title)` succeeds if there is a menu defined named *Title*.

`disable_menu(Title)` succeeds if there is a menu defined named *Title*. The intended side-effect "disables" menu named *Title* so that it *can not* be selected or activated by the user.

`enable_menu(Title)` succeeds if there is a menu defined named *Title*. The intended side-effect "enables" menu named *Title* so that it *can* be selected or activated by the user.

Title	:atom, name of the menu.

6.4 get_items/2

Templates

```
get_items(+Title, -Items)
```

Description

`get_items(Title, Items)` succeeds if there exists a menu named *Title* and unifies *Items* with the definitions of the items in the menu.

Title	:atom, name of the menu.
Items	:list of *nameAndKey* terms, as discussed above. Each term defines an item in the menu.

6.5 is_item/2, delete_item/2, enable_item/2, disable_item/2

Templates

```
is_item(+Title, ?Item)
delete_item(+Title, +Item)
enable_item(+Title, +Item)
disable_item(+Title, +Item)
```

Description

`is_item(Title, Items)` succeeds when *Item* is the name of an item in a menu named *Title*.

`delete_item(Title, Item)` succeeds when *Item* is the name of an item in a menu named *Title*. The intended side-effect deletes *Item* from the definition of the *Title* menu.

`enable_item(Title, Item)` succeeds when *Item* is the name of an item in a menu named *Title*. The intended side-effect "enables" *Item* in the definition of the *Title* menu. This causes *Item* to be selectable.

`disable_item(Title, Item)` succeeds when *Item* is the name of an item in a menu named *Title*. The intended side-effect "disables" *Item* in the definition of the *Title* menu. This causes *Item* to not be selectable.

Title	:atom, name of the menu.
Item	:atom or variable, name of an item in the specified menu.

6.6 mark_item/2, unmark_item/2, toggle_item/2, marked_item/2

Templates

```
mark_item(+Title, +Item)
unmark_item(+Title, +Item)
toggle_item(+Title, +Item)
marked_item(+Title, +Item)
```

Description

`mark_item(Title, Item)` succeeds when *Item* is the name of an item in a menu named *Title*. The intended side-effect "marks" *Item* in the definition of the *Title* menu. This causes *Item* to be displayed *with* a system-defined marking (such as a check mark).

`unmark_item(Title, Item)` succeeds when *Item* is the name of an item in a menu named *Title*. The intended side-effect "unmarks" *Item* in the definition of the *Title* menu. This causes *Item* to be displayed *without* a system-defined marking (such as a check mark).

`toggle_item(Title, Item)` succeeds when *Item* is the name of an item in a menu named *Title*. The intended side-effect toggles the marking of *Item* in the definition of the *Title* menu between "marked" and "unmarked".

`marked_item(Title, Item)` succeeds when *Item* is the name of a marked item in a menu named *Title*.

Title	:atom, name of the menu.
Item	:atom or variable, name of an item in the specified menu.

6.7 rename_item/2

Templates

```
rename_item(+Title, +OldItem, +NewItem)
```

Description

`rename_item(Title, OldItem, NewItem)` succeeds if there exists a menu named *Title* and unifies *OldItem* with the definition of an item in the menu. The intended side-effect renames *OldItem* to *NewItem*.

Title	:atom, name of the menu.
Items	:list of *nameAndKey* terms, as discussed above. Each term defines an item in the menu.

7 Predefined Dialogs

There are several predicates in XGP that provide basic dialogs for interacting with users from prolog programs. These predicates are modeled after the LPA MacProlog32 system's dialog predicates.

7.1 banner/2, banner/4

Templates

```
banner(+Call, +Message)
banner(+Call, +Message, +Top, +Left)
```

Description

`banner(Call, Message, Top, Left )` succeeds if *`Call`* succeeds. The intended side-effect presents a dialog box containing *`Message`*. The dialog box is positioned with its top left corner positioned at *`Top`* and *`Left`*

`banner(Call, Message)` is similar to banner/4. Its dialog box is centered in the screen.

`Call`	:term representing the goal to be executed
`Message`	:a list of terms, the message to be displayed.
`Top`	:integer, top of the banner.
`Left`	:integer, left side of the banner.

7.2 errormessage/1, warning/1, message/1

Templates

```
errormessage(+Message)
warning(+Message)
message(+Message)
```

Description

`errormessage(Message)` always throws an exception. The intended side-effect presents a dialog box containing *`Message`*, then throws the `xgp_stop` exception when the "Stop" button of the dialog is clicked.

`warning(Message)` succeeds if the user clicks "Continue" and throws the `xgp_stop` exception if the "Stop" button of the dialog is clicked. The intended side-effect presents displays *`Message`* in a dialog with "Continue" and "Stop" buttons.

`message(Message)` always succeeds. It finishes evaluation when the user clicks the "Continue" button of the dialog. The intended side-effect presents displays *`Message`* in a dialog with a "Continue" button.

`Message`	:a list of terms, the message to be displayed.

7.3 scroll_menu/4, scroll_menu/7, scroll_menu/8

Templates

```
scroll_menu(+Prompt, +MenuList, +Preselected, +Selected)
scroll_menu(+Prompt, +MenuList, +Preselected, +Selected, +Font, +Size, +Style)
scroll_menu(+Prompt, +MenuList, +Preselected, +Selected, +Font, +Size, +Style, +Color)
```

Description

`scroll_menu(Prompt, MenuList, Preselected, Selected)` succeeds if the user clicks on the "OK" button and *`Selected`* unifies with the items in *`MenuList`* that the user selected, it fails if the user selects "Cancel". The intended side-effect presents a dialog containing display the *`Prompt`*, a scrolling menu selection of the items in *`MenuList`*, and "OK" and "Cancel" buttons. The items in *`Preselected`* that are in *`MenuList`* are displayed as "selected" in the scrolling menu.

`scroll_menu(Prompt, MenuList, Preselected, Selected, Font, Size, Style)` is similar to the `scroll_menu/4` predicate. The *`Font`*, *`Size`*, and *`Style`* terms specify the appearance the text in the displayed dialog.

`scroll_menu(Prompt, MenuList, Preselected, Selected, Font, Size, Style, Color)` is similar to the `scroll_menu/7` predicate. The *`Color`* term specifies the color of the text in the displayed dialog.

`Prompt`	list of terms: the concatenation of the terms in this list, with a space between consecutive terms, is the string to display in the interaction dialog.
`MenuList`	list of atoms: these atoms are placed in the given order in the scrolling menu of the interaction dialog.
`Preselected`	list of atoms: these atoms are selected in the scrolling menu of the interaction dialog.
`Selected`	variable or list of atoms: unifies with the atoms that are selected in the scrolling menu of the interaction dialog when the "OK" button is clicked.
`Font`	atom: name of the font of text to use in the prompt and scrolling menu of the interaction dialog.
`Size`	integer: size in "points" of the font of text to use in the prompt and scrolling menu of the interaction dialog.
`Style`	integer: style number of the font of text to use in the prompt and scrolling menu of the interaction dialog. 0 is plain, 1 is bold, 2 is italic (others...)
`Color`	atom: color of the font of text to use in the prompt and scrolling menu of the interaction dialog. 'red', 'blue', 'green', 'yellow', 'black' (others...)

7.4 yesno/1

Templates

```
yesno(+Message)
```

Description

`yesno(Message)` succeeds if the user clicks "Yes" and fails if the "No" button of the dialog is clicked. The intended side-effect display *`Message`* in a dialog with "Yes" and "No" buttons.

`Message`	:a list of terms, the message to be displayed.

7.5 ask/2

Templates

```
ask(+Message, -Input)
```

Description

`ask(Message, Input)` succeeds if the user clicks "OK" and *`Input`* unifies with list of tokens created by parsing the input by the user and fails if the "Cancel" button of the dialog is clicked. The intended side-effect display *`Message`* in a dialog with "OK" and "Cancel" buttons and a text edit field into which the user can enter text.

`Message`	:a list of terms, the message to be displayed.
`Input`	:a list of token terms, the result of parsing the text entered by the user into the text edit field of the dialog.

7.6 prompt_read/2, prompt_read/3

Templates

```
prompt_read(+Message, -Result)
prompt_read(+Message, -Result, +Type)
```

Description

`prompt_read(Message, Result)` succeeds if the user clicks "OK" and *`Result`* unifies with Prolog term created by parsing the input by the user and fails if the "Cancel" button of the dialog is clicked. The intended side-effect display *`Message`* in a dialog with "OK" and "Cancel" buttons and a text edit field into which the user can enter text.

`prompt_read(Message, Result, term)` is interpreted the same as `prompt_read(Message, Result)` .
`prompt_read(Message, Result, tokens)` is interpreted the same as `ask(Message, Result)` .

`Message`	:a list of terms, the message to be displayed.
`Result`	:a term, the result of parsing the text as Prolog source entered by the user into the text edit field of the dialog.
`Type`	:atom, either `term` or `tokens`. This specifies how the text input is to be parsed to produce

8 Custom Dialogs

There are two predicates provided for building custom dialogs,`dialog/7` and `dialog/8`. These are based on the dialog item format descriptors described below.

8.1 Predicates

8.1.1 `dialog/7`, `dialog/8`, `mdialog/6`

Templates

```
dialog(+Title, +Top, +Left, +Depth, +Width, +Items, -Btn)
dialog(+Title, +Top, +Left, +Depth, +Width, +Items, -Btn, +Goal)
mdialog(+Top, +Left, +Depth, +Width, +Items, -Btn)
```

Description

`dialog(Title, Top, Left, Depth, Width, Items, Btn)` and `dialog(Title, Top, Left, Depth, Width, Items, Btn, Goal)` display a modeless dialog defined by the *`Items`* list. The control items defined by *`Items`* are implicitly numbered 1 to N and are referred to by this number. The first two items are usually buttons (e.g. 'OK' and 'Cancel'). If the first item is a button, then the enter/return key will select it. If the second item is a button, then selecting it causes the dialog predicate to fail. Selecting any other button item causes the dialog predicate to succeed, and the number of the selected button item is returned in *`Btn`*.

`mdialog(Title, Top, Left, Depth, Width, Items, Btn)` is a wrapper for `dialog(Title, Top, Left, Depth, Width, Items, Btn)` with a default name of 'Dialog'. This creates a modeless dialog, although in LPA MacProlog32 it created a modal one.

`Title`	:atom, to be displayed as the title of the dialog window.
`Top, Left, Depth, Width`	:integers (in pixels), specifies the upper left corner and size of the dialog.
`Items`	:list of dialog format descriptor terms, defines the layout and behavior of the dialog window.
`Btn`	:atom or variable, unifies with the name of the button clicked by the user.
`Goal`	:call term, form `Test(a1, a2, ..., ak)` where `Test/(k+2)` is defined by user and is invoked as `Test(DialogName, ButtonID, a1, a2, ..., ak)`.

8.1.2 `getditem/3`, `getditem/7`

Templates

```
getditem(+Title, +ItemNumber, -Value)
getditem(+Title, +ItemNumber, -Type, -Abled, -Setting, -Value, -Rect)
```

Description

`getditem(Title, ItemNumber, Value)` succeeds if there is a dialog named *`Title`*, it has an item with number *`ItemNumber`*, and the value of that item unifies with *`Value`*.

`getditem(Title, ItemNumber, Type, Abled, Setting, Value)` has the same interpretation as `getditem(Title, ItemNumber, Value)` with *Type* unifying with the type of *Value*, *Abled* indicating whether the item is enabled or not, and *Setting* being the state of the item (a special kind of value).

Title	:atom, the title of a dialog window.
ItemNumber	:integer, the number identifying an item of the dialog window.
Value	:term, unifies with the term associated as a value with the *ItemNumber*'th item of the dialog window.
Type	:term, unifies with the type of the *ItemNumber*'th item of the dialog window.
Abled	:atom, either `enabled` or `disabled`, unfiies with the enablement of the *ItemNumber*'th item of the dialog window.
Setting	:atom, either `on` or `off` or `null`, unfiies with the setting of the *ItemNumber*'th item of the dialog window.
Rect	:term of the form `rect(T, L, D, W)`, unifies with the rectangle/box of the display of the *ItemNumber*'th item of the dialog window.

8.1.3 `setditem/3`

Templates

```
setditem(+Title, +ItemNumber, -Value)
```

Description

`setditem(Title, ItemNumber, NewValue)` succeeds if there is a dialog named *Title*, it has an item with number *ItemNumber*. The intended side-effect sets the value of the item to *NewValue*.

The format of the *NewValue* term varies depending on the type of the *ItemNumber*'th item. Generally, these are the same forms as used for cw_get_item/3. In addition, for 'menu' items the *NewValue* may be a list of two items, `[Menu, Preselected]`. *Menu* is a list of items that becomes the new menu items for the *ItemNumber*'th item and *Preselected* is a list of items in *Menu* that are selected. Also, for 'text'/'edit' items the *NewValue* may be `replace(Location, Length, Text)`, where *Location* and *Length* define a range ((-1, 0) indicating the entire item) and *Text* is the new text that is to replace the text currently in that range.

Title	:atom, the title of a dialog window.
ItemNumber	:integer, the number identifying an item of the dialog window.
NewValue	:term, sets the term associated as a value with the *ItemNumber*'th item of the dialog window. The format of this term is defined above.

8.2 Custom Dialog Format Descriptors

8.2.1 `button/6`

Templates

```
button(+T, +L, +D, +W, +Label, +KeyEquivalent)
```

Description

`button(T, L, D, W, Label, KeyEquivalent)` specifies a button control at rectangle *T, L, D, W* labeled *Label*. The *KeyEquivalent* identifies a key that will select/activate this button. The *KeyEquivalent* can be '' meaning no key.

8.2.2 check/7

Templates

```
check(+T, +L, +D, +W, +Label, +InitValue, -FinalValue )
```

Description

`check(T, L, D, W, Label, InitValue, FinalValue)` specifies a check (toggle) button control at rectangle *T, L, D, W* labeled *Label*. *InitValue* is either 'on' or 'off'. *FinalValue* is a variable that will be bound to 'on' or 'off'.

