Garnet Toolkit Version 2.2 Changes Document
(Beta Release)

August 1993

Abstract

This is the change document for V2.2 of the Garnet toolkit. It describes all the differences from V2.1. This change document should be used with the manual *The Garnet Reference Manuals, Revised for Version 2.1*, November 1992, CMU-CS-90-117-R3.

Copyright © 1993 - Carnegie Mellon University

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. Additional support for Garnet has been provided by NEC, Apple, Adobe, and General Electric.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

1. Summary Of Changes

This document contains a description of all the changes from Garnet version 2.1 to version 2.2. The two sections below list the most important changes in the new version, split into "primary" and "secondary" lists. The primary changes are those that are most likely to be noticed and used by all Garnet users. The primary changes are not repeated in the list of secondary changes.

1.1. Primary Changes

Better Performance - The lowest level of Garnet, the KR object system, has been completely redesigned and is now significantly faster. Users should expect their Garnet applications to run two to three times faster.

Type-Checking - Type-checking is now performed on most user-settable slots in the pre-defined Garnet objects. So if you accidently set the `:left` of a `rectangle` to be something other than an integer, Garnet will generate a continuable error, with a thorough description of the offending slot and value. Users can impose type restrictions on their own slots, using new KR functions (see section 4.2).

Constant Formulas - A set of functions has been added that will help you identify which slots in your objects should be declared constant. Declaring constants should significantly reduce the space required by your application. See section 9.3.1.

Mutlifont-Text - The multifont object has been enhanced with several new features: automatic indentation when lisp code is used as the string; support for color and marks; hypertext-like imbedding of other Garnet objects in the string; many new editing commands (like `^-shift-B` to toggle bold font) to make typing strings easier; support for drag-through selection by the `multifont-text-interactor`; and a new slot in the multifont interactors called `:after-cursor-moves-func` which may contain a function to be executed whenever the mouse is moved in a multifont object.

New opal:text - The `opal:text` object has been reimplemented to handle multiple lines and cursors. References to the old `opal:cursor-text`, `opal:multi-text`, and `opal:cursor-multi-text` objects should be converted to `opal:text` references.

New Aggrelists - The strategy for laying out aggrelists has been completely rewritten. Instead of relying on formulas for positioning, aggrelist components are now laid out procedurally during update. The `opal:notice-items-changed` method should need to be called much less often, since the work that used to be done in that method is now done automatically when you update the window containing the aggrelist.

Debugging Tool - There is a powerful new tool called the `Inspector` which allows you to examine and change slot values of your objects without typing into the lisp listener. This tool can be invoked by hitting the HELP key while the mouse is positioned above the object to be examined.

Gilt - The Gilt Interface builder has been significantly rewritten, and made more robust. The standard and Motif versions have been unified, so that there is now only one Gilt that includes both sets of gadgets. The new version includes all of the pop-up gadgets, like menubars and option-buttons.

More Support - Garnet now runs on LispWorks and CLISP platforms. See sections 3.3 and 3.4 for special instructions on running Garnet in these lisps.

1.2. Secondary Changes

GLOBAL:

- A standardized declaration syntax has been created for the definition of special slots in Garnet schemas. Previously, various syntaxes were used for values of the slots `:constant`, `:maybe-constant`, `:update-slots`, `:ignored-slots`, `:local-only-slots`, and `:sorted-slots`. Now these special slots (and several new ones) can be defined using the new :DECLARE keyword in the `create-instance` call (see section 4.3).
- All objects now contain a list of slots that are user-settable in their `:parameters` slot. For example, the `opal:rectangle`'s parameters list includes `:left`, `:top`, and `:line-style`, along with other slots that determine its graphical appearance. The parameters list is a valuable data resource for objects like the `gg:prop-sheet`, which allow the user to change the value of certain slots in an object.
- The functions `garnet-load` and `garnet-compile` now look up their prefix parameters in an expandable association list. You can add your own prefixes and pathnames to this alist, so that the functions can be used for your own directories.
- There is a new variable in `garnet-loader.lisp` that sets the optimization flags for lisp, so that your compiled Garnet binaries will be as fast as possible.

KR:

- The macro `gv` can now be used at the top-level, where previously it was only allowed inside of formulas. `Gv` at the top-level now acts like `g-value`, so novice users only need to learn about `gv`. `G-value` can still be used inside of formulas when the user does not want to establish a dependency on a referenced slot.
- The macro `s-value` can now take multiple slots in its argument list, like `g-value` and `gv` have always supported.
- You can now store "meta-information" in formulas. This new feature allows formulas to be thought of more like schemas, with information stored in slots of the formulas. Meta-information could also be thought of as different "views" of the formula, where the cached value of the formula is one particular view. The lambda expression associated with a formula is an example of meta-information.
- The function `ps`, used for printing the slots and values of schemas, now supports the printing of slot types. Array values are now abridged when they are printed with `ps`, so that very large arrays are not printed in their entirety. Also, `ps` now returns the value of the schema that it was given to print, instead of just returning NIL.
- A new reader macro for `#k` has been provided, so that lisp can read the `#k<...>` syntax printed by some Garnet functions (like `ps`). The new `#f` reader macro allows you to abbreviate calls to `(o-formula (...))` with `#f(...)`.
- A variety of bugs in KR have been fixed in this version. A list of the most noteworthy bug fixes are listed in section 4.7.

OPAL:

- All Opal prototypes have a new `:parameters` slot that indicates the user-settable slots for the object.
- The PS module for generating PostScript files from Garnet windows has been made more robust, and many limitations of the previous version have been eliminated. The function `opal:make-ps-file` can now put two types of simulated window-manager frames around your rendered images, and can print pixmaps in full color. A list of many more new features appears in Section 5.7.

- Several new features have been added concerning cursors of windows. Normally the cursor, also called the pointer, is in the shape of an arrow when it is over a Garnet window. Several new functions have been provided that make it easier to change the cursor of Garnet windows. One function temporarily makes the cursor into the shape of an hourglass, which can be useful when an application is performing a long computation.

- A "quarantine slot" now exists in Garnet windows. If there was a crash during the last update, then the window will stop being updated automatically along with the other Garnet windows, until you can fix the problem and update the window successfully. This should alleviate the problem of being thrown endlessly into the debugger while trying to figure out what's going wrong.

- The `window` prototype is no longer exported from Opal, and should not be referenced unless absolutely necessary. Creating instances of `opal::window` instead of `inter:interactor-window` is strongly discouraged. The `inter:interactor-window` should be the window prototype referenced by user programs.

- Window reparenting has been made more flexible, so that the `:parent` of a window can be changed after it has been created.

- The function `opal:update-all`, which calls `opal:update` on all existing Garnet windows, has been given a new *total-p* parameter, so that you can do a total update on all Garnet windows.

- Internal functions that use the `:hit-threshold` slot have been reimplemented to conform to their documented specifications. Objects that rely on the `:hit-threshold` and `:select-outline-only` slots may be affected by this change.

- The new `opal:cursor-font` object is an instance of `opal:font-from-file`. Regular text strings can be printed in this font, but its primary use is for changing the cursors of Garnet windows by indexing into the font. This is the font used by the window manager for its own cursor shapes.

- Several lighter shades of the Motif colors have been added (e.g., `opal:motif-light-blue`) along with their corresponding filling-styles. It should be easier to read text against these lighter backgrounds.

- Three utility functions have been added to Opal. `Opal:shell-exec` is used to execute Unix commands in a shell from lisp. `Opal:make-image` is used to save an image of the current lisp session. `Opal:directory-p` takes a pathname and returns T if it is a valid directory.

- Update-slots of graphical objects can no longer be changed just by editing the value of the `:update-slots` list. As a consequence of optimizing the system, there is now a function that must be called to add or remove update-slots from an object, discussed in section 4.6.1.

- Opal now properly handles the screen parameter of the DISPLAY environment variable.

- You can now stack fast redraw objects of all three types -- `:rectangle`, `:redraw`, and `T`.

INTERACTORS:

- Keyboard "accelerators" are now provided that allow you to attach functions to specific keyboard keys. Some default accelerators are bound that will raise, lower, or destroy a window when the appropriate key sequence is typed.

- Entering and leaving a window with the mouse can now be detected as an event. You can turn this feature on and off for individual windows.

- The `Agate` gesture trainer has been reimplemented, and has a new format for gesture transcripts.

- The text editing interactors now automatically stop editing the current string whenever the mouse is pressed *outside* the string. One implication of this feature is that you will typically not have to type RETURN before hitting the OK button, since the down press on the OK button will stop editing and still operate the OK button.

- Some new features have been provided with the standard Interactors debugging functions that should make debugging Interactors easier.

- A new scheme for controlling the priority of interactors has been implemented, through the `:sorted-interactors` slot of priority levels. Now you can assign your interactors numerical rankings, corresponding to their precedence over other interactors using the same priority level.

- The `:final-feedback-obj` slot of menu and button-interactors can contain a formula. This is useful for computing a different feedback object depending on the currently selected object.

- A new *verbose* key has been added to the parameter list of `inter:Transcript-Events-From-File`.

GADGETS:

- All gadget prototypes have a new `:parameters` slot that contains a list of all the user-settable slots in the gadgets.

- A collection of editing functions has been provided in the `Standard-Edit` module (in the `garnet-gadgets` package). This module works in conjunction with the `gg:Multi-Graphics-Selection` gadget to implement many standard editing routines such as cut, copy, and paste. Now users can share the same editing routines that are used by GarnetDraw, Gilt, and Marquise.

- Two new gadgets called `gg:MouseLine` and `gg:MouseLinePopup` are used to display help strings when the mouse is is held over a particular object. They are analogous to the ‘‘Mouse Documentation Line’’ on Symbolics Lisp machines, and the ‘‘Balloon Help’’ feature of the Macintosh System 7.

- There have been extensive changes to the `gg:Prop-Sheet-For-Obj` gadget that allow it to use the new `:parameters` and type information in Garnet objects.

- A new gadget called `gg:Motif-Prop-Sheet-For-Obj-With-Done` is similar to the `gg:Motif-Prop-Sheet-For-Obj-With-OK` gadget, but with a "Done" button.

- A new gadget, called the `gg:Popup-Menu-Button`, is a combination of a button and a popup menu. The menu appears when you press the button, and the menu goes away when you select an item from it.

- The `gg:motif-menu` has been enhanced to allow variable-width fonts for the menu items. Previously, the gadget could not underline the correct accelerator character in a menu item when the font was variable-width.

- The new `gg:motif-menubar` is a Motif version of the standard `gg:menubar` gadget. The interface is similar to the old `menubar`, though it is not identical. The documentation included here should be supplemented with the Gadgets Manual from the Garnet 2.1 release.

- Menubars have been made more robust. Previous implementations were rather fragile, and required strict adherence to the examples provided in the manual. Users should find the new implementations to be less fragile.

- The new `gg:motif-trill-device` gadget is a Motif version of the standard `gg:trill-device` gadget. The interface is virtually identical to the old gadget.

- The `polyline-creator` gadget now supports polyline *editing*. Calling the new function `gg:toggle-polyline-handles` on a polyline will display selection handles at all points of the polyline, allowing the user to drag the points. Adding and removing points of the polyline are also supported.
- The standard and Motif versions of the `save-` and `load-gadget` have been made more robust, and several new features have been added.
- Four new functions are provided which check user input for errors, and raise an `error-gadget` if an error is found. These functions are analogous to the less portable `ignore-errors` function.
- An `:item-to-string-function` slot has been added to the `gg:menu` gadget.
- The slots `:format-string` and `:enum-format-string` have been added to the `gg:gauge` and `gg:motif-gauge` gadgets.

DEBUGGING:

- New functions will perform various actions (like printing notification messages, or going into the debugger) whenever particular slots are set. These functions are useful when you want to know who is setting a particular slot.
- The function `gd:look-inter` has been fixed, and can be used to print useful information about interactors, including all the interactors that work on a particular Opal object.

GILT:

- A new extension of Gilt called the Value-Control module has been written that makes it easy to define constraints among the `:value` slots of the gadgets in the work-window. An analogous Enable-Control module allows you to control when gadgets are enabled and disabled (active or "grayed-out"). This should make the creation of dialog boxes significantly easier.
- Output from the GarnetDraw application can now be read into Gilt.

DEMOS:

- The `Demos-Controller` application, which is a nice interface for bringing up the other demos, now features the new `gg:MouseLine` gadget. When you keep the mouse still over one of the x-buttons for about 2 seconds, a window will pop up with a short description of the corresponding demo.
- The old `demo-calculator` has been replaced by the new `garnet-calculator`. While this module contains the standard `do-go` routine, the `garnet-calculator` is designed to be a self-contained interface tool that can be part of a larger Garnet interface.
- `GarnetDraw` uses the new `Standard Edit` module from `garnet-gadgets`, and can share "cut" and "pasted" objects with Gilt. The new version uses the new `motif-menubar` gadget with keyboard accelerators, and supports the new polyline editing feature.
- `Demo-Multifont` uses the new `motif-menubar` gadget with accelerators, and demonstrates the new, improved keyboard editing commands for `multifont-text`. A menubar selection allows you to toggle the new `lisp-mode` feature of multifont, for automatic indentation of lisp code.
- The old `Demo-Circle` has been renamed `Demo-Virtual-Agg`, and still shows a window full of overlapping circles created with an `opal:virtual-aggregate`. `Demo-Pixmap` still shows an example of a two-dimensional, non-overlapping `virtual-aggregate`.

LAPIDARY:

- Several new operations are supported in Lapidary, including the ability to supply multiple feedback objects for interactors, and the ability to write out interactors that are attached to a new gadget.
- A new self-contained version of the Box and Line Constraint dialog boxes can be used outside of Lapidary. The full documentation for this new gadget is included in Chapter 12.

2. Non-Backwards-Compatible Changes

- It is now illegal to have slot names that are neither keywords nor symbols. Applications that used other lisp objects as slot names should be edited to use keyword names for slots.
- The `:strings` slot of the `opal:multifont-text` object has been renamed `:initial-text`. The `opal:set-strings` function has been renamed `opal:set-text`.
- The internal functions that rely on the `:hit-threshold` slot (the Opal `point-in-gob` methods) have been rewritten to conform to documented specifications in the Garnet 2.1 Reference Manual. Objects that use the `:hit-threshold` or `:select-outline-only` slots may be affected. See section 5.14.
- The `opal::window` prototype is no longer exported from Opal. Users should reference the `inter:interactor-window` prototype instead.
- Update-slots of graphical objects can no longer be changed just by editing the value of the `:update-slots` list. The function `kr::add-update-slots` should be used to change the update-slots of an object (see section 4.6.1).
- Old gesture transcript files cannot be recognized by the new Agate trainer. All gesture transcripts should be regenerated from scratch.
- A reimplementation of the layout procedure for aggrelists may require some `:left` and `:top` slots of item-prototypes to be given dummy values of zero. Although the need for `opal:notice-items-changed` has been eliminated under usual circumstances, new invocations of this method may be required before requesting the location of aggrelist components. Also, dependencies among the layout parameters in aggrelists have changed so that a reassessment of constant declarations is warranted.
- When generating PostScript files from Garnet windows, the *clip-p* parameter for `opal:make-ps-file` now treats a list of coordinates as screen-based, rather than window-based.

3. Global Changes

3.1. Compiler Optimization Settings

The new variable `user::*default-garnet-proclaim*` holds a list of compiler optimization flags and default values, which determine things like the size and speed of your resulting Garnet binaries. For example, the default value of this variable in Allegro is:

```
'(optimize (speed 3) (safety 1) (space 0) (debug 3))
```

This optimization causes Allegro to generate compiled binaries that are as fast and small as possible. The *safety* setting of 1 means that the compiled code will allow keyboard interrupts if you somehow go into an infinite loop, and the *debug* setting of 3 means you will get the most helpful error messages that Allegro can give you when you are thrown into the debugger.

Different implementations of lisp require different values for the optimization flags, and `garnet-loader.lisp` provides values for Allegro, Lucid, and CMUCL that we have found work particularly well. You can override the default optimizations by redefining the `*default-garnet-proclaim*` variable before loading `garnet-loader.lisp`. A value of NIL for this variable means that you want to maintain the declarations that are already in effect for your lisp.

3.2. Garnet-Load and Garnet-Compile

The functions `user::garnet-load` and `user::garnet-compile` now look up their prefix parameters in an association list called `user::Garnet-Load-Alist`. Its structure looks like:

```
(defparameter Garnet-Load-Alist
  '(("gg" . Garnet-Gadgets-PathName)
    ("demos" . Garnet-Demos-PathName)
    ...))
```

This alist is expandable so that you can include your own prefixes and pathnames. Prefixes can be added with the following function:

`user::Add-Garnet-Load-Prefix` *prefix pathname* [*Function*]

For example, after executing `(add-garnet-load-prefix "home" "/usr/amickish/")`, you would be able to do `(garnet-load "home:my-file")`.

3.3. New Support for LispWorks

Many small modifications have been made to the Garnet code to allow it to run in the Harlequin LispWorks environment. If our LispWorks users notice any unexpected behavior from Garnet, or are able to contribute any bug fixes with respect to running on LispWorks, we would be happy to include them in future releases of Garnet.

There is one peculiarity about LispWorks that requires an additional step before executing the main-event-loop background process of Garnet. LispWorks has an unconventional "initialization phaze" to multiprocessing, which requires tha a special function be called before launching a background process. There are two ways to initialize multiprocessing in LispWorks.

One way is to start the big window-oriented LispWorks interface by executing `(tools:start-lispworks)`. This will cause a menu to appear, and you can open a lisp listener as a selection from the menu. From this listener, you can load `garnet-loader.lisp`, and Garnet's main-event-loop process will be launched by default.

If you do not need all the functionality of the LispWorks interface, you can initialize multiprocessing with much less overhead. Before loading Garnet, load the file `"src/utils/lispworks-process.lisp"` and execute the function `guarantee-processes` to start multiprocessing. For example, at the LispWorks prompt you could type:

```
> (garnet-load "utils-src:lispworks-process.lisp")
> (guarantee-processes)

;; At this point, a new lisp listener has been spawned
> (load "garnet/garnet-loader")
```

It is important to realize that when you call `guarantee-processes`, a new lisp listener is spawned, and all subsequent commands will be typed into the second listener. Putting the `guarantee-processes` call at the top of the `garnet-loader.lisp` file will not work, because the first listener will remain hung at the `guarantee-processes` call, while the second process is waiting for user input.

Whenever you enter the debugger of the new listener spawned by `guarantee-processes`, you will get restart options that include:

```
...
5 (abort) return to level 0.
6 Return to top level
7 Return from multiprocessing
```

When you want to exit the debugger, you should choose either "(abort) return to level 0," or "Return to top level", since both of these options will return you to the top-level LispWorks prompt. If you ever choose "Return from multiprocessing", then you will kill both the second listener and the main-event-loop-process, and you will have to call `guarantee-processes` and `opal:launch-main-event-loop-process` to restart Garnet's main-event-loop process.

It is not necessary to load `"lispworks-process.lisp"` or execute `guarantee-processes` if you instead choose to execute `tools:start-lispworks`.

3.4. New Support for CLISP

Thanks to Bruno Haible, Garnet now works in the CLISP environment. There is a special file in the `src/` directory called `README.CLISP` that contains special instructions for running Garnet in CLISP. If our CLISP users have any additional modifications they would like to see in Garnet that would make it run better in CLISP, we would be happy to add them to the next release.

4. KR

In this version, we have completely redesigned the internals of the object system in KR. As a result, the system is now significantly faster. Typical KR operations are up to *five times faster* than in version 2.1; operations at the Garnet user level are typically between two and three times faster.

The new version of KR also supports complete slot type-checking. Any slot in any object can now be declared of a certain type. Every time the value of such a slot changes (either directly or as a result of formula evaluation), KR checks that the new value is compatible with the declared type of the slot. This feature may be turned off completely, for maximum performance in finished systems. However, the performance overhead is small, and we recommend that you keep the type-checking enabled. This ensures early detection of problems that might otherwise be difficult to track down.

This version also supports a uniform syntax to declare properties of slots. The macro `create-instance`, in particular, now supports the new keyword :DECLARE. This keyword is used to declare the type of slots, the fact that certain slots are constant, `:maybe-constant`, update slots, parameters, ignored when printing the object with `ps`, etc. This uniform syntax unifies the different methods that were used previously to declare different features.

In addition to these major changes, we have also made many smaller changes, plus a number of bug fixes. These changes are described in more detail in the following sections.

4.1. Performance Improvements

We have completely redesigned the internals of the object system in KR to make the system about five times faster.

The primary change was in the internal representation of object slots. Different slots are now stored in different bins in an object, and bins are kept small so searching for slots is very efficient. Typical slot access times are now under 5 microseconds for simple slots that do not contain formulas. Slots are assigned to different bins based on their print names. As a result of this change, it is now illegal to have slot names that are neither keywords nor symbols (the practice of using other Lisp objects as slot names was never encouraged, although it "worked" in some versions of the system).

Another important change was the use of Lisp structures to store slots internally. This led to significant performance improvements in the internal KR code, because whole slot structures are now manipulated by the internal functions. This change eliminated many unnecessary recomputations and searches for the same slot within an object.

The distinction between special and non-special slots has also been removed. This means that any slot is now as efficient as what used to be built-in slots (e.g., `:is-a` and `:left`). Additionally, there is no longer any wasted space in objects that do not use those built-in slots.

All of these changes are completely backwards-compatible, except for the limitations to keywords or symbols as slot names. No user-level code change is necessary.

4.2. New Type System

The new version of KR supports complete type-checking for slots. Any slot in any object can now be declared of a certain type. Slots can be declared with one of the pre-defined types Garnet provides, which cover most of the commonly occurring situations, or new types may be created as needed using the macro `def-kr-type` (see below). Type expressions use the same syntax as in the Common Lisp type system. Type declarations are inherited, so it is generally not necessary to specify types for the slots of an instance (unless, of course, the instance is to behave differently from the prototype).

Every time the value of a typed slot changes, KR checks that the new value is compatible with the declared type of the slot. If not, a continuable error is generated. More specifically, the type of a value is checked against the type specification for a slot under the following circumstances:

- when the slot is first created using `create-instance`: if a value is specified and the value is of the wrong type, an error is generated;
- when a slot is set to a certain value using `s-value`;
- when the value in a slot is computed by a formula, and the formula is evaluated;
- when the type of a slot that already contains a value is changed using `s-type` (see below).

This mechanism ensures that potential problems are detected immediately; without type-checking, it would be possible for a bad value in a slot to cause hard-to-track errors later on. For example, if a slot in an object is supposed to contain an integer value, a formula in another object would typically assume that the value is correct, and compute an expression such as `(+ 10 (gv obj :left))`. If the value in the slot `:left` is incorrectly set to NIL, however, this would not cause an error until much later, when the formula is actually recomputed and the operator + is given a null value. When type-checking is enabled, on the other hand, the user would see an error immediately when the value is set to NIL, because NIL does not meet the "integer" declaration.

Type-checking may be turned off completely, for maximum performance in finished systems, by setting the variable `kr::*types-enabled*` to NIL. However, the performance overhead associated with type-checking is small, and we recommend that you always keep type-checking enabled. This ensures early detection of problems that might otherwise be difficult to track down.

4.2.1. Creating Types

New types may be declared as needed with the macro `def-kr-type`, which is exported from the KR package. The syntax of the macro is as follows:

`kr:Def-Kr-Type` *name-or-type* `&optional` *args body doc-string* [*Macro*]

This macro defines a new type for KR's type-checking mechanism. Every type used in slot declarations must have been defined with `def-kr-type`. However, Garnet already predefines the most common types, so you do not have to worry about those.

The macro may be called in two different styles, one named, one unnamed. The first style is used to define types that have a name; you may then use either the name or the corresponding expression in actual type declarations. The second style simply defines a type expression, which is not named and hence must be used verbatim in type declarations. Here are examples of the two styles:

```
(def-kr-type my-type () '(or keyword null))

(def-kr-type '(or keyword null))
```

The first style uses the same syntax as Lisp's `deftype`; the *body* should be a type expression acceptable to `deftype`, and is used for typechecking when the name is used. In the current implementation of the type system, the *args* parameter should always be NIL.[1] With either example above you could then specify some object's type to be `'(or keyword null)`. With the first style, however, you could also specify the type to be `'my-type`, which may be more convenient and easier to maintain in the long run.

[1]The presence of the *args* parameter is to maintain consistency of syntax with the standard lisp function `deftype`. If you need to pass a parameter to your predicate, then define the predicate using `satisfies`.

The named style also allows a *doc-string* to be specified. This is a human-readable documentation string that is associated with the type, and is useful for debugging purposes. For example, the first call above could be written as:

```
(def-kr-type my-type () '(or keyword null)
   "Either NIL or a keyword")
```

4.2.2. Declaring the Type of a Slot

Types are associated with slots either statically or dynamically. The former mechanism is by far the most common, and is done at object creation time using the :DECLARE option in `create-instance`. For example, consider the following code:

```
(create-instance 'R1 opal:rectangle
  :declare (:type (integer :left :top)
                  ((integer 0) :width :height)
                  ((or keyword null) :link-name))
  (:link-name :PARENT)
  (:left 10) (:height (+ 15 (o-formula (gvl :width)))))
```

The example declares that the values contained in slots `:left` and `:top` must be integers, the values in slots `:width` and `:height` must be positive integers, and the value in slot `:link-name` must be either a keyword or NIL. Note that this declaration is legal, as the type `(or keyword null)` was declared above using `def-kr-type`. Note also that the declarations for slots `:left`, `:top`, `:width`, and `:height` are, in fact, not necessary, as they would normally be inherited from the prototype.

Types can also be associated with slots dynamically, i.e., after object creation time. This is done with the function

`kr:S-Type` *object slot type* `&optional (`*check-p* `T)` [*Function*]

This function changes the type declaration for the *slot* in the *object* to the given *type*. If *check-p* is non-NIL (the default), the function signals a continuable error if the value currently in the *slot* does not satisfy the new type. Setting *check-p* to NIL disables the error. Note that this should only be used with caution, as it may leave the system in an inconsistent state (i.e., the *slot* may in fact contain an illegal value). The function returns the *type* it was given.

The type associated with a slot can be retrieved by the function

`kr:G-Type` *object slot* [*Function*]

If a type is associated with the slot, it is returned (more precisely, if the type is named, the name is returned; otherwise, the type expression is returned). If there is no type, the function returns NIL.

4.2.3. Type Documentation Strings

Given a type (for example, something returned by `g-type`), its associated documentation string can be retrieved using:

`kr:Get-Type-Documentation` *type* [*Function*]

This function returns the human-readable type documentation string, or NIL if there is none.

Given a type, it is also possible to modify its string documentation, using the function:

`kr:Set-Type-Documentation` *type doc-string* [*Function*]

This function associates the *doc-string* with the *type*. When an error message which concerns the type is printed, the documentation string is printed in addition to the raw type.

4.2.4. Explicit Type-Checking

In addition to KR's built-in type checking, which happens when the value in a slot is changed, it is also possible to check whether a value is of the right type for a slot. This can be done with the function:

`kr:Check-Slot-Type` *object slot value* `&optional (`*error-p* `T)` [*Function*]

The function checks whether the given *value* is of the right type for the *slot* in the *object*. If not, it raises a continuable error, unless *error-p* is set to NIL; in this case, it returns a string which describes the error. This function is called automatically by KR any time a slot is modified, so you normally do not have to call it explicitly.

4.2.5. Temporarily Disabling Types

It is possible to execute a piece of code with type-checking temporarily disabled, using the macro

`kr:With-Types-Disabled &body` *body* [*Macro*]

This macro is similar to others, such as `with-constants-disabled`. During the execution of the *body*, type-checking is disabled, and no errors are given if a value does not meet the type specification of its slot. Just as with `with-constants-disabled`, this macro should only be used with caution, as it may leave the system in an inconsistent state.

4.2.6. System-Defined Types

The following new type predicate can be used to declare types:

`kr:Is-A-P` *prototype* [*Type Predicate*]

This is a new type predicate, NOT a function or macro; it can only be used within type specifiers. This predicate declares that the value in a slot should be an instance of the *prototype*, either directly or indirectly. The predicate is true of all objects for which a call to the function `is-a` would return true. For example, the following definition can be used as the type of all rectangles:

```
(def-kr-type rect-type () '(is-a-p opal:rectangle))
```

Garnet defines a number of types, which cover the types of the most commonly used slots. This is the list of pre-defined basic types:

- T - Any value satisfies this type.
- KR-BOOLEAN - Same as `T`, but specifically intended for slots which take a NIL or non-NIL value, often used as boolean variables.
- NULL - Only the value NIL satisfies this type.
- STRING - Strings satisfy this type.
- KEYWORD - All Lisp keywords satisfy this type.
- INTEGER - All integers (fixnums and bignums) satisfy this type.
- NUMBER - This type includes all numbers: integers, floating point, complex numbers, and fractions.
- LIST - Any list satisfies this type.
- CONS - Any cons cell (lists and dotted pairs) satisfy this type.
- SCHEMA - Any non-destroyed KR object satisfies this type.

Garnet also defines many non-basic types, which are typically used in by many objects throughout the system. Some of the types are named for convenience; the following is the list of named types and their associated documentation strings.

