Garnet Reference and User Manual

Short Contents

Table of Contents

3   Garnet Tutorial 41

4   KR: Constraint-Based Knowledge Representation 97

8   Garnet Gadgets   399

1 Overview

1.1 Introduction

The Garnet research project in the School of Computer Science at Carnegie Mellon University is creating a comprehensive set of tools which make it significantly easier to create graphical, highly-interactive user interfaces. The lower levels of Garnet are called the "Garnet Toolkit," and these provide mechanisms that allow programmers to code user interfaces much more easily. The higher level tools allow both programmers and non-programmers to create user interfaces by just drawing pictures of what the interface should look like. Garnet stands for **G**enerating an **A**malgam of **R**eal-time, **N**ovel **E**ditors and **T**oolkits.

At the time of this writing, the Garnet toolkit is in use by about 80 projects throughout the world. This document contains an overview, tutorial, and reference for the Garnet System.

This chapter describes version 3.0 of Garnet. It replaces all previous versions and the change documents for versions 1.3, 1.4, 2.0, 2.1, and 2.2.

Garnet is written in Common Lisp and can be used with either Unix systems running X windows or on the Mac. Therefore, Garnet is quite portable to various environments. It works in virtually any Common Lisp environment, including Allegro (Franz), Lucid, CMU, Harlequin, CLISP, AKCL, and Macintosh Common Lisps. The computers we know about it running on include Sun, DEC, HP, Apollo, IBM RT, IBM 6000, TI, SGI, NeXTs running X11, PC's running Linux, Macs, and there may be others. Currently, Garnet supports X11 R4 through R6 using the standard CLX interface. Garnet does *not* use the standard Common Lisp Object System (CLOS) or any X toolkit (such as Xtk or CLIM).

Garnet has also been implemented using the native Macintosh QuickDraw and operating system. To run the Macintosh version of Garnet, you need to have System 7.0 or later, Macintosh Common Lisp (MCL) version 2.0.1 or later, and at least 8Mb of RAM. The system takes about 10 megabytes of disk space on a Mac, not including the documentation (which takes an additional 8 megabytes). We find that performance of Garnet on MCL is acceptable on Quadra's, and fine on a Quadra 840 A/V. It is really too slow on a Mac II. To do anything useful, you probably need 12mb of memory. A PowerPC Mac does not work well for Lisp (see discussion on `comp.lang.lisp.MCL`).

More details about Garnet are available in the Garnet FAQ: `ftp://a.gp.cs.cmu.edu/usr/garnet/garnet/FAQ` which is posted periodically.

This document is a technical reference manual for the entire Garnet system. There have been many conference and journal papers about Garnet (See [Garnet Articles], page 22, for a complete bibliography). The best overview of Garnet is *GarnetIEEE,,Myers and others, 1990*. Section [Garnet Articles], page 22, includes instructions for retrieving some Garnet papers via FTP.

The project Amulet is sister project of Garnet with features similar to those in Garnet, but implemented in C++. See [Planned Future Extensions], page 21, for descussion how to get more information about the Amulet project.

1.2 Garnet Bulletin Board

There is an international bboard for Garnet users named `comp.windows.garnet`. Topics discussed on this bboard include user questions and software releases. There is also a mailing list called `garnet-users@cs.cmu.edu` which carries exactly the same messages as the bboard. If you cannot read the bboard in your area, please send mail to `garnet-users-request@cs.cmu.edu` to get on the mailing list.

You can report bugs to `garnet-bugs@cs.cmu.edu` which is read only by the Garnet developers.

1.3 Important Features of Garnet

Garnet has been designed as part of a research project, so it contains a number of novel and unique features. Some of these are:

- The Lapidary tool is the only interactive tool that allows application-specific graphics and new widgets to be created without programming.

- The Garnet Toolkit is designed to support the *entire* user interface of an application; both the contents of the application window and its menus and dialog boxes. For example, Garnet directly supports selecting graphical objects with the mouse, moving them around, and changing their size.

- It is *look-and-feel independent.* Garnet allows the programmer to define a new graphical style, and use that throughout a system. Alternatively, a pre-defined or standard style can be used, if desired.

- It uses a *prototype-instance* object model instead of the more conventional class-instance model, so that the programmer can create a *prototype* of a part of the interface, and then create instances of it. If the prototype is changed, then the instances are updated automatically. Garnet's custom object system is called KR. Garnet does not use CLOS.

- *Constraints* are integrated with the object system, so any slot (also called an "instance variable") of any object can contain a formula rather than a value. When a value that the formula references changes, the formula is re-evaluated automatically. Constraints can be used to keep lines attached to boxes, labels centered within rectangles, etc. (see Figure ⟨undefined⟩ [toolkit-SampleFig], page ⟨undefined⟩). Constraints can also be used to keep application-specific values connected with the values of graphical objects, menus, scroll bars or gauges in the user interface.

- Objects are *automatically refreshed* when they change. Pictures are displayed by creating graphical objects which are retained. If a slot of an object is changed, the system automatically redraws the object and any other objects that overlap it. Also, the system handles window refresh requests from X and the Mac.

- The programmer specifies the handling of input from the user at a high level using abstract *interactor* objects. Typical user interface behaviors are encapsulated into a few different types of interactors, and the programmer need only supply a few parameters to get objects to respond to the mouse and keyboard in sophisticated ways.

- There is built-in support for laying out objects in *rows and columns*, for example, for menus, or in *graphs or trees*, for example, to show a directory structure or a dependency graph.

- Two complete sets of *gadgets* (also called widgets or interaction techniques) are provided to help the programmer get started. These include menus, buttons, scroll bars, sliders, circular gauges, graphic selection, scrollable windows, and arrows. One set has the Garnet look and feel, and the other has the Motif look and feel. The Motif set is implemented entirely in Lisp on top of Garnet, to provide maximum flexibility. Note: There are no Macintosh look-and-feel gadgets. When you use Garnet on the Macintosh, the gadgets will **not** look like standard Macintosh widgets.

- Garnet is designed to be *efficient.* Even though Garnet handles many aspects of the interface automatically, an important goal is that it execute quickly and not take too much memory. We are always working to improve the efficiency, but Garnet can currently handle dozens of constraints attached to objects that are being dragged with the mouse.

- Garnet will automatically produce PostScript for any picture on the screen, so the programmer does not have to worry about printing.

- Gesture recognition (such as drawing an "X" over an object to delete it) is supported, so designers can explore innovative user interface concepts.

1.4 Coverage

Garnet is designed to handle interfaces containing a number of graphical objects which the user can manipulate with the mouse and keyboard.

Garnet is suitable for applications of the following kinds:

- User interfaces for expert systems and other AI applications.

- Box and arrow diagram editors like Apple Macintosh MacProject (which helps with project management).

- Graphical Programming Languages where computer programs can be constructed using icons and other pictures (a common example is a flowchart).

- Tree and graph editing programs, including editors for semantic networks, neural networks, and state transition diagrams.

- Conventional drawing programs such as Apple Macintosh MacDraw.

- Simulation and process monitoring programs where the user interface shows the status of the simulation or process being monitored, and allows the user to manipulate it.

- User interface construction tools (Garnet was implemented using itself).

- Some forms of CAD/CAM programs.

- Icon manipulation programs like the Macintosh Finder (which allows users to manipulate files).

- Board game user interfaces, such as Chess.

Figure 5.1 shows a simple Garnet application that was created from start to finish (including debugging) in three hours. The code for this application is shown in the sample program chapter, which begins on page Figure 5.1.
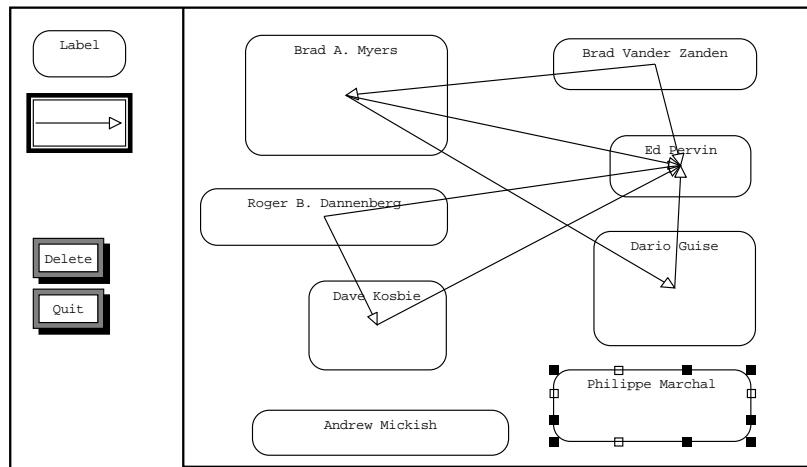
Figure 1.1: A sample Garnet application. The code for this application is listed the "Sample Garnet Program" section of this chapter, starting on page ⟨undefined⟩ [sampleprog-first-page.], page ⟨undefined⟩

Other examples of applications created using Garnet appear in the picture section of this chapter, starting on page ⟨undefined⟩ [apps], page ⟨undefined⟩.

1.5 Running Garnet From /afs

If you are running Garnet in X windows from CMU, or if you have access to AFS, you can access Garnet directly on the `/afs` servers. We maintain binaries of the official release version in machine- and lisp-specific subdirectories of `/afs/cs/project/garnet/`. If you are at CMU, you can skip section [retrieving], page 5, altogether, and just start lisp and load Garnet with:

    (load "/afs/cs/project/garnet/garnet-loader.lisp")

The CMU version of `garnet-loader.lisp` will attempt to determine what kind of machine and lisp you're using, and load the appropriate binaries for you. You will not have to supply or customize any pathnames.

1.6 How to Retrieve and Install Garnet

Garnet is available for free by anonymous FTP. There are different instructions for obtaining the software depending on whether it will be installed on a Mac or a Unix system (the code is the same, but the packaging is different).

1.6.1 Installation on a Mac

[Retrieving the Stuffit Files:]

Garnet 3.0 is available in `Stuffit` files that include the sources, the library files, the binary files compiled for Macintosh Common Lisp 2.0.1, and documentation. To download the Garnet collection that includes MCL binaries, use FTP or Fetch (a Mac file transfer utility) to connect to `a.gp.cs.cmu.edu` (128.2.242.7) and login as ''anonymous'' with your e-mail address as the password. Change to "binary" mode for FTP, or stay in "automatic" mode for Fetch, and download the `Stuffit` archive `/usr/garnet/garnet/mac.sit` Alternatively, you can get the BinHex version in text mode by retrieving `/usr/garnet/garnet/mac.sit.hqx` If you are using Fetch, it will automatically convert the BinHex file into a binary `.sit` file after it is installed on your Mac. If you used FTP to get the `.hqx` file, you will need to BinHex4 Decode the file. You should also retrieve one version of the documentation file:

    /usr/garnet/garnet/doc.sit
    /usr/garnet/garnet/doc.sit.hqx

If you do not have a version of `Stuffit`, you can also download the copy of `Stuffit_Expander` from the same directory to uncompress the Garnet archive. The `Stuffit` utility is a self-extracting archive that you only need to double-click on to install on your Mac. Be sure to use binary transfer mode in FTP if you are retrieving `StuffIt_Expander_.sea`.

[Unpacking the Stuffit Files:]

Once you have downloaded the `.sit` or `.sit.hqx` archives (and installed the `Stuffit_Expander`, if necessary), launch the `Stuffit` utility. Next, "Expand..." or "Open" the `mac.sit` archive, and choose a folder into which the uncompressed Garnet folder will be expanded. The instructions below assume you have installed the uncompressed folder at the top-level of your hard drive, and that your hard drive is named "Macintosh HD" (i.e., the uncompressed folder will become "Macintosh HD:Garnet:"). It is a good idea to expand

the `doc.sit` archive in the Garnet folder that was created by the first archive. For further instructions about printing the documentation, consult the README file in the doc folder.

[Preparing MCL Before Loading Garnet:]

When using Garnet, you may need to increase the amount of memory that is claimed by the Lisp application. You can change the memory claimed by MCL by selecting the MCL application in the Macintosh Finder and choosing "Get Info" from the Finder's "File" menu. Most Garnet applications will require that MCL use at least 6Mb of RAM, and using at least 12Mb is recommended. The default "Preferred size" for MCL is 3072K, so you will need to edit that value to be upwards of 6000K. You are only allowed to change this information when the application is NOT running, and it should be done before proceeding with the rest of these instructions. Note: All Lisp images saved from the MCL application will retain the new "Preferred size" value.

Before loading Garnet, you will need to compile several MCL library files that are used by Garnet. A compiler script for this procedure is provided in the Garnet collection. In the fresh MCL listener, load the file "Macintosh HD:Garnet:compile-MCL-libraries.lisp" (replacing the hard drive prefix with whatever is appropriate for your machine). After the script is finished, quit MCL and then launch MCL again.

[Loading Garnet:]

Using the MCL text editor, edit the file `garnet-loader.lisp` from the new Garnet folder (choose "File...", "Open..." from the MCL menubar to edit a file). Find the definition of the variable `Your-Garnet-Pathname` and set its value to the path of the new Garnet folder you created with `Stuffit`. All other subfolders of Garnet will be computed relative to this pathname. Save the new version of the `garnet-loader.lisp` file.

In the fresh lisp listener, load "Macintosh HD:Garnet:garnet-loader.lisp" (using whatever prefix is appropriate instead of "Macintosh HD:Garnet:"). Garnet will inform you as it loads each module, and will finally return with a prompt. At this point, Garnet is fully loaded and you are ready to try the Tour or some demos as discussed later in this chapter.

1.6.2 Installation on a Unix System

When running on X windows, Garnet uses the CLX interface from Lisp to X11. CLX should be supplied with every Lisp, and the following instructions assume that CLX has been installed correctly on your system. If you need help with CLX, you need to contact your Lisp vendor. We cannot help you acquire, compile, or install CLX, sorry.

Retrieving the TAR Files:

The Garnet software is about 9 megabytes. In order to make it easy to copy the files over, we have created TAR files, so to use the mechanism below requires double the storage area. Therefore, you first need to find a machine with enough room, and then create a directory called `garnet` wherever you want the system to be:

```
% mkdir garnet
```

Then, cd to the `garnet` directory.

```
% cd garnet
```

Now, ftp to `a.gp.cs.cmu.edu (128.2.242.7)`. When asked to log in, use `anonymous`, and your name as the password.

```
% ftp a.gp.cs.cmu.edu
Connected to A.GP.CS.CMU.EDU.
220 A.GP.CS.CMU.EDU FTP server (Version 4.105 of 10-Jul-90 12:07) ready.▇
Name (a.gp.cs.cmu.edu:bam): anonymous
331 Guest login ok, send username@node as password.
Password:
230 Filenames can not have '/..' in them.
```

Then change to the `garnet` directory (note the double `garnet`'s) and use binary transfer mode:

```
ftp> cd /usr/garnet/garnet/
ftp> bin
```

The files have all been combined into TAR format files for your convenience. These will create the appropriate sub-directories automatically. We have both compressed and uncompressed versions. For the regular versions, do the following:

```
ftp> get src.tar
ftp> get lib.tar
ftp> get doc.tar
```

To get the compressed version, do the following:

```
ftp> get src.tar.Z
ftp> get lib.tar.Z
ftp> get doc.tar.Z
```

Now you can quit FTP:

```
ftp> quit
```

Installing the Source Files:

If you got the compressed versions, you will need to uncompress them:

```
% uncompress src.tar.Z
% uncompress lib.tar.Z
% uncompress doc.tar.Z
```

Now, for each tar file, you will need to "untar" it, to get all the original files:

```
% tar -xvf src.tar
% tar -xvf lib.tar
% tar -xvf doc.tar
```

This will create subdirectories will all the sources in them. At this point you can delete the original tar files, which will free up a lot of space:

```
% rm *.tar
```

Now, copy the files `garnet-loader.lisp`, `garnet-compiler.lisp`, `garnet-prepare-compile.lisp`, and `garnet-after-compile` from the src directory into the `garnet` directory:

```
% cp src/garnet-*  .
```

Customizing the PathNames:

The file `garnet-loader.lisp` contains variables that should be set with the pathnames of your Garnet directory and the location of CLX for your lisp. You will now need to edit `garnet-loader.lisp` in an editor, and set these variables. Comments in the file will direct you how to do this. At the top of the file are the two variables you will need to set: `Your-Garnet-Pathname` and `Your-CLX-Pathname`. NOTE: If CLX is already loaded in your lisp image, you do not need to set the CLX variable.

Compiling Garnet to Make Binary Files:

Lisp requires very large address spaces. We have found on many Unix systems, that you need to expand the area that it is willing to give to a process. The following commands work in many systems. Type these commands to the C shell (csh). You might want to also put these commands into your `.login` file.

```
% unlimit datasize
% unlimit stacksize
```

Now, you will need to compile the Garnet source to make your own binaries. This is achieved by loading the compiler scripts. There is more information on compiling in section [compilinggarnet], page 16, below, and special instructions for compiling Garnet in CLISP are in section [clisp], page 13.

```
        ;; Only LispWorks users need to do the next two commands. See sec-
tion [lispworks], page 12.
        lisp> #+lispworks (load "src/utils/lispworks-processes.lisp")
        lisp> #+lispworks (guarantee-processes)

lisp> (load "garnet-prepare-compile")
lisp> (load "garnet-loader")
lisp> (load "garnet-compiler")
```

Now Garnet is all compiled and loaded, but a shell script still needs to be executed to separate the binary files from the source files. To set up for the next time, it is best to quit lisp now, and run the `garnet-after-compile` shell script. If your sources are not in a directory named `garnet/src` or your binaries should not be in a directory named `garnet/bin`, then you will need to edit `garnet-after-compile` to set the directories. Also, if your compiler produces binary files that do not have one of the following extensions, then you need to edit the variable `CompilerExtension` in `garnet-after-compile`: ".fasl", ".lbin", ".sbin", ".hbin", ".sparcf", ".afasl", or ".fas". Otherwise, you can just execute the file as it is supplied (NOTE: this is run from the shell, not from Lisp). You should be in the `garnet` directory.

```
% csh garnet-after-compile
```

Now you can start lisp again, and load Garnet:

```
lisp> (load "garnet-loader")
```

Details about how to customize the loading of Garnet are provided in section [loadinggarnet], page 16.

1.7 Directory Organization

All of the information about where various files of Garnet are stored is in the file `garnet-loader.lisp`. This file also defines the Garnet version number:

```
* user::Garnet-Version-Number
"3.0"
```

You may need to edit the `garnet-loader` file to tell Garnet where all the files are. Normally, there will be a directory called `garnet` with sub-directories called `src`, `lib` and

`bin`. In the `src` and `bin` directions will be sub-directories for all the parts of the Garnet system:

> `utils` - Utility files and functions.
>
> `kr` - KR object system.
>
> `gworld` - Mac routines for off-screen drawing (only used on the Mac)
>
> `gem` - Garnet's interface to machine-specific graphics routines (X and Mac)
>
> `opal` - Opal Graphics management system.
>
> `inter` - Interactors input handling.
>
> `aggregadgets` - Files to handle aggregates and lists.
>
> `gadgets` - Pre-defined gadgets, such as menus and scroll bars.
>
> `gesture` - Tools for handling gestures as input.
>
> `ps` - Functions for printing Garnet windows with PostScript.
>
> `debug` - Debugging tools.
>
> `demos` - Demonstration programs written using Garnet.
>
> `gilt` - The Gilt interface builder.
>
> `c32` - A spreadsheet for editing constraints among objects.
>
> `lapidary` - The Lapidary interactive tool.
>
> `contrib` - Files contributed by Garnet users that are not supported by the Garnet group, but just provided for your use.

1.8 Site-Specific Changes

If you are transferring Garnet to your site, you will need to make a number of edits to files in order for Garnet to load, compile and operate correctly. All users will need to edit the Garnet pathnames as discussed in section [pathnames], page 10, but relatively few users should need the other sections [optimization-settings], page 11, - [clisp], page 13. Garnet has been adjusted to load on the widest possible variety of lisps and operating systems with minimum modification.

Of course, if you change any `.lisp` files in the Garnet subdirectories (not including `garnet-loader.lisp`), you will need to recompile them (section [garnet-load], page 17), even if you do not need to recompile other parts of Garnet.

1.8.1 Pathnames

After you have copied Garnet to your machine and untar'ed the source files, the top level Garnet directory will contain the file `garnet-loader.lisp`. This one file contains the file names for all the parts of Garnet. You should edit this file to put in your own file names. The best way to do this is to set the `Garnet-Version` to be `:external` and edit the string at the top of the file called `Your-Garnet-Pathname` to say where the files are. This change is normally done during the compile procedure, already described in section [retrieving], page 5.

1.8.2 Compiler Optimization Settings

The variable `user::*default-garnet-proclaim*`, defined in `garnet-loader.lisp`, holds a list of compiler optimization flags and default values. These flags 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, CMUCL, LispWorks, and MCL that we have found work particularly well. You can override the default optimizations by defining 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.

1.8.3 Fonts in X11

In X11 R4 through R6, there are almost always a full set of fonts available with standard names. Garnet relies on these fonts being available on the standard font paths set up by X11. You can try loading Garnet and see if it finds the standard fonts.

If not, look in the file `garnet/src/opal/text-fonts.lisp`. This file constructs font names according to the standard X11 format (with lots of `"-*-*-*"`'s). You will have to substitute the names of fonts that are available at your installation.

1.8.4 Keyboard Keys

If your keyboard has some specially-labeled keys on it, Garnet will allow you to use these as part of the user interface. The file

```
define-keys.lisp
```

which is in the `garnet/src/inter` sub-directory, defines the mappings from the codes that come back from X11 and the Mac to the special Lisp characters or atoms that define the keys in Garnet.

For many machines, such as Suns, HP's, DECStations, and Macs, we have built in mappings for all of the keyboard keys. Since there are no Lisp characters for the special keys, they are named with keywords such as `:uparrow` and `:F1`. If some keys on your keyboard are not mapped to keywords, you can use the following mechanism to set this up.

To find the correct codes to use for each undefined key, load the Find-Key-Symbols utility with

```
(garnet-load "inter-src:find-key-symbols.lisp")
```

After loading this file, simply type the keys you need to find mappings for while input is focused on the Find-Key-Symbols window (you may have to click on the window's title-bar to change the input focus). Garnet will print out the code number of the keys you type.

Then, you can go into the file `define-keys.lisp` and edit it so the codes you found map to appropriate keywords.

Next, you might want to bind these keys to keyboard editing operations. If you want these to be global to all Garnet applications, then you can edit the files `textkeyhandling.lisp` and `multifont-textinter.lisp` which contain the default mappings of keyboard keys to text editing operations. The Interactors chapter contains full more information on how this works.

[If you surround your changes to all these files with `#+<your-switch>` and mail them back to us (`garnet@cs.cmu.edu`), then we will incorporate them into future versions so you won't need to continually edit the files.]

1.8.5 Multiple Screens

If you are working on a machine with only one screen, you need not pay attention to this section. However, certain machines, such as the color Sun 3/60, have more than one screen. The color Sun 3/60 has both a black-and-white screen (whose display name is "unix:0.0") and a color screen (whose display name is "unix:0.1"). If you type "echo $DISPLAY" in a Unix shell, you will get the display name of the screen you are working on; that name should look like "unix:0.*" where * is some integer.

Garnet assumes that the DISPLAY environment variable has this form of "display-name:displaynumber.screennumber", and extracts the display and screen numbers from that. If any fields are missing, then the missing display or screen number defaults to zero.

1.8.6 OpenWindows Window Manager

If you are running OpenWindows from Sun, you will need to add the following line to your `.Xdefaults` file to make text input work correctly:

```
OpenWindows.FocusLenience:   True
```

1.8.7 LispWorks

LispWorks is the Common Lisp sold by Harlequin Ltd. There is one peculiarity about Lisp-Works that requires an additional step before executing the `main-event-loop` background process of Garnet (Garnet uses multiprocessing by default in LispWorks – see the Interactors chapter, section "The Main Event Loop" for details). You need to perform this step both when **compiling** and **loading** Garnet (the appropriate steps are mentioned during the standard compile procedure in section [retrieving], page 5).

LispWorks has an unconventional "initialization phaze" to multiprocessing, which requires that 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.

On the other hand, it has been reported that putting the special steps for LispWorks in a `.lispworks` file may serve to automate the process a bit. To automatically initialize multiprocessing whenever LispWorks is started, put the following lines in your `.lispworks` file:

```
(progn
  (load "<your-garnet-pathname>/src/utils/lispworks-process.lisp")
  (guarantee-processes))
```

You will not be able to call `garnet-load` from your `.lispworks` file because the function will not have been defined when the file is read.

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`.

1.8.8 CLISP

CLISP is a Common Lisp (CLtL1) implementation by Bruno Haible of Karlsruhe University and Michael Stoll of Munich University, both in Germany. There are a couple of additional steps you must take to run Garnet in CLISP that are not required in other lisps.

Renaming .lisp files to .lsp

If you have an older version of CLISP, you will have to rename all of the source files from ".lisp" to ".lsp" before starting the procedure to compile Garnet. A `/bin/sh` shell script has been provided to automate this process in the file `src/utils/rename-for-clisp`. This script requires that you `cd` into the `src` directory and execute

```
% sh utils/rename-for-clisp
```

The script will rename all of the `"src/*/*.lisp"` files to `".lsp"`, so that they can be read by CLISP.

Obtaining CLX

If you are already using CLISP, you may need to additionally retrieve the CLX module. CLX for CLISP can be retrieved via `ftp` from `ma2s2.mathematik.uni-karlsruhe.de`, in the file `/pub/lisp/clisp/packages/pcl+clx.clisp.tar.z`.