8.2.3 color/7

Templates

```
color(+T, +L, +D, +W, +Label, +InitValue, -FinalValue )
```

Description

`color(T, L, D, W, Label, InitValue, FinalValue)` specifies a color "well" button control at rectangle *T, L, D, W* labeled *Label*. *InitValue* is any GDL Color Term (section 11.4.1, page 70). *FinalValue* is a variable that will be bound to a GDL Color Term (section 11.4.1, page 70).

8.2.4 radio/8

Templates

```
radio(+T, +L, +D, +W, +Preselected, -Selected, +RowsPerColumn )
```

Description

`radio(T, L, D, W, Items, Preselected, Selected, RowsPerColumn)` specifies a matrix of the given number of *RowsPerColumn* of radio button controls within the given rectangle with labels from *Items*, the first column taking its labels in top down order first from *Items*, then the second column, continuing until *Items* is exhausted. *RowsPerColumn* is an integer. *Selected* must be a variable, it will be bound to the label of the radio button in the matrix that is 'on'. (This is *not* the LPA format.)

8.2.5 text/5, text/8, text/9

Templates

```
text(+T, +L, +D, +W, +Text)
text(+T, +L, +D, +W, +Text, +Font, +Size, +Style)
text(+T, +L, +D, +W, +Text, +Font, +Size, +Style, +Color)
```

Description

`text(T, L, D, W, Text)` specifies a text field control with the given rectangle. Text indicates the value of the field (specified below).

`text(T, L, D, W, Text, Font, Size, Style)` is similar to `text(T, L, D, W, Text)` where *Font*, *Size*, and *Style* specify the appearance of the text. *Font* is an atom that names a 'font family' (such as 'Times'). *Size* is an integer giving the point size. *Style* is an integer specifying the style (specified below).

`text(T, L, D, W, Text, Font, Size, Style)` is similar to `text(T, L, D, W, Text, Font, Size, Style, Color)` where *`Color`* specifies the color of the text (color specification below).

Text formats:

`Value is atom`	*`Value`* is written
`write(ITerm)`	*`ITerm`* is written as atom
`writeq(ITerm)`	*`ITerm`* is written as atom with quoting
`writeq(ITerm, CommaVariablePairs)`	*`ITerm`* is written as atom with quoting with variables named according to *`CommaVariablePairs`*. E.g. `CommaVariablePairs = [('X', _13), ('Y', _32)]`
`wseq(TermList)`	*`TermList`* is written as atom with spaces between terms.
`bytes(ByteList)`	*`ByteList`* is a list of integer codes of characters, the corresponding characters are written as an atom.
`tokens(TokenList)`	*`TokenList`* is a list of gprolog token specifiers. These are "stripped" and written with intervening spaces as a single atom.
`words(Words)`	*`Words`* is a list. It is written with spaces between items.

Style numbers:

`0`	normal
`1`	bold
`2`	italic
`4`	underline
`8`	small caps
`16`	other

Color names:

- white
- black
- red
- blue
- yellow
- green
- cyan
- magenta

8.2.6 scrolltext/5, scrolltext/8, scrolltext/9

Templates

```
scrolltext(+T, +L, +D, +W, +Text)
scrolltext(+T, +L, +D, +W, +Text, +Font, +Size, +Style)
scrolltext(+T, +L, +D, +W, +Text, +Font, +Size, +Style, +Color)
```

Description

These format descriptors specify a scrolling text field. The arguments have the same interpretation as for the 'text' descriptors.

8.2.7 edit/6, edit/9, edit/10

Templates

```
edit(+T, +L, +D, +W, +InitValue, -FinalValue)
edit(+T, +L, +D, +W, +InitValue, -FinalValue, +Font, +Size, +Style)
edit(+T, +L, +D, +W, +InitValue, -FinalValue, +Font, +Size, +Style, +Color)
```

Description

Same interpretation as the 'text' items, but the *InitValue* is the initial value of the text field and *FinalValue* must be a nonground term that is bound to the final value of the text field. *FinalValue* may have one of the following forms (*RawValue* is the final text in edit field considered as a single atom).

Text read formats:

`Value is variable`	`RawValue = Value`
`read(FTerm)`	bind *FTerm* to a prolog term read from *RawValue*.
`eread(Fterm, CommaVariablePairs)`	bind *FTerm* to a prolog term read from *RawValue* and bind *CommaVariablePairs* to the name/variable pairs for the variables in *FTerm*.
`bytes(Bytes)`	bind *Bytes* to the list of character codes for *RawValue*.
`words(Words)`	bind *Words* to the list of stripped tokens read from *RawValue*.
`tokens(Tokens)`	bind *Tokens* to the list of tokens read from *RawValue*.

8.2.8 scrolledit/6, scrolledit/9, scrolledit/10

Templates

```
scrolledit(+T, +L, +D, +W, +InitValue, -FinalValue)
scrolledit(+T, +L, +D, +W, +InitValue, -FinalValue, +Font, +Size, +Style)
scrolledit(+T, +L, +D, +W, +InitValue, -FinalValue, +Font, +Size, +Style, +Color)
```

Description

Same interpretation as the 'edit' items, but the control item has scrolling.

8.2.9 menu/6, menu/9, menu/10

Templates

```
menu(+T, +L, +D, +W, +Items, +Preselected, -Selected)
menu(+T, +L, +D, +W, +Items, +Preselected, -Selected +Font, +Size, +Style)
menu(+T, +L, +D, +W, +Items, +Preselected, -Selected, +Font, +Size, +Style, +Color)
```

Description

`menu(T, L, D, W, Items, Preselected, Selected )` specifies a scrolling menu control item in the dialog with the given rectangle. The menu has the given *Items* in it, with the items identified in *Preselected* highlighted (selected). If *Preselected* is not a list, then only a single item may be selected in the menu. If *Preselected* is a list, then the menu control allow multiple selections. The *Selected* argument must be a variable and it will be bound to the items selected.

The menu may be 'simple' or hierarchical. If all of the *Items* are atoms, then the menu is simple. A hierarchical item is a structure with a single argument that is a list, where the functor of the structure is the name of the item and the argument is a list of sub-items. A hierarchical item is identified in *Preselected* and *Selected* by a nesting of item structures.

For example:

```
menu(10,20, 100, 300, [a([s1, s2]), b, c([t1([p1, p2]), t2)], [c([t1([p2])])], Selected
)
```

This defines a hierarchical menu with the top level containing [a, b, c]. The `a` item has a submenu of `[s1, s2]`. The `b` item has no submenu. The `c` item has a submenu of `[t1, t2]`. The `t1` item has a submenu of `[p1, p2]`. The preselected item is `p2`.

`menu(T, L, D, W, Items, Preselected, Selected, Font, Size, Style )` and `menu(T, L, D, W, Items, Preselected, Selected, Font, Size, Style, Color )` specify the same control as `menu(T, L, D, W, Items, Preselected, Selected )` with the *Font*, *Size*, *Style,*, and *Color* arguments have the same interpretation as in the 'text' item (section 8.2.5, page 37).

8.2.10 popup/8

Templates

```
popup(+T, +L, +D, +W, +Label, +tems, +Preselected, -Selected)
```

Description

`menu(T, L, D, W, Label, Items, Preselected, Selected )` specifies a text field containing *Label* with a popup button control to its right. The text field is just large enough to hold *Label*, the rest of the given rectangle holds the popup control. The popup controls menu contains *Items* with the item named by *Preselected* as the initial selection. The *Selected* argument must be a variable and it will be bound to the final selection.

8.2.11 table/8

Templates

```
table(+T, +L, +D, +W, +ColumnNames, +Rows, +RowStartIndex, -SelectedRowIndices)
```

Description

`table(T, L, D, W, ColumnNames, Rows, RowStartIndex, SelectedRowIndices)` specifies a tabular data display, with column headers (defined by *`ColumnNames)`* and row numbers on the left side (with initial row index determined by *`RowStartIndex)`*. The row data is given by *`Rows`*, a list of rows where each row is a list of atoms.

If the *`RowStartIndex`* is `none`, then there is no row numbering displayed.

The *`SelectedRowIndices`* is a list of the indices of the rows that are selected (and shown highlighted). The value of the table control is the *`SelectedRowIndices`*. This can be useful for getting information from a table: if the user clicks on the table (selecting a row), then getting the value of the table item returns the index of the selected row.

8.2.12 Example dialog

```
dialog(foo, 100, 100, 200, 300, [button(150, 20, 20, 55, 'OK')], Btn, dialog_test_btn_handler).
```

9 Control Windows

The "control window" predicates provide a way to further customize the behavior of dialogs. These are used to implement the `dialog/7` and `dialog/8` predicates described in "Custom Dialogs" (section 8, page 35). These predicates are more difficult to use than the dialog predicates, but the control window predicates support interface possibilities well beyond what can be done with the dialog predicates.

9.1 Predicates

9.1.1 `cw_create/6`

Templates

```
cw_create(+Window, +T, +L, +D, +W, +Specifications )
```

Description

`cw_create(Window, T, L, D, W, Specifications )` succeeds if there is not already a window named *`Window`*. Its intended side-effect creates a "control window" named *`Window`* with rectangle *`(T, L, D, W)`* and options as defined by *`Specifications`*.

`Window`	:atom, to be displayed as the title of the control window.
`T, L, D, W`	:integers (in pixels), specifies the upper left corner and size of the dialog.
`Specifications`	:list of control window creation specifiers (section 9.2, page 47).

9.1.2 `cw_kill/1`

Templates

```
cw_kill(+Window)
```

Description

`cw_kill(Window )` always succeeds. Its intended side-effect kills (closes) a control window named *`Window`*.

`Window`	atom: name of a control window

9.1.3 `cw_add_item/3`

Templates

```
cw_add_item(+Window, +ItemName, +ItemDescriptor )
```

Description

`cw_add_item(Window, ItemName, ItemDescriptor)` succeeds if there is a window named *`Window`* and *`ItemDescription`* is a valid control item specification. Its intended side-effect adds a control item to the control window named *`Window`* according to the *`ItemDescriptor`* as defined in Format Descriptions (section 9.3, page 48). The first 'edit' or 'scrolledit' item added to a conrol window is set as the "first responder" for that window, so initial key strokes go to that item.

Window	atom or list of two atoms. If *Window* is an atom then it is the name of a control window. If Window is a list of two atoms then the first atom names a control window and the second atom names a tab within that Window.
ItemName	atom, name to be used to identify an item.
ItemDescriptor	term, of the format described Format Descriptions (section 9.3, page 48).

9.1.4 cw_get_item/3, cw_get_item/4

Templates

```
cw_get_item(+Window, +ItemName, -Value )
cw_get_item(+Window, +ItemName, -Value, +TextValueMode )
```

Description

cw_get_item(Window, ItemName, Value, TextValueMode) succeeds if there is a window named *Window* with and item named *ItemName* and *Value* unifies with the value of that item.

If the item is an edit, scrolledit, text, or scrolltext type item, then the *TextValueMode* determines the form of the *Value*. In this case, if the *TextValueMode* is atom then *Value* is an atom. If the *TextValueMode* is chars then the *Value* is a list of "character" atoms.

cw_get_item(+Window, +ItemName, -Value) operates the same as cw_get_item/4 with a *TextValueMode* of atom.

Window	atom or list of two atoms. If *Window* is an atom then it is the name of a control window. If Window is a list of two atoms then the first atom names a control window and the second atom names a tab within that Window.
ItemName	atom, name to be used to identify an item.
Value	term, unifies with the value of the named item. The form of this term depends on the type of the item named by *ItemName*. These forms are the same as discussed for the 'value' arguments of the item descriptors used in cw_add_item/3.
TextValueMode	atom, either atom or chars. This is only applied when the type of the item is edit, scrolledit, text, or scrolltext. If atom then Value is an atom. If chars then the Value is a list of "character" atoms.

9.1.5 cw_get_item_desc/3, cw_get_item_desc/4

Templates

```
cw_get_item_desc(+Window, +ItemName, -Desc )
cw_get_item_desc(+Window, +ItemName, -Desc, +TextValueMode )
```

Description

cw_get_item_desc(Window, ItemName, Desc, TextValueMode) succeeds if there is a window named *Window* with item named *ItemName* with a description that unifies with *Desc*. The Desc term has the same form as the item descriptors used by cw_add_item/3 (section 9.3, page 48).

If the item is an `edit`, `scrolledit`, `text`, or `scrolltext` type item, then the *`TextValueMode`* determines the form of the text value argument in the *`Desc`* term. In this case, if the *`TextValueMode`* is `atom` then the text value is an atom. If the *`TextValueMode`* is `chars` then the text value is a list of "character" atoms.

`cw_get_item_desc(Window, ItemName, Desc)` operates the same as `cw_get_item_desc(Window, ItemName, Desc, TextValueMode)` with a TextValueMode of `atom`.

9.1.6 cw_get_item_type/3

Templates

```
cw_get_item_type(+Window, +ItemName, -Type )
```

Description

`cw_get_item_type(Window, ItemName, Type)` succeeds if there is a window named *`Window`* and *`ItemDescription`* is a valid control item specification and the type of the item unifies with *`Type`*.

`Window`	atom or list of two atoms. If *`Window`* is an atom then it is the name of a control window. If Window is a list of two atoms then the first atom names a control window and the second atom names a tab within that Window.
`ItemName`	atom, name to be used to identify an item.
`Type`	atom, unifies with the type of the named item. The type is one of: `button, radio, check, color, text, scrolltext, edit, scrolledit, menu, popup`, or `table`.

9.1.7 cw_set_item/3

Templates

```
cw_set_item(+Window, +ItemName, +NewValue )
```

Description

`cw_set_item(Window, ItemName, NewValue)` succeeds if there is a window named *`Window`* and *`NewValue`* is a valid value for the item.

The type of the item determines the valid format of the value. The value formats are the same as those used for cw_get_item/3, with some additional options. For `menu` and `popup` items the *`NewValue`* may be a list of two items, `[Menu, Preselected]`. `Menu` is a list of items that becomes the new menu items for the *`ItemName`* item and `Preselected` is a list of items in `Menu` that are selected. Also, for `text`/`edit` items the *`NewValue`* may be `replace(Location, Length, Text)`, where *`Location`* and *`Length`* define a range ((-1, 0) indicating the entire item) and *`Text`* is the new text that is to replace the text currently in that range.