KNOWN-AS-TYPE - a keyword (this type is used in the `:known-as` slot)

AGGREGATE - an instance of `opal:aggregate`

AGGREGATE-OR-NIL - either an instance of `opal:aggregate` or NIL

BITMAP - an instance of `opal:bitmap`

BITMAP-OR-NIL - either an instance of `opal:bitmap` or NIL

COLOR - an instance of `opal:color`

COLOR-OR-NIL - either an instance of `opal:color` or NIL

FONT - either an instance of `opal:font` or `opal:font-from-file`

FONT-FAMILY - one of `:fixed`, `:serif`, or `:sans-serif`

FONT-FACE - one of `:roman`, `:bold`, `:italic`, or `:bold-italic`

FONT-SIZE - one of `:small`, `:medium`, `:large`, or `:very-large`

FILLING-STYLE - an instance of `opal:filling-style`

FILLING-STYLE-OR-NIL - either an instance of opal:filling-style or NIL

LINE-STYLE - an instance of `opal:line-style`

LINE-STYLE-OR-NIL - either an instance of `opal:line-style` or NIL

INTER-WINDOW-TYPE - a single `inter:interactor-window`, or a list of windows, or T, or NIL.

WINDOW - an instance of `inter::interactor-window`

WINDOW-OR-NIL - either an instance of `inter:interactor-window` or NIL

FILL-STYLE - one of `:solid`, `:stippled`, or `:opaque-stippled`

DRAW-FUNCTION - one of `:copy, :xor, :no-op, :or, :clear, :set, :copy-inverted, :invert, :and, :equiv, :nand, :nor, :and-inverted, :and-reverse, :or-inverted, :or-reverse`

H-ALIGN - one of `:left`, `:center`, or `:right`

V-ALIGN - one of `:top`, `:center`, or `:bottom`

DIRECTION - either `:vertical` or `:horizontal`

DIRECTION-OR-NIL - either `:vertical`, `:horizontal`, or NIL

ITEMS-TYPE - list of items: ("Label2"...)

ACCELERATORS-TYPE - list of lists: ((#\r "Alt-r" #\meta-r)...)

FILENAME-TYPE - a string that represents a pathname

PRIORITY-LEVEL - an instance of `inter:priority-level`

The following types do not have a name. They are often used for slots in Opal fonts, line styles, etc. Because they are predefined, you don't need to call `def-kr-type` for them.

```
'(real 0 1)
'(integer 0 1)
'(integer 0)
'(integer 1)
'(integer 2)
'(member 0 1 2 3)
'(or null integer)
'(or null (integer 0))
'(or keyword (integer 0))
'(or number null)
'(member :even-odd :winding)
'(or (member :below :left :right) list)
'(or keyword character list)
'(or list string)
'(or list (member t))
'(or list (satisfies schema-p))
'(or string atom)
'(or string (satisfies schema-p))
'(or function symbol)
'(or list integer function symbol)
'(or null function symbol)
'(or null keyword character)
'(or null string)
'(or null (satisfies schema-p))
'(or null string keyword (satisfies schema-p))
'(or string keyword (satisfies schema-p))
```

4.2.7. Extensions to the PS Function

The function `ps` now prints out type information, if desired. This can be specified with a non-null value for the new keyword parameter *types-p* (the default value is NIL). Type declarations are printed in square brackets. For example:

```
<cl> (create-instance 'P NIL :declare (:type (integer :left :top))
       (:left 10) (:top 100))
#k<P>
<cl> (ps P)              ; prints out:
{P
  :LEFT =  10
  :TOP =  100
  }

<cl> (ps P :types-p T)     ; prints out:
{P
  :LEFT =  10  [integer]
  :TOP =  100  [integer]
  }
```

The function also has an additional keyword, *all-p* (default value NIL). If non-NIL, this keyword causes `ps` to print out all slots, including ones that do not currently have any value.

`Ps` now internally binds the variable `*print-length*`. This ensures that printing of arrays that are the value in a slot is truncated after the first 10 elements (this is what the Lisp printer does when it prints such arrays at the top level). Before this change, very large arrays would be printed out in their entirety.

Finally, the function `ps` now returns the object it was given. This allows top-level Lisp expressions such as * to be used after an object is printed.

4.3. Uniform Declaration Syntax

It is now possible to use one syntax for all kinds of declarations associated with slots in an object. Previously, most mechanisms used a somewhat different syntax, making the declaration system harder to learn and remember. While we are currently retaining the old-style syntax for backwards compatibility, we strongly encourage users to convert to the new, uniform declaration syntax whenever possible.

Declarations are generally specified at object creation time. In some cases (notably, in the case of types), it is also meaningful to modify declarations after an object has been created; in such cases, a separate function (such as `s-type`) is provided.

The general syntax for declarations in `create-instance` is as follows:

```
(create-instance instance prototype
  :DECLARE ((declaration-1 [slot1 slot2 ...])
            (declaration-2 [slot1 ...])
            ...)
  [:DECLARE ((declaration-3 [slot1 ...])
            ...)]
  slot-specifiers ...)
```

The keyword :DECLARE introduces a list of declarations. The keyword may appear more than once, which allows separate groups of declarations. Each group of declarations may contain one or more declarations; if there is only one, a level of parentheses may be omitted. Each declaration in a list consists of a keyword, which specifies what property is being declared, followed by any number of slot names (including zero). All slots are declared of the given property.

Consider the following, rather complex example:

```
(create-instance 'REC A
  :declare ((:type (vector :BOX)
                   (integer :LEFT :TOP)
                   ((or (satisfies schema-p) null) :PARENT))
            (:type ((member :yes :no) :VALUE))
            (:update-slots :LEFT :TOP :WIDTH :HEIGHT :VALUE))
  :declare (:type (list :IS-A-INV))

  (:left (o-formula (+ (gvl :parent :left) (floor (gvl :width) 2))))
  (:top 10))
```

The first declaration group defines types (in two separate lists) and the list of update-slots for the object. Slot `:box` is declared as a Lisp vector; left and top are declared as integers; slot `:parent` must be either null or a valid KR object; and slot `:value` must contain either the value :yes or the value :no. The second declaration group shows the simplified form, in which only one declaration is used and therefore the outside parentheses are dropped.
The following keywords can be used to declare different slot properties:

:CONSTANT - The slots that follow are declared constant. Note that (in this case only) the special value T indicates that the slots in the prototype's `:maybe-constant` slot should be used.

:IGNORED-SLOTS - The slots that follow will not be printed by the function `ps`.

:LOCAL-ONLY-SLOTS - The slots that follow should never be inherited by this object's instances.

:MAYBE-CONSTANT - Specifies the list of slots that can be made constant in this object's instances simply by specifying the special value T.

:OUTPUT - Specifies the list of output slots for the object, i.e., the slots that are computed by formulas and may provide useful output values for communication with other objects.

:PARAMETERS - Specifies the list of parameters for the object, i.e., the slots designed to allow users to customize the appearance or behavior of the object. This slot is used extensively in the Garnet Gadgets to indicate user-settable slots.

:SORTED-SLOTS - Specifies the list of slots (in the appropriate order) that `ps` should always print first.

:TYPE - Introduces type declarations for one or more slots, as explained above.

:UPDATE-SLOTS - The list of update slots for the object, i.e., the slots that should trigger the `:invalidate-demon` when modified.

4.3.1. Declarations in Instances

Most inherited declarations follow the standard KR scheme, where a :MAYBE-CONSTANT or :UPDATE-SLOTS declaration in an instance will completely override the declaration in the prototype. One important exception is the :TYPE declaration, which is *additive* from prototype to instance. That is, all of the types declared in a prototype will be valid in its instances, along with any new type declarations in the instance. So you do not need to repeat type declarations in the instances of an object.

For other kinds of declarations besides :TYPE, a convenient syntax has been provided for specifying declarations in instances. If you want all the declarations in a prototype to be inherited by the instance along with several new ones, you could either retype all the declarations in the instance, or you could use the `T` and `:except` syntax. This is the same syntax that has been in use for the :CONSTANT declaration. For example, it is now possible to write

```
(create-instance 'REC A
  :declare ((:output T :new-slot)
            (:parameters T :except :left)))
```

to indicate that object REC's list of output slots includes all the ones declared in object A, plus the `:new-slot`. Also, the list of parameter slots is equal to the one in A, minus the slot `:left`.

Declarations made in a prototype can be eliminated with an empty declaration in an instance. This may be particularly convenient for declarations such as :MAYBE-CONSTANT. For example, the expression

```
:declare ((:TYPE) (:MAYBE-CONSTANT))
```

in a call to `create-instance` would clear the :MAYBE-CONSTANT declarations from the prototype, and eliminate all type declarations.

However, note that redefining the :CONSTANT declaration may not yield the expected results. When a slot becomes constant in a prototype, that slot will be constant for all instances. This makes sense because any formulas in the prototype that relied on the constant slot have been eliminated, and cannot be restored in the instance. See the "Constants" section in the Garnet 2.1 KR Reference Manual for a more elaborate discussion of constant slots.

4.3.2. Examining Slot Declarations

The following functions may be used to determine what slot declarations are associated with a particular slot in an object, or to retrieve all slot declarations for an object. Note that there is no function to alter the declaration on an object after the object has been created, as most properties can only be set meaningfully at object creation time.

`kr:Get-Declarations` *object selector* [*Function*]

Returns a list of all the slots in the *object* that have associated declarations of the type given by *selector*, which should be one of the keywords listed above. If *selector* is `:type`, the return value is a list of lists, such as

```
((:left (or integer null)) (:top (or integer null)))
```

If *selector* is one of other keywords, the function returns a list of all the slots that have the corresponding declaration.

`kr:Get-Slot-Declarations` *object slot* [*Function*]

This function returns a list of all the declarations associated with the *slot* in the *object*. The list consists of keywords, such as :CONSTANT and :UPDATE-SLOT, or (in the case of type declarations) a list of the form `(:type` *type-specification*`)`.

4.4. Extensions to gv, s-value, and g-local-value

It is now possible to use the macro `gv` at the top level (i.e., outside any formula). Used this way, `gv` does not record any dependency, and effectively behaves exactly like `g-value`. Now, the only time you need to use `g-value` is in the rare case when you explicitly want to avoid establishing a dependency on a slot from inside a formula. This additional flexibility should be especially useful for novice Garnet users, who will only have to learn about `gv`.

The macro `s-value` may now be given more than one slot in its argument list, to access a slot that is several levels away from an object. This makes it more like `g-value`: all slots but the last one are used to access objects, and then the value of the last slot is set in the resulting object. For example:

```
(s-value object :parent :parent :left 100)
```

sets the `:left` slot of the object's parent's parent to 100. An appropriate error message is given if any of the intervening objects is found to be NIL.

One of the advantages of this extension is that it supports `setf` with a multi-level `g-value` expression; for instance, it is now possible to write

```
(setf (g-value object :parent :parent :left) 100)
```

A new (non-exported) macro is available which can be used to prevent the evaluation of `gv` and `gvl` inside formulas from setting up dependencies. Inside the body of the macro, `gv` and `gvl` effectively behave (temporarily) exactly like `g-value`. The syntax is:

`kr::With-Dependencies-Disabled &body` *body* [*Macro*]

This macro should be used with great care, as it may cause formulas not to be re-evaluated if dependencies are not set up correctly.

We have also defined a `setf` form for the function `g-local-value`, which simply expands into `s-value`. This makes the function more similar to others such as `g-value`.

4.5. Formula Meta-Information

It is now possible to associate arbitrary information (sometimes known as meta-information) with formulas, for example for documentation or debugging purposes. Meta-information is internally represented by a KR object which is associated with the formula; this allows essentially any slot to be added to formulas. Meta-information can be inherited from parent formulas, and is copied appropriately by functions such as `copy-formula`.

In addition, it is now possible to access built-in formula information (such as the lambda expression that was used to create the formula) using exactly the same mechanism that is used to access meta-information. This provides a single, well-documented way to access all information associated with a formula.

4.5.1. Creating Meta-Information

Meta-information can be specified statically at formula creation time, and also dynamically for already existing formulas. Static meta-information is specified by additional parameters to the functions `formula` and `o-formula`. The additional parameters are slot specifications, in the style of `create-instance` (except that, of course, special `create-instance` keywords such as :DECLARE or :OVERRIDE are not supported). For example, the expression:

```
(o-formula (gv a :top) 15
  (:creator 'GILT) (:date "today"))
```

creates a new formula with initial value 15, and two meta-slots named `:creator` and `:date`.

Note that in order to specify meta-information statically, one has to specify the default initial value for the formula, which is also an optional parameter.

Meta-information may also be created dynamically, using the function

`kr:S-Formula-Value` *formula slot value* [*Function*]

This function sets the value of the meta-slot *slot* in the *formula* to be the specified *value*. If the *formula* does not already have an associated meta-object, one is created.

It is not possible to use this function to alter one of the built-in formula slots, such as the formula's lambda expression or its list of dependencies.

4.5.2. Accessing Meta-Information

Meta-information can be retrieved using the function `g-formula-value`. In addition to slots that were specified explicitly, this function also makes it possible to retrieve the values of all the special formula slots, such as the formula's parent or its compiled expression.

`kr:G-Formula-Value` *formula slot* [*Function*]

The function returns the value of meta-slot *slot* for the *formula*. If the latter is not a formula, or the meta-slot is not present, the function returns NIL. If the *formula* inherits from some other formula, inheritance is used to find the meta-slot.

As a convenience, *slot* can also be the name of an internal formula slot, i.e., one of the structure slots used by KR when handling formulas. Such slots should be treated strictly as read-only, and should never be modified by application programs. The built-in slot names are:

:DEPENDS-ON - returns the object, or list of objects, on which the formula depends.

:SCHEMA - returns the object on which the formula is currently installed.

:SLOT - returns the slot on which the formula is currently installed.

:CACHED-VALUE - returns the current cached value of the formula, whether or not the formula is currently valid.

:VALID - returns T if the formula is currently valid, NIL otherwise.

:PATH - returns the path accessor associated with the formula, if any.

:IS-A - returns the parent formula, or NIL if none exists.

:FUNCTION - returns the compiled formula expression.

:LAMBDA - returns the original formula expression, as a lambda list.

:IS-A-INV - returns the list of formulas that inherit from the *formula*, or NIL. If there is only one such formula, a single value (not a list) is returned.

:NUMBER - returns the internal field which encodes the valid/invalid bit, and the cycle counter.

:META - returns the entire meta-object associated with the formula, or NIL if none exists.

The function `ps` has been extended so that when given a formula, it prints any associated meta-information. The latter is printed as a schema, immediately after the formula itself. For example:

```
<cl> (create-instance 'A NIL
       (:left (o-formula (gvl :parent :left) 100
                         ;; Supply meta-information here
                         (:name "Funny formula")
                         (:creator "Application-1"))))
#k<A>

<cl> (ps (get-value A :left))          ; prints the following:
{F8
  lambda:        (gvl :parent :left)
  cached value:  (100 . NIL)
  on schema A, slot :LEFT
  }
  ---- meta information (S7):
{S7
  :NAME =  "Funny formula"
  :CREATOR =  "Application-1"
  }
```

4.6. Other Changes

4.6.1. Internal Representation of Update Slots

The internal representation of update slots has been changed to improve the system's performance. Update slots, i.e., slots that should trigger the invalidate and the pre-set demon when modified, are internally represented by a bit associated with each slot. As a consequence, it is no longer sufficient to modify the value of the slot :UPDATE-SLOTS to make a slot into an update slot after object creation time. Instead, it is necessary to call the function:

`kr::Add-Update-Slots` *object slot* `&optional (`*turn-off* `NIL)` [*Function*]

If *turn-off* is NIL (the default), the *slot* in the *object* is declared as an update slot; if *turn-off* is non-NIL, the slot is no longer an update slot. In addition to setting or resetting the internal bit, the function also modifies the :UPDATE-SLOTS slot accordingly, by adding or removing the *slot* from the list.

4.6.2. New Reader Macros

A new reader macro is defined for the `#k<...>` notation, which is produced when the variable `kr::*print-as-structure*` is non-NIL (the default). This macro allows objects written with the `#k` notation to be read back in as a KR object.

If necessary, this feature may be disabled by recompiling KR after pushing the keyword `:no-kr-reader` onto the `*features*` list.

A second reader macro is defined for convenience. This reader macro allows o-formulas to be entered using the `#f()` notation, which expands into a call to `o-formula`. For example, one may write:

```
(s-value a :left #f(gvl :top))
```

instead of the equivalent expression

```
(s-value a :left (o-formula (gvl :top)))
```

4.6.3. Slot Printing Functions

It is now possible to use the basic mechanism used by the function `ps` to print or format objects in a customized way. This facility is used by applications such as the `garnet-debug:Inspector`, which need full control over how objects are displayed. This mechanism is supported by the following function.

`kr::Call-On-PS-Slots` *object function* `&key (`*control* `T)` *inherit* [*Function*]
`(`*indent* `NIL)` *types-p all-p*

The *function* is called in turn on each slot that would be printed by `ps`. The keyword arguments have exactly the same meaning as in `ps`. The *function* should take nine arguments, as follows:

```
(lambda (object slot formula is-inherited valid real-value types-p bits indent limits))
```

When the function is called, the first argument is the object being displayed; the second argument is bound to each slot in the object, in turn. The *formula* is set to NIL (for slots that contain non-formula values), or to the actual formula in the *slot*. The parameter *is-inherited* is T if the value in the *slot* was inherited, NIL if the value was defined locally. The parameter *valid* is NIL if the *slot* contains a formula whose cached value is invalid; it contains T if the slot contains a valid formula, or any non-formula value. The parameter *real-value* is whatever g-value would actually return. The parameter *types-p* is set to T if the *function* should process type information for the *slot*; its value simply reflects the value passed to `kr::call-on-ps-slot`. The parameter *bits* contains the internal bitwise representation of the slot's features and type, as an integer. The parameter *indent* is the level of indentation. The parameter *limits* is a number (the maximum number of values from the *slot* that are to be processed by the *function*), or NIL if all values in the slot should be processed.
A similar function is used when only one slot in an object is to be processed:

`kr::Call-On-One-Slot` *object slot function* [*Function*]

This function returns T if the slot exists and the *function* was called, and NIL otherwise.

4.6.4. Converting to O-Formulas

A new function was added that modifies formulas created using the function `formula` to behave as if they were created using `o-formula`. This is useful for tools like Lapidary that need to construct formulas on the fly. The converted formulas will be handled properly by functions such as `opal:write-gadget`. It is also possible to specify that the formula's expression be compiled during the transformation. The syntax is:

`kr::Make-Into-O-Formula` *formula* `&optional` *compile-p* [*Function*]

The given *formula* is modified and will look as though it had been created by `o-formula`. If *compile-p* is non-NIL, the *formula*'s expression is also compiled in the process.

4.6.5. Tracking Formula Dependencies

The new function `kr::i-depend-on` can be used to find out all the objects and slots upon which a certain formula depends directly. The syntax is:

`kr::I-Depend-On` *object slot* [*Function*]

If the *slot* in the *object* does not contain a formula, this function returns NIL. Otherwise, the function returns a list of dotted pairs of the form `(obj . slot)`, which contains all the slots upon which the formula depends. Note that this is the list of only those slots that are used by the formula directly; if some of those slots contain other formulas, `kr::i-depend-on` does not pursue those additional formulas' dependencies.

Example:

```
(create-instance 'A NIL (:left 7))
(create-instance 'B A (:left 14) (:top #f(+ (gvl :left) (gv a :left))))
(g-value b :top)        ; set up the dependencies

(kr::i-depend-on B :top)
    ==> ((B . :LEFT) (A . :LEFT))
```

4.6.6. Old Values of Formulas

A new function has been provided that returns the old value of a formula. This function should be called from within a `formula` or `o-formula` expression.

`kr::Self-Old-Value` [*Function*]

An important application of this function is to reduce the amount of consing that occurs when a function returns a list. For example, if a formula returns a list of structures, the naive implementation would be to always cons up a list as the return value of the expression. However, by using `kr::self-old-value`, you could get the old list of structures, and iterate over the elements, setting the fields of the old structures with new values.

4.6.7. Compiling of Formula Expressions

An important aspect of using `formula` was present in Garnet 2.1, but was inadvertently left undocumented. When a formula is created using `formula`, the lambda expression for the formula is compiled. (This is in contrast to `o-formula`, which compiles its lambda expression at compile-time, along with the rest of the file.) This may cause noticable delays at run-time, when objects and their formulas are being created.

In Garnet versions 2.0 and older, the `formula` expressions were not compiled at all.

4.7. Bug Fixes

We have fixed a number of bugs that were found in the previous version. Most of these changes fixed rather obscure problems. The following are the most noteworthy bug fixes.

Several of the macros defined by KR used to bind commonly used symbols, such as `schema` and `slot`. This could cause bugs for programs that used those symbols as local variable names; the problem was especially severe for the symbol `schema`, which is exported from the KR package and therefore could easily "capture" local bindings. All the exported macros defined by KR now use `gensym` to create local variables, eliminating any chance for undesired interactions with user variable names.

We fixed a problem which caused `gv` to refuse `:self` as a valid first argument. It is now possible to write expressions such as

```
(o-formula (gv :self :left))
```

which behaves exactly like `(o-formula (gvl :left))`.

We replaced `break` with `cerror` in all the functions of the `g-value` family. This generates a true continuable error, and can be handled programmatically if so desired.
We added better error checking to `create-instance` for illegal arguments. In particular, the code now gives an error for expressions such as

```
(create-instance NIL 'PROTO)
```

where the prototype name is incorrectly quoted.

We also added a check for the case when a slot contains a formula that was never evaluated, and the value of the slot is set directly with `s-value`. This causes the formula to become valid, and because dependencies were never set up, the formula will never be evaluated. The current code detects this condition and prints a warning, which informs the user about the fact that the formula will never be evaluated.

5. Opal

5.1. Multifont

NOTE: This section is written to completely replace the multifont documentation in the Garnet 2.1 Opal manual. We begin with a summary of the new features and non-backwards-compatible changes in this new version of `multifont-text`, and continue with the complete documentation for the new version.

New Features:

- Segments of the multifont string can now have different colors. The syntax for changing the color of a segment is analogous to the syntax for changing fonts. There is a new function for changing the color of the selected region.
- The multifont object can now contain marks in the string. Marks are special characters that can hold a place in the string, and can be useful for indicating region boundaries.
- Other Garnet objects (like text-buttons) can be imbedded in a multifont string. This feature is useful for creating hypertext-like applications.
- If you are typing lisp code into the multifont object, the string will be automatically indented according to lisp indentation conventions. Also, when you type a closing right parenthesis, the multifont object will indicate which open left parenthesis is being matched. These features are activated by the `:lisp-mode-p` and `:match-parens-p` slots of the multifont interactors.
- String segments can now contain newlines. The object will automatically break the string into separate fragments for you.
- There are many new editing commands, and the default keybindings have been changed to be more "rational".
- The `multifont-text-interactor` now supports drag-through selection.
- The new slot `:after-cursor-moves-func` can contain a function to be executed whenever the cursor is moved in the multifont text string. This feature is used by the `multifont-text` object when it is in lisp mode to highlight matching parentheses.

Non-Backwards-Compatible Changes

- The `:strings` slot has been renamed `:initial-text`, since this slot is not maintained dynamically while the string is being edited.
- The `opal:set-strings` function has been renamed to `opal:set-text`, to be consistent with the new `:initial-text` slot name.

The New opal:Multifont-Text

```
(create-instance 'opal:Multifont-Text opal:aggregate
   (:left 0)
   (:top 0)
   (:initial-text ...)
   (:word-wrap-p NIL)
   (:text-width 300)
   (:current-font ...)
   (:current-fcolor ...)
   (:current-bcolor ...)
   (:fill-background-p T)
   (:show-marks NIL))
```

The `multifont-text` object is loaded by default, since it is used by the new `garnet-debug:Inspector`. If you are not already loading the `Inspector`, you can load `multifont-text` and all of its interactors with `(garnet-load "opal:multifont-loader")`.

The `opal:multifont-text` object is designed to allow users to create more complicated editing applications. The object is similar to the `opal:cursor-multi-text` with many added abilities. As the name implies, the `opal:multifont-text` object can accept text input in multiple fonts. Also, the object has a word wrap mode to permit word-processor-like editing as well as the ability to highlight text for selection.

Positioning the object is performed with `:left` and `:top` as with most Garnet objects. The slots `:width` and `:height` are read-only and can be used to see the size of the object, but should not be changed by the user. The `:initial-text` slot is used to initialize the contents of the `multifont-text`. The format of the `:initial-text` slot is complicated enough that the next section is devoted to discussing it. If the user is not particular about the font of the initial contents, a simple string is sufficient for the `:initial-text` slot. The slots `:word-wrap-p` and `:text-width` control the word wrap mode. If `:word-wrap-p` is T, the text will wrap at the pixel width given in the `:text-width` slot. If `:word-wrap-p` is NIL, word wrap mode will not be activated and no wrapping will occur. In this case, your string should contain `#\newline`s wherever required. Both `:word-wrap-p` and `:text-width` can be modified at run time.

The `:current-font` slot can be used to control what font newly added characters will appear as. Also, the `:current-font` slot can be polled to determine the last font of the character the cursor most recently passed over. The slots `:current-fcolor` and `:current-bcolor` act similarly for the foreground and background colors of the text. The slot `:fill-background-p` controls the background of the characters. If `:fill-background-p` is T, the background of the character will be drawn in the `:current-bcolor`. If `:fill-background-p` is NIL, the background of the glyphs will not be drawn at all (allowing whatever is behind the multifont text object to show through). The slot `:show-marks` turns on and off the visibility of text marks. If `:show-marks` is T, text-marks will be visible, appearing as little carats pointing to the character to which they are stuck. When `:show-marks` is NIL, the marks will be invisible.

Along with the multi-font text object are a pair of special interactors that make them editable (see section 5.1.3). The font object and the two interactors are combined into the `multifont-gadget` gadget for convenience (section 5.1.6).

There are two demos that show off multifont capabilities. `Demo-text` shows how to use the `multifont-text` object with the `multifont-text-interactor`. `Demo-multifont` shows how to use multiple text fields in a single window with the `focus-multifont-textinter` and `selection-interactor`, and demonstrates the new indentation and paren-matching features of lisp mode.

5.1.1. Format of the :initial-text Slot

The format used in the `:initial-text` slot of `multifont-text` is also used by many of the procedures and functions that can be called using the multifont object.

In its simplest form, the `:initial-text` format can be a single string. In this form, the default font and colors are used. The simplest values for `:initial-text` are:

```
"Here is my example string."

"An example string
with multiple lines."
```

All other formats require a list structure. The outermost list is the list of lines: `(list line1 line2 ... )`. A line can either be a string in which case the the default font and colors are used, or a line can be a list of fragments: `(list frag1 frag2 ... )`. Each line acts as though it ends with a newline character. If the `multifont-text` has word wrap activated, each line will also be broken at places where the length of the text exceeds the `:text-width`, thus the user need not compute how to break up the text to be placed in the window. A fragment is the unit that allows the user to enter font data into the `:initial-text` format. A fragment can be one of the following:

- a string, in which case the defaults are used.
- a "cons"ing of a string with a Garnet font: `(cons "string" garnet-font)`.
- a list of a string, font, foreground color, and background color: `(list "string" font f-color b-color)`. If *font* or *color* is NIL, the default will be used.
- a `view-object` (see 5.1.2.6).
- a mark, in the form `(list :mark sticky-left name info)` (see 5.1.2.7).

Note that only the fragment level contains font or color information. For instance, a single line in bold font may look like this:

```
'((("Here is my example string" . ,(opal:get-standard-font :fixed :bold :medium))))
```

Here is a set of sample values for the `:initial-text` slot. Each of these examples are pictured in Figure 5-1. Details on using fonts, colors, marks, and graphical objects are given in section 5.1.2.

```
; Define some fonts for brevity, and a circle to use in a string.
(setf ITALIC (opal:get-standard-font :fixed :italic :medium))
(setf BOLD   (opal:get-standard-font :fixed :bold :medium))
(create-instance 'MY-CIRCLE opal:circle)

; A pair of lines.  Both lines are strings.
'("An example string" "with multiple lines")

; Same pair of lines in italics.
'((("An example string" . ,ITALIC))
  (("with multiple lines" . ,ITALIC)))

; A single line with multiple fragments.  Note fragments can be strings
; when default font is desired.
'(("Here " ("is" . ,ITALIC) " my " ("example" . ,BOLD) " string."))

; A single line containing a graphical object
'(("Here is a circle:" ,MY-CIRCLE))

; A single line with colored fragments
'(("Here is " ("yellow" ,BOLD ,opal:yellow) " and " ("red" ,BOLD ,opal:red) " text"))

; A single line with marks.  Note: make marks visible by setting :show-marks to T.
'(("The " (:mark NIL) "(parentheses)" (:mark T) " are marked")))
```

5.1.2. Functions on Multifont Text

The `opal:multifont-text` differs from most objects in that it has a great number of functions that operate on it. The functions range from mundane cursor movement to complicated operations upon selected text. Very few operations can be performed by manipulating the slots of a multifont object.