Making a Garnet image

Once you have installed the CLX module, you can make a restartable image of Garnet with the following procedure (NOTE: this is different from other lisps). This is the standard procedure for compiling Garnet, followed by a dump of the lisp image.

```
clisp -M somewhere/clx.mem
> (load "garnet-prepare-compile.lsp")
> (load "garnet-loader.lsp")
> (load "garnet-compiler.lsp")
> (opal:make-image "garnet.mem" :quit t)
```

The saved image can then be restarted with the command:

```
clisp -M garnet.mem
```

1.8.9 AKCL

Some of the default parameters for the AKCL lisp image are insufficient for running Garnet. You may be able to change some of these parameters in the active lisp listener, but it is probably better to rebuild your AKCL image from scratch with the following parameter values:

```
MAXPAGES for AKCL should be at least 10240, and

(SYSTEM:ALLOCATE-RELOCATABLE-PAGES 800)
(SYSTEM:ALLOCATE-CONTIGUOUS-PAGES 45 T)
(SYSTEM:ALLOCATE 'CONS   3500 t)
(SYSTEM:ALLOCATE 'SYMBOL  450 t)
(SYSTEM:ALLOCATE 'VECTOR  150 t)
(SYSTEM:ALLOCATE 'SPICE   300 t)
(SYSTEM:ALLOCATE 'STRING  200 t)
```

Garnet runs about half as fast in AKCL as on other Common Lisps. Increasing the RAM in your machine may help. Users have reported that 16MB on a Linux-Box 486 yields unacceptable performance.

1.9 Mac-Specific Issues

1.9.1 Compensating for 31-Character Filenames:

There are several gadgets files that normally have names that are longer than 31 characters. Mac users may continue to specify the full-length names of these files by using `user::garnet-load`, described in section [garnet-load], page 17, which translates the regular names of the gadgets into their truncated 31-character names so they can be loaded. It is recommended that `garnet-load` be used whenever any Garnet file is loaded, so that typically long and cumbersome pathnames can be abbreviated by a short prefix.

1.9.2 Directories:

Unlike the Unix version, the Macintosh version stores all the binary and source files together in the various subdirectories under "src". This difference will not matter when a Garnet application is moved between Unix and Mac platforms as long as `garnet-load` is being used to load Garnet files. `Garnet-load` will always knows where to find the files.

1.9.3 Binding Keys:

We have assigned Lisp keywords for most of the keys on the Macintosh keyboard. Thus, to start an interactor when the "F1" key is hit, use `:F1` as the interactor's `:start-event`. If you want to know what a key generates, you can use the small utility Find-Key-Symbols which has been ported to the Mac. Execute (`garnet-load "inter-src:find-key-symbols"`) to bring up a window which can perceive keyboard events and prints out the resulting characters. The data you collect from this utility can be used in the `:start-where` slot of interactors to describe events that will start the interactor, and can be used to modify the characters generated by the keyboard key by editing the file `src:inter:mac-define-keys.lisp`.

1.9.4 Simulating Multiple Mouse Buttons With the Keyboard:

Most of the Garnet demos assume a three button mouse. To simulate this on the Macintosh, we use keyboard keys to replace a three-button mouse. By default, the keys are `F13`, `F14`, and `F15` for the left, middle, and right mouse buttons, respectively. The real mouse button is also mapped to `:leftdown`.

You can redefine the keys to be any three keys you want by setting `inter::*leftdown-key*`, `inter::*middledown-key*`, and `inter::*rightdown-key*` after loading Garnet or by editing the file `src:inter:mac-define-keys.lisp` directly. These variables should contain numerical key-codes corresponding to your desired keys. Some key-codes are shown on p. I-251 of *Inside Macintosh Volume I*, but you can also do (`garnet-load "inter:find-key-symbols"`) to run a utility program that tells you the key-code for any keyboard key. The utility will generate numbers that can be used directly in `src:inter:mac-define-keys.lisp`.

To facilitate Garnet's use with keyboards not equipped with function keys, Garnet supplies another utility program called `mouse-keys.lisp`, which is in the top-level Garnet directory in the Mac version (and is in `src/utils/mouse-keys.lisp` if you acquired the Unix-packaged version of Garnet). When loaded, this utility creates a window that allows you to toggle between using the function keys and arrow keys for the simulated mouse buttons. If you are frequently switching between using Garnet on an Extended Keyboard and a smaller laptop keyboard, you may use this utility a lot to tell Garnet which keys should be used for middle-down and right-down.

1.9.5 Modifier Keys:

Like MCL itself, Garnet treats the `Option` key as the "Meta" key. Also, you currently cannot get access to the `Command` (Open-Apple) key from Garnet.

1.9.6 Things to Keep in Mind When You Want Your Garnet Programs

to Run on Both X Windows and the Mac:

Use `user::garnet-load` instead of `load` when loading gadget files

Only supply `:face` values for fonts that run on both systems – this typically restricts you to using only the standard faces available in Garnet 2.2 and earlier versions.

The `#+apple` and `#-apple` reader macros can be used to indicate code that should be used only for Macs and only for non-Macs, respectively. When defining fonts, for example, you may want to provide the slot description `(:face #+apple :underline #-apple :bold)` to indicate that the font will be underlined on the Mac but bold in X.

The default place for windows is at (0,0) which unfortunately puts their title bars under the Macintosh menubar, so you cannot even move them using the mouse! (You can still `s-value` the position from the Lisp Listener.) Therefore, never create a window on the Mac with a `:top` less than 45 or it will not be movable.

Remember that many Mac screens are much smaller than most workstations' screens. Positioning windows perfectly may not be possible, and a better goal may be to simply keep the window title-bars within reach of the mouse so that the windows can be moved.

1.10 Compiling Garnet

After executing the compile procedure in section [retrieving], page 5, the result should be that all the files are compiled and loaded. (If there was a problem and you need to restart the compile procedure, please see below.) The compiler scripts do *not* check for compile errors. We have attempted to make Garnet compile without errors on all Common Lisps, but some lisps generate more warnings than others.

The compiler scripts compile the binaries into the same directories as the source files. For example, all the interactor binaries will be in `garnet/src/inter/` along with the source (`.lisp`) files. Therefore, after the compilation is completed, you will need to move the binaries into their own directory (e.g., `garnet/bin/inter`). To do this, use the c-shell script

```
csh garnet-after-compile
```

The `garnet-after-compile` file will normally be in the top level garnet directory. Note that this is typed to the shell, not to Lisp. Even if you normally run the "regular" (Bourne) shell (sh), the above command should work.

To prevent certain parts of Garnet from being compiled, set `user::compile-`*xxx*`-p` to `nil`, where *xxx* is replaced with the part you do not want to compile. See the comments at the top of the file `garnet-prepare-compile` for more information.

If you ever have to restart the compile process, you do not have to start from scratch. If you have not yet moved the binary files out of the `src/` directory (i.e., you have not yet run `garnet-after-compile`), then you can use the files that have been compiled already instead of compiling them again. Restart lisp, and for each Garnet module that has been compiled, set the variable `user::compile-`*xxx*`-p` to `nil` to indicate that it should not be compiled again. Then load the three script files again in the usual order. Note: if a module has been only partially compiled, then you must recompile the whole module.

1.11 Loading Garnet

To load Garnet, it is only necessary to load the file:

```
(load "garnet-loader")
```

(Of course, you may have to preface the file name with the directory path of where it is located. It is usually in the top level `garnet` directory.)

To prevent any of the Garnet sub-systems from being loaded, simply set the variable `user::load-`*xxx*`-p` to `nil`, where *xxx* is replaced by whatever part you do not want to load. Normally, some parts of the system are not loaded, such as the gadgets and demos. This is because you normally do not want to load or use all of these in a session. Files that use gadgets will load the appropriate ones automatically, and the `demos-controller` program loads the demos as requested.

It is possible to save an image of lisp after loading Garnet, so that when you restart lisp, Garnet will already be loaded and you will not have to load `garnet-loader.lisp`. For details about making lisp images, see the function `opal:make-image` in the Opal chapter.

1.12 Loader and Compiler Functions

1.12.1 Garnet-Load and Garnet-Compile

There are two functions that allow you to save a lot of typing when you load and compile files. When you supply `garnet-load` and `garnet-compile` with the Garnet subdirectory that you want to get a file from (e.g., "gadgets"), the functions will automatically append your Garnet pathname to the front of the specified file.

```
user::Garnet-Load "prefix:filename" [function], page 90


user::Garnet-Compile "prefix:filename" [function], page 90
```

These functions are defined in `garnet-loader.lisp` and are internal to the `user` package.

The *prefix* parameter corresponds to one of the Garnet subdirectories, and the *filename* is a file in that directory. A list of the most useful prefixes appear in section [garnet-load-alist], page 18, and a full list can be seen by evaluating the variable `user::Garnet-Load-Alist` in your lisp (after loading Garnet). Examples:

```
* (garnet-load "gadgets:v-scroll-loader")
Loading #p"/afs/cs/project/garnet/bin/gadgets/v-scroll-loader"
Loading V-Scroll-Bar...
...Done V-Scroll-Bar.

T
* (garnet-compile "opal:aggregates")
Compiling #p"/afs/cs/project/garnet/src/opal/aggregates.lisp"
for output to #p"/afs/cs/project/garnet/bin/opal/aggregates.fasl"
...
; Writing fasl file "/afs/cs/project/garnet/bin/opal/aggregates.fasl"
; Fasl write complete

NIL
*
```

There are two groups of prefixes that `garnet-load` accepts – the "bin" prefixes and the "src" prefixes. `Garnet-load` assumes that when you load files, you will want to load the compiled binaries. Therefore, when you use prefixes like "gadgets", `garnet-load` uses the Garnet-Gadgets-Pathname variable to find the file you want. If you really want to load a file from your source directory, you should use the subdirectory name with "-src" tacked on. For example,

```
* (garnet-load "gadgets-src:motif-parts")
Loading #p"/afs/cs/project/garnet/src/gadgets/motif-parts"
...


T
*
```

`Garnet-compile` does not accept "-src" prefixes, because it always assumes that you want to take a lisp file from your source directory, compile it, and output it to your bin directory. Note: do not specify ".lisp" or ".fasl" with your filename – `garnet-compile` will supply suffixes for you. `Garnet-compile` attempts to determine your correct binary extension (".fasl", ".lbin", etc.) from the kind of Lisp that you are using. If `garnet-compile` ever gets the extension wrong, you can change it by setting the variable *compiler-extension*, which is defined in the `user` package.

1.12.2 Adding Your Own Pathnames

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

```
(defparameter Garnet-Load-Alist
  `(("opal" . Garnet-Opal-Pathname)        ; For loading the  multifont-loader
    ("gg" . Garnet-Gadgets-PathName)       ; For loading gadgets
    ("gestures" . Garnet-Gestures-PathName) ; For loading  agate
    ("debug" . Garnet-Debug-PathName)      ; For loading the  Inspector
    ("demos" . Garnet-Demos-PathName)      ; For loading demos
    ("gilt" . Garnet-Gilt-PathName)        ; For loading high-level tools...
    ("c32" . Garnet-C32-PathName)
    ("lapidary" . Garnet-Lapidary-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], page 90

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

1.13 Overview of the Parts of Garnet

Garnet is composed of a number of sub-systems, some of which can be loaded and used separately from the others. Most of the subsystems also have their own separate packages. The following list shows the components of Garnet, the package used by that component, and the page number of the corresponding section in this chapter.

> `KR` - Package `kr`. system. ⟨undefined⟩ [kr], page ⟨undefined⟩.

`Gem` - Package `gem`. routines that allow the system to run on the Mac or on X11. We do not support user code directly calling Gem, so it is not described further in this chapter.

`Opal` - Package `opal`. The graphical object system. ⟨undefined⟩ [Opal], page ⟨undefined⟩.

`Interactors` - Package `inter`. mouse and keyboard input. ⟨undefined⟩ [inter], page ⟨undefined⟩.

`Gestures` - Package `inter`. Code to handle gesture recognition and training. Described in the interactors chapter, ⟨undefined⟩ [inter], page ⟨undefined⟩.

`Aggregadgets` - Package `opal`. Support for creating instances of collections of objects, and for rows or columns of objects. [aggregadgets], page 55.

`AggreGraphs` - Package `opal`. Support for creating graphs and trees of objects. Also described in the aggregadgets chapter, [aggregadgets], page 55.

`Gadgets` - Package `garnet-gadgets`, nicknamed `gg`. gadgets, including menus, buttons, scroll bars, circular gauges, graphics selection, etc. ⟨undefined⟩ [gadgets], page ⟨undefined⟩.

`Debugging tools` - Package `garnet-debug`, nicknamed `gd`. including the Inspector. ⟨undefined⟩ [Debug], page ⟨undefined⟩.

`Demonstration programs` - Each demonstration program is in its own package. ⟨undefined⟩ [demos], page ⟨undefined⟩.

`Gilt` - Package `gilt`. interface builder. ⟨undefined⟩ [gilt], page ⟨undefined⟩.

`C32` - Package `c32`. A spreadsheet interface for editing constraints. [c32], page 642.

`Lapidary` - Package `Lapidary`. sophisticated interactive design tool. ⟨undefined⟩ [lapidary], page ⟨undefined⟩.

`Contrib` - A set of file contributed by Garnet users. These have not been tested by the Garnet group, and are not supported. Each file should have a comment at the top describing how it works and who to contact for help and more information.

1.14 Overview of this Technical Report

In the programer's reference to all the parts of the Garnet toolkit listed above, this technical report also contains:

- A guided on-line tour of the Garnet system that will help you become familiar with a few of the features of the Garnet toolkit. ⟨undefined⟩ [tour], page ⟨undefined⟩.

- A tutorial to teach you the basic things you need to know to use Garnet. ⟨undefined⟩ [tutorial], page ⟨undefined⟩.

- The code for a simple graphical editor, as a sample of code written for Garnet. ⟨undefined⟩ [sampleprog], page ⟨undefined⟩.

- The Hints chapter starting on page ⟨undefined⟩ [hints], page ⟨undefined⟩, includes some suggestions that have been collected from the experience of Garnet users for making Garnet programs run faster. If you have ideas for things to add to this section, let us know.

1.15 What You Need To Know

Although this is a large technical report, you certainly do not need to know everything in it to use Garnet. Garnet is designed to support many different styles of interface. Therefore, there are many options and functions that you will probably not need to use.

In fact, to run the **Tour** (page ⟨undefined⟩ [tour], page ⟨undefined⟩), it is not necessary to read any of the programer's reference chapters. The tour is self-explanatory.

Next, you should probably read the **Tutorial** (page ⟨undefined⟩ [tutorial], page ⟨undefined⟩), since it tries to provide enough information about most of Garnet so that you don't need the other chapters right away.

To run the Gilt Interface Builder, you do not need to know about the rest of the system either. The Gilt chapter should be sufficient. When you are ready to set some properties of the gadgets, you will need to look up the particular gadget in the Gadgets chapter to see what the properties do.

Even when you are ready to start programming, you will still not need most of the information described here. To start, you should probably do the following:

Read this overview.

Run the tour, to get a feel for Garnet programming.

Read the tutorial.

You might try creating a few dialog boxes using Gilt. This will familiarize you with the Gadgets. See the Gilt chapter (⟨undefined⟩ [gilt], page ⟨undefined⟩).

After that, you can look at the sample program at the end of this technical report, to see what you need more information about.

You could now try to start writing your own programs, and just use the rest of the chapters as reference when you need information.

Next, look at the introduction and the following functions in the KR document: `gv`, `gvl`, `s-value`, `formula`, `o-formula`, and `create-instance`. The KR chapter documents the entire KR module, but Garnet does not use every feature that KR provides. Some concepts (like demons), will never be used by the typical Garnet user. Once you have gained some familiarity with the system, you may want to return to the KR chapter and read about object-oriented programming, type-checking, and constants.

Next, skim the first five chapters of the Opal chapter, and look at the various graphical objects, so you know what kinds are provided. The primary functions you will use from Opal are: `add-component`, `update`, and `destroy`, as well as the various types of graphical objects (`rectangle`, `line`, `circle`, etc.), drawing styles (`thin-line`, `dotted-line`, `light-gray-fill`, etc.) and fonts.

Next, in the Interactors chapter, you will need to skim the first four chapters to see how interactors work, and then see which interactors there are in the next chapter. You will probably not need to take advantage of the full power provided by the interactors system.

Aggregadgets and Aggrelists are very useful for handling collections of objects, so you should read their chapter. They support creating instances of groups of objects.

You should then look at the gadget chapter to see all the built-in components, so you do not have to re-invent what is already supplied.

User interface code is often difficult to debug, so we have provided a number of helpful tools. The Inspector is mentioned briefly in the Tutorial, and it is discussed thoroughly in the debugging chapter. You will probably find many debugging features very useful.

The demo programs can be a good source of ideas and coding style, so the document describing them might be useful.

If all you want Garnet for is to display menus and gauges that are supplied in the gadget set, you can probably just read the KR, Gadgets and Gilt chapters, and skip the rest.

1.16 Planned Future Extensions

We expect 3.0 to be the last release of the lisp version of Garnet. No enhancements of the lisp version are planned. However, if you need something and would like to sponsor its development, write to `garnet@cs.cmu.edu`.

The group is now working on a C++ system called Amulet, which will have many features similar to those found in Garnet. Watch for announcements about the Amulet project on `comp.windows.garnet` and `comp.lang.c++`. To sign up for the new Amulet mailing list, please send mail to `amulet-users-request@cs.cmu.edu`.

1.17 Garnet Articles

A number of articles about Garnet have been made available for FTP from the directory
`/usr/garnet/garnet/doc/papers/` on `a.gp.cs.cmu.edu`. There is a README file in that
directory, indicating which `.ps` files correspond to the Garnet bibliography citations.

2 On-line Tour Through Garnet

by: Brad A. Myers
10 May 2020

2.1 Abstract

This chapter provides an on-line tour through some of the features of the Garnet toolkit. It serves as an introduction to the toolkit and how to program with it. This document and tour do *not* assume that the reader has read the reference chapters. The tour only assumes that the reader is familiar with Common Lisp and has loaded the Garnet software.

2.2 Introduction

The Garnet User Interface Development Environment contains a comprehensive set of tools that make it significantly easier to design and implement highly-interactive, graphical, direct manipulation user interfaces. The lower layers of Garnet provide an object-oriented, constraint-based graphical system that allows properties of graphical objects to be specified in a simple, declarative manner and then maintained automatically by the system. The dynamic, interactive behavior of the objects can be specified separately by attaching high-level "interactor" objects to the graphics. The higher layers of Garnet include a number of tools to allow various parts of the interface to be specified without programming. The primary tools are Gilt, an interface builder, and Lapidary, a tool which helps you build gadgets and dialog boxes.

This document will help users get acquainted with the Garnet software by leading them through a number of exercises on line. This entire exercise should take about an hour. This tour assumes that the user is familiar with Lisp, although even non-Lispers might be able to type in the expressions verbatim and get the correct results.

Clearly, in this short tour, a great many parts of Garnet will not be covered, so the interested reader will need to refer to other parts of this chapter for details.

2.3 Getting Started

Garnet is a software package written in Common Lisp for X11 and the Mac, so the first thing to do is to run X11 and lisp on your Unix machine, or start MCL on your Mac. At Carnegie Mellon University, the Garnet software is available on the AFS file server. Elsewhere, you will have to copy the software onto your machine, and load it into your Lisp. See the discussion in the Overview document for an explanation of loading Garnet and special considerations for particular machines.

2.4 Typing

Many of the names in Garnet contain colons ":" and hyphens "-". These are part of the names and must be typed as shown. For example, `:filling-style` is a single name, and must be typed exactly.

In this document, the text that the user types (e.g, you) is shown underlined in the code examples. Most of the code looks like the following:

```
* '(+ 3 4)'
```

```
    7
```
The `"*"` is the prompt from Lisp to tell you it is ready to accept input (your Lisp may use a different prompt). Do not type the `"*"`. Type `"'(+ 3 4)'"`. The next line (here 7) shows what Lisp types as a response.

If you don't like to type, you might have the Appendix of this document displayed in an editor and just copy the commands into the Lisp window. In X, you can use the X cut buffer (copy the lines one-by-one into the X cut buffer, then paste them into the Lisp window); on the Mac, you can edit a file using the MCL editor and do the usual copy-and-paste operations. The Appendix contains a list of all the commands you need to type, to make it easier to copy them. The appendix code by itself is stored in the file `tourcommands.lisp` which is stored in the `demos` source directory (usually `garnet/src/demos/tourcommands.lisp`). *Note: do not just load* `tourcommands`*, since it will run all the demos and quickly quit; just copy-and-paste the commands one-by-one from the file.*

2.5 Garbage Collection

Most Common Lisp implementations use a garbage collection mechanism that occasionally interrupts all activity until it is completed. At various times during your tour, Lisp will stop and print something like the following message:

```
[GC threshold exceeded with 2,593,860 bytes in use. Commencing GC.]
```

You will then have to wait until it finishes and types something like:

```
[GC completed with 538,556 bytes retained and 2,055,356 bytes freed.]
[GC will next occur when at least 2,538,556 bytes are in use.]
```

This can happen at any time, and it causes the entire system to freeze (although the cursor will still track the mouse). Therefore, if nothing is responding, Lisp and Garnet may not have crashed. Wait for a minute and see if they come back.

2.6 Errors, etc.

It is quite common to end up in the Lisp debugger. This might be caused by a bug in Garnet or because you made a small typing error. To get out of the debugger, you will need to type the specific command for that version of Common Lisp (`q` on CMU Common Lisp, `:reset` in Allegro Common Lisp, and Command-period in MCL). For special instructions about the LispWorks debugger, see the section `"LispWorks"` in the Overview chapter.

Often, you can just try whatever you were doing again. However, some errors might cause Garnet or even Lisp to get messed up. In order of severity, you can try the following recovery strategies after leaving the debugger:

If Lisp does not seem to be responding, try typing `^C` (or whatever your break character is – Command-comma in MCL) *to the lisp window* (move the mouse cursor to the Lisp window first).

If you typed a line incorrectly, try typing it again the correct way.

If that does not work, try destroying the object you were creating and starting over from where you first started creating the object. To destroy an object that you created using (`create-instance 'xxx ...`), just type (`opal:destroy xxx`). Note that on the `create-instance` there is a quote mark, but not on the destroy call.

If you were in the first part of the tour (section ⟨undefined⟩ [LearnGarnet], page ⟨undefined⟩), then if that does not work, try destroying the window and starting over from the top: (opal:destroy MYWINDOW). If you were in the Othello part, try typing (stop-othello).

If that does not work, try quitting Lisp and restarting. For CMU Common Lisp, type (quit) to get out of Lisp; for Lucid, type (system:quit); for LispWorks, type (bye); for Allegro, type :ex; and for MCL type (quit). See section [startlisp], page 23, about how to start Lisp, and section [quitting], page 36, about quitting.

Finally, you can always logout and log back in.

In the Appendix of this document is a list of all the commands you are supposed to type in. This will be useful if you need to start over and don't want to have to read through everything to get to where you were. If you are starting at the Othello part (section [Othello], page 34), you do not have to execute any of the commands before that (except to load Garnet and the tour).

If Lisp seems to be stuck in an infinite loop, you can break out by typing the break character (often ^C — control-C) or the abort command in MCL (Command-comma). It will throw you into the debugger.

If you start something over, or retype a command, you may see messages like:

    Warning - create-schema is destroying the old #k<MGE::TRILL>.

This is a debugging statement is you can just ignore it.

There are a large number of debugging functions and techniques provided to help fix Garnet toolkit code, but these are not explained in this tour. See the debugging chapter.

```
****************************
LOGGING IN
****************************


Ask for user's name and login and e-mail address.
Automatically send bam the name in a mail message.