`Window`	atom or list of two atoms. If *`Window`* is an atom then it is the name of a control window. If Window is a list of two atoms then the first atom names a control window and the second atom names a tab within that Window.
`ItemName`	atom, name to be used to identify an item.
`NewValue`	atom, unifies with the type of the named item. The type is one of: `button, radio, check, color, text, scrolltext, edit, scrolledit, menu, popup`, or `table`.

9.1.8 cw_set_item_tooltip/3

Templates

```
cw_set_item_tooltip(+Window, +ItemName, +Tooltip )
```

Description

`cw_set_item_tooltip(Window, ItemName, Tooltip)` succeeds if there is a window named *Window*. Its intended side-effect sets the tooltip displayed when the user holds the mouse cursor over the associated item to *Tooltip*.

Window	atom or list of two atoms. If *Window* is an atom then it is the name of a control window. If Window is a list of two atoms then the first atom names a control window and the second atom names a tab within that Window.
ItemName	atom, name to be used to identify an item.
Tooltip	atom, text for the tooltip.

9.1.9 cw_delete_item/2, cw_del_item/2

Templates

```
cw_delete_item(+Window, +ItemName)
```

Description

`cw_delete_item(Window, ItemName)` succeeds if there is a window named *Window* and an item named *ItemName*. Its intended side-effect removes the item from the window.

Window	atom or list of two atoms. If *Window* is an atom then it is the name of a control window. If Window is a list of two atoms then the first atom names a control window and the second atom names a tab within that Window.
ItemName	atom, name to be used to identify an item.

9.1.10 cw_set_program/3

Templates

```
cw_set_program(+Window, +HandlerType, +HandlerFunctor)
```

Description

`cw_set_program(Window, HandlerType, HandlerFunctor )` succeeds if there is a window named *Window*. Its intended side-effect sets the *HandlerType* handler functor for *Window* to *HandlerFunctor*. When an event of *HandlerType* occurs, then a predicate constructed using *HandlerFunctor* is invoked.

The way *HandlerFunctor* is invoked depends on the *HandlerType*. For HandlerType of `click`, the predicate is `HandlerFunctor(Window, ItemName)`, where *ItemName* is the name of the item that was clicked.

For *HandlerType* of `key`, the predicate (not yet implemented) is `HandlerFunctor(Window, Code, Modifiers)`, where *Code* is the ASCII code of the key pressed and *Modifiers* is an integer indicating which modifier keys were used:

`0`	none
`256`	Command key
`512`	Shift key
`1024`	Caps Lock key
`2048`	Option key

Parameters

`Window`	atom: name of a control window
`HandlerType`	atom, one of `click`, `key`, or `help`. Currently, only `click` is implemented.
`HandlerFunctor`	term, defines the program to be invoked when *`HandlerType`* event occurs.

9.1.11 `cw_await_button/2`

Templates

```
cw_await_button(+Window, ?ButtonName)
```

Description

`cw_await_button(Window, ButtonName)` succeeds if there is a window named *`Window`* and a button named *`ButtonName`* is clicked. Its intended side-effect is to wait until a button is clicked to finish evaluating.

Parameters

`Window`	atom: name of a control window
`Button`	variable/atom, name of the button clicked.

9.2 Creation Specifiers

The control window creation specifiers are used in the cw_create_window/6 predicate. A list of specifiers must have at most one specifier from each category: visibility, goaway/close box, window style, and tabs.

9.2.1 visibility

`visible`	make the window visible
`invisible`	do not make the window

9.2.2 goaway/close box

`goaway`	add a close box (goaway box) control to the window

9.2.3 window style

`dialog`	
`document`	This style of control window is resizable.
`alert`	
`plain`	
`shadowed`	this is for compatibility with LPA only, it is treated as a synonym for plain.

9.2.4 tabs

`tabs(Names)`	creates a "tabbed" window. *`Names`* is a list of atoms that are the names of the tabs.

9.3 Format Descriptors

The control window format descriptors are used in the `cw_add_item/3` predicate. They have essentially the same interpretation as the dialog item format descriptors, with the removal of the "output" final value argument.

The `edit` and `scrolledit` items have a *bezeled* appearance (without an outlining rectangle) when displayed in a *textured* window (such as `dialog`). Otherwise, they are not bezeled and have an enclosing rectangle.

```
button(T, L, D, W, Label, KeyEquivalent )

check(T, L, D, W, Label, InitValue )

color(T, L, D, W, Color)

radio(T, L, D, W, Items, Preselected, RowsPerColumn )

text(T, L, D, W, Text)

scrolltext(T, L, D, W, Text)
scrolltext(T, L, D, W, Text, Font, Size, Style)
scrolltext(T, L, D, W, Text, Font, Size, Style, Color)

edit(T, L, D, W, InitValue)
edit(T, L, D, W, InitValue, Font, Size, Style)
edit(T, L, D, W, InitValue, Font, Size, Style, Color)

scrolledit(T, L, D, W, InitValue)
scrolledit(T, L, D, W, InitValue, Font, Size, Style)
scrolledit(T, L, D, W, InitValue, Font, Size, Style, Color)

menu(T, L, D, W, Items, Preselected )
menu(T, L, D, W, Items, Preselected, Font, Size, Style )
menu(T, L, D, W, Items, Preselected, Font, Size, Style, Color )

popup(T, L, D, W, Label, Items, Preselected )

table(T, L, D, W, ColumnNames, Rows, RowStartIndex, SelectedRowIndices)
```

9.3.1 Example

```
cw_create(foo, 100, 100, 200, 300, [visible, dialog]),
cw_add_item(foo, btn, button(150, 20, 20, 55, 'OK', '')),
cw_set_program(foo, click, dialog_test_btn_handler),
cw_await_button(foo, Btn).
```

9.4 Macros

The cwmacro predicates are simpler ways to work with control windows thanthe basic cw_* predicates. The price for this simplicity is less control over exactly how the control window works.

9.4.1 cwmacro_add_labeled_item/6

Templates

```
cwmacro_add_labeled_item(+Window, +BoxIN, +ItemID, +Label, +RelativeLocation, +BaseItem)
```

Description

`cwmacro_add_labeled_item(Window, BoxIN, ItemID, Label, RelativeLocation, BaseItem)` succeeds if there is a window named *Window*. Its intended side-effect adds a control *BaseItem* labeled with text *Label* to *Window*, where the *Label* and the *BaseItem* fit within *BoxIN*. The *Label* is placed relative to *BaseItem* according to *RelativeLocation*. The *BaseItem* is added to *Window* with name *ItemID*.

There are three defined forms for the *RelativeLocation*: `left`, `left(W)`, and `above`. `left` puts Label to the left of the *BaseItem* and divides *BoxIN* such that the Label portion is just wide enough for the *Label* and the *BaseItem* portion is the rest of *BoxIN*. `left(W)` puts *Label* to the left of *BaseItem* and sets the width of the *BaseItem* portion of *BoxIN* to *W* and the *Label* portion is the rest of *BoxIN*. `above` puts *Label* above the *BaseItem* where the *Label* portion of the *BoxIN* is just deep enough to hold the *Label* and the *BaseItem* portion is the rest of *BoxIN*.

Parameters

Window	atom: name of a control window
BoxIN	term of the form `box(T, L, D, W)`. *T* is the top of the box, *L* is the left side of the box, *D* is the depth, and *W* is the width. This box encloses both the *Label* and *BaseItem*.
ItemID	atom, name of the *BaseItem* in *Window*.
Label	term. Text to be displayed as the label for the *BaseItem*, suitable for the `text` control item.
RelativeLocation	term. One of `left`, `left(W)`, or `above`.
BaseItem	a control window format descriptor.

9.4.2 cwmacro_add_item/s6

Templates

```
cwmacro_add_items (+ItemInfos, +AddItemFunctor, +Window, +Gutter, +FirstEntryBox, -FinalEntryBox)
```

Description

`cwmacro_add_items (ItemInfos, AddItemFunctor, Window, Gutter, FirstEntryBox, FinalEntryBox)` adds items to *`Window`* by applying *`AddItemFunctor`* to each element in ItemInfos. The first item is placed in *`FirstEntryBox`*. The next "box" is created by advancing (vertically) the current "box" by *`Gutter`*.

Each element of ItemInfos is a list of one or more elements. The *`AddItemGoal`* is created from the *`AddItemFunctor`* by:

```
AddItemGoal =..  [AddItemFunctor, Window, CurrentBox | ItemInfo]
```

9.4.3 Example using cwmacro_add_items/6 and cwmacro_add_labeled_item/6

The example_dialog/0 predicate uses `cwmacro_add_items/6`, `example_add_item/3`, and `cwmacro_add_labeled_item/6` to create a control window that has four `edit` boxes, labeled `first`, `second`, `third`, and `fourth`.

```
example_dialog :-
cw_create('Example', 50, 50, 150, 300, [goaway, visible, dialog]),
cwmacro_add_items(
[[first], [second], [third], [fourth]],
example_add_item,
'Example', 5, box(5, 5, 25, 275), _FinalBox).
```

The `example_add_item/3` predicate uses the `Box-175` term to specify that the `edit` item is to fill the right 175 points of `Box`.

```
example_add_item(Window, Box, Label) :-
proper(Label, ProperLabel),
concat([ProperLabel, ':'], DisplayLabel),
cwmacro_add_labeled_item(Window, Box-175, Label, DisplayLabel, left, edit('')).
```

9.4.4 cwmacro_add_labeled_buttons/6

Templates

```
cwmacro_add_labeled_buttons(+Window, +Box, +ButtonDepth, +Label, left-horizontal, +ButtonGroups)
```

Description

`cwmacro_add_labeled_buttons (Window, Box, ButtonDepth, Label, left-horizontal, ButtonGroups)` adds the *`Label`* and buttons defined in *`ButtonGroups`* to *`Window`* according to *`Layout`*. The *`Label`* is placed left of *`ButtonGroups`*. Each button group is placed horizontally, all of the buttons in a group fitting within the width of *`Box`*, each of the buttons the same width, and each button wide enough to hold all of the text in its label if possible. The *`Box`* should be `box(T, L, D, W)`, where *`D`* is unbound. It will be bound on completion to specify the overall depth of the buttons.

Parameters

`Window`	atom: name of a control window
`Box`	term of the form `box(T, L, D, W)`. *`T`* is the top of the box, *`L`* is the left side of the box, *`D`* is the depth, and *`W`* is the width. This box encloses both the *`Label`* and *`ButtonGroups`*. *`D`* is unbound, it is bound to the overall depth.
`ButtonDepth`	integer, depth to be used for all buttons.

Label	term.*Text* to be displayed as the label for the *ButtonGroups*, suitable for the `text` control item.
ButtonGroups	List of items of the form `GeneralID-Name`. A list of the `GeneralID-Name` terms is a "button group". The *GeneralID* is either `ID/Key` or just `ID`. The *ID* is the identifier to use for the button item. The *Key* (if any) is the key shortcut to associate with the button.

10 Graphics Predicates

The XGP graphics services provide utilities for creating a graphics document, creating images within that document, and managing interactions with that document and its defined images. There is a declarative language, XGDL (XGP Graphic Description Language), for describing pictures (XGDL is an adaption of LPA MacProlog32's GDL to the XGP environment.)

There are many predicates absent from XGP but which should be implemented eventually. Among these are ones for getting, deleting, selecting, and changing pictures.

10.1 Fundamental Picture Manipulation Predicates

10.1.1 `add_pic/3`, `chg_pic/3`, `chg_pic/4`, `record_qpic/3`

Templates

```
add_pic(+Document, +PictureName, +Description)
chg_pic(+Document, +PictureName, +Description)
chg_pic(+Document, +PictureName, +Description, +Shift)
record_qpic(+Document, +PictureName, +Description )
```

Description

`add_pic(Document, PictureName, Description)` succeeds if *`Document`* exists. The intended side-effect adds a picture named *`PictureName`* and defined by an XGDL *`Description`* to *`Document`*.

`chg_pic(Document, PictureName, Description)` succeeds if *`Document`* exists. The intended side-effect changes a picture named *`PictureName`* in *`Document`* to be defined by an XGDL *`Description`*.

`chg_pic(Document, PictureName, Description, Shift)` succeeds if *`Document`* exists. The intended side-effect changes a picture named *`PictureName`* in *`Document`* to be defined by an XGDL *`Description`* and shifts the origin of the picture to *`Shift`*.

`record_qpic(Document, PictureName, Description )` is alias for `add_pic(Document, PictureName, Description)` . Eventually it should have a slightly different side effect, where *`Description`* is "compiled" and stored, but not displayed (a separate predicate would cause the display). The compiled form of a graphic, a 'qpic' (originally an abbreviation for "QuickDraw Picture" in MacProlog32), is faster to draw and takes less storage than the graphic as stored by add_pic/3, but cannot be extracted and decomposed into its original definition. This is equivalent to the not-yet-implemented:

```
record_pic(Document, PictureName, Description, (0,0), compiled)
```

In MacProlog32 the Mode was `quickdraw` instead of `compiled`.

`Document`	atom: name of a graphics document
`PictureName`	atom: name of a picture in *`Document`*.
`Description`	term: An XGDL (section 11, page 63) description of a picture.

10.1.2 `del_pic/2`

Templates

```
del_pic(+Document, +Name )
```

Description

`del_pic(Document, PictureName )` succeeds if *PictureName* exists in *Document*. The intended side-effect deletes a picture named *PictureName* from *Document*.

Document	atom: name of a graphics document
PictureName	atom: name of a picture in *Document*.

10.1.3 `del_pic_num/2`

Templates

`del_pic_num(+Document, +Position)`

Description

`del_pic_num(Document, Position)` succeeds if *Position* exists in *Document*. The intended side-effect delets the picture that is at *Position* in the display order of pictures for *Document*..

Document	atom: name of a graphics document
Position	integer: The number indicating the position of a picture in the display ordering of pictures for *Document*.