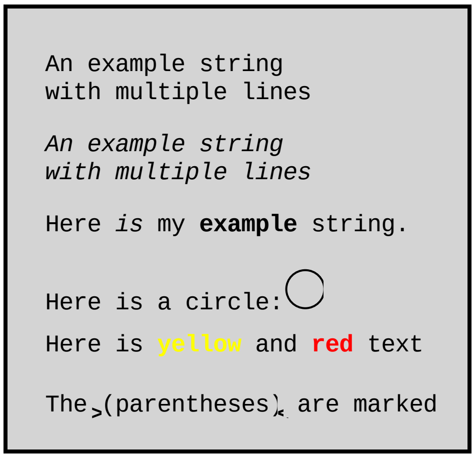


Figure 5-1: Examples of the multifont-text object

5.1.2.1. Functions that Manipulate the Cursor

`opal:Set-Cursor-Visible` *text-obj vis* [*Function*]

This makes the cursor of a `multifont-text` visible or invisible, depending on whether *vis* is T or NIL. Having a visible cursor is not required for entering text, but is recommended for situations requiring user feedback. This function does not return any useful value.

`opal:Set-Cursor-To-X-Y-Position` *text-obj x y* [*Function*]

`opal:Set-Cursor-To-Line-Char-Position` *text-obj line# char#* [*Function*]

These move the cursor to a specific location in the `multifont-text`. The function `set-cursor-to-x-y-position` sets the cursor to the position nearest the <x, y> pixel location. The function `set-cursor-to-line-char-position` tries to place the cursor at the position indicated (zero-based). If the line or character position is not legal, it will try to find a reasonable approximation of the location given. Neither function returns any useful value.

`opal:Go-To-Next-Char` *text-obj* [*Function*]

`opal:Go-To-Prev-Char` *text-obj* [*Function*]

`opal:Go-To-Next-Word` *text-obj* [*Function*]

`opal:Go-To-Prev-Word` *text-obj* [*Function*]

`opal:Go-To-Next-Line` *text-obj* [*Function*]

`opal:Go-To-Prev-Line` *text-obj* [*Function*]

These functions move the cursor relative to where it is currently located. The functions `go-to-next-char` and `go-to-prev-char` move the cursor one character at a time. The functions `go-to-next-word` and `go-to-prev-word` move the cursor one word at a time. In this case, a word is defined by non-whitespace characters separated by whitespace. A whitespace character is either a space or a newline. These functions will skip over all non-whitespace until they reach a whitespace character. They will then skip over the whitespace until they find the next non-white character. The functions `go-to-next-line` and `go-to-prev-line` moves down and up one line at a time. The horizontal position of the cursor will be maintained as close as possible to its position on the original line. The functions `go-to-next-char`, `go-to-prev-char`, `go-to-next-word`, and `go-to-prev-word` all return the characters that were passed over including newlines as a simple string. NIL will be returned if the cursor does not move as a consequence of being at the beginning or end of the text. The functions `go-to-next-line` and `go-to-prev-line` do not return useful values.

`opal:Go-To-Beginning-Of-Line` *text-obj* [*Function*]

`opal:Go-To-End-Of-Line` *text-obj* [*Function*]

`opal:Go-To-Beginning-Of-Text` *text-obj* [*Function*]

`opal:Go-To-End-Of-Text` *text-obj* [*Function*]

These functions move the cursor to a position at the beginning or end of something. The functions `go-to-beginning-of-line` and `go-to-end-of-line` move the cursor to the beginning or end of its current line. The functions `go-to-beginning-of-text` and `go-to-end-of-text` move the cursor to the beginning or end of the entire document. None of these functions return a useful value.

5.1.2.2. Functions for Text Selection

`opal:Toggle-Selection` *text-obj mode* [*Function*]

This will turn off and on the selection mode. When selection mode is on, moving the cursor will drag the selection highlight to include characters that it passes over. Moving the cursor back over selected text will unselect and unhighlight the text. Setting *mode* to T turns on selection mode, and setting it to NIL turns off selection mode. Turning off selection mode will unhighlight all highlighted text.

`opal:Set-Selection-To-X-Y-Position` *text-obj x y* [*Function*]

`opal:Set-Selection-To-Line-Char-Position` *text-obj line# char#* [*Function*]

These functions are similar to the functions `set-cursor-to-x-y-position` and `set-cursor-to-line-char-position`. The selection highlight has two ends. One end is bound by the cursor; here, the other end is called the selection end. To move the cursor end of the highlight, use the cursor functions. To move the selection end, use these two functions. The function `set-selection-to-x-y-position` sets the selection end based on pixel position. The function `set-selection-to-line-char-position` is based on line and character position. Neither function returns a useful value.

`opal:Copy-Selected-Text` *text-obj* [*Function*]

`opal:Delete-Selection` *text-obj* `&optional` *lisp-mode-p* [*Function*]

These functions are used to manipulate the selected text. The `copy-selected-text` function just returns the selected text without affecting the multifont object. The function `delete-selection` removes all selected text from the multifont object and returns it. Both functions return the text in the `text` format described above. The function `delete-selection` will also automatically turn off selection mode. Since special bookkeeping is done to keep track of parentheses and function names in lisp-mode, you must supply a value of T for *lisp-mode-p* when the interactors currently working on the *text-obj* are in lisp-mode.

`opal:Change-Font-Of-Selection` *text-obj font* `&key` *family size italic bold* [*Function*]

The font of selected text can be updated using this function. There are two options. The new font can be given explicitly using the *font* parameter, or it can be updated by setting *font* to NIL and using the key parameters.

Valid values for *family* are:

- `:fixed` - makes font fixed width
- `:serif` - makes font variable-width with "serifs" on the characters
- `:sans-serif` - makes font variable-width with no serifs on the characters

Values for *size* are:

- `:small` - makes font smallest size
- `:medium` - makes font medium size

- `:large` - makes font large size
- `:very-large` - makes font the largest size
- `:bigger` - makes font one size larger than it is
- `:smaller` - makes font one size smaller than it is

Values for *italic* and *bold* are:

- `T` - makes font italic or bold
- `NIL` - undoes italic or bold
- `:toggle` - toggles italic or bold throughout the selected region.
- `:toggle-first` - looks at the first character of the selection, and changes the entire region by toggling based on the bold or italic of that character

The function `change-font-of-selection` is also used to change the value of the slot `:current-font` even if there is no text selected.

`opal:Change-Color-Of-Selection` *text-obj foreground-color background-color* [*Function*]

This function will change the color of the selected text. If only one of foreground-color and background-color needs to be changed, the other should be sent as NIL. This function also changes the values of the slots `:current-fcolor` and `:current-bcolor`.

5.1.2.3. Functions that Access the Text or Cursor

`opal:Get-String` *text-obj* [*Function*]

`opal:Get-Text` *text-obj* [*Function*]

These functions return the entire contents of the `multifont-text` object. The function `get-string` returns the contents as a single string with `#\newlines` separating lines. The function `get-text` returns the contents in the `:initial-text` slot format.

`opal:Get-Cursor-Line-Char-Position` *text-obj* [*Function*]

`opal:Get-Selection-Line-Char-Position` *text-obj* [*Function*]

These return the position of the cursor or the selection end of a highlight. The values are returned using multiple return values: (*values line char*).

`opal:Fetch-Next-Char` *text-obj* [*Function*]

`opal:Fetch-Prev-Char` *text-obj* [*Function*]

These return the character before or after the cursor. The function `fetch-next-char` returns the character after the cursor, and `fetch-prev-char` returns the character before the cursor. Neither function affects the text of the object. The functions will return NIL if the cursor is at the beginning or end of the text where there is no character before or after the cursor.

5.1.2.4. Adding and Editing Text

`opal:Add-Char` *text-obj char* `&optional` *font foreground-color background-color lisp-mode-p* [*Function*]

`opal:Insert-String` *text-obj string* `&optional` *font foreground-color background-color* [*Function*]

`opal:Insert-Text` *text-obj text* [*Function*]

These functions are used to add text to a multifont object. The function `add-char` adds a single character, the function `insert-string` adds a whole string possibly including newline, and `insert-text` adds text that is in `:initial-text` slot format.

The optional *font* and *color* parameters indicate the font and color of the new text. If any of these parameters are NIL, the newly added text will use the value of the `:current-font`,

`:current-fcolor`, and/or `:current-bcolor` slots, which can be set manually or allowed to take on the font and colors of the character over which the cursor last passed.

The optional *lisp-mode-p* argument indicates whether the interactors currently working on the multifont object are in lisp-mode. Extra operations are performed on the string to keep track of parentheses and function names when in lisp-mode, and this parameter is required to keep the bookkeeping straight.

`opal:Delete-Char` *text-obj* [*Function*]

`opal:Delete-Prev-Char` *text-obj* [*Function*]

`opal:Delete-Word` *text-obj* [*Function*]

`opal:Delete-Prev-Word` *text-obj* [*Function*]

These functions are used to delete text from a multifont object. The functions `delete-char` and `delete-prev-char` delete a single character after or before the cursor. The functions `delete-word` and `delete-prev-word` delete a single word. A word is defined the same way as in the functions `go-to-next-word` and `go-to-prev-word`. The word will be deleted by deleting whitespace characters up to the first non-whitespace character and then deleting all non-whitespace up to the next whitespace character. The value returned by these functions is the characters deleted. NIL is returned if no characters are deleted.

`opal:Delete-Substring` *text-obj start-line# start-char# end-line# end-char#* [*Function*]

`opal:Kill-Rest-Of-Line` *text-obj* [*Function*]

These functions are used to delete larger portions of text. The function `delete-substring` removes all characters within the given range. If the start position is after the end position, nothing will happen. The function `kill-rest-of-line` deletes all characters from the cursor to the end of the current line. When word wrap is on, the end of a wrapped line is where the wrap occurs. Both functions return the deleted text as a string.

`opal:Set-Text` *text-obj text* [*Function*]

This function is used to reset everything in the multifont object. All previous text is deleted and the new *text* is put in its place. The *text* parameter uses the `:initial-text` slot format. The new cursor position will be at the beginning of the text. This function does not return a useful value.

5.1.2.5. Operations on :initial-text Format Lists

`opal:Text-To-Pure-List` *text* [*Function*]

`opal:Pure-List-To-Text` *list* [*Function*]

These functions converts text in the `:initial-text` slot format into a format that is similar but uses a list representation for fonts, colors, marks, and view-objects. Converting the fonts from Garnet objects to lists makes operations such as reading or writing text objects to files easier. To convert from `:initial-text` format to list use `text-to-pure-list` and to convert back use `pure-list-to-text`.

`opal:Text-To-String` *text* [*Function*]

This function converts text in the `:initial-text` format into a regular character string, losing all font, color, and mark information.

`opal:Concatenate-Text` *text1 text2* [*Function*]

This function is like the lisp function `concatenate` for arrays. The function will return the concatenation of *text2* onto the end of *text1*. The function will not affect *text1* or *text2*.

5.1.2.6. Using View-Objects as Text

`opal:Add-Object` *gob object* [*Function*]

`opal:Get-Objects` *gob* [*Function*]

`opal:Notice-Resize-Object` *object* [*Function*]

These functions are useful when you want to include a shape or other view-object in the multifont text. The function `add-object` will insert a view-object at the cursor. The object will act just like a character; the cursor can move over it, and it can be selected, deleted, etc. The function `get-objects` will return a list of all the objects currently in the text. When the size of an object which is in the text changes, the function `notice-resize-objects` should be used to notify multifont of the change.

5.1.2.7. Using Marks

Another feature of the multifont object is the ability to use text-marks. The function `insert-mark` will insert a mark at the cursor. Marks are invisible to the cursor as you are typing, and are primarily used as place-holders in the text. The lisp-mode feature uses marks to keep track of parentheses when it is paren-matching. To make all of the marks in a multifont object visible (so you can see them), set the `:show-marks` slot to `T`.

`opal:Insert-Mark` *gob sticky-left* `&key` *name info* [*Function*]

The *sticky-left* parameter should be T if the mark should stick to the character on its left, and NIL if it should stick to the one on its right. When a mark "sticks" to a character, the cursor cannot be inserted between the character and the mark. This makes the position of the mark equivalent to the position of the character, so it is easy to determine whether the cursor is on the left or right side of the mark.

One implication of "stickiness" is that a mark moves through the string along with the character that it is stuck to (i.e., if you are typing with the cursor in front of the mark, the mark will be pushed forward along with the character in front of it). Another implication is that when a character is deleted, the mark(s) stuck to it will be deleted as well.

The *name* parameter is a useful way to differentiate between marks, and *info* can be used to let the mark carry any additional information that might be useful.

`opal:Search-For-Mark` *gob* `&key` *name info* [*Function*]

`opal:Search-Backwards-For-Mark` *gob* `&key` *name info* [*Function*]

`opal:Between-Marks-P` *gob* `&key` *name info* [*Function*]

The functions `search-for-mark` and `search-backwards-for-mark` will return the mark which is nearest to the cursor. Leaving out the keywords will search for any mark, or include a *name* or *info* to search for a specific type of mark. The function `between-marks-p` can help to use marks as a type of region. It will search right and left, and will return T if the mark found to the left is sticky-left and the one on the right is sticky-right.

5.1.3. Interactors for Multifont Text

It may seem strange to find a section about interactors in the Opal chapter, Since the interactors mentioned here are integral to using the `opal:multifont-text` object, it was decided to include their description here, near the description of the `multifont-text`. If you are not familiar with the basic principles of interactors, you will be best served if you read the interactors manual first, particularly the parts about the `inter:text-interactor` and the slots of all interactors.

There are three interactors for multifont-text objects. The `multifont-text-interactor` is similar to the standard `text-interactor`, and is used in much the same way. Two other interactors, the `focus-multifont-textinter` and `selection-interactor` are designed to work together in more complicated situations, like when there are two or more multifont objects being edited in the same window.

The convenient `multifont-gadget` (section 5.1.6) combines the `focus-multifont-textinter` and `selection-interactor` with a `multifont-text` object, so you might be able to use it rather than explicitly creating the interactors below. However, the gadget is only useable when you have exactly one `multifont-text` object in a window. If you want more than one text object, then you should create the interactors explicitly because there should still be only one pair of *interactors* in each window, and the interactors should be set up so the `:start-where` will return one of the multifont objects. So, it could be an `:element-of...` type specification or a `:list-of...` or whatever that will return multifonts, just so long that it doesn't return other types of objects.

5.1.3.1. Multifont Text Interactor

```
(create-instance 'inter:Multifont-Text-Interactor inter:text-interactor
   (:edit-func #'inter::MultiFont-Text-Edit-String)
   (:lisp-mode-p NIL)              ; See section 5.1.3.4
   (:match-parens-p NIL)           ; "    "     "
   (:match-obj ...)                ; "    "     "
   (:drag-through-selection? T)    ; See below
   (:button-outside-stop?    T)    ; See section 6.4
   (:stop-action #'inter::MultiFont-Text-Int-Stop-Action)
   (:after-cursor-moves-func NIL) ; (lambda (inter text-obj))
   )
```

This interactor was designed to appeal to people familiar with the `inter:text-interactor`. The interactor is started when you click the mouse on a text object, and it stops when you type the stop-event, like #\RETURN. The editing commands (listed below) are similar to `inter:text-interactors`'s commands, with many additional ones.

The new slot `:drag-through-selection?` controls whether dragging through the string with the mouse will cause the indicated region to become selected. You can apply all the standard multifont commands to a region that is selected this way. Note: since we use "pending-delete" like the Macintosh, if you type anything when something is selected, the selected text is deleted.

The words in upper case are labelings of the keys (on the Sun keyboard). If your keyboard has keys labeled differently, let us know and we will insert them into the code.

`^f ^b ^d ^h` = forward, backwards, delete forwards, delete backwards char
`leftarrow`, `rightarrow` = backwards, forwards
`META-f, META-b, META-d, META-h` = same but by words
`^p` = previous line, `^n` = next line
`uparrow`, `downarrow` = previous line, next line
`^,` or `HOME` = beginning of document
`^.` or `END` = end of document
`^a` = beginning of line
`^e` = end of line

`^k` = kill line, `^u` = delete entire string, `^w`, `CUT` = delete selection
`META-w`, `COPY` = copy selection to interactor cut buffer
`^c` = copy entire string to X cut buffer
`^y`, `PASTE` = yank interactor cut buffer or X cut buffer into string
`^Y`, `^PASTE` = yank X buffer
`META-y`, `META-PASTE` = yank interactor cut buffer

The following ones extend the selection while moving:
`^leftarrow`, `^rightarrow` = prev, next char selecting
`META-leftarrow`, `META-rightarrow` = prev, next word selecting
`^uparrow`, `^downarrow` = up-line, down-line selecting

^HOME, ^END = beginning, end of string selecting
^* = select all

CONTROL-META is Lisp stuff if you have lisp mode on (see below):
^-META-b, ^-META-leftarrow = prev lisp expression
^-META-f, ^-META-rightarrow = next lisp expression
^-META-h, ^-META-backspace, ^-META-delete = delete prev s-expr
^-META-d = delete next s-expr

^-shift- is for font stuff:
^-shift-B = toggle bold
^-shift-I = toggle italic
^-shift-F = fixed font (courier)
^-shift-T = times font (serif)
^-shift-H = helvetica font (sans-serif)
^-shift-< = smaller font
^-shift-> = bigger font
^1 ^2 ^3 ^4 = small, medium, large, and very-large fonts

Of course, you can change the mapping of all these functions, using the standard inter:bind-key mechanism described with the regular text-interactor.

5.1.3.2. Focus Multifont Text Interactor

```
(create-instance 'inter:Focus-Multifont-Textinter inter:interactor
   (:obj-to-change NIL)
   (:stop-event NIL)
   (:lisp-mode-p NIL)
   (:match-parens-p NIL)
   (:match-obj ...)
   (:final-function NIL)          ; (lambda (inter obj final-event final-string x y))
   (:after-cursor-moves-func NIL)  ; (lambda (inter text-obj))
   )
```

For applications where one wants the user to be able to type text into a multifont text object without first having to click on the object, the focus-multifont-textinter was created. This interactor provides a feel more like a text editor. The demo demo-text shows how to use the focus-multifont-textinter to create and edit multifont-text objects. The demo-multifont text editor shows how to use this interactor along with the selection-interactor described in the next section.

Unlike other interactors, this interactor never goes into the "running" state. The interactor can only "start." This means that aborting this interactor, or setting the :continuous slot to non-NIL is meaningless. The only way to stop the interactor is either to deactivate it (set the :active-p slot to NIL) or to destroy it. If two or more of these interactors are in the same window, all of the interactors will fetch the keyboard events and send them to their corresponding multifont text objects. Extreme caution is urged when having two or more focus interactors in the same window to avoid having keystrokes go to multiple objects. Ways to avoid having keystrokes go to multiple destinations are to have non-overlapping :start-where positions for all the interactors or to make certain that all idle interactors have their :obj-to-change slot set to NIL.

Usually this interactor will continue running until it is destroyed, but you may want to execute a final function whenever a particular key is pressed. Whenever the user issues the event specified in the :stop-event slot (like #\RETURN), the function in :final-function is executed. The parameters to the final-function are the same as for the standard text-interactor:

```
(lambda (an-interactor obj-being-edited final-event final-string x y))
```

When a `focus-multifont-textinter` is in a window, all keyboard input will be fed directly into the multifont text object that is in its `:obj-to-change` slot. If the `:obj-to-change` slot is NIL, then no multifont text object has the focus.

The `inter:focus-multifont-textinter` has the same key bindings as the `inter:multifont-text-interactor`.

The `inter:focus-multifont-textinter` also has several functions that can be used on it. These functions are used mainly to manipulate the multifont text that the interactor is focused upon.

`inter:Set-Focus` *interactor multifont-text* [*Function*]

This function changes the focus of a `focus-multifont-textinter` from one text object to another. The cursor of the newly activately text object will become visible indicating that it is ready to accept text. The cursor of the previous text object will become invisible and any selected text will become unselected. If the *multifont-text* parameter is NIL, then the currently selected text object will become unselected and no object will have the focus. This function does not return any useful value.

`inter:Copy-Selection` *interactor* [*Function*]

`inter:Cut-Selection` *interactor* [*Function*]

`inter:Paste-Selection` *interactor* [*Function*]

These functions perform cut, copy, and paste operations upon the text object that currently has the focus. The `cut-selection` and `copy-selection` operations copy the selected text into the cut-buffer. `Cut-selection` will delete the selected text, but `copy-selection` will leave it unaffected. `Paste-selection` inserts the cut buffer at the position of the cursor.

5.1.3.3. Selection Interactor

```
(create-instance 'inter:Selection-Interactor inter:interactor
   (:focus-interactor ...)
   (:match-parens-p NIL)
   (:match-obj ...))
```

The `selection-interactor` is a complementary interactor to the `focus-multifont-textinter`. The `selection-interactor` controls mouse input so that the user may click and drag the mouse in order to select text and choose a new multifont object to edit. The `:focus-interactor` slot must be filled with a valid `inter:focus-multifont-textinter` interactor. It is the interactor in that slot that will be used to reset the focus if a new multifont object is clicked upon. The `:start-where` slot must include all possible multifont objects that the `selection-interactor` operates upon. If a new multifont object is clicked upon the `selection-interactor` will reset the focus to the new object and place the cursor at the point where the mouse was clicked. If the mouse is clicked in the multifont object that contains the cursor, the cursor will be moved to position of the click. Dragging the mouse across a multifont object will select the text that was passed over by the mouse. Clicking the mouse while holding the shift key (or clicking the mouse with the right button instead of the left) causes the selection highlight to extend to the newly clicked position.

The `selection-interactor` uses a key translation table to decode different types of clicking operations. The current table translates `:leftdown` to `:start-selection` and `:shift-leftdown` and `:rightdown` to `:start-selection-continue`. These combinations can be changed and other combinations added by using the `inter:bind-key` function.

5.1.3.4. Lisp Mode

Multifont supports a special text-entry mode which is useful for typing Lisp functions or programs. This mode can be used by setting the `:lisp-mode-p` slot of the `multifont-text-interactor` or `focus-multifont-textinter` to T. When in lisp mode, lines of text will tab to the appropriate spot, and semicolon comments will appear in italics. It is important that the fonts of the text are not changed during lisp-mode, since certain fonts hold special meaning for tabs and parenthesis-matching.

`inter:Indent` *string how-many how-far* [*Function*]

This function can be used to define a special indent amount for your own function. The argument *string* is the name of the function, *how-many* is the number of arguments (starting with the first) that should be indented the special amount, and *how-far* is an integer signifying how many spaces from the start of the function name these special arguments should be placed. If *how-far* is -1, then the indent will line up with the first argument on the line above it. The argument following the last special argument will be placed one space in from the start of the function name, and all following arguments will line up with the first argument on the line above it. Here are some examples of the default indentations:

```
(indent "defun" 2 4)                    (indent "do" 2 -1)
(indent "create-instance" 2 4)          (indent "cond" 0)
(indent "let" 1 4)                      (indent "define-method" 3 4)
```

There are several keys which are bound specially during lisp mode:

`^-META-f`, `^-META-rightarrow` = skip forward lisp expression

`^-META-b`, `^-META-leftarrow` = skip backward lisp expression

`^-META-d` = delete lisp expression

`^-META-h`, `^-META-backspace` = delete previous lisp expression

Also helpful in lisp mode is setting the `:match-parens-p` of the interactors to T. Now when the cursor is next to a close parenthesis, the corresponding open-parenthesis will be highlighted in boldface. Also, if the interactors' `:match-obj` is set to another multifont object, that object's text will be set to the text of the line that the matching open-parenthesis is on.

`inter:Turn-Off-Match` *interactor* [*Function*]

This function can be used to externally turn off a matched parenthesis, since it will only be automatically turned off when the cursor is moved away from the close parenthesis.

`inter:Add-Lisp-Char` *text-obj char* `&optional` *new-font new-foreground-color new-background-color* [*Function*]

`inter:Delete-Lisp-Region` *text-obj* [*Function*]

Because lisp mode does some extra things during addition and deleting of text, these special functions should be used when in lisp mode in the place of `opal:add-char` and `opal:delete-selection`. If changes are made externally without using these functions, future tabs and parenthesis-matching may not work properly. Note: you can also use the *lisp-mode-p* parameter of `opal:add-char` and `opal:delete-selection` to indicate that the operation is taking place while lisp-mode is active.

`inter:Lispify` *string* [*Function*]

This function takes a plain string and will return text which will work in lisp mode. The returned text is in `:initial-text` format, and can be used with functions such as `set-text`. The text will already be indented and italicized properly.

5.1.4. Auto-Scrolling Multifont Text Objects

A companion to the word-wrap feature is the vertical auto scroll feature. The auto scroll option can be utilized when a multifont-text object is used inside a scrolling-window along with a focus-multifont-textinter, multifont-text-interactor, or selection-interactor.

The interface for auto-scrolling `multifont-text` is the same as for `cursor-multi-text`, which is described in the Garnet 2.1 Opal Reference Manual.

5.1.5. After Cursor Moves

To support the new lisp-mode, there is a new slot of the three multifont interactors (`multifont-textinter`, `focus-multifont-textinter`, `selection-interactor`) called `:after-cursor-moves-func`. If non-NIL, it should be a function called as `(lambda (inter text-obj)` and will be called whenever the cursor moves, or the text to the left of the cursor changes.

If the function in this slot is overridden with a user-supplied function, the new function should do a `(call-prototype-method ...)` to ensure that the default lisp-mode indentation function is executed, also.

5.1.6. A Multifont Text Gadget

Putting a gadget description into the Opal section is fairly strange. Just as the interactors section above, it was decided that the `multifont-gadget` should be described in the `multifont-text` section.

```
(create-instance 'gg:Multifont-Gadget opal:aggregadget
   (:left 0)
   (:top 0)
   (:initial-text (list ""))
   (:fill-background-p NIL)
   (:word-wrap-p NIL)
   (:text-width 300)
   (:stop-event NIL)
   (:selection-function NIL))
```

This gadget is <u>not</u> automatically loaded by the `multifont-loader`. Instead, you should load `multifont-gadget-loader` from the gadgets directory to load the gadget and all of the required multifont files.

The `multifont-gadget` is a conglomeration of a `multifont-text`, a `focus-multifont-textinter`, and a `selection-interactor`. These are all put together to take some of the trouble out of assembling the pieces by hand. The slots of the gadget are the same as the `multifont-text`. To use the gadget just create it and go. The keyboard and mouse handling are built in. The trouble with this gadget is that you cannot have more than one `multifont-gadget` per window. If you have more than one, all the gadgets will receive the same keystrokes; thus, all the gadgets will respond to the keyboard at the same time.

Usually the gadget will continue running until it is destroyed, but you may want to execute a selection function whenever a particular key is pressed. Whenever the user issues the event specified in the `:stop-event` slot (like #\RETURN), the function in `:selection-function` is executed. The selection function takes the usual parameters (the gadget and its value), where the value is the pure text representation of the gadget's current string.

There is a small demo of how to use the multifont text gadget in the gadget file. To run it, execute `(garnet-gadgets:multifont-gadget-go)`.

5.2. New opal:text

The functionality of all the text objects has been merged into the opal:text object. Specifically, you can now set the :cursor-index slot of a regular opal:text object to make a cursor appear, and the :string slot of opal:text can contain multiple lines. Put another way, inter:text-interactor will now work on instances of opal:text.

We strongly encourage users to eliminate all references to opal:cursor-text, opal:mutli-text, and opal:cursor-multi-text and replace them with opal:text.

5.3. Reparenting Windows

In previous versions of Garnet, it was not possible to change the :parent of a window after it had been created. This restriction has been removed, and you can change the parent of an existing subwindow, or convert a top-level window into a subwindow by setting its :parent slot.

When a top-level window with a window-manager frame is reparented, the low-level CLX drawable is destroyed, and a new drawable is created for the subwindow. This will wipe out any non-Garnet CLX modifications that have been made to the window.

5.4. New Motif Colors

Some new shades of the Motif colors have been added, along with their corresponding filling styles. It should be easier to read text on top of these lighter backgrounds.

```
opal:MOTIF-LIGHT-GRAY          opal:MOTIF-LIGHT-GRAY-FILL
opal:MOTIF-LIGHT-BLUE          opal:MOTIF-LIGHT-BLUE-FILL
opal:MOTIF-LIGHT-GREEN         opal:MOTIF-LIGHT-GREEN-FILL
opal:MOTIF-LIGHT-ORANGE        opal:MOTIF-LIGHT-ORANGE-FILL
```

5.5. Opal:Cursor-Font

The new object opal:cursor-font has been provided for accessing the font used by your window manager to display cursors. This object is an instance of opal:font-from-file, and may not be fully portable on different machines. Regular text strings may be printed in this font, but it is specifically intended for use when changing the cursor of Garnet windows (see section 5.6.1).

5.6. Window Cursors

The default cursor shape for Garnet windows is an arrow pointing to the upper left. However, it would be nice to change this shape sometimes, particularly when an application is performing a long computation and you would like to display an hourglass cursor. Several new functions and objects have been added that make it easy to change the cursors of Garnet windows.

The following sections discuss how to change window cursors, starting with some background at the lowest level of the cursor interface. The later sections, particularly 5.6.3, describe the high-level functions that allow you to change the cursor with a single function call.