System will also Use-package kr, kr-debug


****************************
BASICS
****************************
```

2.7 Learning Garnet

2.8 LearnGarnet

2.9 A Note on Packages

The Garnet software is divided into a number of Lisp packages. A *package* may be thought of as a module containing procedures and variables that are all associated in some way. Usually, the programmer works in the user package, and is not aware of other packages in

Lisp. In Garnet, however, function calls are frequently accompanied by the name of the package in which the function was defined.

For example, one of the packages in Garnet is `opal`, which contains all the objects and procedures dealing with graphics. To reference the `rectangle` object, which is defined in `opal`, the user has to explicitly mention the package name, as in `opal:rectangle`.

On the other hand, the package name may be omitted if the user calls `use-package` on the package that is to be referenced. That is, if the command `(use-package :OPAL)` or `(use-package "OPAL")` is issued, then the `rectangle` object may be referenced without naming the `opal` package.

The recommended "Garnet Style" is to `use-package` only one Garnet package – `KR` – and explicitly reference objects in other packages. This convention is followed in the code examples below. The file `tour.lisp` that you loaded contains the line `(use-package :KR)`, which implements this convention. You will probably want to put this line at the top of all your future Garnet programs as well.

The packages in Garnet include:

`KR` - contains the procedures for creating and accessing objects. This contains the functions `create-instance`, `gv`, `gvl`, `s-value`, and `o-formula`.

`Opal` - contains the graphical objects and some functions for them.

`inter` - contains the interactor objects for handling the mouse.

`Garnet-Gadgets` - (nicknamed `gg`) contains a collection of predefined "gadgets" like menus and scroll bars.

`Garnet-Debug` - (nicknamed `gd`) contains a number of debugging functions. These are not discussed in this tour, however.

2.10 A Note on Refresh

In X11 and Mac QuickDraw, pictures drawn to windows need to be redrawn if the window is covered and then uncovered. Garnet handles this automatically for you by through a background process which detects this situation and redraws windows when necessary. In most lisps, Garnet launches this `main-event-loop` process itself. On the Mac, MCL runs a background process anyway, and Garnet supplies the necessary functions that handle graphics redrawing. This function is also responsible for processing mouse and keyboard input to Garnet windows.

The `main-event-loop` background process starts without any special attention in most lisps, including Allegro, Lucid, CMUCL, and MCL. If you are running LispWorks, then there is an initialization procedure for multiprocessing that you must perform before loading Garnet. Please consult the "LispWorks" section of the Overview chapter, the first section in this Garnet programer's reference chapters.

Unfortunately, if you are not running a recent version of Allegro, Lucid, CMUCL, MCL, or LispWorks, your Lisp may not support background processes. In this case, you must explicitly run the function yourself. If you notice that windows are not refreshing properly after becoming uncovered (or de-iconified), or that Garnet is completely ignoring all your keyboard and mouse input, then type the following into Lisp:

```
* '(inter:main-event-loop)'
```

This function loops forever, so you then have to hit the `F1` key while the cursor is in a Garnet window to exit `main-event-loop`. Alternatively, you can type `^C` or Command-period, or whatever your operating system break character is, in the Lisp window. Also, it is permissible (though unnecessary) to call `main-event-loop` within a version of Lisp which supports background processes – the function first checks if another `main-event-loop` is already running in the background, and if so, it returns immediately.

2.11 Loading Garnet and the Tour

The Overview document discusses how to load the Garnet software. In summary, you will load the file `Garnet-Loader` and this will load all the standard software. After that, you need to load the special file `tour.lisp`, which is in the `src/demos` sub-directory. For example, if the Garnet files are in the directory `/usr/xxx/garnet/`, then type the following:

    * `(load "/usr/xxx/garnet/garnet-loader")`

Which will print out lots of stuff. Then type:

    * `(garnet-load "demos:tour")`

Note that `garnet-load` provided by Garnet to simplify loading Garnet files. It takes one argument (in this case `"demos:tour"`), a two-part string consisting of the a Garnet subdirectory reference (eg, `"demos"`) and the name of a file (eg, `"tour"`), separated by a colon. The procedure searches the directory associated with that package for a Lisp file (either compiled or uncompiled) of that name.

2.12 Basic Objects

Now you are going to start creating some Garnet Toolkit objects.

Garnet is an object-oriented system, and you create objects using the function `create-instance`, which takes a quoted name for the new object, the type of object to create, and then some other optional parameters. First, you will create a window object.

Type the text shown underlined to Lisp. Be sure to start with an open parenthesis and be careful about where the quotes and colons go.

    * `(create-instance 'MYWINDOW inter:interactor-window)`
    #k<MYWINDOW>

You won't see anything yet, because Garnet waits for an `update` call before showing the results. Now type:

    * `(opal:update MYWINDOW)`

and the window should appear.

You can move the window around and change its size just like any other X or Mac window, in whatever way you have your X window manager set up to do this.

Now, you are going to create an "aggregate" object to hold all the other objects you create. An aggregate holds a collection of other objects; it does not have any graphic appearance itself.

    * `(create-instance 'MYAGG opal:aggregate)`
    #k<MYAGG>

This aggregate will be the special top level aggregate in the window, that will hold all the objects to be displayed in the window. You will use the function `s-value` which sets the

value of a "slot" (also called an instance variable) of the object. `S-value` takes the object, the slot and the new value. To read the value of the slot, use the function `gv`, which stants for "get value". All slot names in Garnet start with a colon.

```
* '(s-value MYWINDOW :aggregate MYAGG)'
#k<MYAGG>
* '(gv MYWINDOW :aggregate)'
#k<MYAGG>
```

Now, you will create a rectangle.

```
* '(create-instance 'MYRECT MOVING-RECTANGLE)'
#k<MYRECT>
```

> [Note: MOVING-RECTANGLE is defined in the `user` package by `tour.lisp` as a specialization of the general `opal:rectangle` prototype.]

Again, this is not visible yet. First, the rectangle must be added to the aggregate, and then the update procedure must be called. Adding the rectangle uses the function `add-component` which takes the aggregate and the new object to add to it.

```
* '(opal:add-component MYAGG MYRECT)'
#k<MYRECT>
* '(opal:update MYWINDOW)'
NIL
```

The rectangle should now appear in the window.

All objects have a number of properties, such as their position, size and color. So far, all the objects have used the default values for properties. You will now change the color of the rectangle by setting its `:filling-style` slot. Remember that slot names begin with a colon, and that nothing happens until you do the `update`.

```
* '(s-value MYRECT :filling-style opal:gray-fill)'
#k<GRAY-FILL>
* '(opal:update MYWINDOW)'
NIL
```

The other filling styles that are available include `opal:light-gray-fill`, `opal:dark-gray-fill`, `opal:black-fill`, `opal:white-fill`, and `opal:diamond-fill`. These are all "halftone" shades, which means that they are created by turning some pixels on and others off. If you have a color screen, you might also try `opal:red-fill`, `opal:blue-fill`, `opal:green-fill`, `opal:yellow-fill`, `opal:purple-fill`, etc.

Now, you will create a text object. Here, for the first time, you will supply some extra values for slots when the object is created, rather than just using `s-value` afterward. Objects have a large number of slots and the ones that are not specified use the default values. To specify a slot at creation time, each name and value is enclosed in a separate parenthesis pair. Note that you can type carriage return where-ever you want. After the text is created, add it to the aggregate and update the window.

```
* '(create-instance 'MYTEXT opal:text (:left 200)(:top 80)
      (:string "Hello World"))'
#k<MYTEXT>
```

```
* `(opal:add-component MYAGG MYTEXT)'
#k<MYTEXT>
* `(opal:update MYWINDOW)'
NIL
```

The :top of the string is just its Y value, and the :left is just the X value, and they are, of course, independent.

You can change the position (:left and :top) and string of MYTEXT using s-value if you want, like the following:

```
* `(s-value MYTEXT :top 40)'
40
* `(opal:update MYWINDOW)'
NIL
```

2.13 Formulas

An important property of Garnet is that properties of objects can be connected using *constraints*. A constraint is a relationship that is defined once and maintained automatically by the system. You will constrain the string to stay at the top of the rectangle. Then, when the rectangle is moved, the string will move automatically.

Constraints in Garnet are expressed as *formulas* which are put into the slots of objects. Any slot can either have a value in it (like a number or a string) or a formula which computes the value. The formula can be an arbitrary Lisp expression which must be passed to the Garnet function o-formula. References to other objects in formulas must take a special form. To get the slot slot-name from the object other-object, use the form (gv other-object slot-name), where "gv" stands for "get value." The gv function can be used either inside or outside of formulas. When used from inside a formula, gv will establish a dependency on the referenced slot, causing the formula to reevaluate if the value in the referenced slot ever changes.

Now, set the top of the string to be a formula that depends on the top of the rectangle.

Note that the particular number returned by the s-value call will not be the same as shown below.

```
* `(s-value MYTEXT :top (o-formula (gv MYRECT :top)))'
#k<F3875>   the number will be different
* `(opal:update MYWINDOW)'
NIL
```

After the update, the string should move to be at the top of the rectangle. If you change the top of the rectangle, *both* the rectangle and the string will now move:

```
* `(s-value MYRECT :top 50)'
50
* `(opal:update MYWINDOW)'
NIL
```

If you want to experiment with writing your own formulas, the Lisp arithmetic operators include +, -, floor (for divide), and * (for multiply) and they must be in fully parenthesized expressions, as in (o-formula (+ (gv MYRECT :top) 7)). To get the width and height of an object from inside a formula, use (gv *obj* :width) and (gv *obj* :height). You could

try, for example, to get the text to stay centered in X (`:left`) and Y (`:top`) inside the rectangle.

2.14 Interaction

Now, you will get the objects to respond to input. To do this, you attach an *interactor* to the object. Interactors handle the mouse and keyboard and update graphical objects.

First, you will have the rectangle move with the mouse. To do this, you create a `move-grow-interactor` and tell it to operate on MYRECT. The interactor will start whenever the mouse is pressed `:in` MYRECT, and the interactor works in MYWINDOW. The interactor will continue to run no matter where the mouse is moved while the button is held down.

It is not necessary to call `update` to get interactors to start working; they start as soon as they are created. However, if you are not using a recent version of CMU, Allegro, LispWorks, Lucid, or MCL Common Lisp, interactors only run while the `main-event-loop` procedure is operating. `Main-Event-Loop` does not exit, so you will have to hit the `F1` key while the cursor is in the Garnet window, or type `^C` (or whatever your operating system break character is) while the cursor is in the Lisp window, to be able to type further Lisp expressions.

```
* `(create-instance 'MYMOVER inter:move-grow-interactor
(:start-where (list :in MYRECT))
(:window MYWINDOW))'
#k<MYMOVER>
```

If your Lisp requires it, then type:

```
* `(inter:main-event-loop)'
```

Now you can press with the left button over the rectangle, and while the button is held down, move the rectangle around. (The first time you press on the rectangle, it may take a while, as Lisp swaps in the appropriate code.) Notice that the text string moves up and down also. The text string does not move left and right, however, since there is no constraint on the `:left` of the string, only on the `:top` (unless you have written some extra formulas other than the one described above).

A different interactor allows you to type into text strings. This is called a `text-interactor`. The code below will cause the text interactor to start when you press the right mouse button, and stop when you press the right mouse button again. This will allow you to type carriage returns into the string and to move the cursor point by hitting the left button inside the string. (Before typing these commands, hit the F1 key to exit `main-event-loop` if necessary).

```
* `(create-instance 'MYTYPER inter:text-interactor
(:start-where (list :in MYTEXT))
(:window MYWINDOW)
(:start-event :rightdown)
(:stop-event :rightdown))'
#k<MYTYPER>
```

If your Lisp requires it, then type:

```
* `(inter:main-event-loop)'
```

   Now, if you press with the right mouse button on the string, you can change the string
by typing. The available editing commands include:

`^h, delete, backspace`
               delete previous character.

`^w, ^backspace, ^delete`
               delete previous word.

`^d`   delete next character.

`^u`   delete entire string.

`^b, left-arrow`
               go back one character.

`^f, right-arrow`
               go forward one character.

`^n, down-arrow`
               go vertically down one line.

`^p, up-arrow`
               go vertically up one line.

`^<, ^comma, home`
               go to the beginning of the string.

`^>, ^period, end`
               go to the end of the string.

`^a`   go to beginning of the current line.

`^e`   go to end of the current line.

`^y, insert`
               insert the contents of the X or Mac cut buffer into the string at the current
               point.

`^c`   copy the current string to the X or Mac cut buffer.

`enter, return, ^j, ^J`
               Go to new line.

`left button down inside the string`
               move the cursor to the specified point.

`^G`   Abort the edits and return the string to the way it was before editing started.

   All other characters go into the string (except other control characters which beep). You
can also move the cursor with the mouse by clicking in the string.

   (In X, to type to a window, the mouse cursor must be inside the window, so to type
to the "Hello World" string, the mouse cursor must be inside the Garnet window, and to
type to Lisp, the cursor should be inside the Lisp window. On the Mac, you have to click
the mouse on the title-bar of the window you want to type into, so you will have to click
alternately on the Garnet window and the lisp listener.)

If you make the text string be multiple lines, by typing a carriage return into it, then you can control whether the lines are centered, left or right justified. This is controlled by the `:justification` slot of MYTEXT, which can be `:left`, `:center`, or `:right`. (Before typing these commands, hit the F1 key to exit `main-event-loop` if necessary).

```
* '(s-value MYTEXT :justification :right)'
:RIGHT
* '(opal:update MYWINDOW)'
NIL
* '(s-value MYTEXT :justification :center)'
:CENTER
* '(opal:update MYWINDOW)'
NIL
```

Of course, you can type to the string while it is centered or right-justified, and you can move around the rectangle with the mouse and the string will still follow.

2.15 Higher-level Objects

Now, you are going to create instances of pre-created objects from the "Garnet Gadget Set." The Gadget Set contains a large collection of menus, buttons, scroll bars, sliders, and other useful *interaction techniques* (also called "widgets"). You will be using a set of "radio buttons" and a slider.

First, however, you should make the window bigger (in whatever way you do this in your window manager).

2.15.1 Buttons

First, you will create a set of 3 "radio" buttons that will determine whether the text is centered, left, or right justified. The parameter that tells the buttons what the labels should be is called `:Items`. This slot is passed a quoted list. The radio buttons will appear at the right of the string.

```
* '(create-instance 'MYBUTTONS gg:radio-button-panel
        (:items '(:center :left :right))
(:left 350)(:top 20))'
#k<MYBUTTONS>
* '(opal:add-component MYAGG MYBUTTONS)'
#k<MYBUTTONS>
* '(opal:update MYWINDOW)'
NIL
```

If your Lisp requires it, then type:

```
* '(inter:main-event-loop)'
```

Now, you can click on the radio buttons with the left mouse button, and the dot will move to whichever one you click on.

Next, you will use a constraint to tie the value of the `:justification` field of the text object to the value of the radio buttons. The current value of the radio buttons is conveniently kept in the `:value` field. (Before typing these commands, hit the F1 key to exit `main-event-loop` if necessary).

```
* '(s-value MYTEXT :justification (o-formula (gv MYBUTTONS :value)))'
```

```
#k<F2312>    the number will be different
* '(opal:update MYWINDOW)'
NIL
```

If your Lisp requires it, then type:

```
* '(inter:main-event-loop)'
```

Now, whenever you press on one of the buttons, the text will re-adjust itself.

All of the built-in toolkit items have a large number of parameters to allow users to customize their look and feel. For example, you can change the radio buttons to be horizontal instead of vertical: (From now on, you will have to remember to hit the F1 key to exit `main-event-loop` if necessary before typing commands without these reminders).

```
* '(s-value MYBUTTONS :direction :horizontal)'
:HORIZONTAL
* '(opal:update MYWINDOW)'
NIL
```

Now, change it back to be vertical:

```
* '(s-value MYBUTTONS :direction :vertical)'
:VERTICAL
* '(opal:update MYWINDOW)'
NIL
```

2.15.2 Slider

Next, you will do a similar thing to get the gray shade of the rectangle to be attached to an on-screen slider. First, create a Garnet vertical slider object:

```
* '(create-instance 'MYSLIDER gg:v-slider
(:left 10)(:top 20))'
#k<MYSLIDER>
* '(opal:add-component MYAGG MYSLIDER)'
#k<MYSLIDER>
* '(opal:update MYWINDOW)'
NIL
```

If your Lisp requires it, then type:

```
* '(inter:main-event-loop)'
```

This slider can be operated in a number of ways, all using the left mouse button. Press on the top arrow to move up one unit, and the down arrow to move down one. The double arrow buttons move up and down by five (the increment amount can be changed by using `s-value` on the `:scr-incr` and `:page-incr` slots of MYSLIDER). You can also press on the black indicator arrow and drag it to a new position. Finally. you can press in the top number area, then type a new number value, and then hit carriage return.

Of course the value returned by the slider does not affect anything yet. To change the color of the rectangle, you will use the Garnet function `Halftone`, which takes a number from 0 to 100 and returns a `:filling-style` that is that percentage black. Connect the filling style of the rectangle to the value returned by the slider:

```
* '(s-value MYRECT :filling-style
(o-formula (opal:halftone (gv MYSLIDER :value))))'
```

```
#k<F5940>    the number will be different
* '(opal:update MYWINDOW)'
NIL
```

If your Lisp requires it, then type:

```
* '(inter:main-event-loop)'
```

Now when you change the value of the slider, the color of the rectangle will change. Note that halftone only can generate 17 different gray colors, so a range of numbers for the slider will generate the same color.

2.16 Playing Othello

Now you can play the Othello game we created using the Garnet Toolkit.

To bring up the game, type:

```
* '(start-othello)'
T
```

The game board will appear on the screen. There are various things you can control in the game. You can put new pieces down on the board by just pressing with the left mouse button. In Othello, you can put a piece in a position where you are next to the other player's marker, and one of your markers is in a straight line from where you are going to play. If you try to place your marker in an illegal place, the game will beep. This game does not try to play against you; you must handle both players (or get someone else to play with you). If a player does not want to move (or has no legal moves), then the "Pass" menu item can be selected. This implementation does not detect when the game is over. The current score (which is the number of squares that the player controls) is shown in the top left box.

To start over, press on the menu button marked "Start." This will start a new game with a board that has the number of squares shown by the scroll bar. The default is 8 by 8. To change the scroll bar value, press on the arrows. (Changing the scroll bar does not change the current board; it takes affect the next time you hit "Start" from the menu.)

"Stop" just erases the board, and "Quit" exits the game. (You don't have to quit before going on to the next section.)

2.17 Modifying Othello

We created an editor that allows you to change what the Othello playing pieces look like. This is editor is just a small toy program that was created quickly by David Kosbie in the Garnet group especially for this tour.

If you quit out of the Othello game, bring it back up using (start-othello).

Othello has a tall window on the left side of the screen containing the current 2 Othello playing pieces at the top: a white and a black circle. Underneath is a command button ("Delete") and 3 menus. The top left menu is for different types of objects: rectangles, rounded rectangles, circles and ovals. The bottom left menu is for line styles (the way the outlines of objects are drawn): no outline, dotted outline, thin, thicker or very thick outline. The menu on the right is for how the inside of objects looks: no filling inside, white, grey, black or various patterns.

Press with the left mouse button over any of the menus to change the current mode.

To draw a new object in either playing piece, just use the *right* mouse button to drag out the dimensions for the new object. Press down the right button inside whichever piece you want to modify where you want one corner of the new object to be, move the cursor while holding down, and release at the other corner. The type, line styles, and inside of the new object come from the current values of the menus.

Objects can be selected by pressing over them with the *left* mouse button. (Some objects require that you press on the edge (border) of the object, and others allow you to press anywhere inside.) When an object is selected, 12 small boxes are shown on the borders of the object. (The small boxes are on the bounding rectangle of the object, which may be a little confusing for circles.) The black boxes can be used to change the object's size, and the white boxes are used to move the object. Just press with the left button over one of the boxes, and then adjust the size or position while holding down. The editor will not let you move or grow an object so that it goes outside the game piece area.

The selected object can also be deleted or changed. Delete it by just hitting the Delete button in the menu when the object is selected. If you press on a new line style or filling style while an object is selected, the object's outline and color will change. (You can't change an object's type.) Note that as you select objects, the menus change to show the object's current styles.

Every time you edit one of the playing pieces, the Othello game display also changes to reflect the edits. This is handled automatically by Garnet using inheritance.

2.18 Using GarnetDraw

There a useful utility called `GarnetDraw` which is a relatively simple drawing program written using Garnet. Using this application, you can draw pictures with many of the basic Garnet objects (like circles, rectangles, and lines), and then save the picture to a file. Since the file format for storing the created objects is simply a Lisp file which creates aggregadgets, you might be able to use GarnetDraw to prototype application objects (but Lapidary is probably better for this).

GarnetDraw uses many sophisticated features of Garnet including gridding, PostScript printing, selection of all objects in a region, moving and growing of multiple objects, menubars, and the `save-gadget` and `load-gadget` dialog boxes.

To load and start GarnetDraw, type:

```
* '(garnet-load "demos:garnetdraw")'

* '(garnetdraw:do-go)'
```

GarnetDraw works like most Garnet programs: select in the palette with any button, draw in the main window with the right button, and select objects with the left button. Select multiple objects with shift-left or the middle mouse button. Change the size of objects by pressing on black handles and move them by pressing on white handles. The line style and color and filling color can be changed for the selected object and for further drawing by clicking on the icons at the bottom of the palette.

You might want to save a picture to a file, and then bring the file up in your editor to see the kind of code that GarnetDraw generates. There should be a top-level aggregadget that has your drawn objects as components.

To quit GarnetDraw, either select "Quit" from the menubar, or type:

    * `(garnetdraw:do-stop)`

2.19 Cleanup

If you are not in a Lisp which supports background processes, and you are running something in Garnet, then you need to type F1 in a Garnet window or ^C in your Lisp window to get back to the Lisp read-eval-print loop.

To get rid everything at once (MYWINDOW, the Othello game, and the editor for the game pieces), just type:

    * `(stop-tour)`
    `"Thank you for your interest in the Garnet Project"`

Otherwise, to just get rid of Othello and the editor, you can hit on the "Quit" menu button or type `(stop-othello)` to Lisp. To just get rid of MYWINDOW, type `(opal:destroy MYWINDOW)`.

The command that exits Lisp is different for different implementations. For CMU Common Lisp, type: '`* (quit)`'

for Lucid Common Lisp, type: '`* (system:quit)`'

for LispWorks, type: '`* (bye)`'

for Allegro Common Lisp, type: '`* :ex`'

and for MCL, type: '`* (quit)`'

This returns you to the shell (or to the finder on the Mac), and you can log out. It is not necessary to run `(stop-othello)` or `(stop-tour)` before quitting Lisp.

If the quit command doesn't work for any reason, you can probably quit by typing ^Z to pause to the shell and then kill the lisp process (or just log out).

2.20

We hope you have enjoyed your tour through Garnet. There are, of course, many features and capabilities that have not been demonstrated. These are described fully in the various chapters and papers about the Garnet project and its parts. The next step might be to run the Gilt interface builder, since it does not require that you learn much about how Garnet works. See the Gilt chapter.

⟨undefined⟩ [Appendix: List of commands], page ⟨undefined⟩, This appendix lists all the commands that the tour has you type. This is useful as a quick reference if you need to restart due to an error. These commands are stored in the file `tourcommands.lisp` which is stored in the `demos` source directory (usually `garnet/src/demos/tourcommands.lisp`). If you have this document in a window on the screen, you can copy-and-paste to move text from below into your Lisp window. *Note: do not just load* `tourcommands`*, since it will run all the demos and quickly quit; just copy the commands one-by-one from the file.*

This listing does not show the prompts or Lisp's responses to these commands.

[First, load the Garnet software. You will have to replace `xxx` with your directory path to Garnet:]

    `(load "/xxx/garnet/garnet-loader")`

```
(garnet-load "demos:tour")
```

[Start here after Garnet and the tour software is loaded:]

```
(create-instance 'MYWINDOW inter:interactor-window)
(opal:update MYWINDOW)

(create-instance 'MYAGG opal:aggregate)
(s-value MYWINDOW :aggregate MYAGG)
(gv MYWINDOW :aggregate)
(create-instance 'MYRECT MOVING-RECTANGLE) ; In the USER package
(opal:add-component MYAGG MYRECT)
(opal:update MYWINDOW)

(s-value MYRECT :filling-style opal:gray-fill)
(opal:update MYWINDOW)

(create-instance 'MYTEXT opal:text (:left 200)(:top 80)
(:string "Hello World"))
(opal:add-component MYAGG MYTEXT)
(opal:update MYWINDOW)

(s-value MYTEXT :top 40)
(opal:update MYWINDOW)

(s-value MYTEXT :top (o-formula (gv MYRECT :top)))
(opal:update MYWINDOW)

(s-value MYRECT :top 50)
(opal:update MYWINDOW)

(create-instance 'MYMOVER inter:move-grow-interactor
  (:start-where (list :in MYRECT))
  (:window MYWINDOW))
#-(or cmu allegro lucid lispworks apple)  ;only do this if your Lisp is NOT a recent█
(inter:main-event-loop)                   ;version of CMU, Allegro, Lu-
cid, or LispWorks
                                          ;type F1 or ^C to exit when finished.█

(create-instance 'MYTYPER inter:text-interactor
  (:start-where (list :in MYTEXT))
  (:window MYWINDOW)
  (:start-event :rightdown)
  (:stop-event :rightdown))
#-(or cmu allegro lucid lispworks apple)  ;only do this if your Lisp is NOT a recent█
(inter:main-event-loop)                   ;version of CMU, Allegro, Lu-
cid, or LispWorks
                                          ;type F1 or ^C to exit when finished.█
```

```
(s-value MYTEXT :justification :right)
(opal:update MYWINDOW)

(s-value MYTEXT :justification :center)
(opal:update MYWINDOW)

(create-instance 'MYBUTTONS gg:radio-button-panel
  (:items '(:center :left :right))
  (:left 350)(:top 20))
(opal:add-component MYAGG MYBUTTONS)
(opal:update MYWINDOW)
#-(or cmu allegro lucid lispworks apple)  ;only do this if your Lisp is NOT a recent
(inter:main-event-loop)                   ;version of CMU, Allegro, Lu-
cid, or LispWorks

                                          ;type F1 or ^C to exit when finished.


(s-value MYTEXT :justification (o-formula (gv MYBUTTONS :value)))
(opal:update MYWINDOW)
#-(or cmu allegro lucid lispworks apple)  ;only do this if your Lisp is NOT a recent
(inter:main-event-loop)                   ;version of CMU, Allegro, Lu-
cid, or LispWorks

                                          ;type F1 or ^C to exit when finished.


(s-value MYBUTTONS :direction :horizontal)
(opal:update MYWINDOW)

(s-value MYBUTTONS :direction :vertical)
(opal:update MYWINDOW)

(create-instance 'MYSLIDER gg:v-slider
  (:left 10)(:top 20))
(opal:add-component MYAGG MYSLIDER)
(opal:update MYWINDOW)
#-(or cmu allegro lucid lispworks apple)  ;only do this if your Lisp is NOT a recent
(inter:main-event-loop)                   ;version of CMU, Allegro, Lu-
cid, or LispWorks

                                          ;type F1 or ^C to exit when finished.