10.1.4 `del_sels/1`

Templates

`del_sels(+Document)`

Description

`del_sels(Document)` succeeds if *Document* exists. The intended side-effect deletes the selected pictures in *Document*.

Document	atom: name of a graphics document

10.1.5 `del_all/1`

Templates

`del_all(+Document)`

Description

`del_all(Document)` succeeds if *Document* exists. The intended side-effect deletes all of the pictures in *Document*.

Document	atom: name of a graphics document

10.1.6 `get_pic/3`, `get_pic/4`, `get_pic/5`

Templates

```
get_pic(+Document, +PictureName, -Description )
get_pic(+Document, +PictureName, -Description, -Shift )
get_pic(+Document, +PictureName, -Description, -Shift, -Selected)
```

Description

`get_pic(Document, PictureName, Description )` succeeds if *Description* unifies with the XGDL description of *PictureName* in *Document*.

`get_pic(Document, PictureName, Description, Shift )` is similar to `get_pic(Document, PictureName, Description )` , additionally unifying *Shift* with the vertical and horizontal shift applied to the *Description*. These shift values are usually 0.

`get_pic(Document, PictureName, Description, Shift, Selected )` is similar to `get_pic(Document, PictureName, Description, Shifgt )` , additionally unifying *Selected* with 1 or 0 to indicate that the picture is selected or not, respectively.

Parameters

Document	atom: name of a graphics document
PictureName	atom: name of a picture in *Document*.
Description	term or variable: unified with an XGDL (section 11, page 63) description of the picture.
Shift	term or variable: Unified with a "point" structure `(Y, X)`. *Y* is the distance *Description* is shifted vertically. *X* is the horizontal shift.
Selected	integer or variable: Unified with 0 or 1. 1 indicates PictureName is selected in Document.

10.1.7 `get_all_pics/2`

Templates

```
get_all_pics(+Document, -PictureNames )
```

Description

`get_all_pics(Document, PictureNames )` succeeds if *PictureNames* unifies with the list of all of the picture names in *Document*.

Parameters

Document	atom: name of a graphics document
PictureNames	list of atoms or variable: list of names of all of the pictures in *Document*.

10.1.8 `sel_pics/2`, `desel_pics/2`

Templates

```
sel_pics(+Document, +PictureNames )
desel_pics(+Document, +PictureNames )
```

Description

`sel_pics(Document, PictureNames )` succeeds if *Document* exists. Intended side-effect of evaluation selects all of the pictures named in *PictureNames* in *Document*.

`desel_pics(Document, PictureNames )` succeeds if *Document* exists. Intended side-effect of evaluation deselects all of the pictures named in *PictureNames* in *Document*.

Parameters

Document	atom: name of a graphics document
PictureNames	list of atoms or variable: list of names of pictures in *Document*.

10.1.9 `sel_all/1`, `desel_all/1`

Templates

```
sel_all(+Document)
desel_all(+Document)
```

Description

`sel_all(Document)` succeeds if *Document* exists. Intended side-effect of evaluation selects all of the pictures in *Document*.

`desel_all(Document)` succeeds if *Document* exists. Intended side-effect of evaluation deselects all of the pictures in *Document*.

Parameters

Document	atom: name of a graphics document

10.1.10 `get_sel_pics/2`, `get_desel_pics/2`

Templates

```
get_sel_pics(+Document, -PictureNames )
get_desel_pics(+Document, -PictureNames )
```

Description

`get_sel_pics(Document, PictureNames )` succeeds if *Document* exists and *PictureNames* unifies with the names of the selected pictures in that document.

`get_desel_pics(Document, PictureNames )` succeeds if *Document* exists and *PictureNames* unifies with the names of the deselected pictures in that document.

Parameters

Document	atom: name of a graphics document
PictureNames	list of atoms or variable: list of names of pictures selected in *Document*.

10.1.11 rename_pic/3

Templates

```
rename_pic(+Document, +OldName, +NewName )
```

Description

`rename_pic(Document, OldName, NewName)` succeeds if a picture named *OldName* exists in *Document* and there is not a picture named *NewName*. The intended side-effect renames the picture named *OldName* to *NewName* in *Document*.

Document	atom: name of a graphics document
OldName	atom: name of a picture in *Document*.
NewName	atom: name of a picture in *Document*.

10.1.12 shift_pic/3, shift_pics/3

Templates

```
shift_pic(+Document, +PictureName, +Shift) shift_pics(+Document, +PictureNames, +Shift)
```

Description

`shift_pic(Document, PictureName, Shift)` succeeds if a picture named *PictureName* exists in *Document*. The intended side-effect translates the current local origin of the picture named *PictureName* in *Document* by the amount indicated by *Shift*.

`shift_pics(Document, PictureNames, Shift)` succeeds if pictures named *PictureNames* exist in *Document*. The intended side-effect translates the current local origin of these pictures by the amount indicated by *Shift*.

Document	atom: name of a graphics document
PictureName	atom: name of a picture in *Document*.
PictureNames	list of atoms: names of pictures in *Document*.
Shift	term of the form `(Y, X)`: *Y* is the distance the picture name *PictureName* is shifted vertically. *X* is the horizontal shift.

10.1.13 pic_frame/2

Templates

```
pic_frame(+Description, -Frame)
```

Description

`pic_frame(Description, Frame)` succeeds if *Frame* is the rectangle bounding the picture defined by *Description*.¡/TD¿

Description	term: XGDL term defining a picture
Frame	term of the form `box(T, L, D, W)`: bounding rectangle of the picture defined by *Description*.

10.2 Picture Display Ordering Predicates

10.2.1 reverse_pics/1

Templates

```
reverse_pics(+Document)
```

Description

`reverse_pics(+Document)` succeeds if *Document* exists. The intended side-effect reverses the display order of all the pictures in *Document*.

Document	atom: name of a graphics document

10.2.2 bring_to_front/2, send_to_back/2

Templates

```
bring_to_front(+Document, + PictureName )
send_to_back(+Document, + PictureName )
```

Description

`bring_to_front(+Document, + PictureName )` succeeds if *PictureName* exists in *Document*. The intended side-effect brings the picture identified by *PictureName* to the front of the display ordering for *Document*.

`send_to_back(+Document, + PictureName )` succeeds if *PictureName* exists in *Document*. The intended side-effect sends the picture identified by *PictureName* to the back of the display ordering for *Document*.

Document	atom: name of a graphics document
PictureName	atom: name of a picture in *Document*.

10.3 Document Predicates

10.3.1 wgcreate/8

Templates

```
wgcreate(+Document, +T, +L, +D, +W, +MaxX, +MaxY, +Vis )
wgcreate(+Document, +T, +L, +D, +W, _Split, +MaxX, +MaxY, +Vis, _Go )
```

Description

`wgcreate(Document, T, L, D, W, MaxX, MaxY, Vis )` succeeds if *Document* does not exist. The intended side-effect creates a "graphics document" named *Document* with rectangle `(T, L, D, W)` and drawing area size of *MaxX* and *MaxY*.

Document	atom: name of a graphics document
T, L, D, W	integers: rectangle frame of the window. *T* is the 'top', *L* is the 'left', *D* is the 'depth', and *W* is the 'width'.

MaxX, MaxY	integers: size of the drawing area (may be larger than *W* and *D*)
Vis	integer, 0 or 1: specifies visibility of the created window. 0 is invisible, 1 is visible.
_Split	[NOT IMPLEMENTED] integer: previously specified the 'split' of the graphics window between the tools pane and the drawing pane. No tools pane is implemented in XGP. Tools will probably be implemented using a per-window toolbar.
_Go	[NOT IMPLEMENTED] integer, 0 or 1: flag specifying presence of the 'goaway' (close) button.

10.3.2 wsize/5

Templates

```
wsize(+Document, ?T, ?L, ?D, ?W )
```

Description

`wsize(Document, T, L, D, W )` succeeds if *Document* exists. If *T*, *L*, *D*, or *W* are ground, then the intended side-effect sets the corresponding value (top, left, depth, or width) of the frame of the window of the *Document*. If any of these are not ground, then they unify with the current setting for the corresponding value of the frame of the window of the *Document*.

Document	atom: name of a graphics document
T, L, D, W	integers: frame of the window of the document (top, left, depth, and width, respectively).

10.3.3 gmax/3

Templates

```
gmax(+Document, +MaxX, +MaxY )
gmax(+Document, -MaxX, -MaxY )
```

Description

`gmax(Document, MaxY, MaxX )` succeeds if *Document* does not exist. If *MaxX* and *MaxY* are ground, then the intended side-effect sets the drawing area size. If they are not ground, then they unify with the current setting for the drawing area size.

Document	atom: name of a graphics document
MaxX, MaxY	integers: size of the drawing area.

10.3.4 gscroll_to/3

Templates

```
gscroll_to(+Document, +Y, +X )
```

Description

`gscroll_to(Document, Y, X )` succeeds if *Document* does not exist. The intended side-effect scrolls *Document* to make *(Y,X)* the upper left corner of the visible rectangle.

Document	atom: name of a graphics document
Y, X	integers: Coordinates of upper left corner of visible rectangle.

10.3.5 gscroll_by/3

Templates

```
gscroll_by(+Document, +Down, +Across )
```

Description

gscroll_to(Document, Down, Across) succeeds if *Document* does not exist. The intended side-effect scrolls *Document* by *Down, Across*.

Document	atom: name of a graphics document
Down, Across	integers: Amounts to scroll the visible rectangle.

10.3.6 refresh_now/1

Templates

```
refresh_now(+Document)
```

Description

refresh_now(Document) succeeds if *Document* exists. The intended side-effect directs the graphic system to immediately redraw all the pictures in *Document*, instead of waiting for the current query evaluation to complete.

Document	atom: name of a graphics document

10.4 Toolbar and Tools

A graphics document has a toolbar that contains a number of tool icons. One tool is selected at a time. Associated with each tool icon is a tool predicate. Various events in the graphics window cause the tool predicate to be invoked with information about the event. Mouse clicks in the graphics document cause the tool predicate to be invoked with information about the mouse location at the time of the click. Selecting a tool icon (and deselecting the previously selected tool icon) causes the tool predicate to be invoked with a 'selected' (respectively 'deselected') argument.

10.4.1 add_tools/2

Templates

```
add_tools(+ Document, +Tools)
```

Description

add_tools(Document, Tools) succeeds if *Document* exists and *Tools* is a well-formed list of tool definitions. The expected side-effect of this predicate is to add tool definitions to the toolbar for *Document* as defined by *Tools*.

Document	atom: name of a graphics document
Tools	list: terms defining tools for the graphics document. Each term is of the form 'Tool(Description)' where Tool is the name of the tool program to call and Description is an XGDL graphics definition of the tool icon (currently ignored).

10.4.2 del_tools/1, del_tools/2

Templates

```
del_tools(+ Document)
del_tools(+ Document, +ToolNames)
```

Description

`del_tools(Document)` succeeds if *Document* exists. The expected side-effect is to remove all of the tools from the toolbar of *Document*.

`del_tools(Document, Tools)` succeeds if *Document* exists. The expected side-effect is to remove the tools name in the list *Tools* from the toolbar of *Document*.

Document	atom: name of a graphics document
Tools	list: terms naming tools for the graphics document. These are the tools to be deleted from the toolbar).

10.4.3 get_tools/2

Templates

```
get_tools(+ Document, -Toolnames)
```

Description

`get_tools(Document, Toolnames)` succeeds if *Document* exists and *Toolnames* unifies with the list of name of tools in the toolbar for *Document*.

Document	atom: name of a graphics document
Toolnames	list: terms naming tools for the graphics document.

10.4.4 get_tool/2

Templates

```
get_tool(+ Document, -Name)
```

Description

`get_tool(Document, Name)` succeeds if *Document* exists and *Name* unifies with the name of the currently selected tool in the toolbar for *Document*.

Document	atom: name of a graphics document
Name	atom: name of the currently selected tool for the graphics document.

10.4.5 set_tool/2

Templates

```
set_tool(+ Document, +Name)
```

Description

`set_tool(Document, Name)` succeeds if *Document* exists and *Name* is the name of a tool in the toolbar for *Document*. The expected side-effect is to make *Name* the new selected tool for *Document* and to deselect the previously selected tool.

Document	atom: name of a graphics document
Name	atom: name of a tool for the graphics document that is to become selected.

11 XGP Graphic Description Language (XGDL)

The XGP Graphic Description Language (SGDL) has three kinds of terms: elementary pictures, transformers, and lists. A list contains any combination of XGDL terms (including other lists). The elementary picture terms define basic pictures such as a square or an oval, or a line. The transformer terms have a XGDL term argument and may have other arguments. The transformer semantics are applied to the picture defined by the XGDL term argument.

11.1 Elementary Pictures

The XGDL elementary picture terms define basic images.

Many of the picture terms use arguments *T, L, D, and W* to describe a rectangle: *T* is the top side coordinate, *L* is the left side coordinate, *D* is the depth of the rectangle, and *W* is the width of the rectangle.

The unimplemented descriptors of LPA's GDL are:

- copyof,
- icon,
- point,
- qpic,
- resource.

11.1.1 arc/6

Templates

```
arc(T, L, D, W , StartAngle, ArcAngle)
```

Description

`arc(T, L, D, W , StartAngle, ArcAngle)` creates an arc as a segment of the oval that inscribes the rectangle *(T, L, D, W)*. The arc begins at *StartAngle* clockwise from "12 o'clock" and continuing clockwise for *ArcAngle*. (The angles are in degrees.)

11.1.2 box/4, box/6

Templates

```
box(T, L, D, W )
box(T, L, D, W, OvalDepth, OvalWidth)
```

Description

`box(T, L, D, W)` creates a hollow box (rectangle) described by *(T, L, D, W)*.

`box(T, L, D, W, OvalDepth, OvalWidth)` creates a hollow box (rectangle) described by *(T, L, D, W)* with rounded corners described by *OvalDepth, OvalWidth*.

11.1.3 `circle/3`

Templates

```
circle(YCenter, XCenter, Radius)
```

Description

`circle(YCenter, XCenter, Radius)` creates a circle centered at *`(YCenter, XCenter)`* with *`Radius`*.