5.6.1. The :cursor Slot

At the lowest level, the cursor of a Garnet window is governed by the value of its :cursor slot. The default value for an inter:interactor-window's :cursor slot is a list of two objects, (#k<OPAL:ARROW-CURSOR> . #k<OPAL:ARROW-CURSOR-MASK>), which are pre-defined bitmaps whose images are read from the `garnet/lib/bitmaps/` directory. The `opal:arrow-cursor` object is the black part of the pointer, and the `opal:arrow-cursor-mask` is the underlying white part.[2]

The `:cursor` slot permits three different syntaxes which all describe a cursor/mask pair for the window. The most basic syntax is used for the default value:

(`list` *bitmap-1 bitmap-2*)

The second syntax allows you to use a font as the source for your cursor, with the primary image and mask specified by indices into the font:

(`list` *my-font index-1 index-2*)

Most machines come with a font specifically for the window manager cursors, and this font can be accessed with the new `opal:cursor-font` object. So you could try the syntax above with the `opal:cursor-font` object and two consecutive indices, like this:

```
(s-value WIN :cursor (list opal:cursor-font 50 51))
```

You have to update the window to make the cursor change take effect. It appears that sequential pairs, like 50 and 51, reliably yield primary cursors and their masks. It is easy to experiment to find a nice cursor.

Since so many cursors are created from the cursor font, a third syntax is provided that is analogous to the previous one:

(`list :cursor` *index-1 index-2*)

Any of these three syntaxes can be used to `s-value` the `:cursor` slot of a window. Changing the `:cursor` slot of a window changes it permanently, until you `s-value` the `:cursor` slot again.

5.6.2. Garnet Cursor Objects

```
(create-instance 'opal:ARROW-CURSOR opal:bitmap
  (:image (opal:Get-Garnet-Bitmap "garnet.cursor")))

(create-instance 'opal:ARROW-CURSOR-MASK opal:bitmap
  (:image (opal:Get-Garnet-Bitmap "garnet.mask")))

(defparameter opal:Arrow-Pair
              (cons opal:ARROW-CURSOR opal:ARROW-CURSOR-MASK))



(create-instance 'opal:HOURGLASS-CURSOR opal:bitmap
  (:image (opal:Get-Garnet-Bitmap "hourglass.cursor")))

(create-instance 'opal:HOURGLASS-CURSOR-MASK opal:bitmap
  (:image (opal:Get-Garnet-Bitmap "hourglass.mask")))

(defparameter opal:HourGlass-Pair
              (cons opal:HOURGLASS-CURSOR opal:HOURGLASS-CURSOR-MASK))
```

The arrow-cursors are used for the default value of the `:cursor` slot in Garnet windows. The Gilt interface builder and the `save-gadget` use the hourglass-cursors when they are busy with file I/O and performing long calculations. Users are free to use these objects in their own applications.

The variables `opal:Arrow-Pair` and `opal:HourGlass-Pair` are provided so that users can avoid cons'ing up the same list repeatedly. Setting the `:cursor` slot of a window to be

[2]Whenever you change the cursor of a window, it is a good idea to have a contrasting mask beneath the primary image. This will keep the cursor visible even when it is over an object of the same color.

`opal:HourGlass-Pair` and then updating the window will change the cursor in the window.

5.6.3. Temporarily Changing the Cursor

Often when the cursor needs to be changed, we will be changing it back to the default very soon (e.g., when the application has finished its computation). Also, usually we want to change all of the windows in an application, rather than just one window. For this situation, the functions `opal:change-cursors` and `opal:restore-cursors` were written to change the cursors of multiple windows without changing the `:cursor` slots.

`opal:Change-Cursors` *cursor-list* `&optional` *window-list* [*Function*]

The *cursor-list* argument is a pair or triplet that adheres to the syntax for the `:cursor` slot, discussed in the previous section. When *window-list* is supplied, the cursor of each window is temporarily set with a cursor constructed out of the *cursor-list* spec. When *window-list* is NIL (the default), then all Garnet windows are set with the temporary cursor. The value of the `:cursor` slot of each window remains unchanged, allowing the window's normal cursor to be restored with `opal:restore-cursors`.

`opal:Restore-Cursors &optional` *window-list* [*Function*]

This function undoes the work of `opal:change-cursors`. Each window is set with the cursor described by the value of its `:cursor` slot (which was not changed by `opal:change-cursors`).

Even the work of calling `opal:change-cursors` and `opal:restore-cursors` can be abbreviated, by using the following macros instead:

`opal:With-Cursor` *cursor* `&body` *body* [*Macro*]
`opal:With-HourGlass-Cursor &body` *body* [*Macro*]

The *cursor* parameter must be a pair or triplet adhering to the `:cursor` syntax. These macros change the cursor of all Garnet windows while executing the *body*, and then restore the old cursors. These are the highest level functions for changing window cursors. To test the `opal:with-hourglass-cursor` macro, bring up any Garnet window (demos are fine) and execute the following instruction:

```
(opal:with-hourglass-cursor (sleep 5))
```

While lisp is sleeping, the cursors of all the Garnet windows will change to hourglass cursors, and then they will change back to normal.

5.7. Printing Garnet Windows

Previous versions of the PS module were able to generate PostScript files for most Garnet windows, supporting all common Garnet graphics. In this version we have eliminated all known limitations in the module, and expect that all graphical operations that can be performed in Garnet can be successfully rendered in a PostScript file with `opal:make-ps-file`.

`opal:make-ps-file` *window-or-window-list filename* [*Function*]
`&key` *position-x position-y left-margin right-margin top-margin bottom-margin left top scale-x scale-y landscape-p borders-p clip-p subwindows-p color-p background-color paper-size title creator for comment*

As before, the only required parameters are the Garnet window (or list of windows) and the filename in which to store the PostScript image. For parameters not described below, see the Garnet 2.1 Opal Manual or do `(describe 'opal:make-ps-file)`.

5.7.1. New Features

The *borders-p* parameter now takes four values: `T`, `NIL`, `:generic`, and `:motif`. The default value of `:motif` gives your image a simulated Motif window manager frame, as in Figure 11-2, on page 76. The value of `:generic` puts a plain black frame around your printed image, with the title of the window centered in the title bar. As before, the value `T` gives the image a thin black border, and `NIL` yields no border at all.

A new *comment* parameter has been added to `make-ps-file`, which allows you to put a single line of text at the top of your PostScript file. In the generated file, the characters `"%%"` are concatenated to the front of your comment, telling PostScript to ignore the text in the line. If you wish to use multiple lines in the comment, you will have to add the `"%%"` to the second line of the string and every line thereafter.

The new *paper-size* parameter is provided mainly for users in the United Kingdom. Allowed values are `:letter`, `:a4`, or a list of `(width height)`. The default value of `:letter` generates a PostScript image for 612x792 point size paper. The `:a4` value generates an image for 594x842 point size paper, which is commonly used in the UK.

When a set of coordinates is supplied as the `:clip-p` parameter, the coordinates are now relative to the entire screen, instead of to a single window. This makes it possible to clip to a region when you are printing multiple windows. Clip regions can be used to make multiple-page PostScript files -- you have to manually divide the image into its component regions, and generate one PostScript file for each region. In the future, we may attempt to automate the process of multiple-page printing.

Pixmaps now print in full color when they are being displayed on a color screen and the *color-p* parameter is T (the default). However, older printers may not know the PostScript command `colorimage` which is required to render a color pixmap. This command is only defined on Level 2 printers. If your printer cannot print your pixmap (it crashes with a "colorimage undefined" error), then try calling `opal:make-ps-file` again with a *color-p* argument of NIL.

5.7.2. Bug Fixes

The PS module has been made more robust and more flexible. Some of the graphical objects and operations that the new version can handle are:

- Printing of `opal:multifont-text`.
- Printing of named colors (e.g., an `opal:color` object created with the symbol `'mauve` in the `:color-name` slot, instead of the usual numerical values in the `:red`, `:green`, and `:blue` slots).
- Printing of very large polylines (with more than 1500 points).
- Printing of `opal:font-from-file` objects as PostScript `Courier`. Since these fonts are non-standard, and are not guaranteed to have a mapping to known PostScript fonts, we have to pick a known font and try to scale the font to fit the string's bounding box.
- Better arrangement of windows that have and do not have window manager frames, as with menubar submenus. Previously, windows that did not have title bars (those created with `:omit-title-bar-p` set to T) were misplaced with respect to windows that did have title bars.
- Transparent filling styles (those with the `:fill-style` set to `:stippled` and a bitmap in their `:stipple` slot) are now transparent in the PostScript image.

5.8. Update Quarantine Slot

A "quarantine slot" named `:in-progress` now exists in all Garnet windows. If there was a crash during the last update of the window, then the window will stop being updated automatically along with the other Garnet windows, until you can fix the problem and update the window successfully.

Usually when there is an update failure, it is while the main-event-loop process is running and it is repeatedly calling `opal:update-all`. Without a quarantine slot, these repeated updates would keep throwing Garnet into the debugger, even as you tried to figure out what the problem was with the offending window. With the quarantine slot, `opal:update-all` first checks to see if the `:in-progress` slot of the next window is T. If so, then the last update to that window must not have terminated successfully, and the window is skipped. After you fix the problem in the window, a successful call to `opal:update` will clear the slot, and it will resume being updated automatically.

Here is an example of a typical interaction involving the quarantine slot.

1. Execute `(garnet-load "demos:demo-multiwin")` and `(demo-multiwin:do-go)`.
2. Artificially create an error situation by executing
   ```
   (kr:with-types-disabled
    (kr:s-value demo-multiwin::OBJ1 :left 'x))
   ```
3. Try to move an object in the demo by clicking on it and dragging with the mouse. Even if you did not click on OBJ1 (the rectangle), the main-event-loop called `opal:update-all`, which caused OBJ1's window to update. This caused a crash into the debugger when `'x` was found in the `:left` slot. Get out of the debugger with `:reset` or `q` or whatever your lisp requires.
4. Now move objects again. As long as your first mouse click is not in the same window as OBJ1, you will not get the crash again. You can even drag objects into and through OBJ1's window, but that window will not be updated.
5. After you give OBJ1's `:left` slot a reasonable value and do a <u>total</u> update on its window -- `(opal:update demo-multiwin::WIN1 T)` -- the window will be treated normally again. Note: the total update is sometimes required because the bad `:left` value can get stored in an internal Opal data structure. A total update clears these data structures.

We have found that this feature makes it much easier to find the source of a problem in a window that cannot update successfully. Without this feature, useful tools like the `Inspector` would not be able to run while there was one broken window, since interacting with the `Inspector` requires repeated calls to `opal:update-all`.

5.9. New Parameter to opal:Update-All

The function `opal:update-all` is used to call `opal:update` on all existing Garnet windows. A new optional parameter has been added to this function that will cause a total update of all Garnet windows, to make it more like the regular `opal:update` method.

`opal:update` *window* `&optional` *total-p* [*Method*]

`opal:update-all` `&optional` *total-p* [*Function*]

When *total-p* is T, then `opal:update-all` will redraw the entire contents of all existing Garnet windows. Since this procedure is expensive, it should only be used in special situations, like during debugging.

5.10. Unexported opal::window

The `window` prototype is no longer exported from Opal, and should not be referenced unless absolutely necessary. Creating instances of `opal::window` is strongly discouraged. The `inter:interactor-window` should be the window prototype referenced by user programs.

5.11. Executing Unix Commands

`opal:Shell-Exec` *command* [*Function*]

The new function `opal:shell-exec` is used to spawn a Unix shell and execute Unix commands. The *command* parameter should be a string of the Unix command to be executed. The spawned shell does not read the `.cshrc` file, in order to save time. The function returns a string of the output from the shell.

5.12. Saving Lisp Images

`opal:Make-Image` *filename* `&key` *quit* `(`*verbose* `T) (`*gc* `T)` [*Function*]

The function `opal:make-image` is used to save an image of your current lisp session. Without `make-image`, you would have to call `opal:disconnect-garnet`, use your implementation-dependent function to save your lisp image, and then call `opal:reconnect-garnet` if you wanted to continue the session. `Opal:make-image` does all of this for you, and also does a total garbage collection before the save if the *gc* parameter is T. If the *quit* parameter is T, then your lisp image will automatically exit after saving itself. The *verbose* parameter controls whether the function should announce when it is in the stages of garbage collection, disconnection, saving, and reconnection.

When you restart the saved image, it will print a banner indicating the time at which the image was saved, and will automatically call `opal:reconnect-garnet`.

5.13. New opal:Directory-P

This new function is used to determine whether a string describes an existing directory or not.

`opal:Directory-P` *string* [*Function*]

The *string* should name a potential directory, like `"/usr/garnet/"`. This function spawns a shell and executes a Unix command to test the directory. There is no other standard way to test directories on different lisps and operating systems.

5.14. New Implementation of Hit-Threshold

In previous versions of Garnet, the implementation of the `:hit-threshold` slot was not conforming precisely to the documented specifications. We have reimplemented the internal Opal functions that use the `:hit-threshold` slot so that they are now working correctly. Additionally, we have changed the default value of the `:hit-threshold` slot from 3 to 0.

The `:hit-threshold` slot controls the sensitivity of the internal Opal `point-in-gob` methods that decide whether an event (like a mouse click) occurred "inside" an object. If the `:hit-threshold` is 3, for example, then an event 3 pixels away from the object will still be interpreted as being "inside" the object. When `:select-outline-only` is T, then any event directly on the outline of the object, or within 3 pixels of the outline, will be interpreted as a hit on the object.

Some users who rely on the `:hit-threshold` and `:select-outline-only` slots may be adversely affected by the change in code. In particular, objects that have their `:select-outline-only` slot set to T may find that now they have to click directly on the outline of the object, instead of close to it. This can be remedied by additionally setting the local value of the `:hit-threshold` slot to 3 or some other reasonable value.

5.15. Miscellaneous Bug Fixes

Previous versions of Garnet did not properly handle the screen parameter of the DISPLAY environment variable when opening the display. This problem has been fixed, and has proven to work on several different platforms.

You can now stack fast redraw objects of all three types -- `:rectangle`, `:redraw`, and `T`.

6. Interactors

6.1. Mouse and Keyboard Accelerators

The Interactors now have a new mechanism to attach functions to specific keyboard keys as *accelerators*. These are processed either before or after interactors, and are either attached to a particular window, or global to all windows. If they are *after* the interactors, then the accelerators are only used if no interactor accepts the event.

(Note: If you are using the `menubar` or `motif-menubar`, then you can use the slot `:accelerator-windows` of those gadgets to tell them which windows should have the keyboard accelerators defined in them.)

By default, a number of *global* accelerators are defined:

```
:SHIFT-F1 -  raise window
:SHIFT-F2 -  lower window
:SHIFT-F3 -  iconify window
:SHIFT-F4 -  zoom window
:SHIFT-F5 -  fullzoom window
:SHIFT-F6 -  refresh window
:SHIFT-F7 -  destroy window

:HELP - INSPECTOR object
:CONTROL-HELP - INSPECTOR next interactor to run
:SHIFT-HELP - print out object under the mouse (also in inspector.lisp)
```

The last three are processed *before* Interactors, and are defined in the debugging file `inspector.lisp`.

To change these, see section 9.1. The first 7 are processed *after* the interactors. To change these bindings, set the variable `*default-global-accelerators*`, which is initially defined as:

```
(defvar *default-global-accelerators* '(
    (:SHIFT-F1 .  raise-acc)
    (:SHIFT-F2 .  lower-acc)
    (:SHIFT-F3 .  iconify-acc)
    (:SHIFT-F4 .  zoom-acc)
    (:SHIFT-F5 .  fullzoom-acc)
    (:SHIFT-F6 .  refresh-acc)
    (:SHIFT-F7 .  destroy-acc)))
```

Applications can also set and maintain their own accelerator keys, using the following functions:

`inter:Add-Global-Accelerator` *key fn* `&key` *replace-existing? first?* [*Function*]
`inter:Add-Window-Accelerator` *win key fn* `&key` *replace-existing? first?* [*Function*]

Will call the function *fn* whenever *key* is hit. If *first?* then the accelerator will be tested before all interactors, otherwise it will be tested if no interactor uses *key*. *Replace-existing*, if non-NIL, will remove any other assignments for *key*. By using the default NIL value, you can temporarily hide an accelerator binding.
The function *fn* is called as:

```
(lambda (event))
```

where event is the interactor event structure that caused the accelerator to happen.

`inter:Remove-Global-Accelerator` *key* `&key` *remove-all? first?* [*Function*]
`inter:Remove-Window-Accelerator` *win key* `&key` *remove-all? first?* [*Function*]

Removes the specified accelerator. If *remove-all?* then removes all the accelerators bound to the *key*, otherwise, just removes the first one.

`inter:Clear-Global-Accelerators` [*Function*]
`inter:Clear-Window-Accelerators` *win* [*Function*]

`inter:Default-Global-Accelerators` *;; sets up the default accelerators* [*Function*]

6.2. Window Enter and Leave Events

Sometimes it is useful to know when the cursor is inside the window. We have added the ability to get events when the cursor enters and leaves a window. To enable this, you must set the `:want-enter-leave-events` slot of the window to T *at window creation time*. Changing the value of this slot after the window has been created will not necessarily work. If the window has this value as non-NIL, then when the cursor enters the window, a special event called `:window-enter` will be generated, and when the cursor exits, `:window-leave` will be generated. For example, the following will change the color of the window to red whenever the cursor is inside the window:

```
(create-instance 'MY-WIN inter:interactor-window
  (:want-enter-leave-events T)
  (:aggregate (create-instance NIL opal:aggregate)))
(opal:update MY-WIN)

(create-instance 'SHOW-ENTER-LEAVE inter:button-interactor
  (:start-event '(:window-enter :window-leave))
  (:window MY-WIN)
  (:continuous NIL)
  (:start-where T)
  (:final-function #'(lambda(inter obj)
                       (declare (ignore obj))
                       (s-value (g-value inter :window)
                                :background-color
                                (if (eq :window-enter (g-value inter :start-char))
                                    opal:red
                                    opal:white)))))
```

6.3. Transcripts

`inter:Transcript-Events-From-File` *filename window-list* [*Function*]
`&key (`*wait-elapsed-time* `T) (`*verbose* `T))`

The new parameter *verbose* has been added to the function that reads and executes a Garnet transcript from a file. When *verbose* is NIL, messages will not be printed out saying what the function is doing, nor will the function warp the pointer. Turning this off can speed up execution significantly, since the format statements tend to dominate the timing of replaying the transcript.

6.4. Outside Stop

A new parameter to all the text editing Interactors (`text-interactor`, and `multifont-text-interactor`), specifies whether when the mouse is pressed *outside* the string, the down event should stop the text editing, but still do the action it would have done if text wasn't being edited. This means, for example, that you typically won't have to type RETURN before hitting the OK button, since the down press will stop editing and still operate the OK button. By default this new feature is enabled, but you can turn it off by setting the `:button-outside-stop?` parameter to NIL.

6.5. Debugging Interactors

There are a number of enhancements that make debugging interactors easier. The primary one is the `Inspector` described in section 9.1. Also, the function `garnet-debug:look-inter` has been fixed (section 9.4).

The `inter:trace-inter` function has been enhanced with two additional special parameters. Executing `(inter:trace-inter :next)` will start tracing when the next interactor runs, and it will trace that interactor.

Executing `(inter:trace-inter :short)` will tell you just the name of the interactor that runs, so it will provide a much smaller listing. This is very useful if you suspect that more than one interactor is accidentally running at a time.

6.6. Priority Levels

As an experiment, and to support the Marquise tool which is in progress, there is a new way to control which interactors run. You can now mark an interactor priority level as having `:sorted-interactors`. When this slot of a priority level is non-NIL, then the interactors in that level run in sorted order by the number in the `:sort-order` slot of each interactor (which can be an integer or float, negative or positive). The lowest numbered interactor runs first. Then, if that interactor has a value in its `:exclusivity-value` slot, then no other interactor with the same value in that slot will be run, but interactors with a different value in that slot will be run in their sorted order. Interactors with NIL in their `:sort-order` and/or `:exclusivity-value` slot will run after all other interactors are run. Note that multiple interactors with the same number in the `:sort-order` slot will run in an indeterminate order (or if they have the same `:exclusivity-value`, then only one of them will run, but no telling which one). The `:stop-when` slot of the priority-level works as always to determine what happens when the interactors in that level are finished.

6.7. Final Feedback Objects

The `:final-feedback-obj` slot of menu and button interactors can now contain a formula, which might compute the appropriate feedback object based on the object selected. The interactor will automatically duplicate the appropriate feedback object if more than one is needed (e.g., if `:how-set` is `:list-toggle`). One use of this is to have different kinds of feedback for different kinds of objects, and another would be to have different feedback objects in different windows, for an interactor that works across multiple windows. To aid in this computation, the `:current-obj-over` slot of the interactor is set with the object the mouse was last over, and the `:current-window` slot of the interactor is maintained with the window of the current event.

An example of this is the day-of-the-week menu in `demo-menu` (MENU2) which chooses between arrows and plusses based on the day.

6.8. Agate

NOTE: This section has been written to completely replace the Agate chapter in the Garnet 2.1 Interactors Reference manual. However, some references remain below to other sections in the old Interactors manual.

Agate is a Garnet application that is used to train gestures for use with the gesture interactor. Agate stands for A Gesture-recognizer And Trainer by Example. Agate is in the gesture area, and can be loaded using `(garnet-load "gestures:agate")`. Then type `(agate:do-go)` to begin.

6.8.1. End-User Interface

To train a gesture classifier, the user first types a gesture name into the `Gesture Class Name` field and then demonstrates approximately 15 examples of the gesture by drawing on the `Canvas` window with one of the mouse buttons pressed. To train another gesture class the user can press on the `New Class` button, type in the new gesture name, and give some examples of the gesture. This is done repeatedly for each of the gestures that the user would like the classifier to recognize.

At any point, the user can try out the gestures trained so far by switching to `Recognize` mode by clicking on the `Recognize` toggle button. After demonstrating a gesture in `Recognize` mode, Agate will print the name of the gesture in the `Gesture Class Name` field, along with numbers that represent the non-ambiguity probability and distance of the example from the mean (see the topics `Min-Non-Ambig-Prob` and `Max-Dist-To-Mean` in the Garnet 2.1 Interactors Manual).

When the gesture classifier performs as desired, it can be saved to a file by clicking on the `Save Classifier` button. Existing classifiers can be modified by first loading them into Agate by clicking on the `Load Classifier` button. Then the user can add more examples to existing gestures or add entirely new gestures to the classifier.

A gesture example can be deleted by first selecting the example (a full-sized version of the gesture will be displayed on the Canvas) and then clicking on the `Delete Example` button. Similarly, an entire class can be deleted by selecting the class (all of the examples will be displayed in the `Examples` window) and then clicking on the `Delete Class` button. A gesture class can be renamed by selecting the class and then editing the name in the `Gesture Class Name` field.

The current gesture classifier can be cleared out by clicking on the `New Classifier`. The user will be prompted to save the classifier if it has not been previously saved.

6.8.2. Programming Interface

Agate V2.0 is a self-contained interface tool that can be integrated within another Garnet application. A designer can call `agate:do-go` with parameters for an initial classifier, an initial name to be displayed in the `Gesture Class Name` field, and a final function to call when the user quits Agate.

`agate:Do-Go &key` *dont-enter-main-event-loop double-buffered-p initial-classifier initial-examples initial-gesture-name final-function* [*Function*]

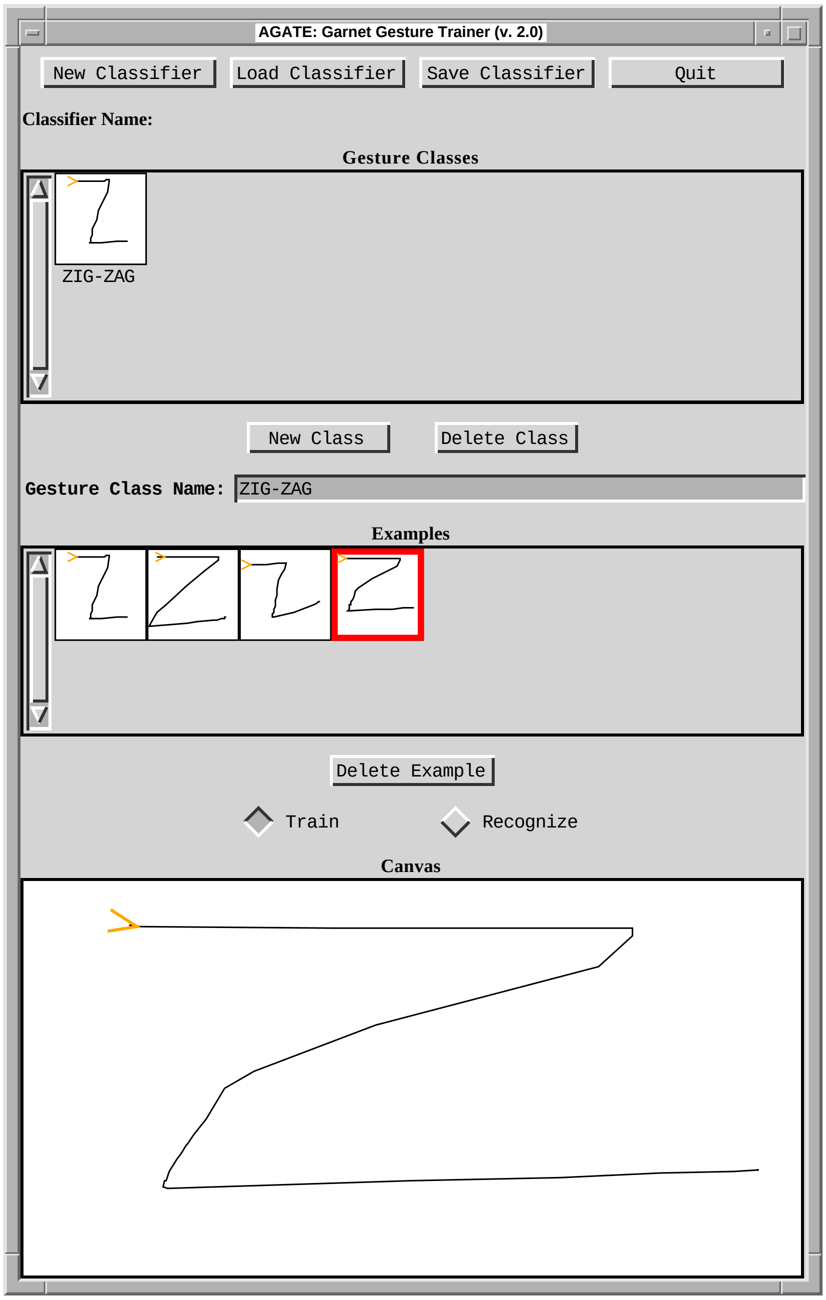


Figure 6-1: An example session with the Agate gesture trainer

---

`Do-go` creates the necessary windows and Garnet objects, and then starts the application. The parameters to `do-go` are as follows:

dont-enter-main-event-loop
- if T, don't enter the main event loop

double-buffered-p - if T, use double buffered windows

initial-classifier - initial classifier to use

initial-examples - initial examples to display *initial-gesture-name* - name to fill in gesture class name field

final-function - function to call on quit

The final function takes the parameters

```
(lambda (last-saved-filename cur-classifier cur-examples saved-p trained-p)
```

last-saved-filename - the last filename saved to

cur-classifier - the current classifier (as of last training)

cur-examples - the current examples (if untrained, will not necessarily correspond to the cur-classifier)

saved-p - has the current classifier been saved?

trained-p - has the current classifier been trained?

7. Aggregadgets and Aggrelists

7.1. Write-Gadget

The `opal:write-gadget` function, used to write out the structure of aggregadgets to a file, supports the writing of new low-level features, like type declarations and meta-information in formulas. It now writes out calls to `opal:get-standard-font` when appropriate, instead of creating new instances of `opal:font` objects. Instances of `opal:multifont-text` can now be written out.

7.2. Aggrelist Layout

7.2.1. The New Implementation

In previous versions of Garnet, formulas were installed in aggrelist components that caused the components to lay out at the proper coordinates, depending on parameters set in the aggrelist. Aggrelists have been completely reimplemented, so that very few formulas remain in the aggrelist prototype, and no formulas are installed in the components. Instead, the `:left` and `:top` slots of the components are set with the appropriate values during update. Setting numerical values procedurally, rather than relying on formulas to compute the appropriate values, saves a significant amount of time and space over the old implementation.

The maintenance of the `:components` list in conjunction with the `:items` list has been automated, so that you do not need to call `opal:notice-items-changed` after setting the `:items` list. The maintenance performed by `opal:notice-items-changed` will be done automatically during update. (If your code really depends on the structure of the aggrelist <u>before</u> you update, you may need to retain your call to `opal:notice-items-changed`, but this would be unusual.)

Since the aggrelist layout function uses `s-value` to change the `:left` and `:top` of the components, you should not declare those slots constant. If the layout parameters in the aggrelist are constant, or a `:parts` list is supplied, then the `:left` and `:top` slots will become constant automatically after the aggrelist is laid out. To find out if all possible formulas have been eliminated from your aggrelist, use the functions `gd:count-formulas` and `gd:why-not-constant`, described in section 9.3.3.

7.2.2. Non-Backwards Compatibility

Unfortunately, we have found that some aggrelists have a constraint structure that causes circularities under the new implementation, and some complicated aggrelists may require some modification. The problem is rooted in the fact that the `:left` and `:top` slots are not set as soon as they used to be.