(s-value MYRECT :filling-style (o-formula
(opal:halftone (gv MYSLIDER :value))))
(opal:update MYWINDOW)
#-(or cmu allegro lucid lispworks apple)  ;only do this if your Lisp is NOT a recent
(inter:main-event-loop)                   ;version of CMU, Allegro, Lu-
cid, or LispWorks

                                          ;type F1 or ^C to exit when finished.
```

[To just get Othello to run, execute the following line. You do not have to enter any of the previous code to run Othello and the editior (except for the software loading, of course).]

```
(start-othello)
```

[To just load and run GarnetDraw, execute the following lines.]

```
(garnet-load "demos:garnetdraw")
(garnetdraw:do-go)
```

[Cleaning up and quitting:]

```
;;; * To quit all editors and demos and destroy all windows

(stop-tour)
(garnetdraw:do-stop)   ; if running

;;; * To leave lisp

#+cmu    (quit)            ; in CMU Common Lisp
#+lucid (system:quit)  ; in Lucid Common Lisp
#+allegro :ex             ; in Allegro Common Lisp
#+lispworks (bye)       ; in LispWorks Common Lisp
#+apple (quit)            ; in MCL
```

```
**********************************************************
Later, have another section on more details of objects, etc.
**********************************************************


**********************************************************
CREATING A GRAPHICS EDITOR (Optional)
**********************************************************


            left button moves objects
            right button edits text
            shift-left creates a new object
            shift-right deletes object under mouse
            scroll bar and menu as before for specifying props of next
                    new object

create another window
create an aggregadget of a rectangle, line, and editable-string as a prototype█
** or use predefined one; would prefer to do this part using Lapidary****
```

create an aggregate to hold the objects

create a new version of the scroll bar and x-boxes to in another new window

define the create and delete functions

create all the interactors

@ref{References}

3 Garnet Tutorial

Andrew Mickish [No value for "date"]

3.1 Abstract

This tutorial has been designed to introduce the reader to the basic concepts of Garnet. The reader should have already taken the Garnet Tour before starting the tutorial.

3.2 Take the Tour

Before beginning this tutorial, you should have already completed the Garnet Tour, available in a separate document. The Tour was a series of exercises intended to acquaint you with a few of the features of Garnet, while giving you a feel for the interactive programming aspects of Garnet. This Tutorial investigates all of those features in greater depth, while explaining the fundamental principles behind objects, inheritance, constraints, interactors, and the actual writing of code.

In the Garnet Tour, you were given some background information about how to load Garnet, how to access the different Garnet packages, garbage collection, the main-event-loop for interactors, etc. It may be helpful to review this information from the first few sections of the Tour before starting the Tutorial.

3.3 Load Garnet

Using the instructions from the Tour, load Garnet into your lisp process. Also, type in the following line so that references to the KR package can be eliminated (we will explicitly reference the other Garnet packages):

```
(use-package :KR)
```

3.4 The Prototype-Instance System

The basic idea behind programming in Garnet is creating objects and displaying them in windows on the screen. An object is any of the fundamental data types in Garnet. Lines, circles, aggregates and windows are all objects. These are all *prototype* objects — you make copies of these objects and customize the copies to have your desired size and position, as well as other graphic qualities such as filling styles and line styles. When you make a customized copy of an object, we say you create an *instance* of the object. Thus, Garnet is a prototype-instance system.

3.5 Inheritance

When instances are created, an inheritance link is established between the prototype and the instance. *Inheritance* is the property that allows instances to get values from their prototypes without specifying those values in the instances themselves. For example, if we set the filling style of a rectangle to be gray, and then we create an instance of that rectangle, then the instance will also have a gray filling style. Naturally, this leads to an inheritance hierarchy among the objects in the Garnet system. In fact, there is one root object in Garnet – the `view-object` – that all other objects are instances of (except for

interactors, which have their own root). Figure [opal-inheritance], page 43, shows some of the objects in Garnet and how they fit into the inheritance hierarchy. (The average user will never be concerned with the actual `view-object` or `graphical-object` prototypes.)
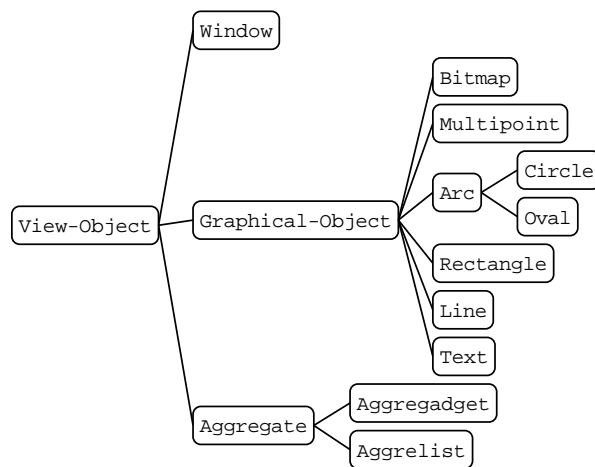
Figure 3.1: The inheritance hierarchy among some of the Garnet prototype objects. All of the standard shapes in garnet are instances of the `graphical-object` prototype. As an example of inheritance, the `circle` and `oval` objects are both special types of arcs, and they inherit most of their properties from the `arc` prototype object. The Gadgets (the Garnet widgets) are not pictured in this hierarchy, but most of them are instances of the `aggregadget` object.

To see an example of inheritance, let's create an instance of a window and look at some of its inherited values. After you have loaded Garnet, type in the following code.

```
(create-instance 'MY-WIN inter:interactor-window
    (:left 800) (:top 100))
(opal:update MY-WIN)  ; To make the window appear
```

The window should appear in the upper-right corner of your screen. In the definition of the MY-WIN schema, we gave a value of 800 to the `:left` slot and a value of 100 to the `:top` slot. Let's check these slots in MY-WIN to see if they are correct. Type in the following lines.

```
(gv MY-WIN :left)     ; Should be 800
(gv MY-WIN :top)      ; Should be 100
```

The function `gv` gets the values of slots from an object. If you got the right values for the `:left` and `:top` slots of MY-WIN, then you see that the values you supplied during the `create-instance` call are still being used by MY-WIN. These are values that are held in the instance itself. On the other hand, try typing in the following lines.

```
(gv MY-WIN :width)
(gv MY-WIN :height)
```

We did not supply values to the `:width` and `:height` slots of MY-WIN when it was created. Therefore, these values are *inherited* from the prototype. That is, they were defined in the `interactor-window` object when it was created, and now MY-WIN inherits those values as its own. We could, however, override these inherited values. Let's change the width and height of MY-WIN using `s-value`, the function that sets the values of slots.

```
(s-value MY-WIN :width 100)
(s-value MY-WIN :height 400)
(opal:update MY-WIN)  ; To cause the changes to appear
```

The dimensions of the window should change, reflecting the new values we have supplied to its `:width` and `:height` slots. If we were to now use `gv` to look at the width and height of MY-WIN, we would get back the new values, since the old ones are no longer inherited.

The inheritance hierarchy which was partially pictured in Figure [opal-inheritance], page 43, is traced from the leaves toward the root (from right to left) during a search for a value. Whenever we use `gv` to get the value of a slot, the object first checks to see if it has a local value for that slot. If there is no value for the slot in the object, then the object looks to its prototype to see if it has a value for the slot. This search continues until either a value for the slot is found or the root object is reached. When no inherited or local value for the slot is found, the value `nil` is returned (which, by the way, looks just the same as a user-defined local value of `nil` for a slot).

Since we are now finished with the example of MY-WIN, let's destroy it so it does not interfere with future examples in this tutorial. Type in the following line.

```
(opal:destroy MY-WIN)
```

3.6 Prototypes

When programming in Garnet, inheritance among objects can eliminate a lot of duplicated code. If we want to create several objects that look similar, we could create each of them

from scratch and copy all the values that we need into each object. However, inheritance allows us to define these objects more efficiently, by creating several similar objects as instances of a single prototype.
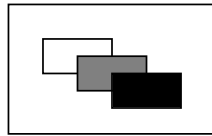
Figure 3.2: Three instances created from one prototype rectangle.

To start, look at the picture in Figure [proto-rects], page 45. We are going to define three rectangles with three different filling styles and put them in a window. First, let's create a window with a top-level aggregate. (For now, just think of an aggregate as an object which contains several other objects.) As we add our objects to this aggregate, they will be displayed in the window.

```
(create-instance 'WIN inter:interactor-window
    (:left 750)(:top 80)(:width 200)(:height 400))
(create-instance 'TOP-AGG opal:aggregate)
(s-value WIN :aggregate TOP-AGG)
(opal:update win)
```

Now let's consider the design for the rectangles. The first thing to notice is that all of the rectangles have the same width and height. Therefore, we will create a prototype rectangle which has a width of 40 and a height of 20, and then we will create three instances of that rectangle. To create the prototype rectangle, type the following.

```
(create-instance 'proto-rect opal:rectangle
    (:width 40) (:height 20))
```

This rectangle will not appear anywhere, because it will not be added to the window. But now we need to create the three actual rectangles that will be displayed. Since the prototype has the correct values for the width and height, we only need to specify the left, top, and filling styles of our instances.

```
(create-instance 'r1 proto-rect
    (:left 20) (:top 20)
    (:filling-style opal:white-fill))

(create-instance 'r2 proto-rect
    (:left 40) (:top 30)
    (:filling-style opal:gray-fill))

(create-instance 'r3 proto-rect
    (:left 60) (:top 40)
    (:filling-style opal:black-fill))

(opal:add-components top-agg r1 r2 r3)  ; give the aggregate three components
(opal:update win)
```

After you update the window, you can see that the instances R1, R2, and R3 have inherited their :width and :height from PROTO-RECT. You may wish to use gv to verify this. With these three rectangles still in the window, we are ready to look at another important use of inheritance. Try changing the width and height of the prototype as follows.

```
(s-value proto-rect :width 30)
(s-value proto-rect :height 40)
(opal:update win)
```

The result should look like the rectangles in Figure [changed-proto], page 47. Just by changing the values in the prototype rectangle, we were able to change the appearance of all its instances. This is because the three instances inherit their width and height from the prototype, even when the prototype changes.
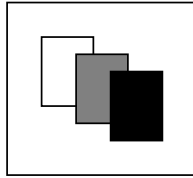
Figure 3.3: The instances change whenever the prototype object changes.

For our last look at inheritance in this section, let's try to override the inherited slots in one of the instances. Suppose we now want the rectangles to look like Figure [override], page 48. In this case, we only want to change the dimensions of one of the instances. The following lines should change the appearance of the black rectangle accordingly.

```
(s-value R3 :width 100)
(opal:update WIN)
```

The rectangle R3 now has its own value for its :width slot, and no longer inherits it from PROTO-RECT. If you change the :width of the prototype again, the width of R3 will not be affected. However, the width of R1 and R2 will change with the prototype, because they still inherit the values for their :width slots. This shows how inheritance can be used flexibly to make specific exceptions to the prototype object.
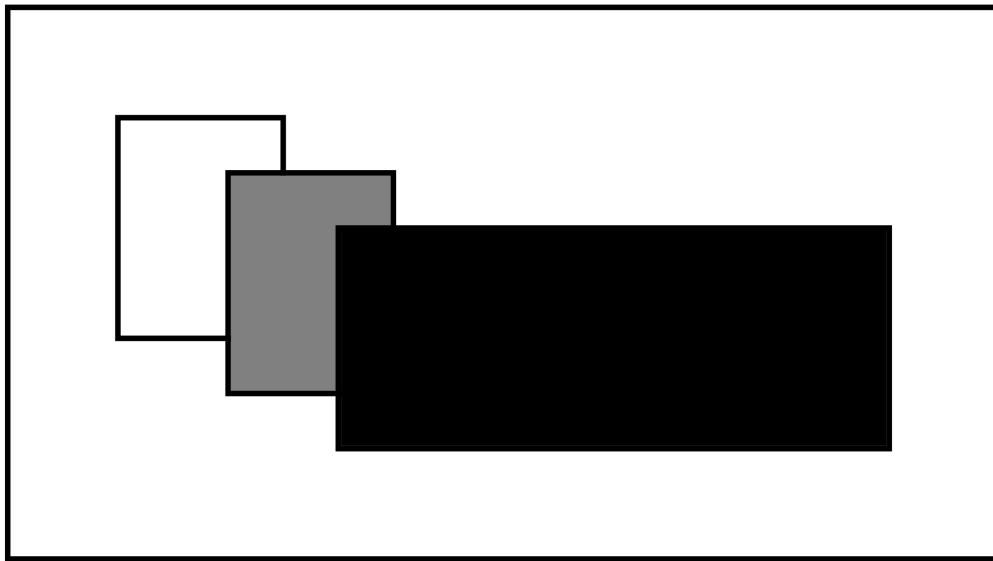
Figure 3.4: The width of R3 is overridden, so it is no longer inherited from the prototype.

3.7 Default Values

Because of inheritance, all instances of Garnet prototype objects have reasonable default values when they are created. As we saw in section [inheritance], page 41, the interactor-window object has its own :width. So, if an instance of it is created without an explicitly defined width, the width of the instance will be inherited from the prototype, and it can be considered a default value.

3.8 The Inspector

An important tool for examining properties of objects is the Inspector. This tool is loaded with Garnet by default, and resides in the package garnet-debug. The Inspector is described in detail in the Debugging chapter that starts on page ⟨undefined⟩ [debug], page ⟨undefined⟩, of this reference chapter.

To run the inspector on our example of three rectangles, position the mouse over R3 (the black rectangle) in the window, and hit the HELP key. If your keyboard does not have a HELP key, or hitting it does not seem to do anything, you can start the Inspector

chapterly by typing `(gd:Inspector R3)` into the lisp listener. The Inspector window that appears will look like figure [inspector], page 49.

```
┌─────────────────────────────────────────────────────────────────────┐
│ ▬                        Showing #k<R3>                         ▫  □  │
├─────────────────────────────────────────────────────────────────────┤
│  ┌───────────┐ ┌───────────┐ ┌─────────┐ ┌────────────┐ ┌────┐ ┌────────┐
│  │Show Object│ │Show in New│ │Re-Fetch │ │Dependencies│ │Done│ │Done All│
│  └───────────┘ └───────────┘ └─────────┘ └────────────┘ └────┘ └────────┘
│  ┌─────┐ ┌──────┐ ┌──────┐ ┌─────┐ ┌────────────┐ ┌───────────────┐ ┌───────┐
│  │Flash│ │Search│ │Notify│ │Break│ │Clear Breaks│ │Inherited Slots│ │Objects│
│  └─────┘ └──────┘ └──────┘ └─────┘ └────────────┘ └───────────────┘ └───────┘
│  #k<R3>                                                               │
│                                                                       │
│  :IS-A = (#k<PROTO-RECT>)                                             │
│  :LEFT = 60                                                           │
│  :TOP = 40                                                            │
│  :WIDTH = 100                                                         │
│  :VISIBLE - #k<F16-259-260-261-264> (V) = T                          │
│  :FILLING-STYLE = #k<OPAL:BLACK-FILL>                                 │
│  :PARENT = #k<TOP-AGG>                                                │
│  :UPDATE-INFO = #<Update-Info dirty-p NIL invalid-p NIL>             │
│  :WINDOW = #k<WIN>                                                    │
└─────────────────────────────────────────────────────────────────────┘
```
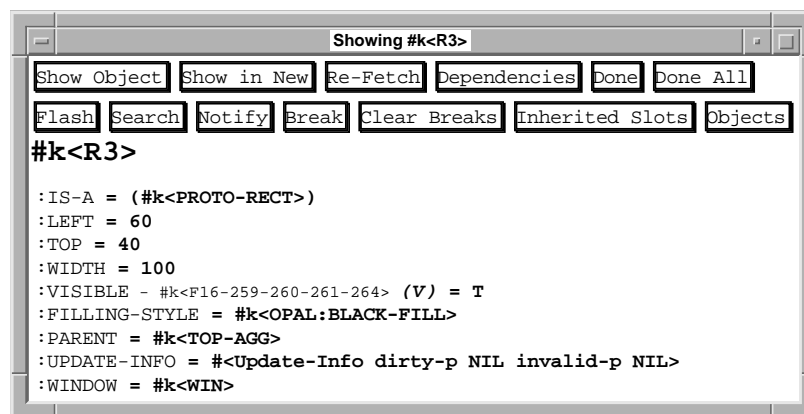
Figure 3.5: The Inspector displaying the slots and values of rectangle R3.

The local slots and values for R3 are shown in the Inspector window. Inherited slots are not shown, like `:height` or `:line-style` (assuming that you did not set these slots yourself,

installing local values in R3). If you have a color screen, some slots are red, indicating that these slots are public "parameters" of the object (we discuss parameters more in section ⟨undefined⟩ [[], page ⟨undefined⟩parameters]).

It is very easy to change properties of an object with the Inspector. For example, to change the `:width` of R3 using the inspector, click the mouse on the value of the `:width` slot (which is 100 in figure ⟨undefined⟩ [[], page ⟨undefined⟩inspector]). Use standard Emacs commands to change the value of the slot to something significantly different, like 20. When you hit RETURN, the change will appear instantly in R3.

To add a new local value to R3 – that is, to override an inherited value with a new local value – you have to add an extra line to the Inspector window. In our example, R3 does not have a local value for `:height`, since its value is inherited from the prototype PROTO-RECT. To override this value, click the cursor at the end of a line, and type `^j` to add a new line to the display. Now you can type `":height = 100"` and hit RETURN to install the new slot/value pair. The change should be reflected instantly in R3.

You can bring up other Inspector windows by positioning the mouse over another object and hitting HELP again, or you can select text that is already displayed in the Inspector and using the `"Show Object"` or `"Show in New"` buttons. For example, to examine the `opal:black-fill` object that is the value of R3's `:filling-style` slot, either click-and-drag or double-click on the `#k<OPAL:BLACK-FILL>` value and press the `"Show in New"` button. The object will be displayed in a new window.

When you are finished with the Inspector, you can click on the `"Done"` or `"Done All"` buttons to make the Inspector windows disappear.

Significantly more detail about the Inspector is included in the Debugging chapter, including how to explore the Prototype/Instance hierarchy of objects, and how to use the Inspector for debugging more compilcated examples.

3.9 Parameters

Most objects in Garnet have a list of *parameters*, which are stored in the `:parameters` slot. This is a list of all customizable properties of the object. For example, `gv opal:rectangle :parameter` yields:

```
(:LEFT :TOP :WIDTH :HEIGHT :LINE-STYLE :FILLING-STYLE :DRAW-FUNCTION
:VISIBLE)
```

These can be considered the "public" slots of `opal:rectangle`, which can be given customized values when instances are created. If values are not supplied for these slots when instances are created, the default values will be inherited from the prototype object.

There are other slots that change when instances of `opal:rectangle` are added to a window, such as the `:window` and `:parent` slots, but these slots are not intended to be set chapterly. Since they are "read-only" slots, they are not included in the `:parameters` list.

Several tools in Garnet rely heavily on the `:parameters` slot. As discussed in section [inspector-sec], page 48, the Inspector displays the parameter slots in red, so that they are easily identified. The `gg:prop-sheet` gadget which is used in Gilt and Lapidary looks at the `:parameters` slot to determine which slots should be displayed for the user to customize. These objects are discussed thoroughly in later sections of this reference chapter.

The typical Garnet user will not have to worry much about the `:parameters` slot. All of the slots that are in the list are documented in this chapter, so it is really just another way to access the same information about properties of objects. For details on defining `:parameters` slots for your own objects, see the KR chapter. Unless you are defining your own list for a special object, the `:parameters` slot should be considered read-only.

3.10 Destroying Objects

Before moving on to the next section, destroy the window so that it does not interfere with future examples in this tutorial. Type the following line.

```
(opal:destroy WIN)
```

Destroying the window will also destroy all of the objects that were added to its aggregate. We can no longer manipulate R1, R2, and R3, since they were destroyed by the previous call. However, the PROTO-RECT was never added to the top-level aggregate, and it was not destroyed. You could destroy this object now with a `destroy` call, but we will be using this object again in Section [destroying], page 51. So, leave the object residing in memory for now.

When an object is destroyed, its variable name becomes unbound and the memory space that was allocated to the object is freed. You can `destroy` any object, including windows. If you destroy a window, all objects inside of it are automatically destroyed. Similarly, if you destroy an aggregate, all objects in it are destroyed. When you destroy a graphical object (like a line or a circle), it is automatically removed from any aggregate it might be in and erased from the screen.

If a prototype object is destroyed (i.e., an object that has had instances created from it), then all of the instances of that object will be recursively destroyed.

Occasionally in the course of developing a program, you may (either accidentally or intentionally) define a new object which happens to have the same name as an old object. When the new object is created, its variable name is set to the new object, and the old object by the same name is destroyed. Also, all of the instances of the old object are recursively destroyed.

For example, in Section [prototypes], page 44, above, we created the object PROTO-RECT, which still exists in memory. If we now enter the following new schema definition for an object by the same name, then the old PROTO-RECT will be destroyed.

```
(create-instance 'PROTO-RECT opal:rectangle)
```

When the new schema is entered, a warning is given that the old object is being destroyed. You can safely ignore this message, assuming that you intended to override the definition of the old schema.

3.11 Unnamed Objects

Sometimes you will want to create objects that do not have a particular name. For example, you may want to write a function that returns a rectangle, but it will be called repeatedly and should not destroy previous instances with new ones. In this case, you should return an unnamed rectangle from the function which can be used just like the named objects we have created earlier in this tutorial.

As an example, the following code creates an unnamed object and internally generates a unique variable name for it. Instead of supplying a quoted name to `create-instance`, we give it the value `nil`.

```
(create-instance NIL opal:rectangle
    (:left 10) (:top 10) (:width 75) (:height 50))
```

When you enter this schema definition, the `create-instance` call will return the generated internal name of the rectangle – something like `RECTANGLE-123`. This name has a unique number as a suffix that prevents it from being confused with other rectangles in Garnet. You can now use the generated name to refer to the object.

```
(gv RECTANGLE-123 :top)  ; Replace this name with the name of your rectangle.█
```

Usually it is convenient to assign an unnamed object to a local variable. The following line creates a circle and assigns it to the new variable MY-CIRCLE.

```
(setf MY-CIRCLE (create-instance NIL opal:circle))
```

Now MY-CIRCLE will have the generated circle as its value. If the same line were entered again, the old circle would not be destroyed, but the variable MY-CIRCLE would still point to a new one. This can be useful inside a function that uses a `let` clause – every time the `let` is executed, new objects are assigned to the local variables, but the old objects still remain in memory and are not destroyed. Section [function], page 90, contains an example of how unnamed objects might be used in a function.

3.12 Lines, Rectangles, and Circles

The Opal package provides different graphical shapes including circles, rectangles, round-tangles, and lines. There are also several different kinds of text, and some special objects like bitmaps and arrowheads. Each graphical object has special slots that determine its appearance, which are documented in the Opal chapter. (For example, the line uses the slots `:x1`, `:y1`, `:x2`, and `:y2`.) See the section "Specific Graphical Objects" in the Opal chapter for details of how each object works. Examples of creating instances of graphical objects appear throughout this tutorial.

3.13 Aggregates

In order to put a large number of objects into a window, we might create all of the objects and then add them, one at a time, to the window. However, this is usually not how we organize the objects conceptually. For example, if we were to create a sophisticated interface with a scroll bar, several buttons, and labels for the buttons, we would not want to add each rectangle in the scroll bar and the buttons individually. Instead, we would think of creating the scroll bar from its composite rectangles, then creating the buttons along with their labels, and then adding the scroll bar assembly and the button assembly to the window.

Grouping objects together like this is the function of the `aggregate` object. Any graphical object can be a component of an aggregate - lines, circles, rectangles, and even other aggregates. Usually all of the components of an aggregate are related in some way, like they are all parts of the same button.

Two other objects, the `aggregadget` and the `aggrelist`, are also used to group objects, and usually appear more often in Garnet programs. `Aggregadgets` and `aggrelists` are

instances of `aggregate`, and they have special features that make them very useful in creating objects. These objects will be discussed further in Section [aggregadgets], page 55.

The top-level object in a window is always an aggregate. This aggregate contains all of the objects in that window. Therefore, for an object to appear in a window it either has to be a component of the top-level aggregate, or it has to be a component of another aggregate which, at the top of its aggregate hierarchy, is a component of the top-level aggregate.

When aggregates have other aggregates as components, an aggregate hierarchy is formed. This hierarchy describes the way that objects are grouped together. Figure [v-scroll-hierarchy], page 54, shows how the objects that comprise a vertical scroll bar might be conceptually organized.
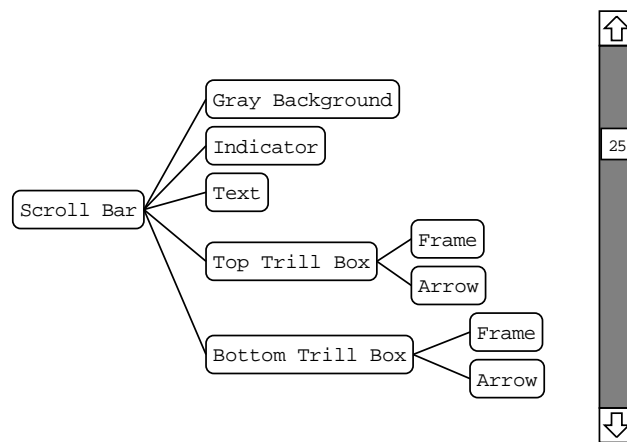
Figure 3.6: One possible hierarchy for the objects that make up a scroll bar.

In the scroll bar hierarchy, all of the leaves correspond to shapes that appear in the scroll bar. The leaves are always Opal graphic primitives, like rectangles and text. The nodes `top-trill-box` and `bottom-trill-box` are both aggregates, each with two components. And, of course, the top-level `scroll-bar` node is an aggregate.

This aggregate hierarchy should not be confused with the inheritance hierarchy that was discussed earlier. Components of an aggregate do not inherit values from their parents. Instead, relationships among aggregates and components must be explicitly defined using constraints, a concept which will be discussed shortly in this tutorial.

When an object is added to an aggregate, its `:parent` slot is set to point to that aggregate. Therefore, in Figure [v-scroll-hierarchy], page 54, the `:parent` of the `bottom-trill-box` is the `scroll-bar` aggregate. This `:parent` slot is called a *pointer* slot because its value is another Garnet object. Pointer slots are discussed further in section [aggregadgets], page 55.

The functions `add-component`, `remove-component`, and `move-component` are used to manipulate the components of an aggregate. Descriptions of these and other functions for components may be found in the "Aggregate Objects" section of the Opal chapter.

3.14 Aggregadgets, Aggrelists, and Aggregraphs

Aggregadgets and aggrelists are types of aggregates. With these objects, an aggregate and its components can basically be defined simultaneously. In aggregadgets, all the components are defined with a list in the `:parts` slot. In an aggrelist, a single object is defined to be an "item-prototype", and the aggrelist automatically generates several instances of that object to make its components. The aggregraph is a type of aggregadget, where all the components are nodes and arcs that make up a graph. Figures [opal-inheritance], page 43, and [v-scroll-hierarchy], page 54, were created using Garnet aggregraphs. For several examples and a complete discussion of how to use aggregraphs, see the Aggregadgets, Aggrelists, and Aggregraphs Reference chapter.

3.14.1 Aggregadgets

When you create an aggregadget, you may list all of the objects that you want as components of the aggregadget in the `:parts` slot. The list is specified using the standard Lisp backquote macro, and there are usually many function calls and objects inside the list that must be evaluated with a comma. As an example of an aggregadget, we will analyze the following schema definition, but it is not necessary to type it in. This code contains a few references that have not been discussed in this tutorial yet, but it serves the purpose of giving us a plain aggregadget to study.