11.1.4 `circle/3`

Templates

```
circle(YCenter, XCenter, Radius)
```

Description

`circle(YCenter, XCenter, Radius)` creates a hollow circle centered at *`(YCenter, XCenter)`* with *`Radius`*.

11.1.5 `fillbox/4`, `fillbox/6`

Templates

```
fillbox(T, L, D, W )
fillbox(T, L, D, W, OvalDepth, OvalWidth)
```

Description

`fillbox(T, L, D, W)` creates a filled box (rectangle) described by *`(T, L, D, W)`*.

`fillbox(T, L, D, W, OvalDepth, OvalWidth)` creates a filled box (rectangle) described by *`(T, L, D, W)`* with rounded corners described by *`OvalDepth, OvalWidth`*.

11.1.6 `fillcircle/3`

Templates

```
fillcircle(YCenter, XCenter, Radius)
```

Description

`fillcircle(YCenter, XCenter, Radius)` creates a filled circle centered at *`(YCenter, XCenter)`* with *`Radius`*.

11.1.7 `filloval/4`

Templates

```
filloval(T, L, D, W )
```

Description

`filloval(T, L, D, W)` creates a filled oval inscribed in *`(T, L, D, W)`*.

11.1.8 fillpoly/1

Templates

```
fillpoly(ListOfPoints )
```

Description

`fillpoly(ListOfPoints)` creates a filled polygon defined by the sequence of points in *`ListOfPoints`*. Points are in the form *`(Y, X)`*. The set of lines will create a closed polygon by adding a line from the last point of the list back to the first point.

Example: `fillpoly([(50, 25), (50, 75), (80, 75), (80, 25)])` describes the same picture as `fillbox(50, 25, 30, 50)`.

11.1.9 fillsquare/1

Templates

```
fillsquare(YCenter, XCenter, Size )
```

Description

`fillsquare(YCenter, XCenter, Size)` creates a filled square centered at *`(YCenter, XCenter)`* with sides of length *`Size`*.

11.1.10 grid/3

Templates

```
grid(YIncrement, XIncrement, box(T, L, D, W))
```

Description

`grid(YIncrement, XIncrement, box(T, L, D, W))` creates a grid of lines inscribed in `box(T, L, D, W)` with horizontal lines *`YIncrement`* apart and vertical lines *`XIncrement`* apart. The first grid point is *`(T, L)`*. *`D`* should be an integer multiple of *`YIncrement`* and *`W`* an integer multiple of *`XIncrement`*.

11.1.11 line/2

Templates

```
line (P1, P2)
```

Description

`line (P1, P2)` creates a line from point *`P1`* to point *`P2`*.

11.1.12 lines/1

Templates

```
lines(ListOfPoints)
```

Description

`lines(ListOfPoints)` creates joined line segments defined by the sequence of points in *`ListOfPoints`*. Points are in the form `(Y, X)`.

11.1.13 `oval/4`

Templates

```
oval(T, L, D, W )
```

Description

`oval(T, L, D, W)` creates a hollow oval inscribed in *`(T, L, D, W)`*.

11.1.14 `picture/6`

Templates

```
picture(T, L, D, W, CompositionMode, Path)
```

Description

`picture(T, L, D, W, CompositionMode, Path)` creates a picture from source file at *`Path`*. Compose using *`CompositionMode`*. The *`Path`* can be absolute or relative path (can use ‘ ’ homedir convention) of an image file with an appropriate suffix (.tiff, .gif, .jpg, .png, etc.)

The *`CompositionMode`* is one of: clear, copy, source_over, source_in, source_out, source_atop, destination_over, destination_in, destination_out, destination_atop, xor, plus_darker, highlight, plus_lighter.

11.1.15 `pixels/12`

Templates

```
pixels(T, L, D, W, Composition, PixelsWide, PixelsHigh, BitsPerSample, SamplesPerPixel,
HasAlpha, ColorSpaceName, Planes)
```

Description

`pixels(T, L, D, W, Composition, PixelsWide, PixelsHigh, BitsPerSample, SamplesPerPixel, HasAlpha, ColorSpaceName, Planes)` creates a picture from a list of pixels using the numbers in *`Planes`* to define each pixel to be displayed. The interpretation of *`Planes`* is determined by *`PixelsWide`*, *`PixelsHigh`*, *`BitsPerSample`*, *`SamplesPerPixel`*, *`HasAlpha`*, and *`ColorSpaceName`*.

`Composition`	determines how the pixels image is overlayed onto another image (if any). The values are given at —(section 11.1.14, page 66).
`BitsPerSample`	number of bits used to specify 1 pixel in a single component of the data. All components are assumed to have the same bits per sample. Should be one of these values: 1, 2, 4, 8, 12, or 16.
`SamplesPerPixel`	number of data components. It includes both color components and the coverage component (alpha), if present. Meaningful values range from 1 through 5. An image with cyan, magenta, yellow, and black (CMYK) color components plus a coverage component would have an spp of 5; a grayscale image that lacks a coverage component would have an spp of 1.

HasAlpha	`1` (true) if one of the components counted in the number of samples per pixel (spp) is a coverage component, and `0` (false) if there is no coverage component. If `1` (true), the color components in the bitmap data must be premultiplied with their coverage component.
ColorSpaceName	`calibrated_white`, `calibrated_black`, `calibrated_RGB`, `device_white`, `device_black`, `device_RGB`, `device_CMYK`, `named`, or `custom`. This determines the color space used to interpret the sample values.
Planes	Either a list of planes (implies IsPlanar = YES), or a single plane (implies IsPlanar = NO). Each plane is a list of float values representing color and alpha values. The actual bits in a sample *ActualSampleValue* are calculated from the *InputSampleValue* by `ActualSampleValue = round(((2**BitsPerSample)-1) * InputSampleValue`.

11.1.16 `pointer/2`, `pointer/3`

Templates

```
pointer(P1, P2)
pointer(P1, P2, Arrow)
```

Description

`pointer(P1, P2, Arrow)` creates a line with one or two arrowheads. Arrow defaults to `right` for `pointer(P1, P2)` form.

`left`	arrow at point *P1*.
`right`	arrow at point *P2*.
`both`	arrows at both points.

11.1.17 `poly/1`

Templates

```
poly(ListOfPoints)
```

Description

`poly(ListOfPoints)` creates a hollow polygon defined by the sequence of points in *ListOfPoints*. Points are in the form `(Y, X)`. The set of lines will create a closed polygon by adding a line from the last point of the list back to the first point.

Example: `poly([(50, 25), (50, 75), (80, 75), (80, 25)])` describes the same picture as `box(50, 25, 30, 50)`.

11.1.18 `square/3`

Templates

```
square(YCenter, XCenter, Size)
```

Description

`square(YCenter, XCenter, Size)` creates a hollow square centered at `(YCenter, XCenter)` with sides of length *Size*.

11.1.19 `textbox/6`, `textbox/9`, `textbox/10`

Templates

```
textbox(T, L, D, W, Justification, Text)
textbox(Font, Size, Style, T, L, D, W, Justification, Text)
textbox(Font, Size, Style, T, L, D, W, Justification, Text, Bordered)
```

Description

`textbox(T, L, D, W, Justification, Text)` creates a rectangular text display, with the first letter at `(T, L)` and the text wrapped to fit the rectangle's width. The depth, *`D`*, must be greater than or equal to 15. If it is less than 15, then the graphics system will treat it as 15.

`Font, Size, Style`	same as for the text item of the Custom Dialog Format Descriptors (section 8.2, page 36). Defaults to: `'Arial', 12,0`.
`Justification`	text justification. 0 for left justified, 1 for right justified, -1 for centered text. [unimplemented]
`Bordered`	either `on` or `off`. Defaults to `off`.

11.1.20 `textline/4`, `textline/7`, `textline/8`

Templates

```
textline(T, L, Justification, Text)
textline(Font, Size, Style, T, L, Justification, Text)
textline(Font, Size, Style, T, L, Justification, Text, Bordered)
```

Description

`textline(T, L, D, W, Justification, Text)` creates a linear text display, with the first letter at `(T, L)` and the text following to the right.

`Font, Size, Style`	same as for the text item of the Custom Dialog Format Descriptors (section 8.2, page 36). Defaults to: `'Arial', 12,0`.
`Justification`	text justification. 0 for left justified, 1 for right justified, -1 for centered text. [unimplemented]
`Bordered`	either `on` or `off`. Defaults to `off`.

11.1.21 `wedge/6`

Templates

```
wedge(T, L, D, W , StartAngle, ArcAngle)
```

Description

`wedge(T, L, D, W , StartAngle, ArcAngle)` creates a filled arc as a segment (pie section) of the oval that inscribes the rectangle `(T, L, D, W)`. The arc begins at *`StartAngle`* clockwise from "12 o'clock" and continuing clockwise for *`ArcAngle`*. (The angles are in degrees.)

11.2 Transformers

The XGDL transformers are described below. There are several different kinds of transformers:

- spatial–size or location of picture.
- pen size–stroke width (size) of outlines.
- pen pattern–pattern of outlines.
- fill pattern–pattern of interiors of filled pictures.
- pen mode–compositing mode of picture with other pictures [NOT YET IMPLEMENTED].
- color–colors of a picture.

11.3 Simplified Transformers

There are simplified single argument and general multi-argument forms of transformers. The single argument form, `Transformer(Description)` applies a named tranformation, *`Transformer`*, to the given picture, `Description`. For example:

```
red( stripesthin( oval( 10, 10, 90, 180 ) ) )
```

There are two single argument transformers, `red` and `stripesthin`, applied to an elementary picture, `oval`.

The single argument transformers are simplified forms of the multiple argument transformers.

11.3.1 color/1

Templates

`Color(Description)`

Description

The *`Color`* is applied to the picture *`Description`*. The *`Color`* is one of: `black, blue, cyan, green, magenta, red, white`, and `yellow`.

11.3.2 fillPattern/1

Templates

`FllPattern(Description)`

Description

The *`FllPattern`* is applied to the picture *`Description`*. The *`FllPattern`* is one of: `alpha, beta, blank, boxed, brick, check, crosses, diag, diamonds, hash, lhash, rdiag, solid, speckled, stripesthick, stripesthin`, and textttwaves.

11.3.3 penTransformer/1

Templates

`PenTransformer(Description)`

Description

The *PenTransformer* is applied to the picture *Description*. The *PenTransformer* is one of: `blankpen, double, hashpen, nilpen, solidpen, thick, thin`, and `triple`.

11.4 General Transformers

(Not-yet-implemented MacProlog32 GDL transformers: fillmode, penmode, thicker, thinner.)

11.4.1 Color term

Several of the general transformers (and the dialog and control window `color` item) take a color term as an argument. A color term has one of the following forms:

`atom`	One of the atoms: `black, blue, cyan, green, magenta, red, white`, and `yellow`.
`rgb(R, G, B)`	*R*, *G*, and *B* are integers in the range 0 to 65535. *R* indicates the red magnitude, *G* the green, and *B* the blue.
`rgb(R, G, B, A)`	*R*, *G*, *B* , and *A* are numbers in the range 0.0 to 1.0. *R* indicates the red magnitude, *G* the green, *B* the blue, and *A* is the 'alpha' (opacity) magnitude (1.0 is opaque, 0.0 is transparent).
`hsb(H, S, B, A)`	*H*, *S*, *B* , and *A* are numbers in the range 0.0 to 1.0. *H* indicates the hue magnitude, *S* the satjuration, *B* the brightness, and *A* is the 'alpha' magnitude.
`white(W, A)`	*W* and *A* are numbers in the range 0.0 to 1.0. *W* indicates the white magnitude (0.0 is black and 1,0 is white) and *A* is the 'alpha' magnitude.

11.4.2 Pattern term

A pattern term as an argument. A pattern term has one of the following forms:

`pattern name atom`	One of the atoms: `alpha, beta, blank, boxed, brick, check, crosses, diag, diamonds, hash, lhash, rdiag, solid, speckled, stripesthick, stripesthin`, and `waves`.
`hexadecimal atom`	An atom that is 16 hexadecimal digits. These describe an 8 by 8 pixel pattern, 1 bit per pixel. Foreground pixels are represented by a 1 and background by 0. Each hexadecimal digit represents 4 bits. Example: Checkerboard pattern is `AA55AA55AA55AA55`. The top row is `10101010` in binary, which is `AA` in hexadecimal. The second row is `01010101` in binary which is `55` in hexadecimal.
`GDL Description term`	Any valid GDL description term. This is a picture that is "tiled" to become a pattern.

11.4.3 backcol/2

Templates

```
backcol(Color, Definition)
```

Description

Set *Color* as the background color (pen and fill) for *Definition*.

11.4.4 fill/2

Templates

```
fill(Details, Definition)
```

Description

Set the various 'fill' transformations in *Details* for *Definition*.

Defintiion of *Details*:

- Any simplified transformer fill pattern (sets the fill pattern).
- Any color term (sets the fill foreground color).
- `fg(Color)`, where *Color* is any color term (sets the fill foreground color).
- `bg(Color)`, where *Color* is any color term (sets the fill background color).
- `on(ForeColor, BackColor)` where *ForeColor* and *BackColor* are color terms (sets the fill foreground color to *ForeColor* and the fill background color to *BackColor*).

11.4.5 fillbg/2

Templates

```
fillbg(Color, Definition)
```

Description

Set *Color* as the background fill color for *Definition*.

11.4.6 fillfg/2

Templates

```
fillfg(Color, Definition)
```

Description

Set *Color* as the foreground fill color for *Definition*.

11.4.7 fillpattern/2

Templates

`fillpattern(Pattern, Definition)`

Description

Set *`Pattern`* as the fill pattern for *`Definition`*.

11.4.8 pen/2

Templates

`pen(Details, Definition)`

Description

Set the various 'pen' transformations in *`Details`* for *`Definition`*.

Defintiion of *`Details`* includes all of the details defined for the fill/2 transformer, plus:

- atom that is one of `thick`, `thin`, `nilpen`.
- `size(D, W)` where *`D`* is the depth of the pen in pixels and *`W`* is the width. The XGP Cocoa pen size is actually a 'stroke line width', so these values are averaged to specify this width.