When the aggrelist tries to figure out where to place the components, it has to compute the `:width` and `:height` of the aggrelist before setting any values. If, during this computation of the dimensions of the aggrelist, the `:left` or `:top` values of the components are needed, they will return inherited values from their prototype, rather than return correct values set by the aggrelist. When the item-prototype is an aggregadget, circularities can be set up among the default dimension formulas. To avoid this problem, default values of zero may be installed in the item-prototype object's `:left` and `:top` slots, since these slots will be overridden with the correct values anyway. Putting these local values in the prototype will prevent the unwanted default formulas from being inherited.

This is not expected to be a common problem, since the `:width` and `:height` formulas of the aggrelist have been carefully written to depend only on the top-level layout parameters and the dimensions of the components. But if the user's own component dimensions depend on the `:left` and `:top` slots, unexpected formulas or other values may appear in these slots. To analyze the problem further, it may be helpful to enable warnings when KR detects a circularity by setting `kr::*warning-on-circularity*` to T, and track the order of evaluation of the constraints in your components.

8. Gadgets

8.1. Standard Edit

There are a number of editing functions that are shared by most graphical editors. The file `standard-edit.lisp` supplies many of these functions in a manner that can probably be used by your graphical editors without change. They support such operations such as cut, copy, paste, delete, duplicate, group, ungroup, refresh, to-top, to-bottom, etc. These functions are designed to work with the `Multi-Graphics-Selection` gadget, and can be invoked from buttons, menus, or a menubar. The `standard-edit` functions are currently used by GarnetDraw, Gilt and Marquise. You don't have to use all the functions in an application. For example, Gilt does not support grouping and ungrouping. (If you find that a small change to a `standard-edit` routine will allow it to be useful to your application, let us know.)

The `standard-edit` routines are in the `garnet-gadgets` package, and can be loaded using `(garnet-load "gg:standard-edit-loader")`.

8.1.1. General Operation

The `standard-edit` routines assume that the graphical objects that are to be edited are all in a single aggregate in a single window (extensions to handle multiple windows are planned, but not in place yet). The routines are tightly tied to the design of the `Multi-Graphics-Selection` gadget. For example, most routines determine which objects to operate on by looking at the current selection, and many change the selection.

`Standard-edit` determines how to edit objects by looking at various slots. The slots listed below are set in the <u>selected</u> objects, not in the selection gadget itself. Most Garnet prototypes already contain the correct default values:

`:line-p` - if non-NIL, then the object is controlled by a `:points` list of 4 values. True by default for `opal:line` and `gg:arrow-lines`.

`:polygon-p` - if non-NIL, then the object is controlled by a `:point-list` list of multiple values. True by default for `opal:polylines`.

`:group-p` - if non-NIL, then the object is a group of objects that the user might be able to get the parts of. True by default for `opal:aggregadgets`. If you allow high-level objects to be added in your editor (e.g., gadgets like buttons), and you supply the `Standard-Ungroup` command, you should set the `:group-p` slot of any objects you don't want the user to ungroup to be NIL.

`:grow-p` - whether the object can change size or not.

If the object has `:line-p` and `:polygon-p` both NIL, then it is assumed to be controlled by a `:box` slot.

The various routines find information they need by looking in a special slots of the gadget that invokes them. This means that all routines must be invoked from the same gadget set, for example, the same `menubar` or `motif-button-panel`.

8.1.2. The Standard-Edit Objects

The `gg:Clipboard-Object` holds the last object that was cut or copied. It also contains some parameters used for pasting and duplicating the objects. Each application can have its own clipboard, or a set of applications can share a clipboard to allow cut and paste among applications. For example, GarnetDraw and Gilt both share the same clipboard, so you can cut and paste objects between the two applications. By default, all applications share the one `gg:Default-Global-Clipboard`.

Note that this does *not* use the X cut buffer, since there is no standard way to copy graphics under X.

```
(create-instance 'gg:Clipboard-Object NIL
  (:value NIL)
  (:x-inc-amt NIL)  ; Offset for duplicate. If NIL, then uses 10
  (:y-inc-amt NIL))

(create-instance 'gg:Default-Global-Clipboard gg:Clipboard-Object)
```

The `Default-Global-Clipboard` is used by default, and allows objects to be copied from one Garnet application to another.

8.1.3. Standard Editing Routines

`gg:Standard-Initialize-Gadget` *gadget selection-gadget agg-of-items* `&key` *clipboard undo-delete?* [*Function*]

This routine must be called once before any of the others are invoked. Typically, you would call this after the editor's windows and objects are created. It takes the `gadget` that is going to invoke the standard-edit routines (e.g., a menubar), the selection gadget that is used to select objects in the graphics editor, and the aggregate that holds the items created in the graphics editor. If you do not supply a `clipboard` object, then `Default-Global-Clipboard` will be used.

Unfortunately, there is not yet a global undo facility, but you can support undoing just the delete operations. The `undo-delete?` flag tells standard-edit whether you want this or not. If non-NIL, then deleted objects are never destroyed, they are just saved in a list.

`gg:Standard-NIY` *gadget* `&rest` *args* [*Function*]

Useful for all those functions that are Not Implemented Yet. It prints "Sorry, Not Implemented Yet" in the Lisp listener window and beeps.

`gg:Standard-Delete` *gadget* `&rest` *args* [*Function*]

Deletes all the selected objects. Makes there be no objects selected.

`gg:Standard-Delete-All` *gadget* `&rest` *args* [*Function*]

Deletes all the objects. Makes there be no objects selected.

`gg:Standard-Undo-Last-Delete` *gadget* `&rest` *args* [*Function*]

If you have initialized standard-edit with `Undo-delete?` as non-NIL, then this function will undo the last delete operation. The objects brought back are selected.

`gg:Standard-To-Top` *gadget* `&rest` *args* [*Function*]

Moves the selected objects to the top (so not covered). They stay selected.

`gg:Standard-To-Bottom` *gadget* `&rest` *args* [*Function*]

Moves the selected objects to the bottom (so covered by all other objects). They stay selected.

`gg:Standard-Refresh` *gadget* `&rest` *args* [*Function*]

Simply redraws the window containing the objects using `(opal:update win T)`.

`gg:Standard-Select-All` *gadget* `&rest` *args* [*Function*]

Causes all of the objects to be selected.

`gg:Standard-Cut` *gadget* `&rest` *args* [*Function*]

Copies the selected objects into the clipboard's cut buffer, and then removes them from the window. Afterwards, there will be no selection.

`gg:Standard-Copy` *gadget* `&rest` *args* [*Function*]

Copies the selected objects into the clipboard's cut buffer, but leaves them in the window. The selection remains the same.

`gg:Standard-Paste-Same-Place` *gadget* `&rest` *args* [*Function*]

Pastes the objects in the clipboard into the window at the same place from which they were cut. Pasting the same objects multiple times will give multiple copies, all in the same place. An application will typically provide either `Standard-Paste-Same-Place` or `Standard-Paste-Inc-Place` as the ‘‘paste’’ operation. The new objects will be selected.

`gg:Standard-Paste-Inc-Place` *gadget* `&rest` *args* [*Function*]

Pastes the objects in the clipboard into the window offset from where they were cut. Pasting the same objects multiple times will give multiple copies, each offset from the previous. The offset amount is determined by the `:x-inc-amt` and `:y-inc-amt` slots of the clipboard object, or, if NIL, then 10 is used. The new objects will be selected.

`gg:Standard-Duplicate` *gadget* `&rest` *args* [*Function*]

Makes a copy of the selected objects, and places them back into the window, offset from the previous objects by `:x-inc-amt` and `:y-inc-amt` (or 10 if these are NIL). The new objects will be selected.

`gg:Standard-Group` *gadget* `&rest` *args* [*Function*]

Creates an `aggregadget` and puts the selected objects into it. The `Multi-Graphics-Selection` gadget will then operate on the group as a whole, and will not let parts of it be manipulated (like MacDraw, but unlike Lapidary). The group (aggregadget) object will be selected.

`gg:Standard-UnGroup` *gadget* `&rest` *args* [*Function*]

Goes through all the selected objects, and for any that have the `:group-p` slot non-NIL, removes all the components from that aggregate and adds the objects directly to the parent of the group. `:Group-p` is true by default for `opal:aggregadgets`. If you allow high-level objects to be added in your editor (e.g., gadgets like buttons), and you supply the `Standard-Ungroup` command, you should set the `:group-p` slot to be NIL for any objects you don't want the user to ungroup.

8.1.4. Utility Procedures

`gg:Sort-Objs-Display-Order` *objs draw-agg* [*Function*]

For many operations, it is important to operate on the objects in display order, rather than in the order in which the objects were selected. `Sort-Objs-Display-Order` takes a list of objects (*objs*) and an aggregate that contains them (*draw-agg*) and sorts the objects so they are in the same order as in *draw-agg*. The procedure returns a copy of the list passed in, so it is safe to supply the `:value` of the `Multi-Graphics-Selection` gadget, for example.

`gg:Is-A-Motif-Background` *obj* [*Function*]

Tests whether the specified object is a `Motif-Background` object. This procedure is safe even if the motif gadgets have not been loaded.

`gg:Is-A-Motif-Rect` *obj* [*Function*]

Tests whether the specified object is a `Motif-Rect` object. This procedure is safe even if the motif gadgets have not been loaded.

8.2. Prop Sheet

There were extensive changes in the `Prop-Sheet-For-Obj` to make it automatically use the `:parameters` information and the types of slots to determine which slots to use. It also now supports displaying the slots from *multiple* objects at the same time. The new version of Gilt makes heavy use of the new features of the prop-sheet.

```
(create-instance 'gg:Prop-Sheet-For-Obj gg:prop-sheet
    (:maybe-constant '(:left :top :obj :slots :eval-p :set-immediately-p
                       :v-spacing :multi-line-p :select-label-p
                       :label-selected-func :label-select-event :visible
                       :select-value-p :value-selected-func :single-select-p
                       :type-gadgets :union? :error-gadget))
    (:left 5)
    (:top 5)
    (:obj NIL)    ; a single obj or a list of objects
    (:slots NIL) ; list of slots to show. If NIL, get from :parameters
    (:union? T)   ;if slots is NIL and multiple objects, use union or intersection of :parameters?

    (:eval-p T)   ; if T, then evaluates what the user types.  Use T for
                  ; graphical objects.  If NIL, then all the values will be strings.
    (:set-immediately-p T) ; if T then sets slots when user hits return, else doesn't
                           ; ever set the slot.
    (:type-gadgets NIL) ; descriptor of special handling for types
    (:error-gadget NIL) ; an error gadget to use to report errors.

    ;;  plus the rest of the slots also provided by prop-sheet

    (:v-spacing 1)
    (:pixel-margin NIL)
    (:rank-margin NIL)
    (:multi-line-p NIL)       ; T if multi-line strings are allowed
    (:select-label-p NIL)   ; T if want to be able to select the labels
    (:label-select-event :any-mousedown)
    (:label-selected-func NIL)
    (:select-value-p NIL)   ; if want to be able to select the values
    (:value-selected-func NIL)
    (:single-select-p NIL) ; to select more than one value or label

   ; Read-only slots
    (:label-selected NIL)   ; set with the selected label objects (or a list)
    (:value-selected NIL)   ; set with the selected value objects (or a list)
    (:value ...)    ; list of pairs of all the slots and their (filtered) values
    (:changed-values NIL)) ; only the values that have changed
```

The loader for `prop-sheet-for-obj` is "prop-sheet-loader".

The changes since the last version are that the `:obj` slot can contain a list of objects or a single object. If it is a list of objects, then if multiple objects share a slot which is displayed, then the value from the first object is shown. If the values from multiple objects differ, then the slot value is shown in italics. If the user edits the value, then it is set into each object for which that slot is in the `:parameters` list of the object.

Another change is that `:slots` parameter can now be NIL. In this case, the `:parameters` slot of the object or objects is used to determine which slots to display. If there are multiple objects, then you can determine whether to display the slots that are in *any* of the objects, by setting `:union?` to be T (the default), or whether to display only slots that are in *all* of the `:parameters` lists (the intersection of the lists), in which case `:union?` should be NIL. By default, the property sheet will use the

`Horiz-Choice-List` gadget for slots of type `KR-boolean` and `(Member ...)` where the number of options is 5 or less. If the type of a slot has a documentation string, gotten using `kr:get-type-documentation`, then this is displayed as the slot comment field.
Another new parameter is `:type-gadgets` which can be used to modify the default displays for slots from the `:parameters` list. `:Type-gadgets` contains a list which can contain the following entries:

- a slot name - this means never display this slot (omit the slot even though it is in the `:parameters` list).
- a list of `(typ gadget)` - this means whenever a slot of type `typ` is displayed in the prop-sheet, use the specified gadget. For example, Gilt uses this mechanism to display a `Pop-Up-From-Icon` for all slots which contain a font:

```
(list (g-type opal:text :font) (create-instance NIL gg:Pop-Up-From-Icon
                                  (:constant :icon-image :pop-up-function)
                                  (:creator-function 'Show-Font-Dialog)
                                  (:pop-up-function 'Pop-Up-Prop-Dialog)))
```

- a list of `(typ othertyp)` - this means whenever a slot of type `typ` is found, pretend instead that it has type `othertyp`. This is useful, for example, to map types that are complicated to ones that will generate a `member` gadget.

Type checking is performed before setting a slot, and any errors are reported using the optional error gadget which should be placed in the `:error-gadget` slot. If there is no error gadget, then the error message is simply not displayed, but a beep is sounded and the slot value is shown in italics.

8.3. Motif-Prop-Sheet-For-Obj-With-Done

There is a new gadget that displays a property sheet for an object and a ‘‘Done’’ button. When a slot value is edited, the slot is set immediately, rather than waiting for an OK or APPLY to be hit. Thus, the prop-sheet-for-obj slot `:set-immediately-p` is always T. This is especially useful for when the property sheet is displaying multiple objects, since slots which are not edited won’t be set. The Done button simply removes the property sheet window from the screen.

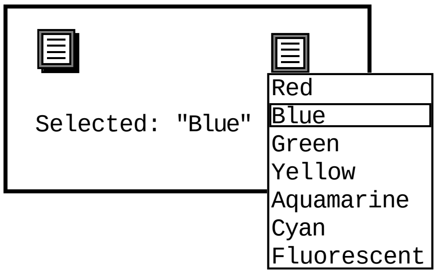
Sorry, there is no Garnet look-and-feel version of this gadget.

The parameters are pretty much the same as for `prop-sheet-for-obj`, with the addition of the `done-function` which is called with the property sheet as a parameter.

```
(create-instance 'motif-prop-sheet-for-obj-with-done opal:aggregadget
    (:maybe-constant '(:left :top :obj :slots :eval-p :done-function
                       :v-spacing
                       :multi-line-p :select-label-p
                       :label-selected-func :label-select-event
                       :select-value-p :value-selected-func :single-select-p
                       :foreground-color :visible
                       :type-gadgets :union? :error-gadget))
    (:left 5) (:top 5)
    (:obj NIL)   ; can be one object or a list of objects
    (:slots NIL) ; list of slots to show.  If NIL uses :parameters
    (:Done-Function NIL) ; called when hit done as (lambda (prop))
    (:eval-p T)  ; evaluate the values of the slots?  Usually T.
    (:error-gadget NIL)  ; used to report errors on evaluation
    (:type-gadgets NIL)  ; modifies the default display of slots
    (:union? T)  ;if slots is NIL and multiple objects, use union or intersection of :parameters?
    (:v-spacing 1)
    (:select-p NIL) ; T if want to be able to select the entries

    (:foreground-color opal:Motif-Gray) ; background color of the window

    (:select-label-p NIL) ; T if want to be able to select the entries
    (:label-selected-func NIL)
    (:label-select-event :any-mousedown)
    (:select-value-p NIL)
    (:value-selected-func NIL)
    (:single-select-p NIL)

      ; read-only slots
    (:label-selected (o-formula (gvl :propsheet :label-selected)))
    (:value-selected (o-formula (gvl :propsheet :value-selected)))
    (:value (o-formula (gvl :propsheet :value)))
    (:changed-values (o-formula (gvl :propsheet :changed-values)))
    (:width (o-formula (MAX (gvl :done-panel :width)
                            (gvl :propsheet :width))))
    (:height (o-formula (+ 2 (gvl :done-panel :height)
                           (gvl :propsheet :height))))
```

The loader for `motif-prop-sheet-for-obj-with-done` is "motif-prop-sheet-win-loader".

8.4. Popup-Menu-Button

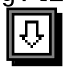



```
(create-instance 'gg:Popup-Menu-Button gg:text-button
  (:left 0)
  (:top 0)
  (:string gg:lines-bitmap)
  (:items '("Item 1" "Item 2" "Item 3" "Item 4"))
  (:v-spacing 0)
  (:h-align :LEFT)
  (:item-font opal:default-font)
  (:item-to-string-function
   #'(lambda (item)
       (if item
           (if (or (stringp item) (schema-p item))
               item
               (string-capitalize (string-trim ":" item)))
           "")))
  (:min-menu-width 0)
  (:shadow-offset 2)
  (:text-offset 3)
  (:gray-width 2)
  (:selection-function NIL)
  (:value (o-formula ...))
  (:position :below)
  (:keep-menu-in-screen-p T)
```

The loader file for the `popup-menu-button` is `popup-menu-button-loader`, and you can see a demo by executing `(gg:popup-menu-button-go)`. (Sorry, there isn't a Motif version yet.)

This is a combination of a button and a popup menu. When you press on the button, the menu is shown, and then you can select something from the menu, and then the menu disappears. If you release outside of the menu, the menu just goes away and keeps its current value. The button itself can show a string or an arbitrary Garnet object (e.g., a bitmap, as shown here).

The `:left` and `:top` determine when the button goes.

The `:string` slot determines what is shown in the button. It can be a regular string (e.g., `"Value"`) or an arbitrary Garnet object. The default value is the `gg:lines-bitmap` shown above. Another bitmap provided is `gg:downarrow-bitmap` which looks like .

The `:items` slot holds the items that are shown in the popup menu. It can have the standard format for items (e.g., a list of strings, objects, pairs of strings and functions, etc.). See section 1.4.3.1 of the Garnet 2.1 Gadgets Manual for more information.

The `:v-spacing`, `:h-align`, and `:item-font` control the display of the menu items. See the Gadgets manual for menus for more information.

The `:min-menu-width` slot can contain a minimum width for the popup menu. You might use this to make the menu line up with a text entry field.

The `:item-to-string-function` can be used to convert the values in the `:items` list into strings.

The `:shadow-offset`, `:text-offset` and `:gray-width` parameters control the appearance of the button itself.

When the user selects a menu item, the `:selection-function` is called with parameters:
`(lambda (gadget value))`, where the gadget is the popup-menu-button and the value is the appropriate item from `:items`. The `:value` slot will also be set with the appropriate item.

The position of the menu with respect to the button is controlled by the `:position` parameter. Legal options are:

- `:below` - the menu is below and left justified with the button (the default).
- `:left` - the menu will be centered vertically at the left of the button.
- `:right` - the menu will be centered vertically at the right of the button.
- a list of two numbers (x y) - the menu will be at this location. The `:position` slot can contain a formula that calculates these numbers.

If `:keep-menu-in-screen-p` is non-NIL, then the position computed based on the `:position` argument will be adjusted so the menu always stays in the screen. Otherwise, the menu might extend off the screen edges.

8.5. Mouseline

There are two new gadgets that will show a help string attached to any object. The string can be shown in a fixed location in a window using the `MouseLine` gadget, and therefore is like the **mouse documentation line** on Symbolics Lisp machines (sometime called the ‘‘mode line’’ or ‘‘who line’’). Alternatively, the help string can pop up in a window using the `MouseLinePopup` gadget, and therefore be like **Balloon Help** in the Macintosh System 7. You can also control whether the string appears immediately or only after the mouse is over an object for a particular period of time.

An example of the use of the two mouseline gadgets is `gg:mouseline-go` which is at the end of the `mouseline.lisp` file. The standard `demos-controller` which you get when you load `garnet-demos-loader` also uses the `MouseLinePopup` gadget to show what the different demos do.

Note: the mouseline gadget is implemented in a rather inefficient manner. It has the potential to significantly slow down applications, especially when the delay feature is used (`:wait-amount` non-zero). If this proves to be a big problem in practice, please let us know.

Note 2: the delay feature is implemented with multiple processes, which are only supported in Allegro and Lucid lisp.

8.5.1. MouseLine gadget

```
(create-instance 'gg:MouseLine opal:aggregadget
  (:left 5)
  (:top (o-formula (- (gvl :window :height)  ; default is bottom of window
                      (gvl :label :height)
                      5)))
  (:windows (o-formula (gvl :window)))  ; default is the window containing the mouseline gadget
  (:wait-amount 0)   ; how long to wait before displaying the string
```

The loader file for the `MouseLine` is `mouseline-loader`.

You create an instance of the `mouseline` gadget and add it to a window. By default it is positioned at the bottom left, but you can override the :top and :left to position it where-ever you want. Once created, the string will display the value of the :help-string field for any object the mouse is over in the window or windows specified in the :windows slot. By default :windows is only the window that the mouseline gadget is in, but it can be any list of windows, or T for all interactor windows.

The gadget first looks at the leaf object under the mouse, and if that does not have a help-string, then its parent (aggregate) is looked at, and so on. The lowest-level help string found is displayed in the string. The string can contain newlines but not font information (the display is a opal:multi-text not a opal:multifont-text). Of course, the :help-string slot can contain a formula, which might, for example, generate a different string when a gadget is disabled explaining why.

If the mouseline gadgets catch on, we might provide a way to specify the help-strings as part of the standard :items protocol for gadgets, but for now you need to s-value the :help-string slots directly. See the demos-controller for how this might be done.

If the :wait-amount slot is non-zero, then it is the number of seconds the mouse must remain over an object before the mouseline string is displayed. This feature relies on the animation-interactor which uses the multi-process mechanism in Lisp, **so the :wait-amount is only currently available in Lucid and Allegro.**

8.5.2. MouseLinePopup gadget

```
(create-instance ’gg:MouseLinePopup opal:aggregadget
  (:start-event :SHIFT-CONTROL-META-LEFTDOWN)
  (:windows (o-formula (gvl :window))) ; default is the window containing the mouseline gadget
  (:wait-amount 3) ; how long to wait before displaying string
```

The loader file for the MouseLinePopup is mouseline-loader.

This displays the same help-string as the mouseline gadget above, but the string is displayed in a window which pops up at the mouse. Therefore it is like ‘‘Balloon Help’’ in the Macintosh System 7. The window is just big enough for the string, and it goes away when you move off of the object. The :wait-amount determines how long in seconds you must keep the mouse over the object before the window appears.

8.6. Menu

An :item-to-string-function-slot has been added to the gg:menu gadget, which is identical in operation to that of the gg:scrolling-menu. If the :items slot does not contain a list of the usual items or item/function pairs, then this function should return the conversion of each element into a valid item.

The default :item-to-string-function assumes that the :items list is composed of the usual items or item/function pairs:

```
(lambda (item)
  (if item
      (if (or (stringp item) (schema-p item))
          item
          (string-capitalize (string-trim ":" item)))
      ""))
```

8.7. Gauges

The slots `:format-string` and `:enum-format-string` have been added to the `gg:gauge` and `gg:motif-gauge` gadgets. Like with the standard slider gadgets, the default value for both strings is "~a". These strings control the formatting of the text in the value-feedback and the tic-marks of the gauges. These slots are mainly of interest to users who need to format floating point numbers in their gauges.

8.8. Motif-Menu

Thanks to Conrad Poelman here at CMU, the `gg:motif-menu` gadget has been enhanced to support variable-width fonts as the `:item-font` for the menu. In previous versions, the gadget could not underline the correct accelerator character in a menu item when the font was variable-width.

8.9. Motif-Menubar

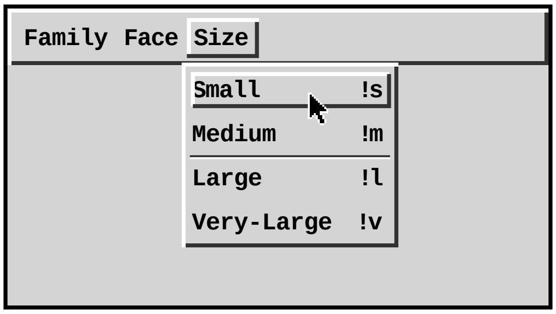


Figure 8-1: An instance of the `motif-menubar` gadget

```
(create-instance 'gg:Motif-Menubar gg::motif-gadget-prototype
  (:left 0)(:top 0)
  (:items NIL)
  (:title-font opal:default-font)
  (:item-font opal:default-font)
  (:min-menubar-width 0)
  (:accelerators NIL)
  (:accelerator-windows (o-formula (gvl :window)))
  (:bar-above-these-items NIL))
```

To load the `motif-menubar`, execute `(garnet-load "gadgets:motif-menubar-loader")`.

The `motif-menubar` is used very much like the standard `menubar`, described in Section 3.9 of the Garnet 2.1 Gadgets Reference Manual. The `motif-menubar` has several additional features, including: slots that allow the menubar to extend across the top of the entire window, keyboard accelerators, and decorative "bars" in the submenus.

A simple demo which uses the `motif-menubar` is loaded along with the `motif-menubar`. To run it, execute `(gg:motif-menubar-go)`. Larger demos also use the `motif-menubar`, including `GarnetDraw` and `Demo-Multifont`.

The `:min-menubar-width` slot specifies how wide the `motif-menubar` should be. If it contains a value greater than the current width of the `motif-menubar`, the bar will extend itself. However, the items will remain fixed (i.e. they won't spread out equidistantly over the bar). This feature is useful when you want the menubar to extend across the top of the entire window, as in Figure 8-1.

8.9.1. Selection Functions

Like in the standard `menubar`, there is no `:value` slot for this gadget. The designer must use the `:selection-function` or the item functions to act on the user's selections.

There are three levels of functions in the `motif-menubar` gadget that may be called when the user makes a selection. Functions can be attached to individual submenu items, whole submenus, or the top level menubar.

All the selection functions take three parameters:

```
(lambda (gadget menu-item submenu-item))
```

The following `:items` list is taken from the `gg:Motif-Menubar-Go` demo, defined at the end of motif-menubar.lisp.

```
(:items
 '((:family ,#'family-fn
            ((:fixed ,#'fixed-fn)(:serif ,#'serif-fn)(:sans-serif ,#'sans-serif-fn)))
   (:face ,#'face-fn
          ((:roman)(:bold)(:italic)(:bold-italic)))
   (:size ,#'size-fn
          ((:small)(:medium)(:large)(:very-large)))))
```

This `:items` list attaches the functions `family-fn`, `face-fn`, and `size-fn` to each of the <u>submenus</u> in the menubar. Whenever the user selects an item from one of those submenus, the corresponding submenu-function will be executed.

Additionally, the functions `fixed-fn`, `serif-fn`, and `sans-serif-fn` are attached to each <u>item</u> in the first submenu. Whenever the user chooses "Fixed", "Serif", or "Sans-Serif" from the "Family" menu, the function associated with that item will be executed.

The order of function execution is as follows: First, the submenu-item function is called, then the submenu function, and then the top-level `:selection-function`. Notice that this is different from the order in which the functions for the regular menubar are called.

8.9.2. Accelerators

Since the `motif-menubar` uses actual instances of the `motif-menu` gadget for its submenus, the "accelerators" feature of the `motif-menu` gadget can be used in the menubar. The syntax for specifying accelerators is a bit more complicated in the menubar, because multiple submenus are used.

An accelerator is a relationship between a keyboard event and an item in the menubar. When a key is typed that corresponds to a menubar item, the function that is associated with the item is executed as if the user had pulled down the submenu and selected the item with the mouse. Each accelerator is specified by its lisp character (e.g., `:F3`), and a string to be shown to the user describing the accelerator key (e.g., "F3"). These string/character pairs are supplied to the menubar in a list, one pair for each item in the menubar. For example,

```
(:accelerators '((("!f" :|META-f|) ("!b" :|META-b|))
                 NIL
                 (NIL NIL ("!x" :|META-x|))))
```

In this accelerators list, the first item in the first submenu has accelerator string "!f", and is selected by the keyboard event, :META-f. The second item in the first submenu has the accelerator string "!b", and keyboard event :META-b. The second submenu has no accelerators. The first two items in the third submenu have no accelerators. The third item in the third submenu has string "!x" and event META-x.

In general, the format for the `:accelerators` slot is:

```
(:accelerators '(((s1,1 k1,1) (s1,2 k1,2) ...)
                 ((s2,1 k2,1) (s2,2 k2,2) ...)
                 ...))
```

where sM,N is the accelerator string for the N-th item in the M-th submenu, and kM,N is the keyboard event for the same.

The `:accelerator-windows` slot by default contains the `motif-menubar`'s window, but may contain a list of windows. When an accelerator event occurs in one of these windows, it will be perceived by the menubar and the item functions will be executed. If the mouse is in a different window, and the accelerator event occurs, the menubar will not notice the event. For this reason, you should put a list of all your application's windows in this slot, if you always want the accelerator to activate the menubar.