```
(create-instance 'AGG opal:aggregadget
   (:left 10) (:top 20)
   (:parts
    '((:my-circle ,opal:circle
                  (:left 60) (:top 70)
                  (:width 100) (:height 100)
                  (:line-style ,opal:dashed-line))
       (:my-rect ,opal:rectangle
                  (:left ,(o-formula (gvl :parent :left)))
                  (:top ,(o-formula (gvl :parent :top)))
                  (:width 80) (:height 40)
                  (:filling-style ,opal:black-fill)
                  (:line-style NIL)))))
```

The `:parts` slot in the AGG object contains a list of lists, with each internal list being a definition of a component. The components of AGG will be a circle and a rectangle, to which we have given the arbitrary names `:my-circle` and `:my-rect`. These names, which are preceded by a colon, will be the names of new slots in the aggregadget. That is, two *pointer* slots will be created in AGG, named `:my-circle` and `:my-rect`, which will have the circle and rectangle objects as their values. We say these are pointer slots because they point to other objects.

Other pointer slots which are automatically created are the `:parent` slots of both the circle and the rectangle. Since these objects are being added as components to the aggregadget, their `:parent` slots are set as with aggregates. Thus, a two-way path of communication is established between the aggregadget and each of its components – the `:parent` slot points up, and the `:my-circle` slot points down.

Notice that the `:parts` list is backquoted (with a ' instead of a '). Using this backquote syntax, we can then use commas to evaluate the names of objects inside the list. The references to be evaluated are the two graphical object prototypes (the `opal:circle` and the `opal:rectangle`) and the graphical qualities (`opal:dashed-line` and `opal:black-fill`). Commas are also used to evaluate the `o-formula` calls, which establish constraints among objects (constraints are discussed in the chapter [constraints], page 683). If the commas were not present inside the `:parts` list, then the names of all the Garnet objects would not be dereferenced, and they would be treated as mere atoms, not objects. Similarly, the calls to `o-formula` would appear as simple quoted lists instead of function calls.

An important difference between aggregates and aggregadgets is that when you create an instance of an aggregadget, the instance will automatically have components that match those in the prototype. That is, if we created an instance of AGG, called AGG-INSTANCE, then AGG-INSTANCE would automatically have a circle and rectangle component just like AGG. In contrast, when you create an instance of an aggregate, the components are not automatically generated, and you would have to create and add them to the instance chapterly.

Other examples of aggregadget definitions can be found in sections [aggregadget-ex], page 65, and [big-example], page 76, in this tutorial.

3.14.2 Aggrelists

An aggrelist allows you to create and easily arrange objects into a nicely formatted graphical list. The motivation for aggrelists comes from the arrangement of groups of objects like button panels, tic-marks, and menu choices, where all the components of an aggregate are similar and should appear in a vertical or horizontal list.

In an aggrelist, a single item-prototype is defined, and then this object is automatically copied several times to make the components of the aggrelist. The `:left`, `:top`, and other slots of the components are automatically given values that will neatly lay out the components in a list, so that the programmer does not have to do any calculations for the positions of the objects.

As with aggregadgets, aggrelists use the backquote syntax to define the item prototype. There are many customizable aspects of aggrelists, such as whether to orient the components vertically or horizontally, the distance between each object, etc. Since there are so many customizable slots, please see the Aggregadgets and Aggrelists Reference chapter for a

discussion of how to use aggrelists. Section [big-example], page 76, in this tutorial includes an example of the definition of an aggrelist.

3.15 Windows

When we want to add an object to a window, what we really mean is that we want to add the object to the window's top-level aggregate (or to an aggregate at a lower level in that window's aggregate hierarchy). Every window has one top-level aggregate, and all objects that appear in the window are components in its aggregate hierarchy.

Any object must be added to a window in order for it to be shown on the screen. Additionally, a window must be updated before any changes made to it (or the objects in it) will appear. Windows are updated when you explicitly issue a call to `opal:update`, and they are also continuously updated when interactors are running and changing objects in the window (during the `main-event-loop`, discussed in Section [interactors], page 693).

3.16 Gadgets

The Garnet gadgets are a set of ready-made widgets that can be treated as regular graphical objects. They have slots that can be customized with user-defined values, and are added to windows just like graphical objects. Generally, they are objects that are commonly found in an interface including scroll bars, menus, buttons and editable text fields. In the Tour, you created instances of the radio button panel and the vertical slider. There are also more sophisticated gadgets like scrolling windows, property sheets (to allow quick editing of the slots of objects), and selection handles (for moving and growing objects).

Most of the gadgets come in two versions – one called the Garnet Style, and one modeled after the OSF/Motif style. Examples of how to use the gadgets are found in demonstration programs at the end of each of the gadget files, which can be executed by commands like (`garnet-gadgets:menu-go`). For detailed descriptions of all the available gadgets, see the Gadgets Reference chapter.

3.17 Constraints

In the course of putting objects in a window, it is often desirable to define relationships among the objects. For example, you may want the tops of several objects to be aligned, or you might want a set of circles to have the same center, or you may want an object to change color if it is selected. Constraints are used in Garnet to define these relationships among objects.

Although all the examples in this section use constraints on the positions of objects, it should be clear that constraints can be defined for filling styles, strings, or any other property of a Garnet object. Many examples of constraints can be found in other sections of this tutorial. Additionally, much of the KR Reference chapter is devoted to the discussion of constraints among objects. The sections "Constraint Maintenance" and "Slot and Value Manipulation Functions" should be of particular interest.

3.18 Formulas

A formula is an explicit definition of how to calculate the value for a slot. If we want to constrain the top of one object to be the same as another, then we define a formula for the

`:top` slot of the dependent object. With constraints, the value of one slot always *depends* on the value of one or more other slots, and we say the formula in that slot has *dependencies* on the other slots.

An important point about constraints is that they are constantly maintained by the system. That is, they are evaluated once when they are first created, and then they are continually *re-evaluated* when any of their dependencies change. Thus, if several objects depend on the top of a certain rectangle, then all the objects will change position whenever the rectangle is moved.
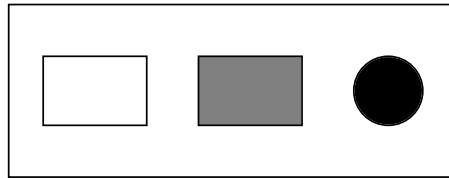
Figure 3.7: Three objects that are all aligned with the same top. The top of the gray rectangle is constrained to the white rectangle, and the top of the black circle is constrained to the top of the gray rectangle.

As our first example of defining constraints among objects, we will make the window in Figure [align-top], page 59. Let's begin by creating the white rectangle at an absolute

position, and then create the other objects relative to it. Create the window and the first
box with the following code.

```
(create-instance 'CONSTRAINTS-WIN inter:interactor-window      ; Create the window
   (:left 750)(:top 80)(:width 260)(:height 100))
(create-instance 'TOP-AGG opal:aggregate)                      ; Create an aggregate
(s-value CONSTRAINTS-WIN :aggregate TOP-AGG)                   ; Assign the ag-
gregate to the window
(opal:update CONSTRAINTS-WIN)                                  ; Make the win-
dow appear

(create-instance 'WHITE-RECT opal:rectangle                   ; Create a rectangle
   (:left 20) (:top 30)
   (:width 60) (:height 40)
   (:filling-style opal:white-fill))

(opal:add-components TOP-AGG WHITE-RECT)                       ; Add the rect-
angle to the window
(opal:update CONSTRAINTS-WIN)                                  ; Make changes in the w
dow appear
```

We are now ready to create the other objects that are aligned with WHITE-RECT.
We could simply create another rectangle and a circle that each have their top at 30,
but this would lead to extra work if we ever wanted to change the top of all the objects,
since each object's `:top` slot would have to be changed individually. If we instead define a
relationship that depends on the top of WHITE-RECT, then whenever the top of WHITE-
RECT changes, the top of the other objects will automatically change, too. Define the
schema for the gray rectangle as follows.

```
(create-instance 'GRAY-RECT opal:rectangle
   (:left 110)
   (:top (o-formula (gv WHITE-RECT :top)))  ; Constrain the top of this rect-
angle to the top of WHITE-RECT
   (:width 60) (:height 40)
   (:filling-style opal:gray-fill))

(opal:add-components TOP-AGG GRAY-RECT)
(opal:update CONSTRAINTS-WIN)
```

You can see that without specifying an absolute position for the top of the gray rectangle,
we have constrained it to always have the same top as the white rectangle. The formula in
the `:top` slot of the gray rectangle was defined using the functions `o-formula` and `gv`. The
`o-formula` function is used to declare that an expression is a constraint. When `gv` is used
inside a formula, it causes a dependency to be established on the referenced slot, so that
the formula will be reevaluated when the value in the referenced slot changes.[1]

---

[1] There is another function called `g-value` that is similar to `gv`, except that it never causes dependencies
to be established. Older versions of Garnet required that `gv` only be used inside formulas, and `g-value`
to be used ouside. The `gv` function has since been enhanced so that it can be used everywhere. It would
be unusual to ever need to use `g-value`.

To see if our constraint is working, try changing the top of the white rectangle with the following instructions and notice how the gray rectangle moves with it. Try setting the top to other values, if you wish.

```
(s-value WHITE-RECT :top 50)
(opal:update CONSTRAINTS-WIN)
```

The important thing to notice is that the value of the :top slot of GRAY-RECT changes as the top of the WHITE-RECT changes. This shows that the formula in GRAY-RECT is being re-evaluated whenever its depended values change.

Now we are ready to add the black circle to the window. We have a choice of whether to constrain the top of the circle to the white rectangle or the gray rectangle. Since we are going to be examining these objects closely in the next few paragraphs, let's constrain the circle to the gray rectangle, resulting in an indirect relationship with the white one. Define the black circle with the following code.

```
(create-instance 'BLACK-CIRCLE opal:circle
   (:left 200)
   (:top (o-formula (gv GRAY-RECT :top)))
   (:width 40) (:height 40)
   (:filling-style opal:black-fill))


(opal:add-components TOP-AGG BLACK-CIRCLE)
(opal:update CONSTRAINTS-WIN)
```

At this point, you may want to set the :top of the white rectangle again just to see if the black circle follows along with the gray rectangle.

3.19 Cached Values

An interesting question might have occurred to you – what happens if you set the :top of the gray rectangle now? Setting the value of a slot which already has a formula in it does **not** destroy the existing constraint. However, it does override the current *cached* value of the formula. Try setting the :top of the gray rectangle now.

```
(s-value WHITE-RECT :top 30)   ; Return everything to its original position█
(opal:update CONSTRAINTS-WIN)

(s-value GRAY-RECT :top 40)
(opal:update CONSTRAINTS-WIN)
```

The position of WHITE-RECT will remain unchanged, since it was defined with an absolute position. However, the new value that we gave for the top of the gray rectangle has repositioned GRAY-RECT and BLACK-CIRCLE. Previously, the formula in the :top slot of GRAY-RECT had correctly computed its own top, getting the value from the :top slot of WHITE-RECT. Now, however, we replaced that cached value with our absolute value of 40.

To show that the formula is still alive and well in the :top slot of GRAY-RECT, try setting the :top slot of WHITE-RECT again.

```
(s-value WHITE-RECT :top 10)
(opal:update CONSTRAINTS-WIN)
```

Since the top of GRAY-RECT depends on WHITE-RECT, its formula will be recomputed whenever the top of WHITE-RECT changes. There is now a new cached value for the `:top` of GRAY-RECT, a result of re-evaluating the formula.

3.20 Formulas and s-value

It is important to distinguish the behavior of `s-value` when it is used on a slot with a formula in it, versus using it on a slot with an absolute value in it (like the number 5). Setting the value of a slot that *already* has a formula in it will not destroy the old formula. Instead, only the cached value of the formula is changed, and the formula will be re-evaluated if any of its dependencies change.

On the other hand, `s-value` will replace one absolute value with another absolute value, and the old value will never appear again. That is, if an object was created with some particular absolute value for a slot, and we changed that slot with `s-value`, then the new value will be permanent until the slot is explicitly set again with `s-value`.

The one exception to the above rules is when the new value is a formula itself. Using `s-value` to set a new formula will always obliterate what was previously in the slot, whether it was an absolute value or a formula.

3.21 Using the :obj-over Slot

When designing an interface, you may want a box to be drawn around an object to show that it is selected. In the usual case, you will want to define only one box that will be drawn around different objects, and it would be nice if the box changed size when it was over objects of different size. The traditional Garnet approach to this problem is to use constraints in the dimension slots of the selection box that depend on the dimensions of the object it is over.
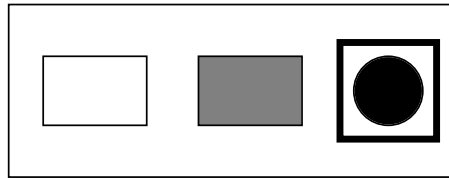
Figure 3.8: A selection box drawn around an object.

In the traditional approach, we use the slot `:obj-over` in the selection box to specify which object the selection box should be drawn around. The `:obj-over` slot is a pointer slot, since it contains an object as its value (pointer slots were discussed in section ⟨undefined⟩ [[], page ⟨undefined⟩aggregadgets]). Then, we define formulas for the dimensions

of the selection box which depend on the `:obj-over` slot. The formulas in the following schema definition should be clear.

```
(create-instance 'SEL-BOX opal:rectangle
   (:obj-over GRAY-RECT)  ; A pointer slot
   (:left (o-formula (- (gvl :obj-over :left) 10)))
   (:top (o-formula (- (gvl :obj-over :top) 10)))
   (:width (o-formula (+ 20 (gvl :obj-over :width))))
   (:height (o-formula (+ 20 (gvl :obj-over :height))))
   (:line-style opal:line-4))  ; A line with a thickness of 4 pixels


(opal:add-components TOP-AGG SEL-BOX)
(opal:update CONSTRAINTS-WIN)
```

Now if you set the `:obj-over` slot of the selection box to be a different object, the position and dimensions of SEL-BOX will change according to the object.

```
(s-value SEL-BOX :obj-over BLACK-CIRCLE)
(opal:update CONSTRAINTS-WIN)
```

You may have noticed that we performed computations in the formulas above, instead of just using values directly as in the GRAY-RECT and BLACK-CIRCLE objects. In fact, formulas can contain any lisp expression. Also, the formulas above use the function `gvl` instead of `gv`, which was used earlier. We use `gvl` here because we are referencing a slot, `:obj-over`, in the same schema as the formula. Previously, we used `gv` to look at a slot in a different object.

Another point of interest in these formulas is the use of indirect references. In the `:left` formula above, the pointer slot `:obj-over` is referenced, and then its `:left` slot is referenced, in turn. It is important to distinguish between this case and the case where a value is stored in the same object as the formula, as in the following example.

Suppose we want the selection box to become invisible if no object is currently selected. Using `s-value`, we can set the `:visible` slot of SEL-BOX to have a formula which will cause the box to disappear when its current selection is `nil`. (We could have also defined this formula in the `create-instance` call.)

```
(s-value SEL-BOX :visible (o-formula (gvl :obj-over)))
```

Clearly, if `:obj-over` is set to `nil`, then the value in the `:visible` slot will also become `nil`, and the box will become invisible. When `:obj-over` is again set to be some object, then the `:visible` slot will have a non-NIL value, and the box will appear in the appropriate position. Previously, if we had set `:obj-over` to `nil`, then updating the window would have caused an error when the formulas tried to access the `:left`, `:top`, `:width`, and `:height` of the non-existent selected object.

We are finished with the objects from this section, but the next section will continue to use the same window. So, to remove the old objects from the window, use the function `remove-components`.

```
(opal:remove-components TOP-AGG WHITE-RECT GRAY-RECT BLACK-CIRCLE SEL-BOX)▮
(opal:update CONSTRAINTS-WIN)
```

3.22 Constraints in Aggregadgets

As mentioned in section [aggregates], page 202, the `:parent` slot of a component is automatically set to its parent aggregate when it is attached. Since aggregadgets are instances of aggregates, all of the components defined in an aggregadget have their `:parent` slot set in this way. In this section, we will examine how this slot can be used to communicate between components of an aggregadget.

The aggregadget we will use in this example will make the picture of concentric circles in Figure [concentric-circles], page 66. Suppose that we want to be able to change the size and position of the circles easily, and that this should be done by setting as few slots as possible.
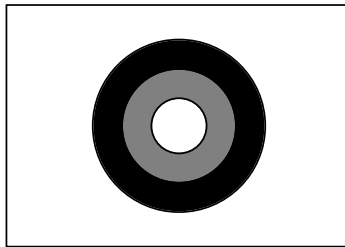
Figure 3.9: An aggregadget with three circles as components.

From the picture, we see that the dimensions of the black circle are the same as the dimensions of the entire group of objects. That is, if a bounding box were drawn around the black circle, all the other objects would be inside the bounding box, too. Therefore, it

will be helpful to put slots for the size and position of this circle in the top-level aggregadget, and have all the circles reference these top-level values through formulas.

To start, let's create an aggregadget with only one component – the black circle – and then redefine the aggregadget with the other components later. The following code creates this one-component aggregate.

```
(create-instance 'CON-CIRCLES opal:aggregadget
   (:left 20) (:top 20)
   (:width 100)
   (:height (o-formula (gvl :width)))
   (:parts
    `((:black-circle ,opal:circle
                     (:left ,(o-formula (gvl :parent :left)))
                     (:top ,(o-formula (gvl :parent :top)))
                     (:width ,(o-formula (gvl :parent :width)))
                     (:height ,(o-formula (gvl :width)))
                     (:filling-style ,opal:black-fill)))))

(opal:add-component TOP-AGG CON-CIRCLES)
(opal:update CONSTRAINTS-WIN)
```

All those commas are needed because we want the Opal objects and the calls to `o-formula` to be evaluated inside the backquoted list. If the commas were not present, then those forms would become inert atoms and lists instead of objects and function calls.

The black circle in the aggregadget gets its position and dimensions from the top-level slots in CON-CIRCLES. The communication link used here is the `:parent` slot, which points from the component to the aggregadget. The function `gvl` is used in the formulas for the black circle because the `:parent` slot is in the same object as the formulas. Notice that the black circle does **not** "inherit" any values from its parent. Creating components in an aggregadget sets up an *aggregate* hierarchy, where values travel back-and-forth over constraints, not inheritance links. If you want a component to depend on values in its parent, you have to define constraints.

The other components of CON-CIRCLES will be defined analogously, but with a little more computation in the formulas to get them to line up properly. Before typing in the new definition of CON-CIRCLES, remove the old aggregadget from the window with the following instruction.

```
(opal:remove-components TOP-AGG CON-CIRCLES)
(opal:update CONSTRAINTS-WIN)
```

And now we are ready to redefine CON-CIRCLES again, this time with an extra top-level slot to reduce redundant calculations, and of course with the other two circles.

```
(create-instance 'CON-CIRCLES opal:aggregadget
   (:left 20) (:top 20)
   (:width 100)
   (:height (o-formula (gvl :width)))
   (:radius/3 (o-formula (round (gvl :width) 6)))
   (:parts
    `((:black-circle ,opal:circle
```

```
                        (:left ,(o-formula (gvl :parent :left)))
                        (:top ,(o-formula (gvl :parent :top)))
                        (:width ,(o-formula (gvl :parent :width)))
                        (:height ,(o-formula (gvl :width)))
                        (:filling-style ,opal:black-fill))
        (:gray-circle ,opal:circle
                        (:left ,(o-formula (+ (gvl :parent :left)
   (gvl :parent :radius/3))))
                        (:top ,(o-formula (+ (gvl :parent :top)
  (gvl :parent :radius/3))))
                        (:width ,(o-formula (- (gvl :parent :width)
    (* 2 (gvl :parent :radius/3)))))
                        (:height ,(o-formula (gvl :width)))
                        (:filling-style ,opal:gray-fill))
        (:white-circle ,opal:circle
                        (:left ,(o-formula (+ (gvl :parent :gray-circle :left)▉
   (gvl :parent :radius/3))))
                        (:top ,(o-formula (+ (gvl :parent :gray-circle :top)▉
  (gvl :parent :radius/3))))
                        (:width ,(o-formula (- (gvl :parent :gray-circle :width)▉
     (* 2 (gvl :parent :radius/3)))))
                        (:height ,(o-formula (gvl :width)))
                        (:filling-style ,opal:white-fill)))))
```

```
    (opal:add-components TOP-AGG CON-CIRCLES)
    (opal:update CONSTRAINTS-WIN)
```

The gray circle gets its size and position from the top-level slots just like the black circle, only it is one-third the size. The white circle is the most interesting case, where it gets its position and dimensions from the gray circle. Not only does the white circle communicate with the aggregadget through the `:parent` slot, but it also uses the slot `:gray-circle` which was automatically created in the aggregadget (see section ⟨undefined⟩ [[], page ⟨undefined⟩aggregadgets]). Thus, the formulas in the white circle trace up the aggregate hierarchy to the parent aggregadget, and then back down into another component.

To examine these pointer slots more closely, try executing the following line.

```
(gv CON-CIRCLES :white-circle)
```

The value returned by this `gv` call is the internally generated name of the white circle. This name was generated with a unique suffix number so that it will not be confused with some other white circle in Garnet (see Section [unnamed-objects], page 51). You can also look at slots of the components directly, by adding slot names to the end of the `gv` call, like

```
(gv CON-CIRCLES :white-circle :top)
```

or even

```
(gv CON-CIRCLES :white-circle :parent)
```

This is the end of the section regarding constraints. Destroy the window we have been using to keep it from interfering with future examples in the tutorial.

```
(opal:destroy CONSTRAINTS-WIN)
```

3.23 Interactors

Interactors are used to communicate with the mouse and keyboard. Sometimes you may just want a function to be executed when the mouse is clicked, but often you will want changes to occur in the graphics depending on the actions of the mouse. Examples include moving objects around with the mouse, editing text with the mouse and keyboard, and selecting an object from a given set.

The fundamental way that the interactors communicate with graphical objects is that they set slots in the objects in response to mouse movements and keyboard key strokes. That is, they generate side effects in the objects that they operate on. For example, some interactors set the `:selected` and `:interim-selected` slots to indicate which object is currently being operated on. When objects are defined with formulas that depend on these special slots, the appearance of the objects (i.e., the graphics of the interface) can change in response to the mouse.

It is important to note that all of the gadgets come with their interactors already defined. Therefore, you do not need to create interactors that change the gadgets.

In this section we will see some examples of how to change graphics in conjunction with interactors. Section [trace-inter], page 96, describes how to use an important debugging function for interactors called `trace-inter`. Although this tutorial only gives examples of using the `button-interactor` and `move-grow-interactor`, the Interactors chapter discusses and provides examples for all six types of Garnet interactors.

3.24 Kinds of Interactors

The design of the interactors is based on the observation that there are only a few kinds of behaviors that are typically used in graphical user interfaces. Currently, Garnet supports seven types of interactive behavior, which allows a wide variety of user actions in an interface. Below is a list of the available interactors, which are described in detail in the Interactors chapter.

Menu-Interactor
> For selecting one or more choices from a set of items. Useful in menus, etc., where the mouse may be held down and dragged while moving over the items to be selected.

Button-Interactor
> For selecting one or more choices from a set of items. Useful in single buttons and panels of buttons where the mouse can only select one item per mouse click.

Move-Grow-Interactor
> For moving and changing the size of an object. Useful in scroll bars and graphics editors.

Two-Point-Interactor
> For obtaining one or two points in a window from the user. Useful in specifying the size and position of a new object to be created.

Angle-Interactor
> For getting the angle that the mouse moves around a point. Useful in circular gauges or for "stirring motions".

Text-Interactor

> For editing strings. Most useful for small strings, since Garnet does not support complicated word-processing applications.

Gesture-Interactor

> For recognizing gestures drawn by the user (e.g., the user draws a rough shape that Garnet recognizes as a square).

Animator-Interactor

> For executing a function at regular intervals, allowing rapid updating of graphics.

There are also several interactors that work with the `multifont-text` object. This object and its associated interactors are discussed in the Opal chapter.

3.25 The Button Interactor

In this example, we will perform an elementary operation with an interactor. We will create a window with a white rectangle inside, and then create an interactor that will make it change colors when the mouse is clicked inside of it. First, create the window with the white rectangle.