11.4.9 penbg/2

Templates

`penbg(Color, Definition)`

Description

Set *`Color`* as the background pen color for *`Definition`*.

11.4.10 penfg/2

Templates

`penfg(Color, Definition)`

Description

Set *`Color`* as the foreground pen color for *`Definition`*.

11.4.11 penpattern/2

Templates

`penpattern(Pattern, Definition)`

Description

Set *`Pattern`* as the pen pattern for *`Definition`*. Pattern terms are defined in (section 11.4.2, page 70).

11.4.12 penscale/3

Templates

```
penscale(H, W, Description)
```

Description

Scale the 'stroke' (the line width) of *`Description`* where the height is multiplied by *`H`* and the width is multiplied by *`W`*.

11.4.13 scale/3

Templates

```
scale(H, W, Description)
```

Description

Scale the *`Description`* where the height is multiplied by *`H`* and the width is multiplied by *`W`*.

11.4.14 trans/3

Templates

```
trans(Y, X, Description)
```

Description

Translate the *`Description`* where the top is shifted by *`Y`* and the left is shifted by *`X`*.

11.4.15 rotate/5

Templates

```
rotate(Angle, CY, CX, Description)
```

Description

Rotate the *`Description`* by *`Angle`* degrees around the point `(CY, CW)`.

12 Graphics Utility Predicates

There are several predicates to manipulate boxes and points. A box is represented by the `box(T, L, D, W)`. A point is tepresented by `(Y, X)`: *Y* is the y-coordinate, *X* is the x-coordinate.

12.1 union_box/3

Templates

```
union_box(+Box1,+Box2, -UnionBox)
```

Description

UnionBox is the union of *Box1* and *Box2*.

12.2 intersect_box/3

Templates

```
intersect_box(+Box1, +Box2, -IntersectBox)
```

Description

IntersectBox is the intersection of *Box1* and *Box2*.

12.3 pt_in_box/2

Templates

```
pt_in_box(+Point, +Box)
```

Description

`pt_in_box(Point, Box)` succeeds if *Point* is in *Box*.

12.4 box_in_box/2

Templates

```
box_in_box(+Box1, +Box2)
```

Description

`box_in_box(Box1, Box2)` succeeds if *Box1* is contained by *Box2*.

12.5 pts_to_box/3

Templates

```
pts_to_box(+Point1, +Point2, -Box)
```

Description

`pts_to_box(Point1, Point2, Box)` succeeds if *`Point1`* and *`Point2`* are at opposite corners of *`Box`*.

12.6 pts_to_polar/4

Templates

```
pts_to_polar(+Point1, +Point2, -Distance, -Angle)
```

Description

`pts_to_box(Point1, Point2, Distance, Angle)` succeeds if *`Distance`* is the cartesian distance bewteen *`Point1`* and *`Point2`* and *`Angle`* is degrees between the vector from the origin to *`Point1`* and the vector from the origin to *`Point2`*.

13 Files

The XGP file services provide utilities for handling path names, inspecting the file system, loading files, and 'optimizing' (compiling) files. These predicates are largely compatible with MacProlog32, albeit incomplete.

13.1 load_files/1, load_files/2

Templates

```
load_files(FileSpecs)
load_files(FileSpecs, _Options)
```

Description

`load_files(FileSpecs)` succeeds if the specified files exist and can be 'consulted'. The intended side-effect loads the specified files into the current XGP environment. The input files may be either Prolog source ('.pl') or byte code ('.wbc').

`load_files(FileSpecs, _Options)` has the same interpretation as `load_files(FileSpecs)`; the *_Options* parameter is ignored–it is only present for backward compatibility with MacProlog32.

Parameter

FileSpecs	list of atoms or terms of the form `Alias(RelPath)`: An atom is interpreted as a pathname. A term of the form `Alias(RelPath)` interprets *RelPath* relative to the path named by *Alias*.

13.2 optimize_files/1

Templates

```
optimize_files(FileInfo)
```

Description

`optimize_files(FileInfo)` succeeds if the specified files exist and can be 'consulted'. The intended side-effed optimizes (compiles) the source files to object (WAM byte code) files.

Parameter

FileInfo	a single term of the form `Source-Object` or a list of terms of that form.

13.3 source_file/1, source_file/2

Templates

```
source_file(File)
source_file(Predicate, File)
```

Description

`source_file(Predicate, File)` succeeds if *File* is the source file of some *Predicate* defined in the XGP environment.

`source_file(File)` succeeds if *File* is a source file in the XGP environment.

Parameter

Predicate	variable or term of the form `Functor / Arity`: predicate specification of a predicate defined in *File*.

13.4 source_load/1

Templates

```
source_load(File)
```

Description

`source_load(File)` succeeds if *File* is a valid Prolog source file . The intended side-effect creates an XGP document for *File*. This will also consult *File* if the 'Consult on Open' option is TRUE.

Parameter

File	atom or term of the form `Alias(Path)`: the pathname of the source file to be opened as a document in XGP.

13.5 old/2, old/5

Templates

```
old(+Type, -File)
old(+Types, -Files, +Directory, +InitFile, +Modes)
```

Description

`old(Type, File)` succeeds if the user selects a value for *File*. The intended side-effect Displays a standard 'open' panel (dialog). This panel allows the user to select a file entry of the given *Type*: the absolute path name of that entry is bound to *File*.

`old(Types, Files, Directory, InitFile, Modes)` succeeds if the user selects a value for *Files*. The intended side-effect displays a standard 'open' panel (dialog). This panel allows the user to select a file entry of the given *Types*: the absolute path names of those entries is bound to *Files*.

Parameters

Type	atom: a type atom is either `''` (the null string atom) or the name of a file extension (such as `txt` or `pl`)
Types	list of *Type* atoms.
File	variable or atom: bound to the pathname of the selected file.
Files	variable or list: bound to a list of the pathnames of the selected files.
Directory	atom: initial directory of the open panel. If empy (`''`), then the previous selection directory is used, or the current working directory of XGP if no previous selection.
InitFile	atom: initial file name for the open panel.
Modes	list of terms of the form `Name(Flag)`: Name is one of `multiple_selection`, `select_directories`, or `select_files` and Flag is either `true` or `false`.

13.6 new/2, new/3, new/4

Templates

```
new(-File, +Prompt)
new(-File, +Prompt, +InitFile)
new(-File, +Prompt, +InitFile, +Directory)
```

Description

`old(Type, File)` succeeds if the user selects a value for *File*. The intended side-effect displays a standard 'save' panel (dialog). This panel allows the user to specify a file pathname bound to *File*. The title of the panel is set to *Prompt*. The initial value for the specified file name is *InitFile*. The working directory for the panel is *Directory*.

Parameters

File	variable or atom: bound to the pathname of a file.
Prompt	atom: title for the displayed panel.
InitFile	atom: initial file name for the open panel.
Directory	atom: initial directory of the open panel. If empy (''), then the previous selection directory is used, or the current working directory of XGP if no previous selection.

13.7 files/2

Templates

```
files(+Directory, -Files)
```

Description

`files(Directory, Files)` succeeds if *Files* is the list of file entries in the directory *Directory*.

Parameters

Directory	atom: path of a directory.
Files	variable: bound to list of atoms that name the file entries in *Directory*.

13.8 folders/1, folders/2

Templates

```
folders(-Folders) folders(+Directory, Folders)
```

Description

`files(Directory, Files)` succeeds if *Folders* is the list of directory entries in the directory *Directory*, if given, in the working directory otherwise.

Parameters

Directory	atom: path of a directory.
Folders	variable: bound to list of atoms that name the folder entries in *Directory*.

13.9 ftype/3

Templates

```
ftype(+Path, -Type, -Creator)
```

Description

`ftype(Path, Type, Creator)` succeeds if *`Type`* unifies with the HFS Type and *`Creator`* unifies with the creator (if any) for the entry at *`Path`*.

Parameters

`Path`	atom: path of an entry, file or directory.
`Type`	variable: bound to HFS type of entry at *`Path`*.
`Creator`	variable: bound to HFS creator of entry at *`Path`*. (not yet working).

13.10 exists_file/1

Templates

```
exists_file(+Path)
```

Description

`exists_file(Path)` succeeds if file at *`Path`* exists.

Parameters

`Path`	atom: path of an entry, file or directory.

13.11 resolve_alias_and_link/2

Templates

```
resolve_alias_and_link(+Path, -ResolvedPath)
```

Description

`resolve_alias_and_link(Path, ResolvedPath)` succeeds if *`ResolvedPath`* unifies with the resolved form of *`Path`*, where the resolved form does not contain any HFS alias or UNIX link components.

Parameters

`Path`	atom: path of an entry, file or directory.
`ResolvedPath`	variable: bound to atom that is the file system path of the entry identified by *`Path`* without any alias or link components.

13.12 resolve_alias/2

Templates

```
resolve_alias(+Path, -ResolvedPath)
```

Description

resolve_alias(Path, ResolvedPath) succeeds if *ResolvedPath* unifies with the resolved form of *Path*, where the resolved form does not contain any HFS alias components.

Parameters

Path	atom: path of an entry, file or directory.
ResolvedPath	variable: bound to atom that is the file system path of the entry identified by *Path* without any alias components.

13.13 absolute_file_name/3

Templates

absolute_file_name(+RelFileSpec, +Options, -AbsFileName)

Description

absolute_file_name(RelFileSpec, Options, AbsFileName) succeeds if *AbsFileName* unifies with the resolved form of the pathname *RelFileSpec* as directed by *Options*:

- ignore_underscores(Flag) – *Flag* is true or false. If true, then underscores are removed from the expanded *RelFileSpec* before other expansions.
- access(Mode) – *Mode* is read, write, append, exist, or none. This mode check is made after all expansions of *RelFileSpec*.
- extensions(Ext) – *Ext* is a list of possible entry extensions (e.g. pl or wbc) to be tried in looking for an existing entry as the expansion of *RelFileSpec*. This option is ignored if *RelFileSpec* already has an extension.
- file_type(Type) – *Type* is text, prolog, or qof. This is an abstract way to specify extensions (qof is an object file, a MacProlog32 value possibly inherited from "Quintus Object Format")

Parameters

RelFileSpec	atom: file system path or XGP alias of the form Alias(RelPath), where *Alias* has an *AliasPath* defined by a file_search_path/2 clause and *RelPath* is defines a path relative to *AliasPath*.
Options	list: none, one or more of the options discussed above: ignore_underscores(Flag), access(Mode), extensions(Ext), file_type(Type) .
AbsFileName	variable: bound to atom that is the expansion of *RelFileSpec*.

14 Persistent Values

The XGP persistent value services provide user default cross-session and (MacProlog32-compatible) per-session utilities for associating property name-value pairs with terms, setting and retrieving values in global variable names, and getting and setting prolog state values.

14.1 Properties

A property has a name and a value and is associated with an object. An object may have any number of distinctly named properties. A property of a particular name for a particular term has at most one value. The same property name may be used for any number of different objects.

14.1.1 `set_prop/3`

Templates

```
set_prop(+Object, +Property, +Value)
```

Description

`set_prop(Object, Property, Value)` always succeeds. Its intended side-effect sets the named property *`Property`* of *`Object`* to have *`value`*.

Parameters

`Object`	term.
`Property`	atom: name of a property.
`Value`	term.

14.1.2 `get_prop/3`

Templates

```
get_prop(+Object, +Property, -Value)
```

Description

`get_prop(Object, Property, Value)` succeeds if the *`Property`* value of *`Object`* unifies with *`Value`*.

Parameters

`Object`	term.
`Property`	atom: name of a property.
`Value`	variable or term.

14.1.3 `del_prop/1`, `del_prop/2`

Templates

```
del_prop(+Object)
del_prop(+Object, +Property)
```

Description

`del_prop(Object)` always succeeds. The intended side-effect deletes all properties from the object.

`del_prop(Object, Property)` succeeds if *`Object`* has *`Property`*. The intended side-effect deletes this property from the object.

Parameters

`Object`	term.
`Property`	atom: name of a property.

14.1.4 `del_cons/1`

Templates

```
del_cons(+Property)
```

Description

`del_cons(Property)` always succeeds. The intended side-effect deletes *`Property`* from all objects.

Parameters

`Property`	atom: name of a property.

14.1.5 `get_cons/2`

Templates

```
get_cons(+Property, -List)
```

Description

`get_cons(Property, List)` succeeds if *`List`* unifies with the list of all of the objects with property named *`Property`*.

Parameters

`Property`	atom: name of a property.
`List`	variable or list: unifies with list of all objects having property named *`Property`*.

14.1.6 `get_props/2`

Templates

```
get_props(+Object, -List)
```

Description

`get_props(Object, List)` succeeds if *`List`* unifies with the list of all of the propery names for *`Object`*.

Parameters

Object	term.
List	variable or list: unifies with list of all property names for given *Object*.

14.1.7 get_all_cons/1

Templates

```
get_all_cons(-Objects)
```

Description

`get_all_cons(Objects)` succeeds if *Objects* unifies with the list of all of the objects with any property.

Parameters

Objects	variable or list: unifies with list of all objects having a property.

14.1.8 get_all_props/1

Templates

```
get_all_props(-Properties)
```

Description

`get_all_props(Properties)` succeeds if *Properties* unifies with the list of all of the property names for all objects.

Parameters

Properties	variable or list: unifies with list of all property names for all objects.

14.1.9 del_all_props/0

Templates

```
del_all_props
```

Description

`del_all_props` always succeeds. The intended side-effect deletes all of the properties for all objects.

14.2 Global Variables

An XGP global variable has a name and a value. It is implemented as a simple MacProlog32-compatible wrapper for gprolog's global variables.

14.2.1 remember/2

Templates

```
remember(+Atom, +Value)
```

Description

`remember(Atom, Value)` always succeeds. The intended side-effect sets the global variable named *`Atom`* to *`Value`*.