8.9.3. Decorative Bars

The "bars" feature of the `motif-menu` can also be used in the `motif-menubar` gadget. The `:bar-above-these-items` slot specifies over which items a horizontal line should appear. For example:

```
(:bar-above-these-items '(("Small")
                          NIL
                          ("Faster" "Warp Speed")))
```

will cause a bar to appear above the item "Small" in the first submenu, and above the items `"Faster"` and `"Warp Speed"` in the third submenu, with no bars in the second submenu. In the `motif-menubar` demo, pictured in Figure 8-1, there is a bar above third item in the last submenu.

8.9.4. Programming the Motif-Menubar the Traditional Garnet Way

There are two approaches to programming a `motif-menubar`. The first, discussed in this section, is the Garnet way, where all the `:items` are provided while creating the menubar. The second approach, discussed in section 8.9.5, requires that all the sub-objects be created individually and attached to the menubar.

The format for the `:items` slot of the `motif-menubar` is the same as in the regular `menubar`. For example,

```
(:items '(("Speed" NIL (("Slow" Slow-Fn) ("Medium" Med-Fn)
                        ("Fast" Fast-Fn) ("Whoa" Too-Fast-Fn)))))
```

This `:items` list creates a menubar with one bar-item, `"Speed"`, which has no submenu selection function. In that bar-item's submenu, are the items `"Slow"`, `"Medium"`, `"Fast"` and `"Whoa"`, which will call `Slow-Fn`, `Med-Fn`, `Fast-Fn` and `Too-Fast-Fn` respectively when selected. Note that in contrast to the example of Section 8.9.1, we did not include #' function specifiers with the selection function names. This is not necessary, because the functions are invoked with `funcall`, and the symbols will be dereferenced when necessary (though it would be faster to include the #', and avoid the dereferencing).

The submenu-items should always be described with lists, even if they have no functions (e.g., `("Slow")` instead of `"Slow"`). Also, the submenu function should either be NIL (as in the above example) or a function. As in the regular menubar, the item functions are optional and may be omitted.

8.9.4.1. An Example

The following example creates the `motif-menubar` pictured in Figure 8-1. Note the behavior of the META-f accelerator and the location of the bar.

```
(create-instance 'WIN inter:interactor-window
  (:background-color opal:motif-gray)
  (:aggregate (create-instance 'TOP-AGG opal:aggregate)))

(defun Fixed-Fn (submenu bar-item submenu-item)
  (format T "Fixed called with ~s ~s ~s~%" submenu bar-item submenu-item))

(defun Face-Fn (gadget menu-item submenu-item)
  (format T "Face called with ~s ~s ~s~%"
          gadget menu-item submenu-item))

(create-instance 'MY-MOTIF-MENUBAR gg:motif-menubar
  (:foreground-color opal:motif-gray)
  (:items
   '((:family NIL
             ((:fixed fixed-fn)(:serif)(:sans-serif)))
     (:face face-fn
            ((:roman)(:bold)(:italic)(:bold-italic)))
     (:size NIL
            ((:small)(:medium)(:large)(:very-large)))))
  (:accelerators
   '((("!f" :|META-f|) ("!e" :|META-e|) ("!a" :|META-a|))
     (("!r" :|META-r|) ("!b" :|META-b|) ("!i" :|META-i|) ("!B" :META-B))
     (("!s" :|META-s|) ("!m" :|META-m|) ("!l" :|META-l|) ("!v" :|META-v|))))
  (:bar-above-these-items
   '(NIL
     NIL
     (:large))))

(opal:add-component TOP-AGG MY-MOTIF-MENUBAR)
(opal:update WIN)
```

8.9.4.2. Adding Items to the Motif-Menubar

Adding items to the `motif-menubar` is very similar to adding items to the regular `menubar`, with the additional ability to add accelerators to the menubar along with the new items.

The add-item method for the motif-menubar may be used to add submenus:

`opal:Add-Item` *menubar submenu* `[:accelerators` *accels*`]` [*Method*]
`[[:where]` *position* `[`*locator*`] [:key` *index-function*`]]`

NOTE: If any accelerators are being added, the `:accelerators` keyword and arguments *must* appear before the `:where` arguments.

The following will add a bar item named "Volume", with a few items and accelerators in it:

```
(opal:add-item MY-MOTIF-MENUBAR
               '("Volume" NIL (("Low") ("Medium") ("High") ("Yowsa")))
               :accelerators '(NIL NIL
                               ("!h" :|META-h|) ("!y" :|META-y|))
               :before :size :key #'car)
```

To add a submenu item, use the function:

`gg:Add-Submenu-Item` *menubar submenu-title submenu-item* [*Method*]
`[:accelerator` *accel*`]`
`[[:where]` *position* `[`*locator*`] [:key` *index-function*`]]`

As with the previous function, if any accelerators are being added, they *must* appear before the `:where`. Also, notice that since only one accelerator is being added for the item, the keyword is `:accelerator`, not `:accelerators`.

The following example will add a submenu item named "Quiet" to the submenu named "Volume", and its accelerator will be META-q:

```
(gg:add-submenu-item MY-MOTIF-MENUBAR "Volume" '("Quiet")
                     :accelerator '("!q" :|META-q|) :before "Low" :key #'car)
```

8.9.4.3. Removing Items from the Motif-Menubar

An item is removed from a `motif-menubar` in exactly the same way as from a regular menubar. To remove an entire submenu, use:

`opal:Remove-Item` *menubar submenu* [*Method*]

For traditional Garnet programming, the *submenu* should be a sublist of the top level `:items` list, or it can just be the title of a submenu.

The following line will remove the "Volume" submenu from the previous examples.

```
(opal:remove-item MY-MOTIF-MENUBAR "Volume")
```

For removing submenu items, use

`gg:Remove-Submenu-Item` *menubar submenu-title submenu-item* [*Method*]

The following will remove the `:small` item from the submenu, `:size`.

```
(gg:remove-submenu-item MY-MOTIF-MENUBAR :size '(:small))
```

8.9.5. Programming the Motif-Menubar with Components

The designer may also choose a bottom-up way of programming the `motif-menubar`. The idea is to create the submenus of the menubar individually using the functions described in this section, and then attach them to a menubar.

8.9.5.1. An Example

This example creates a `motif-menubar` and several components, and attaches them together.

```
(create-instance 'WIN inter:interactor-window
  (:background-color opal:motif-blue)
  (:aggregate (create-instance 'TOP-AGG opal:aggregate)))

; Create the menubar and a bar item
(setf MY-MOTIF-MENUBAR (garnet-gadgets:make-motif-menubar))
(s-value MY-MOTIF-MENUBAR :foreground-color opal:motif-blue)

(setf MAC-BAR (garnet-gadgets:make-motif-bar-item :title "Fonts"))

; Create the submenu items
(setf SUB1 (garnet-gadgets:make-motif-submenu-item :desc '("Gothic")))
(setf SUB2 (garnet-gadgets:make-motif-submenu-item :desc '("Venice")))
(setf SUB3 (garnet-gadgets:make-motif-submenu-item :desc '("Outlaw")))

; Add submenu items to the bar item
(opal:add-item MAC-BAR SUB1)
(opal:add-item MAC-BAR SUB2)
(opal:add-item MAC-BAR SUB3 :after "Venice" :key #'car)

; Add the bar item to the menubar and update the window
(opal:add-item MY-MOTIF-MENUBAR MAC-BAR
               :accelerators '(("!g" :|META-g|) ("!v" :|META-v|) ("!o" :|META-o|)))

; Add the menubar to the top-level aggregate
(opal:add-component TOP-AGG MY-MOTIF-MENUBAR)

(opal:update WIN)
```

When programming a `motif-menubar` by components, you should add accelerators only when you add a bar-item to the menubar, or when adding a submenu item to a bar item that has already been added to a menubar. That is, you cannot add an accelerator to a submenu that has not been attached to a menubar yet.

8.9.5.2. Creating Components of the Motif-Menubar

A `motif-menubar` is essentially the same as a menubar in that there are three components - the menubar itself, bar items containing submenus, and submenu items. Each can be created with the following functions:

`gg:Make-Motif-Menubar` [*Function*]

Will return an instance of `motif-menubar`.

`gg:Make-Motif-Bar-Item &key` *desc font title* [*Function*]

Returns a `motif-bar-item`. Like the regular `menubar`, the `:desc` parameter is a description of the submenu (e.g., `'("Speed" NIL (("Fast") ("Slow") ("Crawl")))`), and the font and title keys specify the font and the heading of the submenu.

`gg:Make-Motif-Submenu-Item &key` *desc enabled* [*Function*]

Creates and returns an instance of `motif-submenu-item`, which is actually a `motif-menu-item`, since each motif-submenu is just a `motif-menu`. The `:desc` parameter describes the item, (e.g., `'("Italic")` or `'("Italic" italic-fn)`). The default for enabled is T.

8.9.5.3. Adding Components to the Motif-Menubar

Two types of components that can be added to the `motif-menubar` are bar-items and submenu-items. The `add-item` method can be used to add new bar-items to the menubar, and can also be used to add new submenu-items to existing bar-items. The `set-...` functions are used to install a collection of components all at once.

`gg:Set-Menubar` *motif-menubar new-bar-items* [*Method*]

This removes the current bar-items from *motif-menubar* and adds the new bar items. This is useful for recycling a menubar instead of creating a new one.

`gg:Set-Submenu` *motif-bar-item new-submenu-items* [*Method*]

Sets the *motif-bar-item* to have the new submenu-items. For more information on these two functions, see the `menubar` section of the Garnet 2.1 Gadgets Manual.

`opal:Add-Item` *motif-menubar motif-bar-item* `[:accelerators` *accels*`]` [*Method*]
`[[:where]` *position* `[`*locator*`] [:key` *index-function*`]]`

Will add *motif-bar-item* to *motif-menubar*. As usual, if any accelerators are being added, the `:accelerators` key *must* be specified before the `:where` key. The `:accelerators` syntax is described in Section 8.9.2.

`opal:Add-Item` *motif-bar-item motif-menu-item* [*Method*]
`[:accelerator` *accels*`]`
`[[:where]` *position* `[`*locator*`] [:key` *index-function*`]]`

Adds the submenu-item, *motif-menu-item* to *motif-bar-item*. If the *motif-bar-item* is not attached to a `motif-menubar`, then no accelerators will be added, regardless of whether any are specified.

The following example shows how bar items are added to a `motif-menubar`:

```
(setf bar1 (gg:make-motif-bar-item
            :desc '("Color" NIL (("Red") ("Blue") ("Polka Dots")))))
(setf bar2 (gg:make-motif-bar-item
            :desc '("Size" NIL (("Small") ("Medium") ("Large")))))
(opal:add-item MY-MOTIF-MENUBAR bar1
               :accelerators '(("!r" :|META-r|) ("!b" :|META-b|) ("!p" :|META-p|)))
(opal:add-item MY-MOTIF-MENUBAR bar2 :before bar1)
(opal:update WIN)
```

This sequence shows how submenu-items can be attached to bar-items:

```
(setf color1 (gg:make-motif-submenu-item :desc '("Maroon")))
(setf color2 (gg:make-motif-submenu-item :desc '("Peachpuff")))
(opal:add-item bar1 color1 :accelerator '("!m" :|META-m|))
(opal:add-item bar1 color2 :after "Blue" :key #'car)
```

8.9.5.4. Removing Components from the Menubar

Bar-items and submenu-items can be removed from the menubar with the `remove-item` method.

In the example from the previous section, to remove `color1` from `bar1`, we say:

```
(opal:remove-item bar1 color1)
```

And to remove the `bar1` itself:

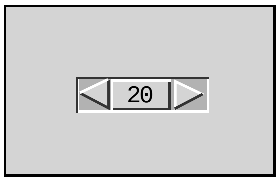
```
(opal:remove-item MY-MOTIF-MENUBAR bar1)
```

8.9.6. Functions Shared with the Regular Menubar

The following functions have the same effect on the `motif-menubar` as they have on the standard `menubar`. Please see the `menubar` section of the Garnet 2.1 Gadgets Manual for more information.

`gg:Menubar-Components` *motif-menubar* [*Method*]
`gg:Submenu-Components` *motif-bar-item* [*Method*]
`gg:Get-Bar-Component` *motif-menubar item* [*Method*]
`gg:Get-Submenu-Component` *motif-bar-item item* [*Method*]
`gg:Find-Submenu-Component` *motif-menubar submenu-title submenu-item* [*Method*]
`gg:Menubar-Disable-Component` *motif-menubar-component* [*Method*]
`gg:Menubar-Enable-Component` *motif-menubar-component* [*Method*]
`gg:Menubar-Enabled-P` *motif-menubar-component* [*Method*]
`gg:Menubar-Get-Title` *motif-menubar-component* [*Method*]
`gg:Menubar-Set-Title` *motif-menubar-component* [*Method*]
`gg:Menubar-Installed-P` *motif-menubar-component* [*Method*]

8.10. Motif-Trill-Device




```
(create-instance 'gg:Motif-Trill-Device gg::motif-gadget-prototype
  (:left 0) (:top 0)
  (:width 150) (:height 40)
  (:val-1 0) (:val-2 100)
  (:value 20)
  (:foreground-color opal:motif-gray)
  (:format-string "~a")
  (:value-feedback-font opal:default-font)
  (:value-feedback-p T)
  (:scroll-incr 1)
  (:selection-function NIL)
```

The loader file for the `motif-trill-device` is "motif-trill-device-loader". The demo `(gg:motif-trill-go)` is loaded by default, and shows an example of the `motif-trill-device`.

The `motif-trill-device` is a simple incrementing/decrementing gadget with trill boxes and a numerical display. The behavior is identical to the standard `trill-device` -- click on the left or right arrows to change the value, and click the left mouse button on the text to edit it.

The slots `:val-1` and `:val-2` contain the upper and lower bounds for the value of the gadget. Either slot may be the minimum or maximum, and either slot may be NIL (indicating no boundary). If a value less than the minimum allowed value is entered, the value of the gadget will be set to the minimum, and analogously for the maximum. Clicking on the left trill box always moves the value closer to `:val-1`, whether that is the max or min, and clicking on the right trill box always moves the value closer to `:val-2`.

The current value of the gadget is stored in the `:value` slot, and may be set directly using `s-value`. The `:scroll-incr` slot specifies the increment for changing the value with the trill boxes. All other slots work the same as in the standard `trill-device`. See the appropriate section of the Garnet 2.1 Gadgets Manual.

The `:foreground-color` slot specifies the color of the object.

8.11. Polyline-Creator

The `polyline-creator` gadget now supports polyline editing, meaning that you can add, remove, and move points of a polyline with the mouse. There are new slots in the `polyline-creator` gadget and the function `gg:toggle-polyline-handles` to control polyline editing. For a full description of the `polyline-creator`, see the Garnet 2.1 Gadgets Reference Manual.

Examples of the polyline editor are in the GarnetDraw demo and the small `(gg:polyline-creator-demo-go)` which is loaded by default with the `polyline-creator`. To load the `polyline-creator`, do `(garnet-load "gadgets:polyline-creator-loader")`.

`gg:Toggle-Polyline-Handles` *polyline-creator-gadget polyline* [*Function*]

This function is used to display square "selection handles" on each point in the polyline to enable editing. The *polyline-creator-gadget* is passed as an argument to this function, since the selection handles to be displayed are components of the gadget.

To move a point, click the left mouse button over the point, move it to a new position, and release the left mouse button. Hitting `control-g` while moving a point will abort the move. Clicking the left mouse button in the middle of a line will add a point, after which the point can be dragged to a different location. Clicking on the background while editing a polyline will turn off the handles for the polyline.

There are several ways to delete points: either hit the middle mouse button over the point, double-click on the point, or hit the DELETE key while moving the point.

When the `toggle-polyline-handles` function is called, it first checks to see if the polyline is already being edited. If it is, it turns off the handles for the polyline. Otherwise, it turns on the handles for the polyline. Note that only one polyline can be edited at a time. If you call this function while a polyline is already being edited, it will turn off the handles for that polyline before turning on the handles for the polyline to be edited.

There are five slots in the polyline gadget which specify what actions cause editing. The slots and their default values are:

```
(:mover-start-event :leftdown)       ; event to start moving a point
(:mover-stop-event :leftup)          ; event to stop moving a point
(:adder-start-event :leftdown)       ; event to add a point
(:deleter-start-event :middledown)   ; event to delete a point
(:threshold 3)                       ; how close to line to add a point
```

The `:threshold` slot specifies how close to the line you have to click for the editor to think you are really trying to add a point to the polyline.

There is a slot in the gadget called `:polyline-being-edited`. This slot will contain the polyline that is currently being edited, or NIL if no polyline is being edited.

8.12. Save and Load Gadgets

Several new features have been added to the standard and Motif versions of the save and load gadgets. For a complete description of these gadgets, see the Garnet 2.1 Gadgets Reference Manual.

- Save and load gadgets change the cursor to the shape of an hourglass while they are becoming visible, since it sometimes takes a few seconds for them to appear. They also display an hourglass cursor while fetching new directory information (when changing the displayed directory).

- The gadgets include a ".." in the menu to make it easier to go up a directory.

- `Save-File-If-Wanted` now takes in two optional arguments -- An initial *filename*, and a *query-string* which specifies what message appears in the `query-gadget`. The default message is "Save file first?"

  `gg:Save-File-If-Wanted` *save-gadget* `&optional` *filename* `(`*query-string* `"Save file first?")` [*Function*]

- `Save-File-If-Wanted` now returns different values depending on whether the user actually completes the save procedure or cancels the procedure. This should be of use to applications that want to do something different depending on whether the user goes through with the save.

  Specifically, calling `Save-File-If-Wanted` causes a dialog box to appear with "Yes", "No", and "Cancel" buttons. If the "No" or "Cancel" buttons are selected, the keywords `:NO` and `:CANCEL` will be returned, respectively. Otherwise, if the user selects "Yes", then the value of the gadget's `:selection-function` will be returned.

- The contents of directories displayed in the save and load gadgets are now automatically sorted. Some lisps were not returning the contents of the directories in sorted order.

8.13. Error-Gadget and Careful Evaluation

There are four new functions that can be used to evaluate lisp expressions that may contain errors, while avoiding a crash into the debugger. These functions may be used to evaluate user input to make sure it is free of errors before passing it on to the rest of an application. If the user input contains an error (i.e., does not successfully evaluate), the functions return a special value and can display an `error-gadget` informing the user of the error.

These functions are more portable and more useful than implementation-dependent functions like `ignore-errors`. These functions are used in many Garnet applications and demos where information is supplied by the user. Examples can be found in the `Inspector`, `demo-graph`, `garnet-calculator`, and the line and filling-style dialog boxes in Gilt.

All of the `careful-eval` functions are defined in `error-gadget-utils`, and are loaded automatically along with the error and query gadgets when you do `(garnet-load "gadgets:error-gadget-loader")` or `(garnet-load "gadgets:motif-error-gadget-loader")`.

These functions were inspired by the `protected-eval` module in the Garnet `contrib` directory, created by Russell G. Almond.

8.13.1. Careful-Eval

`gg:Careful-Eval` *form* `&optional` *error-gadget error-message* [*Macro*]

`Careful-Eval` will evaluate the *form*. If an error is encountered during the eval, then the *error-gadget* will be displayed with the actual lisp error message that was generated, followed by the specified *error-message* (separated by carriage returns).

When the evaluation is successful, `gg:Careful-Eval` returns the evaluated value (which may be multiple values). If there was an error, then `Careful-Eval` returns two values: NIL and the error condition structure. (For a discussion of error conditions, see Chapter 29 of the Second Edition of Guy Steele's *Common Lisp, the Language*.)

Examples:

```
<cl> (gg:careful-eval '(+ 4 5))      ;; evaluates successfully
9
<cl> (gg:careful-eval '(+ 4 y))      ;; signals an error
NIL
#<EXCL::SIMPLE-ERROR.0>
<cl> (multiple-value-bind (val errorp)
          (gg:careful-eval '(+ 4 y))
()        (if errorp    ; perhaps (typep errorp 'condition) is safer
            (format t "An error was encountered~%")
            (format t "Value is ~S~%" val)))
An error was encountered

NIL
<cl>
```

8.13.2. Careful-Read-From-String

`gg:Careful-Read-From-String` *string* `&optional` *error-gadget error-message* [*Function*]

`Careful-Read-From-String` will try to read a symbol or expression from the *string* and return it if successful. If an error is encountered, then the *error-gadget* will be raised and two values will be returned: NIL and the error condition. The message displayed in the error gadget will be a concatenation of the actual lisp error message followed by the *error-message*.

8.13.3. Careful-String-Eval

`gg:Careful-String-Eval` *string* `&optional` *error-gadget error-message* [*Function*]

`Careful-String-Eval` will try to read a symbol or expression from the string and then eval it. If the read and eval are successful, then the evaluated value is returned. If there was an error during either the read or eval, then the *error-gadget* is raised and two values are returned: NIL and the error condition. The message displayed in the error gadget will be a concatenation of the actual lisp error message and the *error-message*.

8.13.4. Careful-Eval-Formula-Lambda

`gg:Careful-Eval-Formula-Lambda` *expr error-gadget error-message the-obj the-slot the-formula warn-p* [*Function*]

`Careful-Eval-Formula-Lambda` evaluates the expression AS IF it were installed in *the-slot* of *the-obj* as a formula. This is useful when the *expr* contains `gvl` calls, which normally require that the *expr* is already installed in an `o-formula` when it is evaluated. If the evaluation is successful, then the evaluated value is returned. If there was an error during the eval, then the *error-gadget* is raised and two values are returned: NIL and the error condition. The message displayed in the error gadget will be a concatenation of the actual lisp error message followed by the *error-message*.

If a formula object has already been created for the expression, then it should be passed as the value of *the-formula*. This will cause dependencies to be established as the `gv`'s and `gvl`'s are evaluated in the expression. *The-formula* may also have the value `:ignore`, which will prevent the establishment of dependencies.

Example:

```
<cl> (create-instance 'R opal:rectangle
        (:my-left 67))
Object R
#k<R>
<cl> (gg:careful-eval-formula-lambda '(gvl :my-left) NIL NIL
                                     R :left :ignore NIL)
67
<cl>
```

9. Debugging

9.1. Inspector

There is a new utility to help with debugging called the `Inspector`. It pops up a window showing an object, and will also show the aggregate and is-a hierarchy for objects, and the dependencies for formulas. Various operations can be performed on objects and slots. In general, the `Inspector` is quite useful for debugging programs, and provides interfaces to many of the other debugging functions in Garnet. A view of an object being inspected in shown in Figure 9-1.

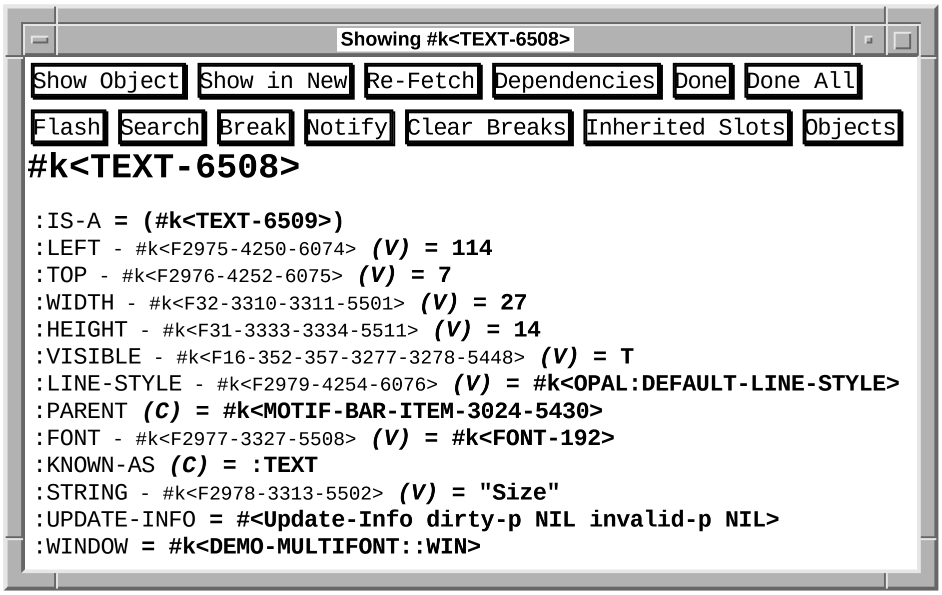


Figure 9-1: The Inspector showing a text object.

The `Inspector` is loaded automatically when you load the debugging tools which is enabled by default in garnet-loader, but it can also be loaded explicitly using `(garnet-load "debug:inspector")` The `Inspector` is in the `garnet-debug` package.

9.1.1. Invoking the Inspector

There are a number of ways to inspect objects. The easiest is to put the mouse over an object and hit the `HELP` keyboard key. This will print a message in the Lisp listener window and pop up a window like Figure 9-1. If you want to use the `HELP` keyboard key for something else, you can set the variable `garnet-debug:*inspector-key*` to a different key (or NIL for none) *before* loading the `Inspector`.

Alternatively, you can explicitly invoke the `Inspector` on an object using either

`garnet-debug:Inspector` *obj* [*Function*]
`gd:Inspector` *obj*

(`gd` is an abbreviation for `garnet-debug`).

To inspect the *next* interactor that runs, you can hit `CONTROL-HELP` on the keyboard (the mouse position is irrelevant), or call the function

`gd:Inspect-Next-Inter` [*Function*]

Hitting `CONTROL-HELP` a second time before an interactor runs will cancel the invocation of the `Inspector`. To change the binding of this function, set the variable `gd:*inspector-next-inter-key*` *before* loading the `Inspector`.

By default, `SHIFT-HELP` is bound to a little function that simply prints the object under the mouse to the Lisp Listener, and does not invoke the `Inspector`. Example output from it is:

```
--> (24,96) = #k<MULTIFONT-LINE-1447> in window #k<INTERACTOR-WINDOW-1371>

--> No object at (79,71) in window #k<INTERACTOR-WINDOW-1371>
```

To change the binding of this function, use the variable `gd:*show-object-key*`.

9.1.2. Schema View

The schema view shown in Figure 9-1 tells all the local slots of an object. To see the inherited slots also, click on the `Inherited Slots` button. For each slot, the display is:

- The slot name.
- A ***(C)*** if the slot is constant.
- An ***(I)*** if the slot is inherited.
- The formula for the slot, if any.
- If there is a formula, then a ***(V)*** if the slot value is valid, otherwise a ***(IV)*** for invalid.
- The current value of the slot, which may wrap to multiple lines if the value is long.
- The entire line is red if the slot is a *parameter* to the object (if it is in the `:parameters` list), otherwise the line is black.

If the object's values change while it is being inspected, the view is *not* updated automatically. To see the current value of slots, hit the `Re-Fetch` button.

To change the value of a slot of an object, click in the value part of the slot (after the =), and edit the value to the desired value and hit return. The object will immediately be updated and the `Inspector` display will be re-fetched. If you change your mind about editing the value before hitting return, simply hit `control-g`. If you try to set a slot which is marked constant, the `Inspector` will go ahead and set the slot, but it gives you a warning because often dependencies based on the slot will no longer be there, so the effect of setting the slot may not work.

If a slot's value is an object and you want to inspect that object, or if you want to inspect a formula, you can double-click the left button over the object name, and hit the `Show Object` button. Also, you can use the `Show in New` button if you want the object to be inspected in a new window.

9.1.3. Object View

Hitting the `Objects` button brings up the view in Figure 9-2. This view shows the name of the object being inspected at the top, then the `is-a` hierarchy. In Figure 9-2, `Text-6509` is the immediate prototype of the inspected object (`text-6508`), and `TEXT-7180` is the prototype of `Text-6509`, and so on. The next set of objects shows the aggregate hierarchy. Here, `text-6508` is in the aggregate `MOTIF-BAR-ITEM-3024-5430`, etc. The last item in this list is always the window that the object is in (even though that is technically not the `:parent` of the top-level aggregate). The final list is simply the list of objects that have been viewed in this window, which forms a simple history of views.

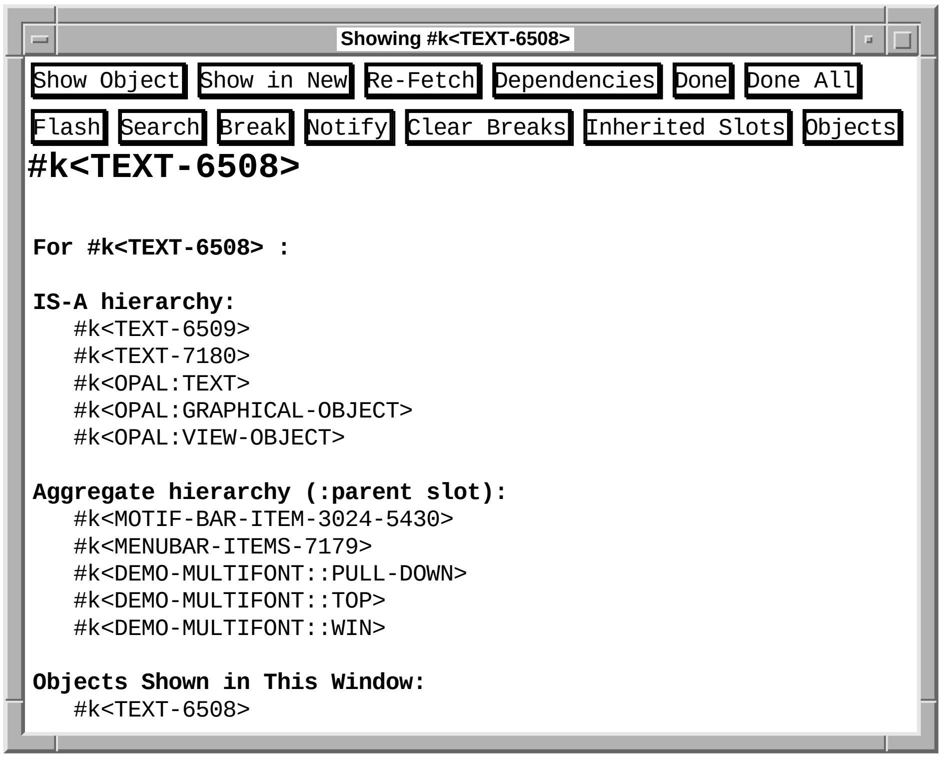


Figure 9-2: The Inspector showing the objects related to the text object of Figure 9-1.