```
(create-instance 'INTER-WIN inter:interactor-window
   (:left 750)(:top 80)(:width 250)(:height 250))
(create-instance 'TOP-AGG opal:aggregate)
(s-value INTER-WIN :aggregate TOP-AGG)
(opal:update INTER-WIN)

(create-instance 'CHANGING-RECT opal:rectangle
   (:left 20) (:top 30)
   (:width 60) (:height 40)
   (:filling-style (o-formula (if (gvl :selected)
 opal:black-fill
 opal:white-fill)))
   (:selected NIL))  ; Set by the interactor

(opal:add-components TOP-AGG CHANGING-RECT)
(opal:update INTER-WIN)
```

From the definition of the `:filling-style` formula, you can see that if the `:selected` slot in CHANGING-RECT were to be set to be non-NIL, then its color would turn to black. Conveniently, setting the `:selected` slot is one of the side effects of the `button-interactor`. The following code defines an interactor which will set the `:selected` slot of CHANGING-RECT, which will cause it to change colors.

```
(create-instance 'COLOR-INTER inter:button-interactor
   (:window INTER-WIN)
   (:start-where (list :in CHANGING-RECT))
   (:start-event :leftdown))

; Unless using CMU, Allegro, Lucid, LispWorks, or MCL
```

```
(inter:main-event-loop)
; Hit F1 while the mouse is in the Garnet window to exit
```

The `main-event-loop` function causes Garnet to start paying attention to events (like clicking the mouse) that trigger the interactors. (A background process in CMU, Allegro, Lucid, LispWorks, and MCL always pays attention to events.) Now you can click on the rectangle repeatedly and it will change from white to black, and back again. From this observation, and knowing how we defined the `:filling-style` of CHANGING-RECT, you can conclude that the `button-interactor` is setting (and clearing) the `:selected` slot of the object. This is one of the functions of the `button-interactor`. When you are ready to resume typing in the Lisp process, you have to hit the F1 key while the mouse is in the Garnet window to get a new prompt. (You may execute the `main-event-loop` call again at any Lisp prompt.)
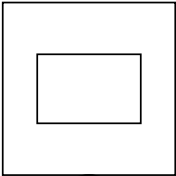
3.26 A Feedback Object with the Button Interactor

The method we used in the previous section with the `button-interactor` involved setting the `:selected` slot of the selected object. There is another way to use the button interactor which involves using feedback objects. A *feedback object* is some object that indicates the currently selected object. For example, it is often desirable that the actual selected object not move or change color, but rather that a separate object follow the mouse or appear over the selection.

In an earlier example in section [obj-over-slot], page 62, we defined a selection box which works just like a feedback object. When the `:obj-over` slot of the selection box was set to the name of the selected object, then the box appeared around the selected object. In this example, we will redefine the objects from that section and create an interactor to work on them.

The following code is analogous to what was presented in section [obj-over-slot], page 62, but here the three selectable objects are defined as components in an aggregadget. Type in the following aggregadget and feedback object, and add them to the current window. Notice that because of the `:visible` formula in FEEDBACK-RECT, that rectangle will be invisible when the window is first updated.

```
(create-instance 'AGG opal:aggregadget
   (:top 100)
   (:parts
    `((:white-rect ,opal:rectangle
                   (:left 20)
                   (:top ,(o-formula (gvl :parent :top)))
                   (:width 60)
                   (:height 40)
                   (:filling-style ,opal:white-fill))
      (:gray-rect ,opal:rectangle
                   (:left 110)
                   (:top ,(o-formula (gvl :parent :top)))
                   (:width 60)
                   (:height 40)
                   (:filling-style ,opal:gray-fill))
      (:black-circle ,opal:circle
                     (:left 200)
                     (:top ,(o-formula (gvl :parent :top)))
                     (:width 40)
                     (:height 40)
                     (:filling-style ,opal:black-fill)))))

(create-instance 'FEEDBACK-RECT opal:rectangle
   (:obj-over NIL)  ; A pointer slot to be set by the interactor
   (:visible (o-formula (gvl :obj-over)))
   (:left (o-formula (- (gvl :obj-over :left) 10)))
   (:top (o-formula (- (gvl :obj-over :top) 10)))
   (:width (o-formula (+ 20 (gvl :obj-over :width))))
   (:height (o-formula (+ 20 (gvl :obj-over :height)))))
```

```
        (:line-style opal:line-4))

    (opal:add-components TOP-AGG AGG FEEDBACK-RECT)
    (opal:update INTER-WIN)
```

Notice that the `:obj-over` slot of FEEDBACK-RECT is a pointer slot, as usual. When `:obj-over` is set with the name of an object, the FEEDBACK-RECT will appear over the object because of the way we defined its position and dimension formulas. In this example, we will not set `:obj-over` by hand, as we did previously. Instead, we will create a `button-interactor` to set the slot for us.

The following code defines an interactor which uses the FEEDBACK-RECT to indicate which object is selected. Since all of the selectable objects are in the same aggregate, we can tell the interactor to start whenever the mouse is clicked over any component of AGG.

```
    (create-instance 'SELECTOR inter:button-interactor
        (:window INTER-WIN)
        (:start-where (o-formula (list :element-of AGG)))  ; Work on the com-
    ponents of AGG
        (:final-feedback-obj FEEDBACK-RECT)
        (:how-set :toggle))

    ; Unless using CMU, Allegro, Lucid, LispWorks, or MCL
    (inter:main-event-loop)
    ; Hit F1 while the mouse is in the Garnet window to exit
```

Now when you click on the objects, the feedback object will appear over the selected object. The reason is that the `button-interactor` sets the `:obj-over` slot of the feedback object. Since the interactor is a toggling interactor (according to its `:how-set` slot), the `:obj-over` slot will be reset to `nil` when the selected object is clicked on again.

If you entered the `main-event-loop`, remember to hit the F1 key before typing in the next example.

3.27 The Move-Grow Interactor

From the previous example, you can see that it is easy to change the graphics in the window using the mouse. We are now going to define several more objects in the window and create an interactor to move them.

The side effect of the `move-grow-interactor` is that it sets the `:box` slot of the selected object (as well as the feedback object, if any) to be a list of four values – the left, top, width, and height of the object. When formulas are defined in the `:left`, `:top`, `:width`, and `:height` slots which depend on the `:box` slot, then the position and dimensions of the object will change whenever the `:box` slot changes.

The idea goes like this: Suppose the current value of the `:box` slot in a rectangle is '(0 0 40 40). Since the `:left` and `:top` slots are constrainted to the `:box` slot, the rectangle appears at the position (0,0). To move the object, the user clicks and drags on the rectangle until it is at position (50,50). When the user lets go, then the interactor automatically sets the `:box` slot to '(50 50 40 40). Since the `:box` slot has changed, the formulas in the `:left` and `:top` slot are re-evaluated, and the rectangle appears at the new position.

The following code creates a prototype circle and several instances of it. With a little study, it should be clear how the position and dimension formulas work with respect to the :box slot. All of the circles are then added to an aggregate, and this aggregate is added as a component to the top-level aggregate.

```
(create-instance 'MOVING-CIRCLE opal:circle
   (:box '(0 0 40 40))  ; This slot will be set by the interactor
   (:left (o-formula (first (gvl :box))))     ; Get the first value in the list
   (:top (o-formula (second (gvl :box))))
   (:width (o-formula (third (gvl :box))))
   (:height (o-formula (fourth (gvl :box)))))

(create-instance 'M1 MOVING-CIRCLE
   (:box '(120 30 40 40)))

(create-instance 'M2 MOVING-CIRCLE
   (:box '(30 100 60 60)))

(create-instance 'M3 MOVING-CIRCLE
   (:box '(120 100 80 80)))

(create-instance 'CIRCLE-AGG opal:aggregate)

(opal:add-components CIRCLE-AGG M1 M2 M3)

(opal:add-components TOP-AGG CIRCLE-AGG)
(opal:update INTER-WIN)
```

If you want to try setting the :box slot of any of these objects, you will see how the position and dimension of each circle depend on it. Be sure you set the :box slot to be a list of four positive numbers, or an error will occur!

Now let's create an instance of the move-grow-interactor, which will cause the moving circles to change position. The following interactor works on all the components of the aggregate CIRCLE-AGG. Remember to execute the inter:main-event-loop call to start the interactors working.

```
(create-instance 'MG-INTER inter:move-grow-interactor
   (:window INTER-WIN)
   (:start-where (list :element-of CIRCLE-AGG)))    ; Work on the compo-
nents of CIRCLE-AGG

; Unless using CMU, Allegro, Lucid, LispWorks, or MCL
(inter:main-event-loop)
; Hit F1 while the mouse is in the Garnet window to exit
```

Now if you press and drag in any of the circles, they will follow the mouse. This is because the interactor sets the :box slot of the object that it is pressed over, and the :left and :top slots of the objects depend on the :box slot.

It is worth noting once again that the `move-grow-interactor` does **not** set the `:left`, `:top`, etc. slots of the selected object. It instead sets the `:box` slot of the selected object, and the user is required to define formulas that depend on the `:box` slot.

3.28 A Feedback Object with the Move-Grow Interactor

Now let's add a feedback object to the window that will work with the moving circles. In this case, the feedback object will appear whenever we click on and try to drag a circle. The mouse will actually drag the feedback object, and then the real circle will jump to the final position when the mouse is released.

Our feedback object will be a circle with a dashed line. The DASHED-CIRCLE object defined below will have two slots set by the interactor. The `:box` slot will be set while the mouse is held down and dragged, and the `:obj-over` slot will be set to point to the circle being dragged. Given our MOVING-CIRCLE prototype, the feedback object is easy to define.

```
(create-instance 'DASHED-CIRCLE MOVING-CIRCLE
   ; Inherit all the :left, :top, etc. formulas that depend on the :box slot.█
   (:obj-over NIL)  ; Set by the interactor
   (:visible (o-formula (gvl :obj-over)))
   (:line-style opal:dashed-line))

(opal:add-components TOP-AGG DASHED-CIRCLE)
```

The `:visible` slot is set with a formula because we only want the feedback object to be visible when it is being used with the interactor. Now, we will redefine the move-grow interactor to use DASHED-CIRCLE as a feedback object. (Redefining the MG-INTER will destroy the old instance, so don't worry if a warning appears.)

```
(create-instance 'MG-INTER inter:move-grow-interactor
   (:window INTER-WIN)
   (:start-where (list :element-of CIRCLE-AGG))
   (:feedback-obj DASHED-CIRCLE))
```

Now when you move the circles with the mouse, the feedback object will follow the mouse, instead of the real circle following it directly.

Since we have finished this section on interactors, destroy the window so that it does not interfere with the next example. Type the following line.

```
(opal:destroy INTER-WIN)
```

3.29 Creating a Panel of Text Buttons

In this section, we will go through a comprehensive example that pulls together all the concepts that have been discussed in this tutorial. The final objective will be the panel of text buttons in Figure [text-buttons], page 431. We will use an aggregadget to assemble a group of objects into a single button, then use an aggrelist to make multiple copies of the text button and organize them into a list for the panel, and finally create a button interactor to manage the panel.

3.29.1 The Limitations of Aggregates

Before starting the aggregadget for this example, let's take a look at the use of an aggregate. This will help to demonstrate the usefulness of aggregadgets. First, create a window with a top-level aggregate and update it:

```
(create-instance 'BUTTON-WIN inter:interactor-window
   (:left 800)(:top 10)(:width 200)(:height 400))
(create-instance 'TOP-AGG opal:aggregate)
(s-value BUTTON-WIN :aggregate TOP-AGG)
(opal:update BUTTON-WIN)
```

The TOP-AGG aggregate is the top-level aggregate for the window. If we want any object to appear in the window, it will have to be added to TOP-AGG, or added to an aggregate further below in TOP-AGG's aggregate hierarchy. We will keep TOP-AGG as the top-level aggregate throughout this example, but we will be changing its components continually.

Now we can begin adding objects to TOP-AGG (throughout this discussion you should periodically check Figure [text-buttons], page 431, to see why we are creating particular objects). Let's start by assembling a button. We will first create a couple of rectangles that have a fixed position so that we get the window in Figure [two-rects], page 78. Since we want the rectangles to have the same dimensions, we can make a prototype object and then create two instances with appropriate position values.

```
(create-instance 'PROTO-RECT opal:rectangle
   (:width 100) (:height 50))

(create-instance 'R1 PROTO-RECT
   (:left 40) (:top 40)
   (:filling-style opal:black-fill))

(create-instance 'R2 PROTO-RECT
    (:left 20) (:top 20)
    (:filling-style opal:gray-fill))

(opal:add-components TOP-AGG R1 R2)
(opal:update BUTTON-WIN)
```
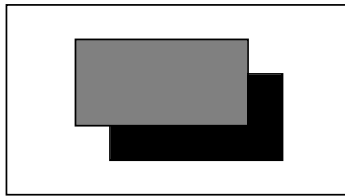
Figure 3.11: The two rectangles R1 and R2, which are instances of PROTO-RECT.

Keeping in mind our goal of making a panel of text buttons, one problem should be immediately clear. In order to make several buttons with this method, we will have to calculate the position of every rectangle in the interface and explicitly create an object for it. This will be time consuming, to say the least, and motivates us to investigate

how constraints will help avoid tedious calculations. So, as the first step in pursuing a more fruitful approach, let's remove the rectangles from the window and move on to aggregadgets. To remove the rectangles, execute:

```
(opal:remove-components TOP-AGG R1 R2)
(opal:update BUTTON-WIN)
```

3.29.2 Using an Aggregadget for the Text Button

When we create an aggregadget, we essentially create an aggregate and add the components (along with pointer slots) all at once. Our task is to build an aggregadget with two rectangles as components which will look like Figure [two-rects], page 78. Since we already know what we want the rectangles to look like in the window, putting a simple aggregadget together using our previously defined R1 and R2 rectangles is straightforward. However, the key to avoiding tedious calculations of the positions of our rectangles is to generalize the code. That is, we want the positions of our components to be formulas rather than absolute numbers.

For the present, let's assume that we will always be giving absolute numbers to our top-level aggregadget (but not its components). The first problem we want to address is how to devise formulas for the positions of the rectangles. By referring back to Figure [two-rects], page 78, we see that the entire aggregate has its upper-left coordinate at one corner of the gray rectangle, and its lower-right coordinate on the shadow. Therefore, it is a reasonable design decision to put the left and top of the gray rectangle at the left, top corner of the aggregadget, and then put the shadow 20 pixels further below and to the right. The following code shows the definition of our new BUTTON aggregadget with formulas defined for the positions of the rectangle components.

```
(create-instance 'BUTTON opal:aggregadget
   (:left 20) (:top 20)
   (:shadow-width 100) (:shadow-height 50)
   (:parts
    '((:shadow ,opal:rectangle
       (:left ,(o-formula (+ 20 (gvl :parent :left))))
       (:top ,(o-formula (+ 20 (gvl :parent :top))))
       (:width ,(o-formula (gvl :parent :shadow-width)))
       (:height ,(o-formula (gvl :parent :shadow-height)))
       (:filling-style ,opal:black-fill))
      (:gray-rect ,opal:rectangle
  (:left ,(o-formula (gvl :parent :left)))
  (:top ,(o-formula (gvl :parent :top)))
  (:width ,(o-formula (gvl :parent :shadow-width)))
  (:height ,(o-formula (gvl :parent :shadow-height)))
  (:filling-style ,opal:gray-fill)))))

(opal:add-components TOP-AGG BUTTON)
(opal:update BUTTON-WIN)
```

After studying the BUTTON schema for a moment, several features stand out. First, there are two slots called `:shadow-width` and `:shadow-height` defined in the top-level schema, which are used by the width and height formulas of the component rectangles.

The presence of these slots at the top-level will make it easier to change the appearance of the button in the future – if we want to make it wider, we only need to change one slot, `:shadow-width`, instead of all the components' `:width` slots.

Next, it should be clear that the formulas in the `:left` and `:top` slots of the components will place the rectangles at the appropriate positions relative to each other, with the shadow further down and to the right. Finally, it is important that the shadow comes first in the order of defining the components. Objects are drawn on the screen in the order that they are added to an aggregate, so we definitely want the gray rectangle to come after the shadow.

Notice that there is no "inheritance" going on in the BUTTON aggregadget. When we want a component to get a value from its parent, we have to explicitly define a constraint that gets that value.

We have just finished the first step in creating a text button. Although there is more code in the aggregadget example than in the previous example with rectangles R1 and R2, the aggregadget code is simple and flexible. Also, almost all of the formulas that you will write in the future will resemble those in this example. The only difference will be the names of the slots and the arithmetic that is appropriate for the situation.

Now we are ready to add more objects to the button. To do this, we will not add objects to BUTTON while it is in the window. Instead, we will remove the old BUTTON aggregadget from the window and write a new one from scratch. Most of the code that we have already written will be reused, however, and if you still have a copy of the previous example on your screen, you will be able to cut-and-paste it. So execute the command that will remove the button from the top-level aggregate:

```
(opal:remove-components TOP-AGG BUTTON)
(opal:update BUTTON-WIN)
```

3.29.3 Defining Parts Using Prototypes

Before constructing an aggregadget with additional components, let's look at another way to define components in aggregadgets. This method will make it easier for us to develop the BUTTON aggregadget by condensing some of the code and eliminating a lot of typing.

In the previous example, the components were instances of rectangles. Another way to define components is to define them as prototypes separate from the aggregadget, and then create instances of those prototypes in the aggregadget `:parts` slot. The following code comes from this alternate method for defining aggregadgets.

```
(create-instance 'SHADOW-PROTO opal:rectangle
   (:left (o-formula (+ 20 (gvl :parent :left))))
   (:top (o-formula (+ 20 (gvl :parent :top))))
   (:width (o-formula (gvl :parent :shadow-width)))
   (:height (o-formula (gvl :parent :shadow-height)))
   (:filling-style opal:black-fill))

(create-instance 'GRAY-PROTO opal:rectangle
   (:left (o-formula (gvl :parent :left)))
   (:top (o-formula (gvl :parent :top)))
   (:width (o-formula (gvl :parent :shadow-width)))
   (:height (o-formula (gvl :parent :shadow-height)))
```

```
         (:filling-style opal:gray-fill))

    (create-instance 'BUTTON opal:aggregadget
       (:left 20) (:top 20)
       (:shadow-width 100) (:shadow-height 50)
       (:string "Button")
       (:parts
        '((:shadow ,SHADOW-PROTO)
          (:gray-rect ,GRAY-PROTO))))


    (opal:add-components TOP-AGG BUTTON)
    (opal:update BUTTON-WIN)
```

Notice that this way of looking at aggregadgets is entirely consistent with the previous aggregadget definition. In the `:parts` slot of our new button, we have created instances just like before, but we have not explicitly defined any slots in the components. It does not matter whether we set slots in the prototype objects or in the parts definitions. Using this abbreviation method for defining aggregadgets, we can now avoid retyping the slot definitions for the old components and move on to talking about new ones.

It should be noted that the SHADOW-PROTO and GRAY-PROTO rectangles can not be added to a window by themselves. If you were to try this, you would get an error when Garnet tried to evaluate any of the formulas that we defined. This is because there is no `:parent` for either the SHADOW-PROTO or the GRAY-PROTO, which is clearly needed by the formulas. But when instances of these rectangles are added to the BUTTON aggregadget, their `:parent` slots are set to be the parent aggregadget.

As usual, remember to remove the current BUTTON from the window using `remove-components`.

3.29.4 The Label of the Button

Referring to Figure [text-buttons], page 431, again, we see that the text button needs a white rectangle to be centered over the gray one, and text should be centered inside the white rectangle. We will want the string of the text object to be a top level slot in the aggregadget so that we can change it easily. Thus, we need to place a constraint in the text object to retrieve it (remember the text object does not "inherit" the string from its parent just because it is a component). Other than that, the addition of the new components to the BUTTON aggregadget is straightforward. Using the abbreviation method for defining aggregadgets, we get the following code (the components SHADOW-PROTO and GRAY-PROTO were defined above).

```
    (create-instance 'WHITE-PROTO opal:rectangle
       (:left (o-formula (+ 7 (gvl :parent :gray-rect :left))))
       (:top (o-formula (+ 7 (gvl :parent :gray-rect :top))))
       (:width (o-formula (- (gvl :parent :gray-rect :width) 14)))
       (:height (o-formula (- (gvl :parent :gray-rect :height) 14)))
       (:filling-style opal:white-fill))

    (create-instance 'TEXT-PROTO opal:text
       (:left (o-formula (+ (gvl :parent :white-rect :left)
```

```
(round (- (gvl :parent :white-rect :width)
  (gvl :width))
      2))))
  (:top (o-formula (+ (gvl :parent :white-rect :top)
      (round (- (gvl :parent :white-rect :height)
 (gvl :height))
     2))))
  (:string (o-formula (gvl :parent :string)))))

(create-instance 'BUTTON opal:aggregadget
   (:left 20) (:top 20)
   (:shadow-width 100) (:shadow-height 50)
   (:string "Button")
   (:parts
    '((:shadow ,SHADOW-PROTO)
      (:gray-rect ,GRAY-PROTO)
      (:white-rect ,WHITE-PROTO)
      (:text ,TEXT-PROTO))))

(opal:add-components TOP-AGG BUTTON)
(opal:update BUTTON-WIN)
```

Although the centering formulas for the `:left` and `:top` slots of the text object are a little more complicated, they are basic calculations that find the proper position of the text based on the dimensions of the white rectangle. Another aspect of the formulas is that they reference not only slots in the parent object, but also slots in their sibling objects. Specifically, in the `white-rect` part, the `:left` formula looks up the aggregate tree to the `:parent`, and then looks down again into the `gray-rect`. The same tracing of the aggregate tree is involved in the `text` formulas for `:left` and `:top`.

3.29.5 Instances of the Button Aggregadget

It should be clear by now that aggregadgets are particularly useful for organizing and defining components. After creating the four prototype objects by themselves, we were able to define BUTTON with the compact aggregadget above. And with our current definition of BUTTON, we will now see another significant use of aggregadgets. The following code creates several instances of the BUTTON aggregadget, which we can use as a prototype. When you add these instances to the window, you see that component rectangles and text are generated automatically for the instances.

```
(create-instance 'BUTTON-1 BUTTON
   (:left 130) (:top 80)
   (:shadow-width 60)
   (:string "abcd"))

(create-instance 'BUTTON-2 BUTTON
   (:left 10) (:top 120)
   (:string "Button-2"))
```

```
(opal:add-components TOP-AGG BUTTON-1 BUTTON-2)
(opal:update BUTTON-WIN)
```
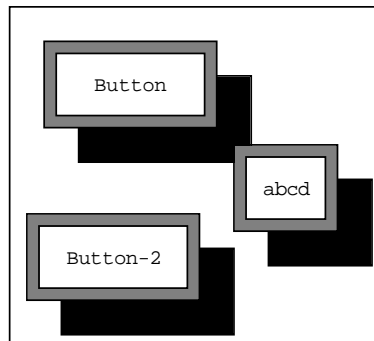

Figure 3.12: The BUTTON object and two instances of it.

This feature of aggregadgets means that you do not need to chapterly create objects individually and add them to the window. Instead, you can create a group of objects, and then create instances of the group.

Before moving to the next section, remember to remove your three button objects from the window with `remove-components`.

3.29.6 Making an Aggrelist of Text Buttons

Even though instances of the aggregadget will automatically generate components, the instances BUTTON-1 and BUTTON-2 show that we still need to chapterly supply coordinates to the aggregadget in order to position it. When we create the finished text button panel in Figure [text-buttons], page 431, we don't want to calculate the position of each text button in the window. (Notice that this is similar to the problem we faced several sections ago – that we didn't want to compute the position of each rectangle within a text button.) The solution to this problem (as before) is to use a special type of aggregate that will generate components for us. This time, we will use an aggrelist.

Just for a start, let's create a simple itemized aggrelist. We will supply an item-prototype and a number to the aggrelist, and it will generate that number of instances of the item-prototype. Specifically, we want the aggrelist to make five copies of the BUTTON aggregadget. So, let's try the following code.

```
(create-instance 'PANEL opal:aggrelist
   (:left 30) (:top 10)
   (:items 5)
   (:item-prototype BUTTON))

(opal:add-components TOP-AGG PANEL)
(opal:update BUTTON-WIN)
```
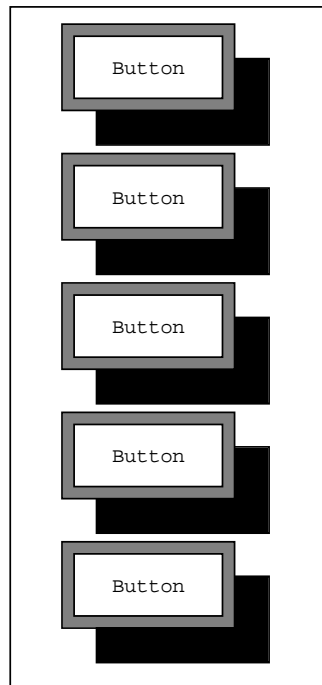
Figure 3.13: An aggrelist with an `:items` value of 5 and the BUTTON aggregadget as its `:item-prototype`.

By supplying the number 5 in the `:items` slot, we tell the aggrelist to make five copies of its item-prototype. And, because this is an aggrelist, all the copies of the prototype are automatically appropriate `:left` and `:top` coordinates. It turns out that we only had to

give the position of the left and top of the aggrelist; all the calculations for the buttons were handled internally. There are many customizable slots for aggrelists that change the appearance of the aggrelist – like whether to make the panel horizontal or vertical, how much space to put between the items, etc. A list of these slots is in the Aggregadgets chapter, which you can try out later.

The next step in the development of our panel is to give each button an appropriate label. To do this, we need to supply a list of labels to the aggrelist. The proper place to do this is in the :items slot. As we just saw, if you give the :items slot a number, the aggrelist generates that number of items. If instead you give it a list, then it will generate the same number of components as there are items in the list. We will also have to change the BUTTON prototype so that its :string slot pays attention to the list of items we supply. The following code makes this change to the BUTTON prototype in the definition of the aggrelist, so we don't have to redefine the BUTTON aggregadget.