Parameters

`Atom`	atom: global variable name
`Value`	term

14.2.2 `recall/2`

Templates

```
recall(+Atom, -Value)
```

Description

`recall(Atom, Value)` succeeds if there is a remembered value for *`Atom`* and it unifies with *`Value`*.

Parameters

`Atom`	atom: global variable name
`Value`	term

14.2.3 `default/3`

Templates

```
default(+Atom, -Value, +Default)
```

Description

`default(Atom, Value, Default)` succeeds if there is a remembered value for *`Atom`* and it unifies with *`Value`* or if there is no remembered value and *`Value`* unifies with *`Default`*.

Parameters

`Atom`	atom: global variable name
`Value`	term
`Default`	term

14.2.4 `forget/1`

Templates

```
forget(+Atom)
```

Description

`forget(Atom)` always succeeds. The intended side-effect forgets (deletes) the value remembered for *`Atom`*, if any.

Parameters

Atom	atom: global variable name

14.3 State Variables

There are a number of XGP prolog state values defined for compatibility with MacProlog32. Some of these state values are just place holders, and some of them map into gprolog state values.

14.3.1 prolog_flag/2, prolog_flag/3

Templates

```
prolog_flag(+Flag, -OldValue)
prolog_flag(+Flag, -OldValue, +NewValue)
```

Description

`prolog_flag(Flag, OldValue)` succeeds if *OldValue* unifies with the value for the value for *Flag*.

`prolog_flag(Flag, OldValue, NewValue)` succeeds if *OldValue* unifies with the value for the value for *Flag*. The intended side-effect sets the value to *NewValue*.

Parameters

Flag	atom: name of an XGP prolog state flag.
OldValue	variable or term: value of an XGP prolog state flag.
NewValue	term: value of an XGP prolog state flag.

14.4 User Defaults

User default values are stored in a per-user database managed by a Mac OS X standard facility. The names provided as the key for these values should be unique, e.g. 'com.goodstuff.special' as the key for a "special" user default.

14.4.1 user_default/2

Templates

```
user_default(+Key, -Value)
user_default(+Key, +Value)
```

Description

`user_default(Key, Value)` succeeds if *Value* is a variable and a value has been recorded for *Key*. The recorded value is unified with *Value*.

`user_default(Key, Value)` always succeeds if *Value* is a non-variable term. The intended side-effect records the value for *Key* as *Value*.

Parameters

Key	atom: name of a user default key.
Value	term or variable: value of a user default key.

15 Utilities

This section describes various prolog utilities availabe in XGP.

15.1 Fonts and Text

These predicates support working with fonts and text.

15.1.1 font_info/7

Templates

```
font_info(+Font, +Size, +Style, -Ascent, -Descent, -MaxWidth, -Leading)
```

Description

`font_info(Font, Size, Style, Ascent, Descent, MaxWidth, Leading)` succeeds if *Ascent, Descent, MaxWidth* and *Leading* unify with the values for the font glyphs for *Font, Size*, and *Style*.

Parameters

Font	atom: font name for glyphs. (e.g. `Lucida Grande`)
Size	number: size of glyphs in points.
Style	number, style of glyphs: 0 - normal, 1 - bold, 2 - italic, 4 - underline, 8 - small caps (was outline), 16 - other (was shadow).
Ascent	variable, unifies with the "ascent" feature for glyphs in the given *Font, Size*, and *Style*.
Descent	variable, unifies with the "descent" feature for glyphs in the given *Font, Size*, and *Style*.
MaxWidth	variable, unifies with the "maximum width" feature for glyphs in the given *Font, Size*, and *Style*.
Leading	variable, unifies with the "leading" feature for glyphs in the given *Font, Size*, and *Style*.

15.1.2 default_font_info_by_type/5

Templates

```
default_font_info_by_type(+Type, +SizeIN, -Font, -SizeOUT, -Style)
```

Description

`default_font_info_by_type(Type, SizeIN, Font, SizeOUT, Style)` succeeds if *Type* is a valid font type and *Font, SizeOUT* and *Style* unify with the values for the *Type* and *SizeIN*.

Parameters

Type	atom: one of the types of fonts defined for Macintosh OS X. Defined types: `boldsystem, controlcontent, label, menu, menubar, message, palette, smallsystem, system, titlebar, tooltips`, or `user`.

SizeIN	number: greater than or equal to 0. If this is 0, then this predicate unifies *SizeOUT* to the default size for the given *Type*. Otherwise, *SizeOUT* unifies with *SizeIN* and the *Type* and *SizeIN* are used to determine the *Font* and *Style*.
Font	atom: font name for glyphs. (e.g. `Lucida Grande`)
SizeOUT	number: size of glyphs in points.
Style	number, style of glyphs: 0 - normal, 1 - bold, 2 - italic, 4 - underline, 8 - small caps (was outline), 16 - other (was shadow).

15.1.3 `text_width/5`

Templates

```
text_width(+Text, +Font, +Style, +Size, -Width)
```

Description

`text_width(Text, Font, Style, Size, Width)` succeeds if *Width* unifies with the width in points required to display *Text* in the given *Font*, *Size*, and *Style*.

Parameters

Text	atom, text which is to have its width (in points) determined.
Font	atom: font name for glyphs. (e.g. `Lucida Grande`)
Style	number, style of glyphs: 0 - normal, 1 - bold, 2 - italic, 4 - underline, 8 - small caps (was outline), 16 - other (was shadow).
Size	number: size of glyphs in points.
Width	variable, unifies with the width of Text in the given Font, Style, and Size.

15.2 Input and Output

These predicates support reading and writing.

15.2.1 ~> /2

Templates

```
Goal ~> Target
```

Description

This succeeds if *Goal* succeeds and *Target* is a valid output destination. The intended side-effect redirects the output from evaluating *Goal* to *Target*. If *Target* is a variable, then the output is converted to an atom and that atom is unified with *Target*.

Parameters

Goal	callable term
Target	term: a stream term, an alias for a stream term, the name of an open window or document, the name of a file, or a variable.

15.2.2 <~/2

Templates

```
Goal <~ Source
```

Description

This succeeds if *Goal* succeeds and *Source* is a valid input source. The intended side-effect redirects the input while evaluating *Goal* from *Source*. If *Source* is a list of character atoms or integer character codes, then the input is read from that list.

Parameters

Goal	callable term
Source	term: a stream term, an alias for a stream term, the name of an open window or document, the name of a file, a list of integer character codes, or a list of single character atoms.

15.2.3 writenl/1, writeqnl/1, writeseqnl/1, writeqseqnl/1, write_list/1, write_list/2, writeq_list/1, writeq_list/2

Templates

```
writenl(+Term)
writeqnl(+Term)
writeseqnl(+Terms)
writeqseqnl(+Terms)
write_list(+Terms)
write_list(+Terms, +Separator)
writeq_list(+Terms)
writeq_list(+Terms, +Separator)
```

Description

`writenl(Term)` always succeeds . The intended side-effect writes *Term* to output (without quoting), followed by a newline.

`writeqnl(Term)` always succeeds . The intended side-effect writes *Term* to output (with quoting), followed by a newline.

`writeseqnl(Terms)` succeeds if *Terms* is a list . The intended side-effect writes each term in *Terms* to output (without quoting), separated by spaces and followed by a newline.

`writeqseqnl(Terms)` succeeds if *Terms* is a list . The intended side-effect writes each term in *Terms* to output (with quoting), separated by spaces and followed by a newline.

`write_list(Terms)` succeeds if *Terms* is a list . The intended side-effect writes each term in *Terms* to output (without quoting), separated by spaces.

`write_list(Terms, Separator)` succeeds if *Terms* is a list . The intended side-effect writes each term in *Terms* to output (without quoting), separated by *Separator*.

`writeq_list(Terms)` succeeds if *Terms* is a list . The intended side-effect writes each term in *Terms* to output (with quoting), separated by spaces.

`writeq_list(Terms, Separator)` succeeds if *`Terms`* is a list . The intended side-effect writes each term in *`Terms`* to output (with quoting), separated by *`Separator`*.

Parameters

`Term`	term
`Terms`	list of terms
`Separator`	term

15.3 Atoms

These predicates work with atoms, constructing or interpreting their names.

15.3.1 `charof/2`

Templates

```
charof(+Char, ?Code)
```

Description

`charof(Char, Code)` succeeds if *`Code`* is the integer ASCII character code of the atom named by the single character of *`Char`*.

Parameters

`Char`	atom: a single character name.
`Code`	integer: ASCII code of Char.

15.3.2 `lower/2`

Templates

```
lower(+TextAtom, ?LowerTextAtom)
```

Description

`lower(TextAtom, LowerTextAtom)` succeeds if *`LowerTextAtom`* is the lowercase ASCII version of *`TextAtom`*.

Parameters

`TextAtom`	atom
`LowerTextAtom`	atom: Lowercased version of *`TextAtom`*.

15.3.3 `upper/2`

Templates

```
upper(+TextAtom, ?UpperTextAtom)
```

Description

`upper(TextAtom, UpperTextAtom)` succeeds if *UpperTextAtom* is the uppercase ASCII version of *TextAtom*.

Parameters

TextAtom	atom
UpperTextAtom	atom: Uppercased version of *TextAtom*.

15.3.4 proper/2

Templates

```
proper(+TextAtom, ?ProperTextAtom)
```

Description

`proper(TextAtom, ProperTextAtom)` succeeds if *ProperTextAtom* is the "proper" case ASCII version of TextAtom. The first character is upper-cased, all other characters are lower-cased.

Parameters

TextAtom	atom
ProperTextAtom	atom: Proper-cased version of *TextAtom*.

15.3.5 pname/2

Templates

```
pname(+Term, ?Atom)
pname (-Term, +Atom)
```

Description

`pname (Term, Atom)` succeeds if *Atom* is named by the string that is the quoted display form of *Term*.

Parameters

Term	term: the interpretation (by `read/1`) of *Atom*.
Atom	atom: this atom's name is the string that is the display of *Term*.

15.3.6 concat_list/2, concat_list/3, concat/2, concat/3

Templates

```
concat_list(+Terms, ?Atom)
concat_list(+Terms, +Separator, ?Atom)
concat(+Terms, ?Atom)
concat(+Term1, +Term2, ?Atom)
```

Description

`concat_list(Terms, Atom)` succeeds if *Atom* is constructed by concatenating together the terms in *Terms*.

`concat_list(Terms, Atom, Separator)` succeeds if *Atom* is constructed by concatenating together the terms in *Terms*, separated by *Separator*.

`concat(Terms, Atom)` succeeds if *Atom* is constructed by concatenating together the terms in *Terms*. (Same as `concat_list/2`.)

`concat(Term1, Term2, Atom)` succeeds if *Atom* is constructed by concatenating together *Term1* and *Term2*.

Parameters

Terms	list of terms
Term1	term
Term2	term
Atom	atom
Separator	term

15.3.7 `gensym/2`

Templates

```
gensym(+Root, -Symbol)
```

Description

`gensym(Root, Symbol)` succeeds if the symbol generated from *Root* unifies with *Symbol*. Intended side-effect increments a counter for *Root* such that the next evaluation of this predicate produces a different value for *Symbol*.

Parameters

Root	atom: prefix for generated symbol atom..
Symbol	variable or atom: this atom's name is concatenation of *Root* followed by an integer.

15.3.8 `genint/2`

Templates

```
genint(+Root, -Integer)
```

Description

`genint(Root, Integer)` succeeds if the integer generated from *Root* unifies with *Integer*. Intended side-effect increments a counter for *Root* such that the next evaluation of this predicate produces a different value for *Integer*.

Parameters

Root	atom: prefix for generated integer atom..
Integer	variable or integer.

15.3.9 `init_gensym/2`

Templates

```
init_gensym(+Root)
init_gensym(+Root, +Integer)
```

Description

`init_gensym(Root)` always succeeds. Intended side-effect sets the integer for *`Root`* to 0.

`init_gensym(Root, Integer)` always succeeds. Intended side-effect sets the integer for *`Root`* to *`Integer`*.

Parameters

`Root`	atom: prefix for generated integer atom..

15.4 Clauses

These predicates work with clauses.

15.4.1 `forall/2`

Templates

```
forall(+Antecedent, +Consequent)
```

Description

`forall(Antecedent, Consequent)` succeeds if *`Consequent`* is true for all solutions of *`Antecedent`*.

Parameters

`Antecedent`	callable term
`Consequent`	callable term

15.4.2 `otherwise/0`

Templates

```
otherwise
```

Description

`forall(Antecedent, Consequent)` always succeeds (same as `true`). This may be used for the 'else' clause of a conditional.

15.4.3 `assert/1`

Templates

```
assert(+X)
```

Description

The same as `assertz(X)`.

Parameters

X	term, not a variable.

15.4.4 `dynamic/1`

Templates

```
dynamic(PredicateSpec)
```

Description

If PredicateSpec is not currently a predicate, then this causes it to become a dynamic predicate with no clauses (by asserting and retracting it). If PredicateSpec is currently a predicate, this predicate is true if it has the `dynamic` property.

Parameters

PredicateSpec	term unifying with `Functor / Arity`, where *Functor* is an atom and *Arity* is a nonnegative integer.

15.4.5 `def/2`, `def/3`

Templates

```
def(?Functor, ?Arity)
def(?Functor, ?Arity, ?MType)
```

Description

`def(Functor, Arity)` succeeds if `Functor/Arity` is currently a predicate.

`def(Functor, Arity, MType)` succeeds if `Functor/Arity` is currently a predicate with MacProlog type *MType*. (The mappings to GProlog types are provisional, they may change.)

- -1 = static incrementally compiled
- -2 = static optimized
- -3 = static assembler
- -4 = static external
- 1 = dynamic incrementally opimized
- 2 = dynamic optimized
- 3 = dynamic assembler
- 4 = dynamic external¡BR¿¡/TD¿

Parameters

Functor	atom
Arity	integer
MType	integer: between -4 and 4.

15.5 List

These predicates choose items from lists, and find uniones, intersections and differences between lists.