To return to the schema view of the current object, use the `Re-Fetch` button. You can double-click on any object and use `Show Object` or `Show in New` to see its fields, or you can hit `Objects` to go to the object view of the selected object.

9.1.4. Formula Dependencies View

If you select a formula or a slot name (by double-clicking on it) and then hit the `Dependencies` button, you get the view of Figure 9-3. This slows the slots used in calculating the value in the formula.

The first lines show the object, the slot, the formula name, and the expression of the formula. Then the dependencies are shown. The outer-most level of indenting are those slots that are immediately used by the formula. In this case, the `:VISIBLE` slot of `#k<MOTIF-BAR-ITEM-3024-5430>`. Note that only non-constant slots are shown, which is why the `:parent` slot of `TEXT-6508` is not listed (it is constant). Indented underneath each slot are the slots it depends on in turn, so `:VISIBLE` of `#k<MOTIF-BAR-ITEM-3024-5430>` depends on its `:parent` and its parent's `:visible`. The `"..."` means that there are more levels of dependencies. To see these, you can double click on the `"..."` or on

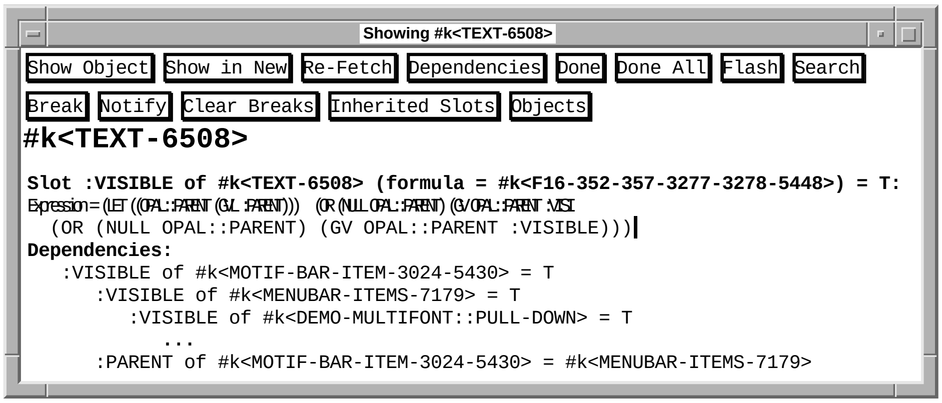


Figure 9-3: The Inspector showing the dependencies of the `:visible` slot of the object shown in Figure 9-1.

any slot name and hit the `Dependencies` button.

9.1.5. Summary of Commands

double-clicking the left button on an object or slot will select it, and it will then be the parameter for further commands.

single-clicking the left button after the = will let you edit the value. Hit return to set the value or control-g to abort.

`Show Object` - displays the selected object in the same window.

`Show in New` - displays the selected object in a new window.

`Re-Fetch` - redisplay the current object, and re-fetch the values of all slots, in case any have changed. This command is also used to get back to the schema view from the object or dependency views.

`Dependencies` - when a formula or a slot containing a formula is selected, then shows the slots that are used to calculate it (see section 9.1.4).

`Done` - get rid of this `Inspector` window.

`Done All` - get rid of all `Inspector` windows.

`Flash` - if an object is selected, flash it, otherwise flash the current object being inspected. The object is flashed by bringing its window to the top and putting an XOR rectangle over it (using the function `gd:flash`).

`Search` - Find a slot of the object and display it at the top of the list. This helps you find slots in a long list, and it will find inherited slots, so you don't have to hit `Inherited Slots` and get the whole list when you are only interested in one slot. After hitting the `Search` button, you will be prompted for the slot name, and you can type in a few letters, hit RETURN, and the `Inspector` will try to fill out the name based on all current slots of the object.

`Notify` - If a slot is selected, then will print a message in the `Inspector` *and* in the Lisp Listener window whenever the selected slot of the object is set. If no slot is selected, then will print a message whenever *any* slot of the object is set. You can be waiting for a Notify or Break on multiple slots of multiple objects at the same time. Note that execution is much slower when there are *any* Breaks or Notifies in effect.

`Break` - If a slot is selected, then will break into the debugger whenever the selected slot of the object is set. If no slot is selected, then will break into the debugger whenever *any* slot of the object is set. You should go to your Lisp Listener window to handle the break, and then *continue* from the break (rather than aborting or popping from the break). The `Inspector` will not operate while you are in the debugger unless you type `(inter:main-event-loop)`.

`Clear Breaks` - Clear all the breaks and notifies. All Breaks and Notifies are also cleared when you hit the `Done` or `Done All` buttons. There is no interface for clearing a single break or notify.

`Inherited Slots` - Toggle the display of inherited slots in the schema view.

`Objects` - Switch to the object view that shows the is-a and aggregate hierarchy (see section 9.1.3).

9.2. Noticing when Slots are Set

It is often useful to be notified when a slot of an object is set, so now we provide a set of debugging functions that do this. There is also an interface to these functions through the `Inspector` (section 9.1) which makes them more convenient to use.

Note that the implementation of this is *very* inefficient and is intended only for debugging. Don't use this as general-purpose demon technique since a search is performed for *every* formula evaluation and every slot setting when any notifies or breaks are set.

`gd:Notify-On-Slot-Set &key` *object slot value* [*Function*]

This will print out a message in the Lisp Listener window whenever the appropriate slot is set. If a value is supplied, then only notifies when the slot is set to that particular value. If an object is provided, then only notifies when a slot of that object is set. If no object is supplied, then notifies whenever *any* object is set. If a slot is provided, then only notifies when that slot is set. If no slot is supplied, then notifies whenever *any* slot is set. If object is NIL, then clears all breaks and notifies. If all parameters are missing, then shows current status. For example,

```
(gd:Notify-On-Slot-Set :object obj :slot :left) ;notify when :left of obj set
(gd:Notify-On-Slot-Set :object obj :slot :left :value 0) ;notify when :left of obj set to 0
(gd:Notify-On-Slot-Set :value NIL) ;notify when any slot of any obj set to NIL
(gd:Notify-On-Slot-Set :object obj) ;notify when any slot of obj set
```

Each call to `Notify-On-Slot-Set` adds to the previous list of breaks and notifies, unless the object is NIL. You can use `clear-slot-set` to remove a break or notify (see below).

`gd:Break-On-Slot-Set &key` *object slot value* [*Function*]

Same as `Notify-On-Slot-Set`, but breaks into the debugger when the appropriate slot is set.

`gd:Call-Func-On-Slot-Set` *object slot value fnc extra-val* [*Function*]

This gives you more control, since you get to supply the function that is called when the appropriate slot is set. The parameters here are not optional, so if you don't want to specify the object, slot or value, use the special keyword `:*any*`. The function `fnc` is called as:

```
(lambda (obj slot val reason extra-val))
```

where the *slot* of *obj* is being set with *val*. The *reason* explains why the slot is being set and will be one of :S-VALUE, :FORMULA-EVALUATION, :INHERITANCE-PROPAGATION or :DESTROY-SLOT. *Extra-val* can be anything and is the same value passed into `Call-Func-On-Slot-Set`.

`gd:Clear-Slot-Set &key` *object slot value* [*Function*]

Clear the break or notify for the object, slot and value. If nothing is specified or object is NIL, then clears all breaks and notifies.

9.3. Choosing Constant Slots

Since the use of constants can significantly reduce the storage requirements and execution time of an application, we have provided several new functions that help you to choose which slots should be declared constant. The following functions are used in conjunction to identify slots that are candidates for constant declarations.

9.3.1. Suggest-Constants

`gd:Record-From-Now` [*Function*]

`gd:Suggest-Constants` *object* `&key` *max* `(`*recompute-p* `T) (`*level* `1)` [*Function*]

To use these functions, bring up the application you want to analyze, and execute `record-from-now`. Exercise all the parts and gadgets of the interface that are expected to be operated during normal use, and then call `suggest-constants`. Information will be printed out that identifies slots which, if declared constant, would cause dependent formulas to be replaced by their actual values.

Keep in mind that it is usually not necessary to declare every reported slot constant. Many formulas will become constant if they depend on constant slots. For example, declaring many of the parameters of a `gg:text-button-panel` constant in the top-level gadget is sufficient to eliminate the internal formulas that depend on them.

Also, it is important to exercise all parts of the application in order to get an accurate list of constant slot candidates. If you forget to operate a certain button while recording, slots may be suggested that would cause the button to become inoperable, since `suggest-constants` would assume it was a static object.

`Suggest-constants` will tell you if a potential slot is in the object's `:maybe-constant` list. When the slot is in this list, then it can be declared constant by supplying the value of T in the `:constant` list. As you add constants, though, you may want to carefully name each slot individually in the `:constant` list to avoid erroneous constant declarations.

The parameters to `suggest-constants` are used as follows:

object - This can be any Garnet object, but it is usually a window or its top-level aggregate. The function examines formulas in *object* and all its children.

max - This parameter controls how many constant slot candidates are printed out. The default is to print all potential constant slots that are found in *object* and all its children.

recompute-p - Set this parameter to NIL if you do not need to reexamine all the objects and you trust what was computed earlier (the same information that was printed out before will be printed out again, without checking that it is still valid).

level - The default value of *level*, which is 1, causes the function to print only pairs which would, by themselves, eliminate some formula. If *level* is made higher, pairs will be printed if the dependent formulas have a total number of depended pairs which is less than or equal to the level.

9.3.2. Explain-Formulas and Find-Formulas

`gd:Explain-Formulas` *aggregate* `&optional (`*limit* `50) eliminate-useless-p)` [*Function*]

`Explain-formulas` is used to analyze all the formulas that were not evaluated since the last call to `record-from-now`. These formulas might have been evaluated when the application was first created, to position the objects appropriately, but are not a dynamic part of the interface, and are thus candidates for constant declarations. If the *eliminate-useless-p* option is non-NIL, then formulas that are in fact unnecessary (i.e., would go away if they were recomputed) are actually eliminated immediately.

`gd:Find-Formulas` *aggregate* `&optional (`*only-totals-p* `T) (`*limit* `50)` *from* [*Function*]

If the function `find-formulas` is called with a non-NIL *only-totals-p* option, it will print out the total number of formulas that have not been reevaluated since the last call to `record-from-now`. If *only-totals-p* is NIL and *limit* is specified, it will print out at most *limit* formula names. If *limit* is NIL, all formula names will be printed out.

You will seldom need to specify the *from* parameter. This allows you to print out formulas that have been unevaluated since *from*. The default value is the number returned by the last call to `record-from-now`; specifying a smaller number reduces the number of formulas that are printed out, since formulas that were evaluated earlier are discarded.

9.3.3. Count-Formulas and Why-Not-Constant

`gd:Count-Formulas` *object* [*Function*]

`Count-formulas` will print a list of all existing formulas in *object* and all its children. It is important to note that formulas are not copied down into an object until they are specifically requested by a `g-value` or `gv` call. Thus, you may not get an accurate count of the real number of formulas in an object until you exercise the object in its intended way. For example, if a prototype `A` has a formula in its `:left` slot and you count the formulas in `B`, an instance of `A`, before asking for `B`'s `:left` slot, then `B`'s `:left` formula will not be counted, because it has not been copied down yet.

`gd:Why-Not-Constant` *object slot* [*Function*]

This function is extremely useful when you are trying to get rid of formulas by declaring constant slots. If `count-formulas` tells you that formulas still exist in your application that you think should go away due to propagation of constants, then you can call `why-not-constant` on a particular slot to find out what its formula depends on. The function will print out a list of dependencies for the formula in the *slot*, which will give you a hint about what other slot could be declared constant to make this formula go away.

9.4. Look-Inter

The function `gd:look-inter` had been broken for a while, so we fixed it. It prints out some useful information about interactors. It takes a parameter which can be:

- an interactor to describe,
- NIL to list all active interactors,
- a window, to list all active interactors on that window,
- an interactor priority-level, to list all active interactors on that level,
- a graphical object, to try to find all interactors that affect that object,
- `:next` to wait and describe the next interactor that runs

As mentioned in section 6.5, the function `inter:trace-inter` takes new parameters `:next` or `:short`. You can also debug the next interactor to run using the `Inspector` by typing control-help or `gd:inspect-next-interactor`.

10. Demos

10.1. Demos-Controller

The `Demos-Controller` application, which is a nice interface for bringing up the other demos, now features the new `gg:mouseline` gadget. When you keep the mouse still over one of the x-buttons for about 2 seconds, a window will pop up with a short description of the corresponding demo.

10.2. Garnet-Calculator

The new demo `garnet-calculator` replaces the old `demo-calculator`. The new calculator has the look and feel of `xcalc`, the calculator supplied by X windows, but it is more robust. The calculator is a self-contained tool, and can be integrated inside a larger Garnet application.

You can load the new demo with `(garnet-load "demos:garnet-calculator")`. To run it, execute `(garnet-calculator:do-go)`.

`garnet-calculator:Start-Calc &key` *double-buffered-p* [*Function*]

`garnet-calculator:Stop-Calc` *app-object* `&optional (`*destroy-app-object?* `T)` [*Function*]

The function `start-calc` creates and returns a calculator "application object" that can be used by a larger interface, and this object should be passed as the *app-object* parameter to `stop-calc`.

10.3. GarnetDraw

The `GarnetDraw` application has been enhanced to use many more features of the Garnet toolkit. The editing functions like Cut, Copy, and Paste are implemented using the new `Standard-Edit` module from `garnet-gadgets`, and objects can be cut and pasted between `GarnetDraw` and `Gilt` (since they share the same clipboard). `GarnetDraw` uses the new `motif-menubar` gadget, with accelerators like `META-x` for Cut and `META-v` for Paste. You can also edit the shape of polylines in the new version: create a polyline, select it, and choose "Reshape" from the "Edit" menu.

10.4. Demo-Multifont

`Demo-Multifont` has been changed to use the new `motif-menubar`. You can test the new keybindings of the `multifont-text` object by following the instructions printed out by the demo.

The new `lisp-mode` feature of `multifont-text` is also shown in this demo. Select "Toggle Lisp Mode" from the "Edit" menu, and type in a lisp expression (like a `defun` definition). As you hit return, the next line will be automatically indented according to standard lisp conventions. Hitting the tab key will re-indent the current line.

11. Gilt

The Gilt Interface builder has been significantly rewritten. This makes Gilt a useful tool that can be used to build real application windows. The major changes include:

- Unifying the two versions, so there is now only one Gilt, and it includes both widget sets.
- Inclusion of more widgets, including the pop-up widgets like menubars and option-buttons.
- Allowing properties to be set on multiple widgets at once.
- Specifying some filtering and enabling conditions interactively using the `Control` commands and dialog boxes

11.1. Unified Gilt

Previously there were two versions of Gilt: the standard `Gilt` which contained the Garnet look-and-feel widgets, and `Motif-Gilt` for the Motif look and feel. Now, there is only one version, so you `(load Garnet-Gilt-Loader)` or `(garnet-load "gilt:gilt-loader")`.

When you start Gilt, you specify which widget set you want to start with: Motif or Garnet as a required parameter to `do-go`:

```
(gilt:do-go :motif)
or
(gilt:do-go :garnet)
```

You can change widget sets after the program has started using the `Load Other Gadgets` command.

The user interface for Gilt itself now only uses the Motif look and feel.

11.2. Gadgets

The one Gilt will now let you place Motif or Garnet look and feel widgets into your window (you can mix and match if you want). To switch back and forth, use the `Load Other Gadgets` command in the `File` menu. You can only see one gadget window at a time.

The new Motif Gadgets window is shown in Figure 11-1.

The gadgets are, from top to bottom, left to right, a menubar, buttons, a fixed menu, a scrolling menu, check boxes (choose one or more), radio buttons (choose only one), OK-Cancel, OK-Apply-Cancel, an option button, which pops up a menu of choices and shows the current choice in the button, a vertical scroll bar, a vertical slider, a horizontal scroll bar, a horizontal "trill device" to input a number, a circular gauge, a text input field, and then a bunch of objects you can use for decorations and labels: a pixmap (a bitmap with depth and color), a regular bitmap, a line, a rectangle, a Motif shadow box (which can either look like it comes out or goes in depending on the setting of the `:depressed-p` parameter), regular text, multi-font text, or a background that fills the window.

The new Garnet Gadgets window is shown in Figure 11-2.

The gadgets are, from top to bottom, left to right, a menubar, buttons, a scrolling menu, an option button, which pops up a menu of choices and shows the current choice in the button, a pop-up-menu button, that also pops up a menu of choices but doesn't show the current value, OK-Cancel, check boxes (choose one or more), radio buttons (choose only one), a regular menu, a horizontal scroll bar, a horizontal slider, a vertical scroll bar, a vertical slider, OK-Apply-Cancel, a circular gauge, a horizontal "trill device" to input a number, a text input field where the box grows depending on the size of the text, a scrolling text input field where the text scrolls and the box stays the same size, and then a bunch of objects you can use for

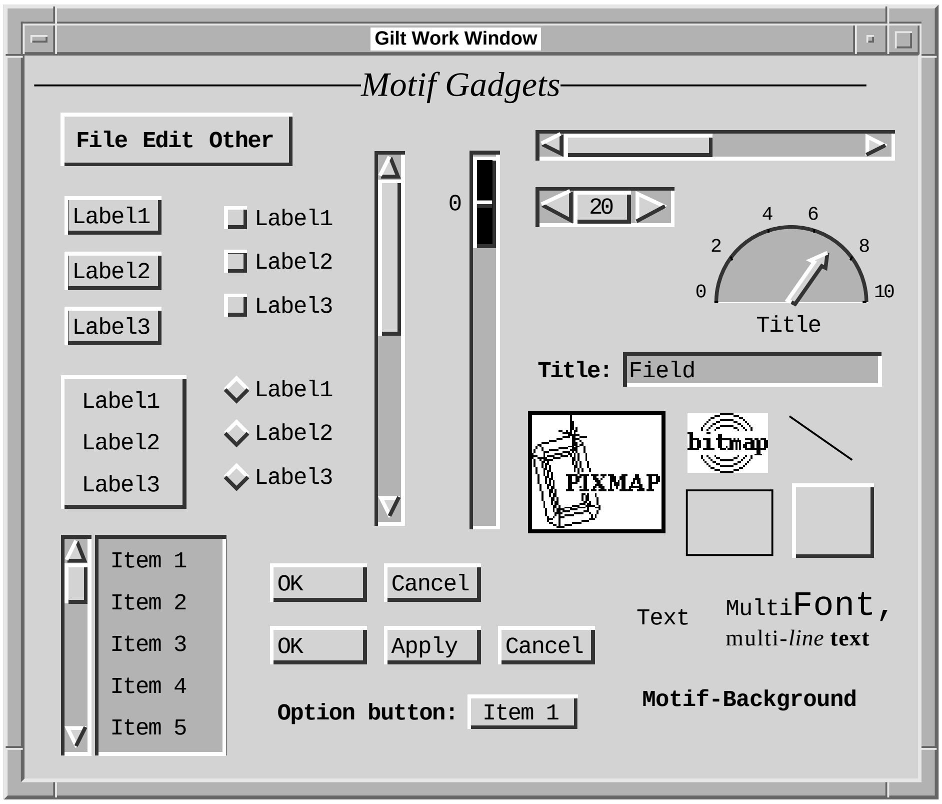


Figure 11-1: The new Gilt Motif gadgets window.

decorations and labels: regular text, multi-font text, a rectangle, a line, a bitmap, and a pixmap (a bitmap with depth and color).

11.3. Editing Strings

To edit string labels, simply click to select them, and then click again with the left button to begin editing. The fonts of multi-font strings can be edited using the keyboard commands described in section 5.1.3.1.

Editing the strings of most gadgets remains as before: select the gadget (to get the selection handles around it) and then click in a string to get the string cursor, and then type the new string, and hit RETURN when done. If you make the string empty (e.g., by typing control-u), and hit return, that button of the gadget will be removed. If you edit the last item of the gadget and hit control-n instead of return, then a new item will be added to the gadget. The strings can also be edited by editing the `:items` property in the property sheet that appears from the `Properties` command.

To edit the strings in a pop-up menu, like a menubar or an option button, click once with the left button to select the gadget, and then click again to pop-up the submenu. You can now click in the submenu to edit any of the items. Use control-n in the last item to add new items or control-u and return to remove items. To edit the top-level labels of a menubar, you need to click the left button three times: once to select the gadget, once to bring up the submenu, and a third time to begin editing. Click outside to make the popped-up menu disappear.

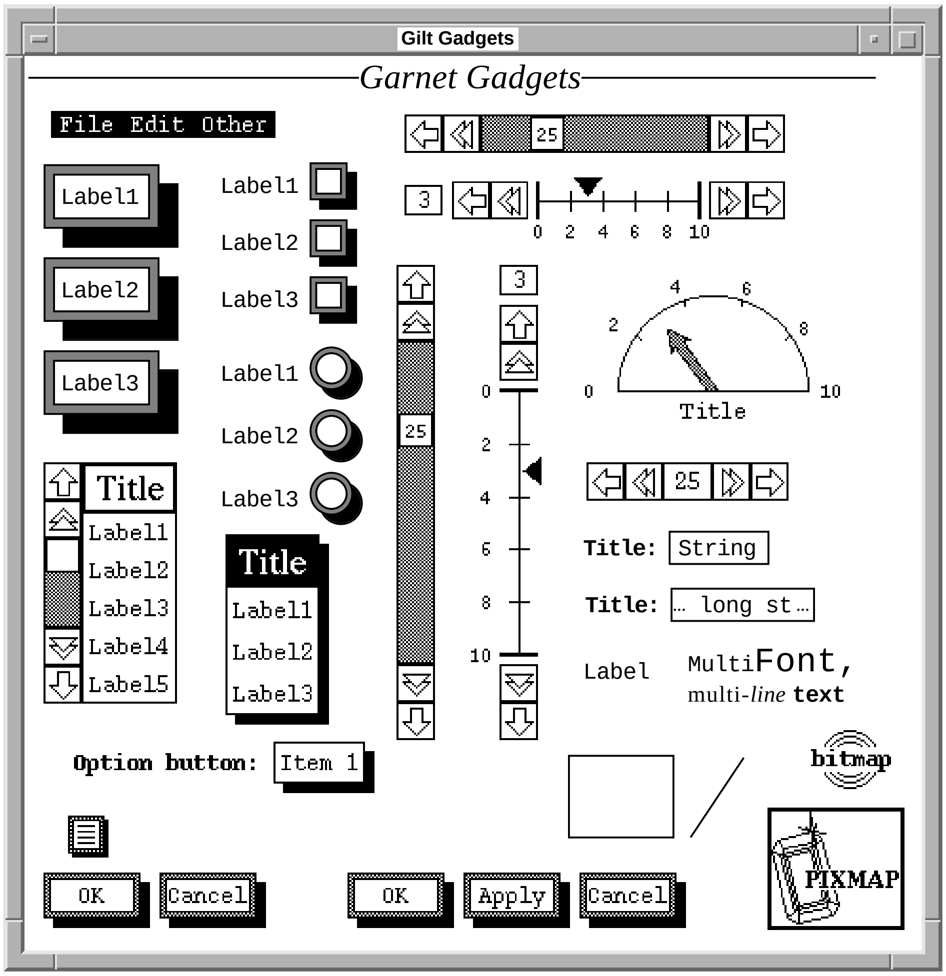


Figure 11-2: The new Gilt Garnet look-and-feel gadgets window.

11.4. Commands

Because there are now many more commands in Gilt, we changed the main command menu to be a menubar at the top of the Commands window, rather than a set of buttons. This also means that you can give the commands using keyboard shortcuts when the mouse is in the main Work Window. The particular shortcuts are listed on the sub-menus of the main menubar.

Many of the editing commands have changed names. The commands are:

`Cut` - remove the selected item(s) but save them in the clipboard so they can later be pasted.

`Copy` - copy the selected item(s) to the clipboard so they can later be pasted.

`Paste` - place a copy of the items in the clipboard onto the window.

`Duplicate` - place a duplicate of the selected items onto the window.

`Delete` - delete the selected objects and don't put them into clipboard. This operation can be undone with the `Undo Last Delete` command.

`Delete All` - delete all the objects in the window. This operation can be undone with the `Undo`

`Last Delete` command.

`Undo Last Delete` - undoes the last delete. All the deletes are saved, so this command can be executed multiple times to bring back objects deleted earlier.

`Select All` - select all the objects in the window (including the background object).

`To Top` - make the selected objects not be covered by any other objects.

`To Bottom` - make the selected objects be covered by all other objects.

`Properties...` - bring up the properties window (see section 11.5).

`Align` - bring up the dialog box to allow aligning of the selected objects with respect to the first of the objects selected.

Many of these commands are now implemented with the functions in the `Standard-Edit` mechanism, described in section 8.1.

11.5. Properties

In this version of Gilt, you can now select multiple items and bring up a property sheet on all of them. The property sheet will show the union of all properties of all objects. If multiple objects have the same property name, then the value of the property for the first object selected is shown.

When you edit the value of a property and then hit return (or when you hit OK for properties that pop up dialog boxes), the property sheet will immediately set that property into all objects for which the property is defined. Thus, you can change the `:foreground-color` of all the objects by executing `Select All`, bringing up the `Properties...`, and then editing the foreground-color property. If you start to edit a property but change your mind, hit `Control-G` if text editing or `Cancel` if a dialog box. The `Done` button hides the property sheet.

The left, top, width and height number boxes displayed in the main Gilt window will now also work on multiple objects. When multiple objects are selected, they show the values for the bounding box of all the objects, and when you edit one and hit RETURN, that value is applied to all objects for which it is settable.

11.6. Value and Enable Control

A sophisticated module for modifying the values of gadgets in the Gilt work-window has been added, along with a corresponding module to modify when a gadget should be active (or grayed-out). These are called the `Value Control` and `Enable Control` modules, and can be invoked from the "Control" submenu in the Gilt menubar.

These modules implement the ideas discussed in Brad Myers' paper "Separating Application Code from Toolkits", presented at the 1991 UIST conference. The paper includes examples of how to use this feature, but a full set of documentation is still pending. We plan to provide details for using these modules in the Beta release of this document, or in the official Garnet 2.2 Reference Manual.

11.7. Miscellaneous

Output from the GarnetDraw utility program can now be read into Gilt programs (using the standard `Open` command), which might be useful for adding decorations to programs.

12. Lapidary

Some new capabilities have been added to Lapidary that give you access to more Garnet features through the dialog boxes, and that give you more options for selecting objects. The Box and Line Constraint dialog boxes are now self-contained gadgets, so you can use them independently of Lapidary. The full documentation for using the new constraint gadget dialog boxes is in section 12.2.