```
(create-instance 'PANEL opal:aggrelist
   (:left 30) (:top 10)
   (:items '("Mozart" "Chopin" "Strauss" "Beethoven" "Vivaldi"))
   (:item-prototype
    '(,BUTTON
      (:string ,(o-formula (nth (gvl :rank) (gvl :parent :items)))))))

(opal:add-components TOP-AGG PANEL)
(opal:update BUTTON-WIN)
```

The :rank slot in the :string formula above is put into each component generated by the aggrelist. Even though there is no :rank in our definition of BUTTON, when the aggrelist generates its components, it ranks the objects in the order that they are created and stores these ranks in the :rank slot (ranks start at 0). This makes it easy to find the item in the :items list that corresponds to each component.

Since we are going to be redefining objects again, remember to remove the PANEL object from the window before going on.

3.29.7 Adding an Interactor

We are almost finished with the text button panel. At this point, the panel that we have written is still a passive graphical object – if you press on it with the mouse, it acts just like a pile of rectangles and does nothing at all. Therefore, the next step is to add an interactor that will cause the appearance of the buttons to change whenever we click the mouse on it. Suppose we choose to use a button-interactor for our interface between the mouse and the panel. By applying the principles discussed in the interactor section of this tutorial, we can write the following code for our interactor.

```
(create-instance 'PRESS-PROTO inter:button-interactor
   (:window (o-formula (gvl :operates-on :window)))
   (:start-where (o-formula (list :element-of (gvl :operates-on))))
   (:start-event :leftdown))
```

The code for this interactor is short and simple. It is a prototype object just like the rectangles, but it happens to be an interactor. The :operates-on reference in the formulas is analogous to the :parent slot in objects, and the slot is automatically created when the

interactor is attached to an aggregadget or aggrelist. In interactors, the `:operates-on` slot points to the aggregadget that it is attached to, just like the `:parent` slot of a graphical object points to its aggregate. Notice that we have supplied values for the two required slots in an interactor. The `:window` slot points to the window of the aggregadget that the interactor will be attached to, which is reasonable since we want the interactor to work in the same window that the graphics appear in. The value in the `:start-where` slot tells the interactor to start whenever the mouse is clicked over any component of the aggrelist (that is, over any button).

Before we attach this interactor to the PANEL aggregadget, we are going to have to change a few of the formulas in the button and its components. The question to ask is – how should the graphics change when we press the mouse over the button? By looking at the full text-buttons picture in Figure [text-buttons], page 431, we see that the gray rectangle should move down and to the right, settling over the shadow. Therefore, we will have to change the formulas for the `:left` and `:top` of the gray rectangle. We do not have to change the `:left` and `:top` slots of the white rectangle or text because they are already constrained to the gray rectangle's position.

As you recall from the "Interactors" section of this tutorial, the `button-interactor` sets the `:selected` slot of the object it operates on when it is clicked on with the mouse. Additionally, the interactor will also set the `:interim-selected` slot of the object while the mouse is being held down over it. With this in mind, it would be useful to make the gray rectangle formulas depend on the `:interim-selected` slot of the aggregadget, since the gray rectangle should settle over the black rectangle when the mouse is held down over the button. A new GRAY-PROTO object that will respond to the interactor follows.

```
(create-instance 'GRAY-PROTO opal:rectangle
   (:left (o-formula (if (gvl :parent :interim-selected)
 (gvl :parent :shadow :left)
 (gvl :parent :left))))
   (:top (o-formula (if (gvl :parent :interim-selected)
(gvl :parent :shadow :top)
(gvl :parent :top))))
   (:width (o-formula (gvl :parent :shadow-width)))
   (:height (o-formula (gvl :parent :shadow-height)))
   (:filling-style opal:gray-fill))
```

The new formulas in the `:left` and `:top` of the new GRAY-PROTO now look at the `:interim-selected` slot of the button. When the mouse is clicked over a button, the button's `:interim-selected` slot will be set to T, causing the gray rectangle to be moved down and to the right. When the mouse is released, the `:interim-selected` slot will be set back to `nil`, and the gray rectangle will return to its normal position.

If you refer back to the definitions of the other components, you will see why the gray rectangle is the only component that we had to change. The white rectangle depended on the position of the gray rectangle, and the text was centered inside the white one. Thus, when the gray rectangle's position changed, the change was propagated to all of the dependent formulas, resulting in uniform movement of the components.

There is one final note to make before we can complete the panel. When the gray rectangle is pushed down onto the shadow, the dimensions of the button are going to

change. That is, when the gray rectangle covers the shadow completely, then the button's aggregate has smaller dimensions than if the two rectangles are spread out a bit. If left unchecked, this will cause unexpected behavior because the aggrelist keeps the components arranged according to their width and height. For this reason, we will have to supply our own `:width` and `:height` values to the BUTTON aggregadget within our definition of the PANEL. To see the problem for yourself, you may want to leave out the `:width` and `:height` values in the following code just to see what happens. Then you will certainly want to try the code again with the values in place.

```
(create-instance 'BUTTON opal:aggregadget
   (:left 20) (:top 20)
   (:width 120) (:height 70)  ; The dimensions of the two rectangles plus their offset
   (:shadow-width 100) (:shadow-height 50)
   (:string "Button")
   (:parts
    '((:shadow ,SHADOW-PROTO)
      (:gray-rect ,GRAY-PROTO)
      (:white-rect ,WHITE-PROTO)
      (:text ,TEXT-PROTO))))

(create-instance 'PANEL opal:aggrelist
   (:left 30) (:top 10)
   (:items '("Mozart" "Chopin" "Strauss" "Beethoven" "Vivaldi"))
   (:item-prototype
    '(,BUTTON
      (:string ,(o-formula (nth (gvl :rank) (gvl :parent :items))))))
   (:interactors
    '((:press ,PRESS-PROTO))))

(opal:add-components TOP-AGG PANEL)
(opal:update BUTTON-WIN)

; Required if you are not using CMU, Allegro, Lucid, LispWorks, or MCL
(inter:main-event-loop)  ; To start the interactors.
                         ; Hit F1 in the Garnet window to exit.
```
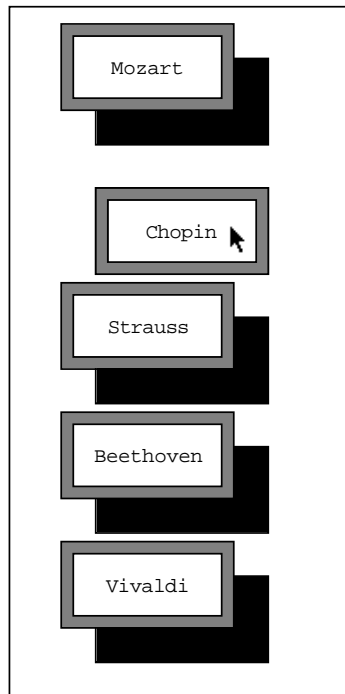
Figure 3.14: The finished text button panel.

Once you have entered the `main-event-loop` (not necessary in CMUCL, Allegro, Lucid, LispWorks, or MCL), you can click on any of the buttons and they will respond. The movement comes from the interactor setting the `:interim-selected` slot of the button that you press on, which causes the `:left` and `:top` slots of the components to be re-

evaluated. When you let go, the `:interim-selected` slot is cleared, and the components return to their original position.

Remember to destroy the window when you are finished with this example.

```
(opal:destroy BUTTON-WIN)
```

3.30 Referencing Objects in Functions

If a function that returns an object is only going to be called once, then it is usually appropriate to explicitly name the objects in each `create-instance` call. This is the method used in the demonstration functions that accompany the gadgets. However, if a function is called repeatedly and returns objects which will be used at the same time, then unnamed objects should probably be used.

For example, the function below will create and return windows with messages in them. If the window in the function was explicitly named (say 'WIN or something), then each call to the function would destroy the previous window instance while creating the new one.

```
(defun Make-Win (left top string)
  ; Create unnamed objects and assign them to local variables
  (let ((win (create-instance NIL inter:interactor-window
(:left left) (:top top)
(:width 100) (:height 100)))
(agg (create-instance NIL opal:aggregate))
(message (create-instance NIL opal:text
    (:left 20) (:top 40)
    (:string string))))
    ; Manipulate the objects according to their local names
    (s-value win :aggregate agg)
    (opal:add-component agg message)
    (opal:update win)
    win))  ; Return the internal name of the new window

(setf Win-List (list (Make-Win 100 100 "Hello")
     (Make-Win 190 120 "Window 2")
     (Make-Win 70 190 "Third Win")))
```

Each time `Make-Win` is called, a window is created, an aggregate is attached, and a text object is added to the aggregate. We kept a list of the internal names of the windows in `Win-List` because we will want to destroy them later. Each of the windows in the list can be manipulated as usual (using `s-value`, etc.) by referring to their generated names.
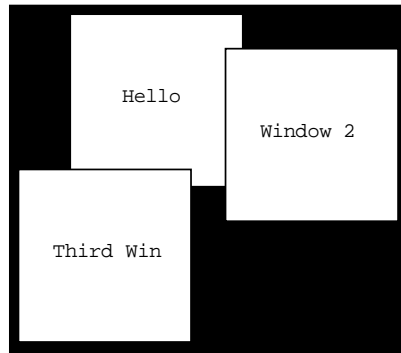
Figure 3.15: Three windows created with the function `Make-Win`.

To clean up the windows generated from `Make-Win`, you could use `dolist` to destroy the whole list, or chapterly destroy the windows individually while referring to their generated names.

3.31 Hints and Caveats

There is a small chapter devoted to optimizations that can be added to your Garnet programs that make them smaller and faster. This section lists a few suggestions that are sometimes **required** by Garnet programs, in addition to helping your programs be more efficient.

3.32 Dimensions of Aggregates

3.32.1 Supply Your Own Formulas to Improve Performance

Although it is usually not necessary to specify the `:width` and `:height` of an aggregate, the programmer can almost always define formulas that will be more efficient than the default formulas for computing the bounding box. The default formulas look at all the components of the aggregate and compute the appropriate bounding box, but they are completely generic and make no assumptions about the arrangement of the components. Since you will know where the components will appear on the screen, you can usually supply simple formulas that depend on only a few of the components.

For example, if you created an aggregadget out of nested rectangles, where there was one outer rectangle and several others inside of it, then you would want to define dimension formulas for the aggregate that depend only on the outer rectangle and ignore the inner ones. Otherwise, the default formulas would check every rectangle before deciding on the correct width and height of the aggregate.

3.32.2 Ignore Feedback Objects in Dimension Formulas

A good reason to define your own formulas for the `:width` and `:height` slots of aggregates is that you usually don't want the feedback object to be considered in the bounding box calculation.

3.32.3 Include All Components in the Aggregate's Bounding Box

Components of aggregates should always be inside the bounding boxes of the aggregates. That is, you should not make the `:left` of an aggregate be 40, and then the left of a component be (- 40 offset). This will put the component outside of the bounding box of the aggregate (too far to the left), and Garnet will not be able to update the aggregate properly.

The solution here is to make the left of the aggregate be the same as the leftmost component, and then make the component inherit that left. Of course, if you have several components which all have different lefts, then you will have to add offsets to the lefts of the other components.

3.33 Dimensions of Windows

Don't make the size of windows depend on the size of the objects inside. This will lead to frequent refreshing of the entire window, causing very poor performance.

3.34 Formulas

3.34.1 The Difference Between formula and o-formula

The difference between `formula` and `o-formula` is somewhat confusing. The preferred form is `(o-formula (expression))` because the expression will be compiled when the the file is compiled. Then, at run-time, the expression for the constraint executes as compiled code when the formula needs to be re-evaluated. (This works by expanding the code in-line to create a lambda expression, for which the compiler generates code.) When the `(formula '(expression))` form is used, the expression is interpreted at run-time, so the constraint executes much slower.

The disadvantage of `o-formula`, however, is that because it is a macro, variable references do not create lexical closures. This means that variables referenced inside an `o-formula` will not be expanded into their actual value inside the expression. The variable name will instead remain inside the expression, and if its value ever changes, the new value will be reflected when the expression is reevaluated.

On the other hand, using the form `(formula '( ... ,*variable* ...))` puts the value of `*variable*` permanently in the formula, and eliminates the reference to `*variable*`. If all your object references use `(gvl ...)` to get values out of slots of the object ("paths" in aggregadgets), then this is not relevant, and you should use `o-formula`.

As an example, let's start with a global variable and two formulas that use the variable. One formula will be an `o-formula`, and one will be a plain `formula`. Note: `lisp>` represents the prompt for the lisp listener.

```
lisp> (setf *width* 100)
100
lisp> (create-schema 'A
 (:left 10)
        (:right1 (o-formula (+ (gvl :left) *width*)))
 (:right2 (formula '(+ (gvl :left) ,*width*))))
Object A

#k<A>
lisp> (gv A :right1)
110
lisp> (gv A :right2)
110
lisp>
```

So in both cases the formula computed the sum of the left and the current value of `*width*`. Now, what happens if we change `*width*`? At first, it seems that nothing happens. Just changing the value of the variable will not cause the formulas to recompute – only things that are `gv`'ed have dependencies, and Garnet doesn't know that the variable's value has changed yet.

```
lisp> (setf *width* 22)
22
lisp> (gv A :right1)
110
lisp> (gv A :right2)
110
```

lisp>

But now let's change the value of the `:left` slot, which will invalidate both formulas and will cause them to recompute.

```
lisp> (s-value A :left 33)
33
lisp> (gv A :right1)
55
lisp> (gv A :right2)
133
lisp>
```

Now they recomputed, and the difference is obvious. In the `o-formula`, the `*width*` variable reference was still hanging around, and that expression used the current value of `*width*`. A `ps` of the `o-formula` shows it's still there:

```
lisp> (ps (get-value A :right1))
#k<F74>
  lambda:        (+ (gvl :left) *width*)
  cached value:  (55 . T)
  on schema #k<A>, slot :RIGHT1


NIL
lisp>
```

On the other hand, the plain formula got rid of the `*width*` variable when the "," dereferenced it.

```
lisp> (ps (get-value A :right2))
#k<F73>
  lambda:        (+ (gvl :left) 100)
  cached value:  (133 . T)
  on schema #k<A>, slot :RIGHT2


NIL
lisp>
```

Notice the 100 replaced `*width*` in the definition of the formula.

One occasion where this distinction between `formula` and `o-formula` comes up is the creation of objects while iterating over a list. The following code correctly dereferences the variable `obj` when constructing formulas.

```
(dolist (obj objlist)
  (create-instance NIL opal:rectangle
    (:left (formula `(gv ,obj :left)))
  ))
```

3.34.2 Avoid Real Number Divide

In all graphical objects, the position and dimension slots `:left`, `:top`, `:width`, and `:height` all take integer values. Therefore, the integer divide functions `round`, `floor`, and `ceiling`, etc. should be used more frequently than `/` for division.

3.35 Feedback Objects

If all of the components of an aggregate are selectable, then any feedback object should be put in a separate aggregate so that the feedback object itself is not selectable.

3.36 Debugging

The Debugging Tools Reference chapter documents many functions that are useful in answering the most common questions that users have when developing Garnet code. The functions will help you find objects, explain the values of particular slots, describe inheritance and aggregate hierarchies, and inspect constraints and interactors. This section describes the most commonly used functions for examining Garnet objects.

3.37 The Inspector

There is a powerful debugging 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.

You can easily try this tool if you have any Garnet window with objects in it. Sections [prototypes], page 44-[inspector-sec], page 48, of this tutorial provide a simple example window with step-by-step interaction with the Inspector.

3.38 PS

`kr:PS` *object*

The function `ps` (which stands for "print schema") is used to examine individual schemas. When `ps` is called with a Garnet object, a list of all the object's slots and values will be printed. By default, any slot whose value is inherited from a prototype is not printed unless `gv` has already been called on that slot.

The `ps` function resides in the KR package, and is fully documented in the KR chapter. There are several switches and global variables that control the amount of information that `ps` prints.

3.39 Flash

`garnet-debug:flash` *object* **&optional** *n* [Function]

The function `flash` helps you to visually locate *object* in a window by flashing the bounding box of *object* from black to white *n* times. The *object* must already be in a window in order for it to flash. If `flash` is unable to flash the object, then the function will try to give you some explanation of why the object will not flash.

3.40 Ident

`garnet-debug:ident`   [Function]

> The function `ident` takes no parameters. After you call `ident`, Garnet waits for the next input event in any Garnet window, like clicking the mouse. If you click over an object, then the name of the object will be printed along with some other information about the object and the window.
>
> Clearly, this function is useful if there are many objects in a window and you forget the names of all of them. A more interesting application is when there are unnamed objects in the window (that is, they were given NIL for a name in their schema definition and now have only internal names) and you want to analyze or manipulate an unnamed object. Then, `ident` will return the internal name of the object clicked on, and that name can be used in `gv` or `s-value` calls as usual.

3.41 Trace-Inter

> `inter:Trace-Inter &optional` *interactor*`[No value for ''[''']function]`
> `inter:Untrace-Inter[No value for ''[''']function]`

The function `trace-inter` is often used to find out why an interactor is not working as you expected. Interactor problems most often arise from improper definitions of either the interactors or the objects they work on. Using `trace-inter` can help to narrow the reasons for the unexpected behavior.

Executing `untrace-inter` will turn off the tracing for interactors.

4 KR: Constraint-Based Knowledge Representation

KR is a very efficient knowledge representation language implemented in Common Lisp. It provides powerful frame-based knowledge representation with user-defined inheritance and relations, and an integrated object-oriented programming system. In addition, the system supports a constraint maintenance mechanism which allows any value to be computed from a combination of other values. KR is simple and compact and does not include some of the more complex functionality often found in other knowledge representation systems. Because of its simplicity, however, it is highly optimized and offers good performance. These qualities make it suitable for many applications that require a mixture of good performance and flexible knowledge representation.

4.0.1 KR: Introduction

This document is the reference manual for the KR system. KR implements objects, also known as *schemata*, which can contain any amount of information and which can be connected in arbitrary ways. All Garnet objects are implemented as KR schemata. KR *kr* can also be used as a very efficient frame-based representation system. Simplicity and efficiency are its main design goals and differentiate it sharply from more conventional frame systems, as discussed in *KR-KER*.

In addition to the basic representation of knowledge as a network of schemata, KR provides object-oriented programming and an integrated constraint maintenance system. Constraint maintenance is implemented through *(*formulas*)*, which constrain certain values to combinations of other values. The constraint system is closely integrated with the basic object system and is part of the same program interface.

Close integration between objects and constraint maintenance yields several advantages. First of all, constraint maintenance is seen as a natural extension of object representation; the same access functions work on regular values and on values constrained by a formula. Second, the full power of the representation language is available in the specification of constraints. Third, since the two mechanisms are integrated at a fairly low level, the constraint maintenance system offers very good performance. These advantages make the KR constraint maintenance system a practical tool for the development of applications that require flexibility, expressive power, and performance comparable to that obtained with conventional data structures.

In addition to being one of the building blocks of the Garnet project, KR can be used as a self-contained knowledge representation system. Besides Garnet, KR is used in the Chinese Tutor *CHINESE-TUTOR CHINESE-TUTOR-SHANGHAI*, an intelligent tutoring system designed to teach Chinese to English speakers, and in speech understanding research *MINDS* currently underway at Carnegie Mellon.

This document describes version 3.0.0+ of KR, which is part of release 3.0.0+ of the Garnet system. Several aspects of this version differ from previous versions of the system, such as the ones described in previous reports *KRTR2 KR*. The present document overrides all previous descriptions.

The orientation of this manual is for users of KR as an object system. Users who are more interested in using KR as a knowledge representation system should consult a previous paper *kr-ker*. This manual begins with a description of the features of the system

that beginners are most likely to need. Some of the less common features are only presented near the end of the document, in order to avoid obscuring the description with irrelevant details. Sections 6 and 7 contain the detailed description of the program interface of KR. This is a complete description of the system and its features. Most application programs will only need a small number of features, described in section 6.

4.1 Structure of the System

KR is an object system implemented in Common Lisp *CommonLisp*. It includes three closely integrated components: *object-oriented* programming), *constraint maintenance*, and *knowledge representation*.

The first component of KR is an object oriented programming system based on the prototype-instance paradigm. Schemata can be used as objects, and inheritance can be used to determine their properties and behavior. Objects can be sent *messages*, which are implemented as procedural attachments to certain slots; messages are inherited through the same mechanism as values.

Instead of the class-instance paradigm, common in object-oriented programming languages, KR implements the more flexible prototype-instance paradigm *liebermanprototypes*, which allows properties of instances to be determined dynamically by their prototypes. This means that the class structure of a system can be modified dynamically as needed, without any need for recompilation.

The second component of KR implements constraint maintenance. Constraint maintenance is implemented through *formulas*, which may be attached to slots and determine their values based on the values of other slots in the system.

Constraint maintenance is closely integrated with the other components. The user, for example, does not need to know which slots in a schema contain ordinary values and which ones are constrained by a formula, since the same access primitives may be used in both cases.

The third component, frame-based knowledge representation, stores knowledge as a network of chunks of information. Networks in KR are built out of unstructured chunks, i.e., *schemata*. Each schema can store arbitrary pieces of information, and is not restricted to a particular format or data structure. Information is encoded via attribute-value pairs.

Values in a schema can be interpreted as links to other schemata. This enables the system to support complex network structures, which can be freely extended and modified by application programs. KR provides simple ways to specify the structure of a network and the relationship among its components.

4.2 Basic Concepts

This section describes the basic elements of KR, i.e., objects. More details about the design philosophy of the system and some of the internal implementation may be found in *KR*, which describes a previous version of the system that did not support constraint maintenance.

4.2.1 Main Concepts: Schema, Slot, Value

An object in KR is known as a *schema*. A schema is the basic unit of representation and consists of an optional *name*, a set of *slots*, and a *value* for each slot. The user can assemble

networks of schemata by placing a schema as the value in a slot of another schema; this causes the two schemata to become linked.

A schema may be namedor unnamed. Named schemata are readily accessible and are most useful for interactive situations or as the top levels of a hierarchy, since their names act as global handles. Unnamed schemata do not have meaningful external names. They are, however, more compact than named schemata and account for the vast majority of schemata created by most applications. Unnamed schemata are automatically garbage-collected when no longer needed, whereas named schemata have to be destroyed explicitly by the user.

The name of a named schema is a symbol. When a named schema is created, KR automatically creates a special variable by the same name and assigns the schema itself as the value of the special variable. This makes named schemata convenient to use.

A schema may have any number of *slots*, which are simply attribute-value pairs. The slot name indicates the attribute name; the slot value (if there is one) indicates its value. Slot names are keywords, and thus always begin with a colon. All slots in a schema must have distinct names, but different schemata may very well have slots with the same name.

Each slot may contain only one value. A value is the actual data item stored in the schema, and may be of any Lisp type. KR provides functions to add, delete, and retrieve the value from a given slot in a schema.

The printed representation of a schema shows the schema name followed by slot/value pairs, each on a separate line. The whole schema is surrounded by curly braces. For example,

```
#k<fido>
  :owner =  #k<john>
  :color =  #k<brown>
  :age =  5
```

The schema is named FIDO and contains three slots named `:owner`, `:color`, and `:age`. The slot `:age` contains one value, the integer 5.

The default printed name of a schema is of the form `#k<name>`, where *name* is the actual name of the schema. This representation makes it very easy to distinguish KR schemata from other objects. Note, however, that this convention is only used when printing, and is not used when typing the name of a schema.

In order to illustrate the main features of the system, we will repeatedly use a few schemata. We present the definition of those schemata at this point and will later refer to them as needed. These schemata might be part of some graphical package, and are used here purely for explanation purposes. In practice, there is no need to define such schemata in a Garnet application, since the Opal component of Garnet (see the Opal manual) already provides a complete graphical object system.

The following KR code is the complete definition of the example schemata:

```
(create-instance 'my-graphical-object nil
  (:color :blue))

(create-instance 'box-object my-graphical-object
  (:thickness 1))

(create-instance 'rectangle-1 box-object
  (:x 10)
  (:y 20))