15.5.1 sort/3

Templates

```
sort(+List, ?SortedList, +KeyPath)
```

Description

`sort(+List, ?SortedList, +KeyPath)` succeeds if *SortedList* unifies with the sorted list of the terms in *List*, compared according to *KeyPath*. *KeyPath* is a list of integers specifying what "path" through the nested structures of each term to following to determine the (sub)term to use to compare (using the `@<` term comparison). An integer of 1 identifies the functor of a term, integers 2 to N+1 identify arguments 1 to N respectively. An empty list specifies the entire term. For example, consider the list:

```
[a(b, c(d)), e(f), g(h(i))]
```

A path of `[]` specifies the list "as is". A path of `[1]` specifies the list `[a, e, g]` for the respective terms of the original list. A path of `[2,1]` specifies: `[b, f, h]`.

Parameters

List	list or variable
SortedList	variable: unifies with the sorted list of term in List.
KeyPath	list of none, one, or more integers: specifies the portion of each term of *List* to use in comparing to determine the sort order for *SortedList*.

15.5.2 choose/4, choose_split/4

Templates

```
choose(+List, ?Item, ?Prefix, ?Suffix)
choose(-List, ?Item, +Prefix, ?Suffix)
choose_split(+List, ?Item, ?Prefix, ?Suffix)
choose_split(-List, ?Item, +Prefix, ?Suffix)
```

Description

These predicates succeed if *Item* unifies with a member of *List* such that *Prefix* is the sublist and *Suffix* is the sublist of *List* that follows Item.

Parameters

List	list or variable
Item	term
Prefix	list or variable
Suffix	list or variable

15.5.3 choose/2, choose/3, choose_once/2, choose_once/3, choose_once/4

Templates

```
choose(+List, ?Item)
choose(+List, ?Item, ?Remainder)
choose_once(+List, +Item, ?Prefix, ?Suffix)
choose_once(+List, +Item, ?Remainder)
choose_once(+List, +Item)
```

Description

`choose(List, Item)` succeeds if *Item* unifies with a member of *List*.

`choose(List, Item, Remainder)` succeeds if *Item* unifies with a member of *List* such that *Remainder* is the sublist of *List* excluding Item.

`choose_once(List, Item, Prefix, Suffix)` succeeds if *Item* unifies with the *first* occurence of the 'Item' term in *List* such that *Prefix* is the sublist and *Suffix* is the sublist of *List* that follows Item.

`choose_once(List, Item, Remainder)` succeeds if *Item* unifies with the *first* occurence of the 'Item' term in *List* such that *Remainder* is the sublist of *List* excluding *Item*.

Parameters

List	list or variable
Item	term
Prefix	list or variable
Suffix	list or variable
Remainder	list or variable

15.5.4 choose_trim/3

Templates

```
choose_trim(+List, ?Item, ?Suffix)
```

Description

`choose_trim(+List, ?Item, ?Suffix)` succeeds if *Item* unifies with a term of List such that *Suffix* is the sublist of *List* following *Item*.

Parameters

List	list or variable
Item	term
Suffix	list or variable

15.5.5 choose_identical/2, choose_identical/3

Templates

```
choose_identical(+List, ?Item)
choose_identical(+List, ?Item, ?Remainder)
```

Description

`choose_identical(List, Item, Remainder)` succeeds if *Item* is identical with a term of *List* such that *Remainder* is the sublist of *List* excluding *Item*.

`choose_identical(List, Item)` succeeds if *Item* is identical with a member of *List*.

Parameters

List	list or variable
Item	term
Remainder	list or variable

15.5.6 difference/3, difference_ordered/3

Templates

```
difference(+Minuend, +Subtrahend, ?Difference)
difference_ordered(+Minuend, +Subtrahend, ?Difference)
```

Description

`difference(Minuend, Subtrahend, Difference)` succeeds if *Difference* is the list of terms remaining after removing items from *Minuend* that are also in *Subtrahend*. A single occurrence of a term in *Subtrahend* removes a single item from *Minuend*. The order of *Minuend* is preserved in *Difference*; if there are more occurrences of a term in *Minuend* than then the *last* of the excess occurrences appear in *Difference*.

E.g. `difference([a,b,a,c,a], [a,a], X)`

yields `X = [b,c,a]` instead of `[a,b,c]` or `[b,a,c]`.

`difference_ordered(Minuend, Subtrahend, Difference)` succeeds if *Minuend* and *Subtrahend* are sorted lists and *Difference* is the sorted list of terms occurring in *Minuend* and not in *Subtrahend*. (This predicate does not use unification for term comparison.)

Parameters

Minuend	list
Subtrahend	list
Difference	list or variable

15.5.7 intersect/3, intersect_ordered/3, intersect_difference/5, intersect_difference_ordered/5

Templates

```
intersect(+A, +B, ?Intersection)
intersect_ordered(+A, +B, ?Intersection)
intersect_difference(+A, +B, ?Intersection, ?AD, ?BD)
intersect_difference_ordered(+A, +B, ?Intersection, ?AD, ?BD)
```

Description

`intersect(A, B, Intersection)` succeeds if *Intersection* is the list of terms occuring in both *A* and *B*. If a term occurs N times in *A* and K times in *B*, then it occurs the minimum of N and K times in *Intersection*. The order of *A* is preserved in *Intersection*; if there are more occurrences of a term in *A* than *B* then the *first* of the common occurrences appear in *Intersection*

E.g. `intersect([a,b,a,c,a], [c,b,a], X)`

yields `X = [a,b,c]` instead of `[b,c,a]` or `[b,a,c]`.

`intersect_difference(A, B, Intersection, AD, BD)` succeeds if *Intersection* is the list of terms that can be unified between *A* and *B*. The terms of *A* are progressively unified with terms of *B*, and successful unifications are placed in *Intersection*. Once a term of *B* is used in a unification it is unavailable for further unifications. The order of *A* is preserved in *Intersection*; if there are more occurrences of a term in *A* than *B* then the *first* of the common occurrences appear in *Intersection*.

AD is the list of terms remaining after removing *Intersection* from *A*. The order of *A* is preserved in *AD*; if there are more occurrences of a term in *A* than *B* then the *last* of the excess occurrences appear in *AD*.

BD is the list of terms remaining after removing *Intersection* from *B*. The order of *B* is preserved in *BD*; if there are more occurrences of a term in *B* than *A* then the *last* of the excess occurrences appear in *BD*.

This deconstruction is *not* reversible.

E.g. `intersect_difference([a,b,a,c,a], [c,d,a,c], C, AD, BD)`

yields `C = [a,c], AD = [b,a,a], BD = [d,c]`.

`intersect_ordered(A, B, Intersection)` succeeds if *A* and *B* are sorted lists and *Intersection* is the sorted list of terms occurring identically in both *A* and *B*. (This predicate does not use unification for term comparison.) This predicate can be a little bit faster than using `intersect_difference_ordered/5` and just ignoring the difference arguments.

`intersect_difference_ordered(A, B, Intersection, AD, BD)` is similar to `intersect_difference(A, B, Intersection, AD, BD)` with the distiinction that all of the arguments are sorted lists.

Parameters

A	list
B	list
Intersection	list or variable
AD	list or variable
BD	list or variable

15.5.8 anonymous_list/2

Templates

`anonymous_list(+K, ?List)`

Description

`anonymous_list(K, List)` succeeds if *List* is a list of *K* distinct unbound (anonymous) variables.

Parameters

K	integer
List	list or variable

15.5.9 append_list/2

Templates

```
append_list(+ListOfLists, ?CombinedList)
```

Description

`append_list(ListOfLists, CombinedList)` succeeds if *CombinedList* is the result of appending together the items of *ListOfLists*.

Parameters

ListOfLists	list of lists
CombinedList	list or variable

15.5.10 subset/2

Templates

```
subset(+Subset, +Superset)
```

Description

`append_list(ListOfLists, CombinedList)` succeeds if the terms of *Subset* have a one-to-one mapping (by way of unification) with items in *Superset*. This mapping is 'into': there may be unmapped items in *Superset*.

Parameters

Subset	list
Superset	list

15.5.11 nth_element/2, nth_element/3

Templates

```
nth_element(-Index, +List, +Item)
nth_element(+Index, ?List, ?Item)
nth_element(-Index, +List, +Item, ?Tail)
nth_element(+Index, ?List, ?Item, ?Tail)
```

Description

`nth_element(Index, List, Item)` succeeds if *Item* unifies with the *Index*-th element of *List*. *List* satisfies `append(Prefix, [Item|_], List)`, where *Prefix* is a list of (*Index*-1) terms.

`nth_element(Index, List, Item, Tail)` succeeds if *Item* unifies with the *Index*-th element of *List*. *List* satisfies `append(Prefix, [Item|Tail], List)`, where *Prefix* is a list of (*Index*-1) terms.

Parameters

Index	list
List	list
Item	term
Tail	list

15.5.12 `insert_ordered/3`, `insert_ordered_unique/3`

Templates

```
insert_ordered(+ListIN, +X, ?ListOUT)
insert_ordered_unique(+ListIN, +X, ?ListOUT)
```

Description

`insert_ordered(ListIN, X, ListOUT)` succeeds if *ListOUT* is the same as *ListIN* except *ListOUT* has one more occurrence of *X* than is in *ListIN*. *ListIN* and *ListOUT* are both sorted.

`insert_ordered_unique(ListIN, X, ListOUT)` succeeds if *ListOUT* is the same as *ListIN* except *X* is in *ListOUT* and is *not* in *ListIN*. *ListIN* and *ListOUT* are both sorted.

Parameters

ListIN	list or variable
X	term
ListOUT	list or variable

15.5.13 `split_list_in_halves3`

Templates

```
split_list_in_halves(?List, ?Front, ?Back)
```

Description

`split_list_in_halves(List, Front, Back)` succeeds if *Front* unifies with the former half of *List* and *Back* unifies with the latter half of *List*.

Parameters

List	list or variable
Front	list or variable
Back	list or variable

15.5.14 union_ordered/3, union_ordered/5

Templates

```
union_ordered(+Set1, +Set2, ?Union)
union_ordered(+Set1, +Set2, ?Union, ?Set1Diff, ?Set2Diff)
```

Description

`union_ordered(Set1, Set2, Union)` succeeds If `Set1` and `Set2` are the ordered representations of two sets and `Union` is unified with the ordered representation of their union. `union_ordered/3` is not defined if `Set1` or `Set2` is insufficiently instantiated or not in standard order.

`union_ordered(+Set1, +Set2, ?Union, ?Set1Diff, ?Set2Diff)` succeeds If `Set1` and `Set2` are the ordered representations of two sets, `Union` is unified with the ordered representation of their union, `Set1Diff` is unified with the ordered representation of the difference of *Set1* minus *Set2* and `Set2Diff` is unified with the ordered representation of the difference of *Set2* minus *Set1*. `union_ordered/5` is not defined if `Set1` or `Set2` is insufficiently instantiated or not in standard order.

Parameters

Set1	list or variable
Set2	list or variable
Union	list or variable

15.6 Graphics and Control Windows

These predicates help with common tasks when working with graphics and control windows.

15.6.1 vertical_shift_box/3

Templates

```
vertical_shift_box(+Box, +ShiftAmount, ?ShiftedBox)
```

Description

`vertical_shift_box(Box, ShiftAmount, ShiftedBox)` succeeds if *ShiftedBox* is *Box* shifted vertically by *ShiftAmount*. A box term has the form `box(Top, Left, Depth, Width)`. Shifting a box vertically is adding *ShfitAmount* to the *Top*.

Parameters

Box	term of the form `box(T, L, D, W)`.
ShiftAmount	number
ShiftedBox	term of the form `box(TShifted, L, D, W)`, where *TShifted* is *T* (of *Box*) + *ShiftAmount*.

15.6.2 horizontal_split_box/5

Templates

```
horizontal_split_box(+Box, +Split, +Gutter, ?LeftBox, ?RightBox)
```

Description

`horizontal_shift_box(Box, ShiftAmount, ShiftedBox)` succeeds if *Box* is split into *LeftBox* and *RightBox* at *Split* with a separation of *Gutter*.

LeftBox and *RightBox* have the same *Top* and *Depth* values. The left value of *LeftBox* is the same as *Box*; its width is *Split*. The left value of *RightBox* is the left value of `Box + Split + Gutter`; its width is the width of *Box* minus `(Split + Gutter)`.

Parameters

Box	term of the form `box(T, L, D, W)`.
Split	number
Gutter	number
LeftBox	term of the form `box(T, L, D, Split)`.
RightBox	term of the form `box(T, L+Split+Gutter, D, W - (Split+Gutter))`.

15.6.3 box_top/2

Templates

```
box_top(+Box, ?Top)
```

Description

`box_top(Box, Top)` succeeds if *Box* has top value *Top*.

Parameters

Box	term of the form `box(T, L, D, W)`.
Top	number

15.6.4 box_left/2

Templates

```
box_left(+Box, ?Left)
```

Description

`box_left(Box, Left)` succeeds if *Box* has left value *Left*.

Parameters

Box	term of the form `box(T, L, D, W)`.
Left	number

15.6.5 box_depth/2

Templates

```
box_depth(+Box, ?Depth)
```

Description

`box_depth(Box, Depth)` succeeds if *Box* has depth value *Depth*.

Parameters

Box	term of the form `box(T, L, D, W)`.
Depth	number

15.6.6 box_width/2

Templates

```
box_width(+Box, ?Width)
```

Description

`box_width(Box, Width)` succeeds if *Box* has width value *Width*.

Parameters

Box	term of the form `box(T, L, D, W)`.
Width	number

References

[1] Nicky Johns MacProlog32 Programming Guide, LPA Ltd 1994.

[2] Nicky Johns MacProlog32 Graphics Guide, LPA Ltd 1994.

[3] D. Diaz and P. Codognet. "Design and Implementation of the GNU Prolog System". *Journal of Functional and Logic Programming*, Vol. 2001, No. 6, October 2001. `ftp://ftp.inria.fr/INRIA/Projects/loco/publications/GNU-PROLOG/jflp01.pdf`

[4] Daniel Diaz. GNU Prolog: A Native Prolog Compiler with Constraint Solving over Finite Domains, Edition 1.8, for GNU Prolog version 1.3.0, 2006. `http://www.gprolog.org/manual/gprolog.pdf`

Index