12.1. New Lapidary Features

12.1.1. Interactor Dialog Boxes

A number of new features have been added to the various interactor dialog boxes:

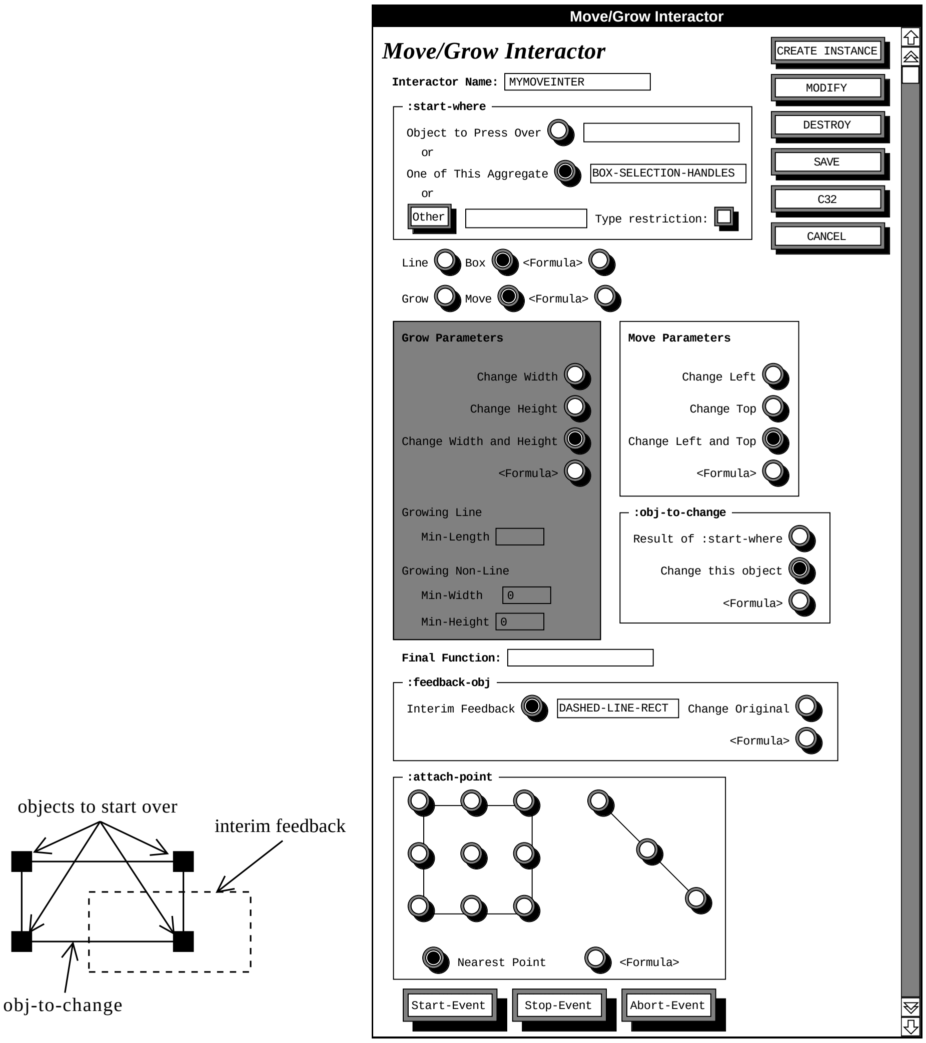
- Multiple feedback objects can be specified for an interactor. If the user selects multiple objects in a drawing window, and then presses one of the feedback radio buttons in an interactor dialog box, Lapidary will pop up a formula window and ask the user to enter a formula that determines when to use each feedback object. All of the selected objects will then be treated as feedback objects for the interactor. To reference one of the feedback objects in the formula, select it as you normally would using C32 and insert it using the `Insert From Mouse` button in the formula window.

- The user can directly choose an obj-to-change in the move/grow and text interactor dialog boxes. Lapidary will automatically construct a formula so that the interactor changes the correct object at run-time. For example, in Figure 12-1, the user wants the move interactor to start over one of the selection handles, but wants the object highlighted by the selection handles moved. The user can specify that the interactor should start over the selection handles by selecting the aggregate containing the selection handles and pressing the `One of This Aggregate` radio button in the move/grow dialog box. The user can specify that the object highlighted by the selection handles at run-time should be the object changed by selecting the example object that the selection handles currently highlight and pressing the `Change this object` button in the move/grow dialog box. Previously users would have had to enter a formula if the obj-to-change was different than the start-where. Occasionally Lapidary may not be able to determine from the start-where objects which object should be changed at run-time. In this case Lapidary will give the user the choice of entering a formula or of having the example object selected as the obj-to-change be the actual object changed at run-time.

- Lapidary now attempts to add interactors to aggregates, so that when aggregates are saved, the interactors will be saved as well. Lapidary queries the user as to whether it is acceptable to insert the interactor into an aggregate, so the user has the option of preventing Lapidary from doing it.

12.1.2. Constraint Menus

The Box Constraint and Line Constraint dialog boxes, used to constrain objects to each other, can now be used separately from Lapidary (see section 12.2). The constraint menus have been changed so that they display the current position and size of a primary selection. By pressing the new `Show Constraints` button in the constraint menus, the user can see what types of constraints are on the slots of an object. If two objects are selected, Lapidary will display the types of the constraints between the two objects. Finally, both box-like and line-like objects can be constrained to the center of a line by selecting the new centering button in the `obj-to-reference` section of the Line Constraint dialog box.



(a) (b)

Figure 12-1: (a) The various parts for a move behavior; and (b) the dialog box for specifying the move/grow interactor. The user can now directly select the obj-to-change and Lapidary will create a formula that automatically selects the correct object to change at run-time.

12.1.3. Selection Techniques

A number of features have been added to make it easier to select objects and to perform instancing/duplication on selected objects. The most obvious change is that users can now select either a ‘‘leaves’’ or ‘‘top-level objects’’ selection mode in the editor menu (Figure 12-2). ‘‘Leaves’’ mode causes Lapidary to select leaf elements of an aggregate, while ‘‘top-level objects’’ mode causes Lapidary to select top-level aggregates (objects that do not belong to an aggregate will be selected in either mode). Additional clicks over an object will cause Lapidary to cycle through the aggregate hierarchy as before. For example, when the user clicks on the label shown in Figure 12-3.a, and Lapidary is in "top-level objects" mode, the entire list element is selected (Figure 12-4.a). If the user clicks on the label again, the label is selected (Figure 12-4.b). Clicking once more with the mouse causes the key-box to become selected (Figure 12-4.c). Finally, one more click causes the list element to be selected, at which point the cycle repeats itself. In "leaves" mode, the label would be the first object selected, then the key box, and finally the list element.

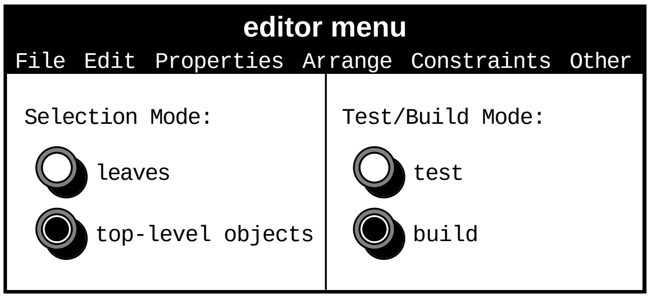
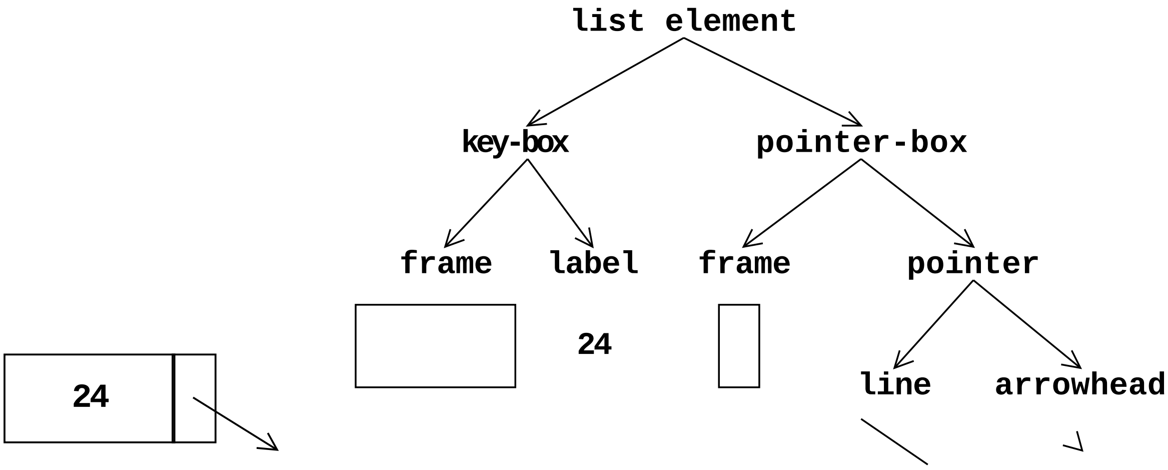


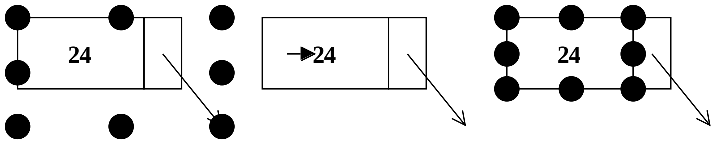
Figure 12-2: The user can cause Lapidary to select leaves of aggregates or top-level aggregates by choosing the appropriate selection mode in the editor menu window.



(a) (b)

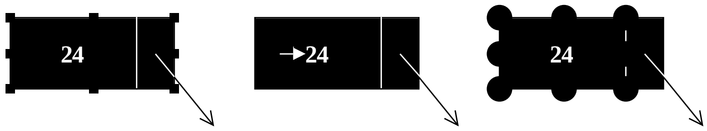
Figure 12-3: A list element (a) and the objects used to build this list element (b).

The user can also select the aggregates of covered objects. Previously only leaf elements of covered objects could be selected. As explained in the Lapidary reference manual, covered objects can be selected using either `shift-control-leftdown` for primary selections or `shift-control-middledown` for secondary selections (these bindings can be changed by modifying the mouse-bindings.lisp file in the lapidary directory). In ‘‘top-level objects’’ mode, the top-level aggregate of a covered aggregate will now be chosen. If the add to selection mouse buttons (shift-leftdown for primary selections and shift-middledown for secondary selections) are clicked while the mouse cursor is over one of the covered object’s selection handles, then the selection will cycle through the aggregate hierarchy (Figure 12-5).



(a) (b) (c)

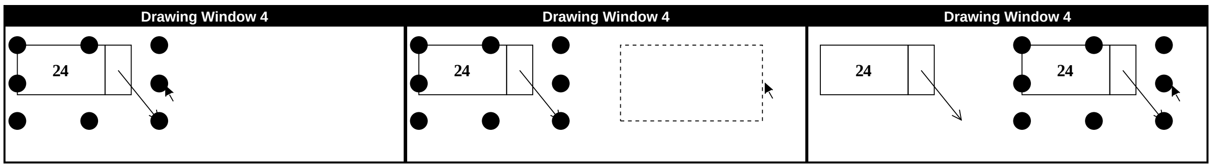
Figure 12-4: As the user repeatedly clicks the mouse button over an object, the selection cycles through the aggregate hierarchy shown in Figure 12-3. If Lapidary is in "top-level objects" mode, then the list element is initially selected (a). A second click selects the label (b), and a third click selects the key-box (c).



(a) (b) (c)

Figure 12-5: Selecting a covered object in ‘‘leaves’’ mode. The label is covered by an xor feedback object, so the feedback object is the initial selection (a). Clicking the `shift-control-leftdown` mouse button pushes the selection down to the covered label (b). Clicking the add to selection button (`shift-leftdown` over the feedback arrow causes the selection to cycle up to the next level in the aggregate hierarchy, in this case, the key-box (c).

Finally, Lapidary now supports duplicating and instancing operations on the selection handles (as before, the operations can also be performed by selecting them from the pulldown ‘‘edit’’ menu in the editor menu window). If the user clicks the appropriate mouse button while over a selection handle (Figure 12-6.a), either a duplicate (`shift-rightdown`) or instance (`control-rightdown`) of the object will be created and the user can move the object to the appropriate location in the window, or even to another window (Figures 12-6.b and 12-6.c). The selection handle constrains the initial movement of the new object, which facilitates the alignment of objects.



(a) (b) (c)

Figure 12-6: Objects can be duplicated or instanced by clicking on one of the selection handles (a), dragging the new object to the appropriate location (b), and dropping it (c).

12.2. The Constraint Gadget

The constraint gadget allows a set of standard constraints to be attached to various Garnet objects. The constraint gadget provides two menus: a box constraint menu for specifying box-type constraints (Figure 12-7) and a line constraint menu for specifying line-type constraints (Figure 12-8). The module can be loaded independently of Lapidary with `(garnet-load "lapidary:constraint-gadget-loader")`.

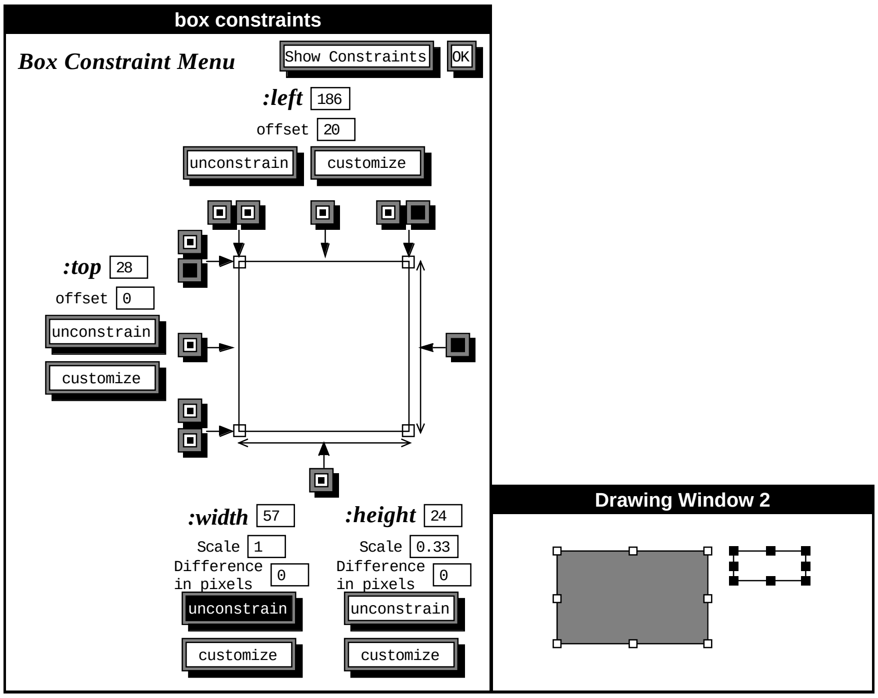


Figure 12-7: The constraint gadget's menu for box-like objects on the left, and a drawing window on the right. The white rectangle in the drawing window is the object to be constrained and the gray rectangle is the object to be referenced in the constraint. The white rectangle is constrained to be offset from the right of the gray rectangle by 20 pixels, and aligned at the top-inside of the gray rectangle. The white rectangle's width is not constrained and it is 33% as tall as the gray rectangle. If the gray rectangle changes, the white one will be adjusted automatically.

12.2.1. Box Constraint Menu

The box constraint menu allows the `:left`, `:top`, `:width`, and `:height` of objects to be constrained. Constraints on the left slot may include:

- Left-Outside - attach the right side of the constrained object to the left side of the reference object;
- Left-Inside - align the left side of the constrained object to the left side of the reference object (this is left alignment);
- Center - center the constrained object with respect to the reference object's center;
- Right-Inside - attach the right side of the constrained object to the right side of the reference object (this is right alignment); and

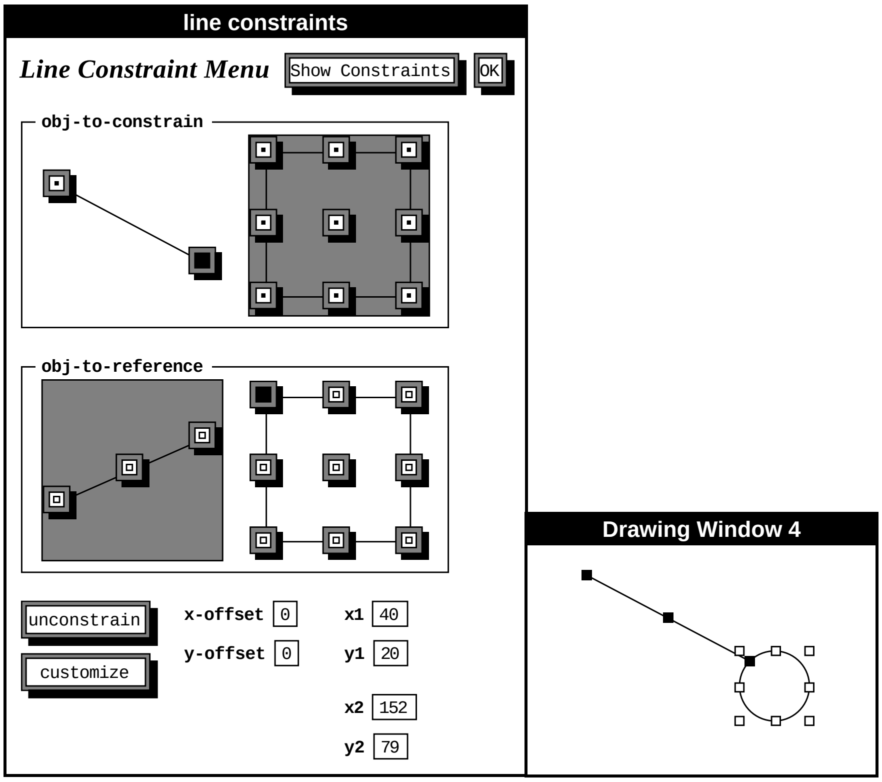


Figure 12-8: The constraint-gadget menu for line-like objects on the left, and a drawing window on the right. The arrow in the drawing window is the object to be constrained and the circle is the object to be referenced in the constraint. In the ‘‘obj-to-constrain’’ section of the constraint menu, the line feedback object has been rotated so that it has the same orientation as the selected arrow, and the box feedback object has been disabled (grayed-out). In the ‘‘obj-to-reference’’ section the line feedback object has been disabled since the object to be referenced in the constraint is a box-like object. The darkened buttons on the right endpoint of the line feedback object and the left corner of the box feedback object indicate that the right endpoint of the arrow is attached to the left corner of the circle.

- Right-Outside - attach the left side of the constrained object to the right side of the reference object.

Constraints on the top slot may include:

- Top-Outside - attach the bottom side of the constrained object to the top side of the reference object;
- Top-Inside - align the top side of the constrained object to the top side of the reference object (this is top alignment);
- Center - center the constrained object with respect to the reference object’s center;
- Bottom-Inside - attach the bottom side of the constrained object to the bottom side of the reference object (this is bottom alignment); and
- Bottom-Outside - attach the top side of the constrained object to the bottom side of the reference object.

An object may be constrained to another object's width or height using the width and height portions of the box constraint menu.

The offset boxes for the left and top slots, and the scale and difference boxes for the width and height slots provide ways of fine-tuning the constraints. The offset boxes for the left and top slots allow the user to add a constant pixel offset to a constraint. For example, the value of 20 in the left offset box in Figure 12-7, causes the white rectangle to be placed 20 pixels to the right of the gray rectangle. When the user presses a center button, the label for the offset box turns to percent, since the user enters a percentage for the center (50% is the default). In this case the constraint has the form ‘‘reference-obj.left + (percent * reference-obj.width) - constrained-obj.width / 2’’. The scale and difference boxes for the width and height slots allow the user to create constraints of the form ‘‘scale * {width, height} + difference’’.

Pressing an `unconstrain` button will remove a constraint from the appropriate slot of the constrained object, if there is one. Pressing a `customize` button will bring up C32 with a formula box for the appropriate slot. More information on using C32 with the constraint gadget is included in Section 12.2.4.

The boxes next to the slot names allow the user to enter specific values for an object's left, top, width, and height. This option is useful when the designer wants specific values for these slots, rather than the approximate values obtained using direct manipulation. When a value is entered in one of these boxes, the constraint in that slot is destroyed if one exists. When an object is selected, these boxes reflect the values of the object's bounding box.

12.2.2. Line Constraint Menu

The line constraint menu supports three types of constraints: line to line, line to box, and box to line. To make it clear what type of constraint is to be applied, the line constraint menu displays the types of the object to be constrained and the object to be referenced in the constraint (either a line or a box). It does so by displaying two sections, an obj-to-constrain section and an obj-to-reference section. Each section displays both a line and a box. The line objects have buttons indicating that endpoints can be constrained and the box objects have buttons indicating that edges, corners, and the center can be constrained. The line in the obj-to-reference section has an additional button in its center, indicating that objects can be constrained to the center of a line (the line in the obj-to-constrain menu does not have a center button, but the user can stll constrain the center of a line by pressing the `customize` button in the line constraint menu). When an object is selected, the line constraint menu grays out whichever feedback object does not correspond to the type of the selected object. In addition, if the selected object is a line, then the appropriate feedback line is rotated, so that its alignment matches the alignment of the selected line. This rotation makes it clear which endpoint of the selected line is being constrained, or being used as a reference in a constraint.

The line constraint menu allows either endpoint of a line to be constrained, and any corner, side, or center of a box object to be constrained. To create a constraint, the designer pushes the appropriate button on the enabled feedback objects in both the obj-to-constrain and the obj-to-reference sections. The connection between two points can be fine-tuned using the `x-offset` and `y-offset` fields. For example, if the designer wanted a line to be connected to the left side of a box, 10 pixels from the top, the designer could constrain the line to the left corner of the box, and enter a y-offset of 10.

The `x1`, `y1`, `x2`, and `y2` boxes allow the designer to specify an exact pixel position for the endpoints of a line. When an object is selected, these boxes are updated to reflect the values of the line's endpoints.

The `unconstrain` button destroys the constraints on the endpoint of a line or the position of a box. If a line is selected, then a button in the `obj-to-constrain` section denoting the appropriate endpoint should be selected. If neither endpoint button is selected, the constraint gadget tries to figure out which endpoint to unconstrain by checking if only one endpoint is constrained. If only one endpoint is constrained, it unconstrains this endpoint. Otherwise, if both endpoints are constrained, the constraint gadget asks the user to select one of the endpoint buttons in the obj-to-constrain section of the line constraint menu.

The `customize` button brings up C32. Since it is unclear whether the user will constrain both endpoints of a line, or just one of the x and y coordinates, the constraint gadget does not bring up formula boxes for the slots.

12.2.3. Programming Interface

The constraint gadget can be created (or made visible, if already created) by executing one of the `-do-go` or `show-` functions described in section 12.2.3.2. Certain slots of the gadget, described in section 12.2.3.1, are then set with the objects to be constrained. When the user operates the buttons in the dialog box, constraints will be set up among the indicated objects.

12.2.3.1. Slots of the Constraint Gadget

The constraint gadget exports one object called `gg::*constraint-gadget*`. This object contains four settable slots:

- `:obj-to-constrain` - The object which should be constrained. This slot expects only one object — it will not take a list.
- `:obj-to-reference` - The object which should be referenced in the constraint. This slot expects only one object — it will not take a list.
- `:top-level-agg` - The top level aggregate containing constrainable objects. If the aggregate associated with a window is the top level aggregate, this slot may be left NIL (the default). However, if, for example, the window contains an editor aggregate and a feedback aggregate, then the `:top-level-agg` slot should be set to the editor aggregate.
- `:custom-function` - A function that is executed whenever a constraint is attached to a slot. The function should take three parameters: an object, a slot, and a formula. The function is called after the formula has been installed on the slot, but before the formula has been evaluated. *This function is not called when the user calls the c32 function and provides a c32-custom-function as a parameter (see Section 12.2.3.2 for details on this function and its parameters*. The function is not called in this case since the constraint gadget does not install the formula if the c32-custom-function is provided.)

It is also possible to prevent either the box-constraint or line-constraint menus from attaching a constraint to a slot by adding the slot's name to a list in the `:do-not-alter-slots` of an object. For example, to prevent a constraint from being attached to the `:width` or `:height` slots of a text object, the list `'(:width :height)` could be placed in the object's `:do-not-alter-slots` slot. If the user tries to attach a constraint to that slot, an error box will be popped up indicating that a constraint cannot be attached to that slot. C32 does not recognize the `:do-not-alter-slots`, and therefore the box-constraint and line-constraint menus cannot prevent the user from inserting a formula into a forbidden slot if a customize button is chosen.

12.2.3.2. Exported Functions

The following functions are exported from the constraint gadget module:

`lapidary:Box-Constraint-Do-Go` [*Function*]

`lapidary:Line-Constraint-Do-Go` [*Function*]

These functions create the Box and Line Constraint dialog boxes. They should not be executed multiple times, since there is only one `constraint-gadget` object. If the user clicks on an "OK" button and makes the dialog boxes invisible, then the following functions can be called to make them visible again:

`lapidary:Show-Box-Constraint-Menu` [*Function*]

`lapidary:Show-Line-Constraint-Menu` [*Function*]

These function make the Box and Line Constraint dialog boxes visible. They can only be called after the `-do-go` functions above have been called to create the dialog boxes.

`gg:c32 &optional` *object slot* [*Function*]
`&key` *left top c32-custom-function prompt*

This function causes c32 to come up, with the *object* displayed in the first panel of the c32 window. The formula in *slot* will be displayed in c32's formula editing dialog box. The keyword parameters are as follows:

left, *top* - Controls placement of query box that users use to indicate that they are done with C32.

c32-custom-function - A function to be executed when the user hits the OK button in a formula window in C32. The function should take three parameters: an object, a slot, and a formula. If a custom function is provided, the formula will not be installed in the slot (thus the function in :custom-function will not be called—it must be called explicitly by the *c32-custom-function* if it should be executed). This gives the *c32-custom-function* an opportunity to defer the installation of the formula. For example, in Lapidary, the user can create formulas that define the values of various slots in an interactor, but until the user presses the ''create-interactor'' or ''modify'' buttons, the formulas should not be installed. Thus the Lapidary *c32-custom-function* places the formulas on a queue, but does not install them.

The constraint gadget stores the links that this formula uses in a meta-slot in the formula called `:links`. Like the formula, the links are not installed. That is, the link slots do not exist (unless another formula already uses them). Because the links have not been installed, the constraint gadget stores the links and the objects they point to in another meta-slot in the formula called `:links-and-objs`. The contents of this slot have the form (list (cons (link-name object))*). Links that already exist because another formula uses them will not be on this list.

The *c32-custom-function* or the application can install the links by calling the c32 function `install-links` which takes a formula and the object that the links should be installed in as arguments (the object that is passed to *c32-custom-function* is the object that should be passed to install-links). Install-links will create the links, and if the link points to an object that is in the same aggregate as the object containing the link, install-links will create a path to the reference object and store it in the link slot. Install-links destroys the `:links-and-objs` slot, so the *c32-custom-function* or application should take care to save the contents of this slot if they need to make further use of this information.

cg-destroy-constraint - Destroys a constraint created by the constraint gadget. Required parameters are an object and a slot.

valid-integer-p - Determines if a string input by a Garnet gadget contains a valid integer. If it does not, the gadget's original value is restored and an error message is printed. Required parameters are a gadget and a string.

`c32::Install-Links` *formula obj* [*Function*]

This function is provided by c32, though it is not exported. As mentioned above, it is useful for installing links when a custom function is provided in c32. The *formula* should have a `:links-and-objs` slot, whose value should be a list of the form ((*link-slot-name object*) (*link-slot-name object*) ...). The *object* parameter names the object which the links should be installed in.

12.2.3.3. Programming with Links

Each constraint contains indirect references to objects rather than direct references. The set of links it uses to make these indirect references is contained in the `:links` meta-slot of the formula and the name of the offset slot it uses is contained in the `:offset` meta-slot. If the formula involves the width or height slots, there is also a `:scale` meta-slot, containing the name of the scale slot that the formula uses. The constraint gadget generates link and offset names by appending the suffixes `-over` and `-offset` to the name of the slot that is being constrained. For example, if the left slot is being constrained, the link name will be `:left-over` and the offset name will be `:left-offset`. These slot names are stored in a formula's `:links` and `:offset` meta-slots. For width and height slots, scale names are generated by appending the suffix `-scale` to the slot name. Thus the scale slot for a height constraint would be named `:height-scale`. When C32 generates link names, it generates them by appending a number to the prefix `link-`. Thus it generates links such as `:link-0` and `:link-1`.

12.2.4. Custom Constraints

When the user selects the custom constraint option in any of the constraint menus, the constraint gadget brings up the C32 spreadsheet and a formula window for the desired slot. The user should enter a formula and press OK (or cancel to stop the operation). Both the OK and cancel buttons in the formula window will make C32 disappear.

The constraint gadget modifies C32 in a number of ways that are important for a user to know. First, it generates indirect references to objects rather than direct references. A direct reference explicitly lists an object in a constraint, whereas an indirect reference accesses the object indirectly through a link. For example, `(gv RECT1 :left)` is a direct reference to `RECT1`, whereas `(gvl :link0 :left)` is an indirect reference to `RECT1` (this assumes that a pointer to `RECT1` is stored in `:link0`). If the user always generates references using the C32 functions `Insert Ref from Mouse` and `Insert Ref from Spread`, then the constraint gadget will automatically generate indirect references and create appropriate link names. The user can edit these link names by finding them in the spreadsheet and modifying them. However, if the user inserts the references by typing them in, then the user should take care to use the `gvl` form and create the appropriate links. When the formula is completed, the constraint gadget checks whether there are any direct references in the formula and generates a warning if there are. At this point the user has the option of editing the formula or continuing with the formula as is. If the user chooses to leave direct references in the formula, the constraint gadget may not be able to generalize it, so the formula may behave strangely if it is inherited.

The second change the constraint gadget makes is in copying formulas. The constraint gadget copies all the links that the formula references to the object which is receiving the copied formula. If the links should point to new objects, the user must manually change them by selecting the `Show All Slots` option in C32 and editing the appropriate links (the names of the links that need to be modified can be found by looking at the formula).

12.2.5. Feedback

The user can determine which constraints are attached to an object by selecting the object and an optional second object that the object may be constrained to, and then selecting the `Show Constraints` option. The appropriate constraint buttons will be highlighted and the offset fields set to the correct values. If only one object is selected, then all constraints that the constraint menu can represent will be shown. For example, the box constraint menu would display the constraints on the left, top, width, and height slots. If there are two selections, a constrained object and a reference object, then only the constraints in the constrained object that depend on the reference object are shown.

Index

Table of Contents