(create-instance 'rectangle-2 box-object
  (:x 34)
  (:y (o-formula (+ (gvl :left-obj :y) 15)))
  (:left-obj rectangle-1))
```

The exact meaning of the expressions above will become clear after we describe the functional interface of the system. Briefly, however, the example can be summarized as follows. The schema `my-graphical-object` is at the top of a hierarchy of graphical objects. The schema `box-object` represents an intermediate level in the hierarchy, and describes the general features of all graphical objects which are rectangular boxes. `box-object` is placed below `my-graphical-object` in the hierarchy, and its `:is-a` slot points to the schema `my-graphical-object`. This is done automatically by the macro create-instance.

Finally, two rectangles (`rectangle-1` and `rectangle-2`) are created and placed below `box-object` in the hierarchy. `rectangle-1` defines the values of the two slots `:x` and `:y` directly, whereas `rectangle-2` uses a formula for its `:y` slot. The formula states that the value of `:y` is constrained to be the `:y` value of another schema plus 15. The other schema can be located by following the `:left-obj` slot of `rectangle-2`, as specified in the formula, and initially corresponds to `rectangle-1`.

Figure 5.1 shows the four schemata after the definitions above have been executed. Relations are indicated by an arrow going from a schema to the ones to which it is related.
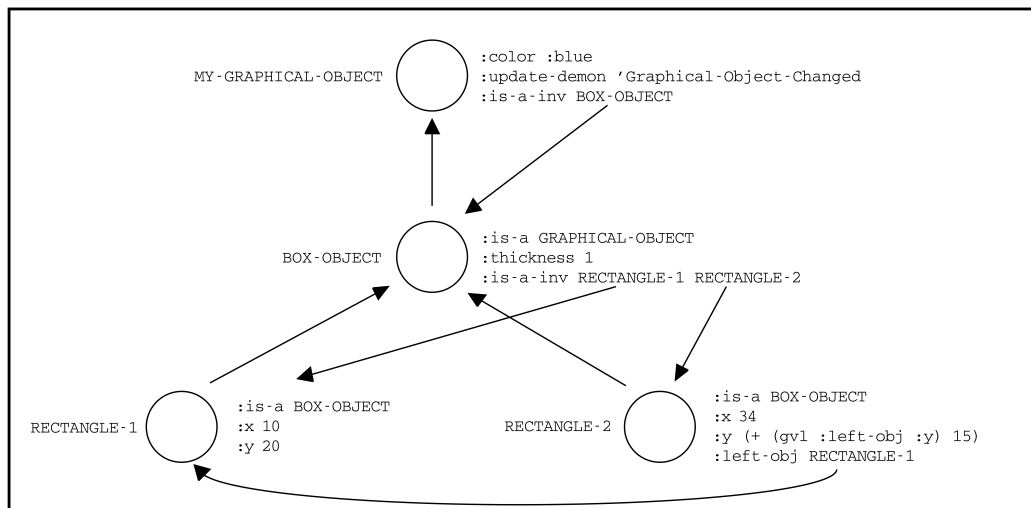
Figure 4.1: The resulting network of schemata

Asking the system to print out the current status of schema `rectangle-2` would produce the following output:

```
#k<RECTANGLE-2>
  :IS-A =  #k<BOX-OBJECT>
  :LEFT-OBJ =  #k<RECTANGLE-1>
  :Y =  #k<F2289>(NIL . NIL)
  :X =  34
```

Note that slot `:y` contains a formula, which is printed as `#k<F2289>(NIL . NIL)`. This is simply an internal representation for the formula and will yield the correct value of `:y` when needed.

4.2.2 Inheritance

The primary function of values is to provide information about the object represented by a schema. In the previous example, for instance, asking the system for the `:x` value of `rectangle-1` would simply return the value `10`.

Values can also perform another function, however: They can establish *connections between schemata*. Consider the `:left-obj` slot in the example above: if we interpret `rectangle-1` as a schema name, then the slot tells us that the schema `rectangle-2` is somehow related to the schema `rectangle-1`. Graphically, this will mean that the position of `rectangle-2` is partially determined by that of `rectangle-1`.

KR also makes it possible to use values to perform *inheritance*, i.e., to control the way information is inherited by a particular schema from other schemata to which it is connected. Inheritance allows information to be arranged in a hierarchical fashion, with lower-level schemata inheriting most of their general features from higher-level nodes and possibly providing local refinements or modifications. A connection that enables inheritance

of values is called an *inheritance relation* . Inheritance relations always contain a list of values; in many cases, this is a list of only one value.

The most common example of inheritance is provided by the `:is-a` relation . If schema A is connected to schema B by the `:is-a` relation,[1] then values that are not present in A may be inherited from B.

Consider the schema `rectangle-1` in our example. If we were to ask "What is the color of `rectangle-1`?", we would not be able to find the answer by just looking at the schema itself. But since we stated that `rectangle-1` is a box object, which is itself a graphical object, the value can be inherited from the schema `my-graphical-object` through two levels of `:is-a`. The answer would thus be `"rectangle-1` is blue." Inheritance is possible in this case because the slot `:is-a` is pre-defined by the system as a relation.

4.3 Object-Oriented Programming

This section describes the object-oriented programming component of KR. This component entails two concepts: the concept of message sending, and the concept of prototype/instance.

4.3.1 Objects

The fundamental data structure in KR is the *schema*, which is equivalently referred to as an *object.* Objects consist of data (represented by values in slots) and methods (represented by procedural attachments, again stored as values in slots). Methods are similar to functions, except that a method can do something different depending on the object that it is called on. A procedural attachment is invoked by "sending a message" to an object; this means that a method by the appropriate name is sought and executed. Different objects often provide different methods by the same name, and thus respond to the same message by performing different actions.

The data and methods associated with an object can be either stored within the object or inherited. This allows the behavior of objects to be built up from that of other objects. The object-oriented component of KR allows some combination of methods , since a method is allowed to invoke the corresponding method from an ancestor schema and to explicitly refer to the object which is handling the message. Method combination, however, is not as developed as in full-fledged object-oriented programming systems such as CLOS *CLOS-X3J13.*

4.3.2 Prototypes vs. Classes

The notion of *prototype* in KR is superficially similar to that of *class* in conventional object-oriented programming languages, since a prototype object can be used to partially determine the behavior of other objects (its *instances*) . A prototype, however, plays a less restricting role than a class. Unlike classes, prototypes simply provide a place from which the values of certain slots may be inherited. The number and types of slots which actually appear in an instance is not in any way restricted by the prototype . The same is true for methods, which are simply represented as values in a slot.

Prototypes in KR serve two specific functions: they provide an initialization method , and they provide default constraints . When a KR schema is created via the function

---

[1] In other words, if schema B appears as a value in the `:is-a` slot of schema A.

`create-instance` , and its prototype has an `:initialize` method, the method is invoked on the instance itself. This results in a uniform mechanism for handling object-dependent initialization tasks.

4.3.3 Inheritance of Formulas

If a prototype provides a constraint for a certain slot, and the slot is not explicitly redefined in an instance, the formula which implements the constraint is copied down and installed in the instance itself. The formula, however, is not actually copied down until a value is requested for that slot (e.g., when `gv` is used). This is a convenient mechanism through which a prototype may partially determine the behavior of its instances. Note that this behavior can be overridden both at instance-creation time (by explicitly specifying values for the instance) and at any later point in time.

4.4 Constraint Maintenance

This section describes the constraint maintenance component of KR. The purpose of constraint maintenance is to ensure that changes to a schema are automatically propagated to other schemata which depend on it.

4.4.1 Value Propagation

The KR constraint system offers two distinct mechanisms to cause changes in a part of network to propagate to other parts of the network. The first mechanism, *value propagation*, ensures that the network is kept in a consistent state after a change. The second mechanism, *demon invocation*, allows certain actions to be triggered when parts of a network are modified. Demons are described in section [demons], page 131.

Value propagation is based on the notion of *dependency* of a value on another. Value dependencies are embodied in formulas. Whenever a value in a slot is changed, all slots whose values depend on it are immediately invalidated, although not necessarily re-evaluated. This strategy, known as lazy evaluation, does not immediately recompute the values in the dependent slots, and thus it typically does less work than an eager re-evaluation strategy. The system simply guarantees that correct values are recomputed when actually needed.

4.4.2 Formulas

Formulas represent one-directional connections between a *dependent value* and any number of *depended values*. Formulas specify an expression which computes the dependent value based upon the depended values, as well as a permanent dependency which causes the dependent value to be recomputed whenever any of the other values change.

Formulas can contain arbitrary Lisp expressions, which generally reference at least one particular depended value. The Lisp expression is used to recompute the value of the formula whenever a change in one of the depended values makes it necessary.

Formulas are not recomputed immediately when one of the depended values changes. This reduces the amount of unnecessary computation. Moreover, formulas are not recomputed every time their value is accessed. Each formula, instead, keeps a cache of the last value it computed. Unless the formula is marked invalid, and thus needs to be recomputed, the cached value is simply reused. This factor causes a dramatic improvement in the performance of the constraint maintenance system, since under ordinary circumstances the rate of change is low and most changes are local in nature.
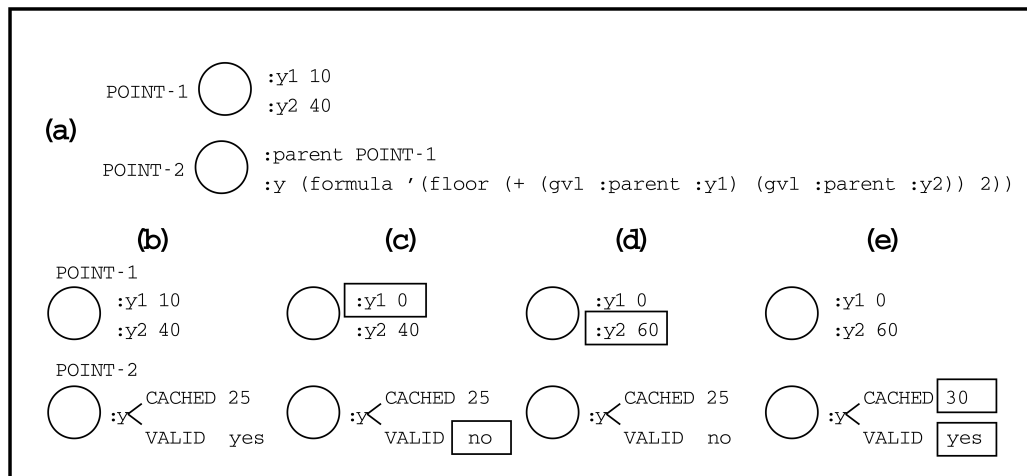
Figure 4.2: Successive changes in depended values

Figure 14.25, part **(a)**, shows an example of a formula installed on slot `:y` of schema `point-2`. The formula depends on two values, i.e., the value of slots `:y1` and `:y2` in schema `point-1`. The formula specifies that slot `:y` is constrained to be the sum of the two values divided by 2, i.e., the average of the two values. Figure 5.1, part **(b)**, shows the internal state of the formula in a steady-state situation where the formula contains a valid cached value. Under these circumstances, any request for the value of slot `:y` would simply return the cached value, without recomputing the formula.

Parts **(c)** and **(d)** show the effects of changes to the depended values. Changes are illustrated by small rectangles surrounding the modified information. The first change is to slot `:y1` and causes the value in the formula to be marked invalid. Note that the formula is not actually recomputed at this point, and the cached value is left untouched. The second change is to slot `:y2` and does not cause any action to occur, since the formula is already marked invalid.

Finally, part **(e)** shows what happens when the value in slot `:y` is eventually needed. The value of the formula is recomputed and again cached locally; the cache is marked as valid. The system is then back to steady state. Note that the formula was recomputed only once, when needed, rather than eagerly after each value changed.

4.4.3 Circular Dependencies

Constraints may involve circular chains of dependency. Slot A, for instance, might depend on slot B, which in turn depends on slot A; see section [degrees], page 143, for an example of a situation where this arises fairly naturally. Circular dependencies may also be used to provide a limited emulation of two-way constraint maintenance.

KR is able to deal with circular dependencies without any trouble. This is handled during formula evaluation; if a formula is evaluated and requests a value which depends of the formula itself, the cycle is broken and the cached value of the formula is used instead. This algorithm guarantees that the network is left in a consistent state, even though the final result may of course depend on where evaluation started from.

4.4.4 Dependency Paths

Typical formulas contain embedded references to other values and schemata. The formula in Figure [formulas], page 104, for example, contains an indirect reference to schema `point-1` through the contents of the `:other` slot. Such references are known as dependency paths . Whenever a formula is evaluated, its dependency paths are used to recompute the updated value.

It is possible for a dependency path to become temporarily unavailable. This would happen, for instance, if schema `point-1` in Figure [formulas], page 104, was deleted, or if slot `:other` in schema `point-2` was temporarily set to `nil`. KR handles such situations automatically. If a formula needs to be evaluated but one of its dependency paths is broken, the current cached value of the formula is simply reused. This makes it completely safe to modify schemata that happen to be involved in a dependency path, since the system handles the situation gracefully.

4.4.5 Constraints and Multiple Values

Unlike earlier versions of KR, version 3.0.0+ supports constraints on multiple values in a slot. The functional interface, however, is not complete and therefore certain operations are not fully supported at the time of this writing. Functions which support constraints on multiple values are easily identified because they accept a *position* parameter which determines what value is affected.

The interaction between constraints and multiple values will be completely specified in future versions of KR. For the time being, most applications should simply be aware that constraints on the first value in a slot are supported universally, whereas some of the functionality may be unavailable for constraints on values other than the first one.

4.5 Functional Interface: Common Functions

This section contains a list of the more common functions and macros exported by the KR interface. It includes the functionality that most Garnet users are likely to need and covers schema representation, object-oriented programming, and constraint maintenance. Section [additional-functions], page 119, describes parts of the system that are less commonly used.

All functions and variables are defined and exported by the KR package. The easiest way to make them accessible to an application program is to execute the following line:
`(use-package :kr)`

Throughout this and the following section, we will use the schemata defined in section [kr-examples], page 99, as examples. All examples assume the initial state described there.

4.5.1 Schema Manipulation

This group includes functions that create, modify, and delete whole schemata.

`kr:create-instance` *object-name prototype* **&rest** *slot-definitions*   [Macro]
      This macro creates an instance of the *prototype* with certain slots filled in; if *prototype* is `nil`, the instance will have no prototype. The instance is named *object-name*. If *object-name* is `nil`, an unnamed object is created and returned. If *object-name* is a symbol, a special variable by that name is created and bound to the new object.

The *slot-definitions*, if present, are used to create initial slots and values for the object. Each slot definition should be a list whose first element is the name of a slot, and whose second element is a value for that slot.

In addition to this basic slot-filling behavior, this macro also performs three operations that are connected to inheritance and constraint maintenance. First of all, create-instance links the newly created object to the *prototype* via the `:is-a` link, thus making it an instance.

Second, if the *prototype* contains any slot with a formula, and the *slot-definitions* do not redefine that slot, create-instance copies the formula down into the instance. This means that the *prototype* can conveniently provide default formulas for any slot that is not explicitly defined by its instances.

Third, if either the *prototype* or the object itself defines the `:initialize` method , create-instance sends the newly created object the `:initialize` message. This is done after all other operations have been completed, and provides an automatic way to perform object-dependent initializations.

Example:

```
(create-instance 'rectangle-4 box-object (:x 14) (:y 15))
```

The following example demonstrates the use of the `:initialize` method at the prototype level[2]:

```
(define-method :initialize box-object (schema)
   (s-value schema :color :red)
   (format t "~s initialized~%" schema))


(create-instance 'rectangle-4 box-object (:x 14) (:y 15))
;; prints out:
#k<RECTANGLE-4> initialized
```

Create-instance understands the `:override` keyword and the `:name-prefix` keyword; see [create-options], page 137, for more details. The uniform declaration syntax with the `:declare` keyword is used to define "local only slots", constant slots, and many others (see section [uniform-syntax], page 120).

kr:ps *schema*   [Function]
This function prints the contents of the *schema*. In its simplest form, described here, the function is called with the *schema* as its sole argument, and prints out the contents of the schema in a standard format. Optional arguments also allow you to control precisely what is printed out; the more complicated form is described in section [print-control-slots], page 139.

The following example shows the simple form of `ps`:

```
(ps RECTANGLE-1)
;; prints out:
#k<RECTANGLE-1>
```

---

[2] Defining methods on Garnet objects is seldom necessary in practice, since real Opal prototypes already have built-in `:initialize` methods.

```
          :Y =  20
          :X =  10
          :IS-A =  #k<BOX-OBJECT>
```

kr:schema-p *thing*   [Function]
>   This predicate returns `t` if *thing* is a valid KR schema, `nil` otherwise. It returns `nil`
>   if *thing* is a destroyed schema. It also returns `nil` if *thing* is a formula.
>
>   (schema-p rectangle-1) ==> t
>   (schema-p 'random) ==> nil

kr:is-a-p *schema thing*   [Function]
>   This predicate returns `T` if *schema* is related to *thing* (another schema) via the `:is-a`
>   relation, either directly or through an inheritance chain. It returns `nil` otherwise.
>
>   Note that *thing* may have the special value `T`, which is used as a "super-class" indica-
>   tor; in this case, `is-a-p` returns `T` if *schema* is any schema. If the *schema* is identical
>   to the *thing*, `is-a-p` also returns `T`.
>   Examples:
>
>   (is-a-p rectangle-1 box-object) ==> t
>   (is-a-p rectangle-1 my-graphical-object) ==> t
>   (is-a-p rectangle-1 rectangle-2) ==> nil
>   (is-a-p rectangle-1 t) ==> t

4.5.2 Slot and Value Manipulation Functions

This group includes the most commonly used KR functions, i.e., the ones that retrieve or
modify the value in a slot. This section presents KR value manipulation functions that deal
with constraints. A different set of primitive functions, which do not deal with constraints,
is described in Section [additional-functions], page 119.

4.5.3 Getting Values with `g-value` and `gv`

When called outside of a formula, `g-value` and `gv` behave identically. When used inside
a formula, the function `gv` not only returns the value of a slot, but also establishes a
dependency for the formula on the slot. This special property of `gv` is discussed in section
[gv-in-form], page 110.

   Novice Garnet users only need to learn about `gv`, but `g-value` is supplied for the rare
case in which you want to retrieve a slot value from inside a formula without establishing
a dependency.

kr:gv *object* **&rest** *slot-names*   [Macro]
kr:g-value *object* **&rest** *slot-names*   [Macro]
>   These macros return the value in a slot of an object. If the slot is empty or not
>   present, they return `nil`. Inheritance may be used when looking for a value. `G-value`
>   and `gv` handle constraints properly: If a formula is currently installed in the slot, the
>   value is computed (if needed) and returned. `G-value` will work in place of `gv` in any
>   of the following examples:
>
>   (gv rectangle-1 :is-a) ==> #k<BOX-OBJECT>

```
(gv rectangle-1 :thickness) ==> 1    ; inherited
(gv rectangle-1 :color) ==> :BLUE
(gv rectangle-2 :y) ==> 35           ; computed formula
```

Although it is common to call `g-value` and `gv` with only one slot name, these macros may actually be given any number of *slot-names*. The macros expand into repeated calls to `g-value` and `gv`, where each slot is used to retrieve another object. The given slot in the final object (which is, in general, different from the *object*) is then accessed. For example:

```
(gv rectangle-2 :left-obj :x)
```

is equivalent to

```
(gv (gv rectangle-2 :left-obj) :x)
```

Both expressions return the value of the `:x` slot of the object which is contained in the `:left-obj` slot of `rectangle-2`. One can think of the slot `:left-obj` as providing the name of the place from which the next slot can be accessed. Such a slot is often called a *link*, since it provides a link to another object which can be used to compute values.

4.5.4 Setting Values with S-Value

`kr:s-value` *object slot more-slots value*   [Function]

This function is used to set a slot with a given value or formula. The *slot* in the *object* is set to contain the *value*. Like with `g-value` and `gv`, `s-value` can be given multiple slots in argument list, when the slot to be set is several levels away from *object*. In the normal case, *value* is an ordinary LISP value and simply supersedes any previous value in the slot. If *value* is a formula (i.e. the result of a call to `o-formula` or `formula`), the formula is installed in the *slot* and internal bookkeeping information is set up appropriately.

If the *slot* already contains a formula, the following two cases arise. If *value* is also a formula, the old formula is replaced and any dependencies are removed. If *value* is not a formula, the old formula is kept in place, but the new *value* is used as its new, temporary cached value. This means that the *slot* will keep the *value* until such time as the old formula needs to be re-evaluated (because some of the values on which it depends are modified).

`s-value` returns the new value of the *slot*.

Note that a `setf` form is defined for `gv` and `g-value`, and expands into `s-value`. This allows a variety of LISP constructs to be used in combination with `gv` and `g-value`, such as the idiom `(incf (gv object slot))` which increments the value of a slot in the object. For example,

```
;; Change value in depended slot from 20 to 21
(incf (gv rectangle-1 :y))
;; The constraint is propagated to RECTANGLE-2:
(gv rectangle-2 :y) ==> 36           ; recomputed
```

Constraint propagation is fully enforced during this operation, just as it would be in the equivalent expression

```
(s-value rectangle-1 :y (1+ (gv rectangle-1 :y)))
```

4.5.5 `formula` and `o-formula`

kr:o-formula *form* **&optional** *initial-value* [Macro]

Given a *form*, this macro returns a formula (formulas are internally represented by special structures). The *form* typically consists of Lisp expressions and `gv` or `gvl` references (see below).

Examples:

```
(o-formula (gvl :above-gadget :x))
(o-formula (min (gvl :above-gadget :x)
                (+ (gvl :other-gadget :width) 10)))
```

The first example creates a formula which causes the slot on which it is installed to have the same value as slot `:x` of the object contained in slot `:above-gadget` of the current object. The second formula is more complex and constrains the slot on which it is installed to have as its value the minimum of two values. One value is computed as before, and the other is computed by adding 10 to the `:width` slot of the object contained in slot `:other-gadget` of the current object.

The *form* can also be an existing formula, rather than a Lisp expression. In this case, the new formula is linked to the existing formula, and inherits the expression from it. No local state is shared by the two formulas. This form of the call should be used as often as possible, since inherited formulas are smaller and more efficient than top-level formulas. An illustration of this case is given by the second call in the following example, which creates a new formula that inherits its expression from the first one:

```
(setf f (o-formula (+ (gvl :above :y)
                      (floor (gvl :above :height) 2))))
(setf g (o-formula f))
```

If an *initial-value* is specified, it is used as the initial cached value for the formula. This cached value is recorded in the formula but marked invalid, and thus it is never used under normal circumstances. The initial value is only used if the formula is part of a circular dependency, or if one of its dependency paths is invalid. Most applications need not be concerned about this feature.

A reader macro has been defined to abbreviate the definition of o-formulas. Instead of typing `(o-formula (...))`, you could type `#f(...)`, 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)))
```

kr:formula *form* **&optional** *initial-value* [Function]

Given a *form*, this function returns a formula. It is similar to `o-formula`, except that the code in *form* is not compiled until run-time, when the `formula` call is actually executed.

Code that can be compiled early should use `o-formula`, which yields more efficient formula evaluation and reduces the amount of storage. `Formula` might be required when local variables are used in *form*, and are not set until run-time. See the "Hints

and Caveats" section of the Tutorial for more discussion of when a formula created with `formula` might be needed.

kr:formula-p *thing*   [Macro]

    a predicate that returns `t` if the *thing* (any Lisp object) is a formula created with `o-formula` or `formula`, `nil` otherwise.

4.5.6 `gv` and `gvl` in Formulas

kr:gv *object* **&rest** *slot-names*   [Macro]

    This macro, which we saw in section [g-value-and-gv], page 107, serves a special purpose when used within formulas.

    In addition to returning a value like `g-value`, `gv` records the dependency path and ensures that the formula in which it is embedded is recomputed whenever the dependency path or the value changes.

    Note that the *object* can be any object, not just the one on which the formula containing `gv` is installed. Specifying the reserved name `:self` for *object* ensures that the path starts from the object on which the formula is installed. This can be achieved more simply via `gvl`, as explained below.

    The following examples show how to use `gv` within formulas:

```
(o-formula (gv rectangle-1 :y))
(o-formula (+ (gv :self :x) 15))
(o-formula (equal (gv :self :other :other :color)
                  (gv :self :color)))
```

    As a special case, the expression (`gv :self`) (without any slot name) may be used within a formula to refer to the object to which the formula is attached. This is sometimes useful for formulas which need a way to explicitly reference the object on which they are currently installed.

kr:gvl *slot* **&rest** *more-slots*   [Macro]

    This is a useful shorthand notation for (`gv :self slot more-slots`). It may only be used within formulas, since it looks for *slot* in the object on which the surrounding formula is installed. For example, the expression (`gvl :color`) returns the current value of the `:color` slot in the object which contains the formula, and is equivalent to the expression (`gv :self :color`).

4.5.7 Object-Oriented Programming

This group includes functions which support objected-oriented programming within KR.

kr:define-method *slot-name object arg-list* **&rest** *body*   [Macro]

    This macro defines a method named *slot-name* for the *object*. While *object* can be any object, and in particular any instance, it is customary to define methods at the level of prototypes; this allows prototypes to provide methods for all their instances.

    The method is defined as a function whose argument list is *arg-list* and whose body is given by the *body*. The method is installed on slot *slot-name*, which is created if needed. In order to facilitate debugging, the function which implements the

method is given a meaningful name formed by concatenating the *slot-name*, the string "`-METHOD-`", and the name of the *object*. Example:

```
(define-method :print box-object (object)
  (format t "A rectangle at (~D, ~D).~%"
    (gv object :x) (gv object :y)))
```

After this, the `:print` method can be inherited by any instance of `box-object`. Sending the message to `rectangle-2`, for example, would cause the following to happen:

```
(kr-send rectangle-2 :print rectangle-2)
;; prints out:
A rectangle at (34, 35).
```

The generated name of the `:print` method, in this example, would be `print-method-box-object`.

kr:kr-send *object slot* **&rest** *arguments*   [Macro]

This macro implements the primitive level of message sending. The *slot* in *object* should yield a Lisp function

object; the function is then called with the arguments specified in *arguments*. Note that the function may be local to the *object*, or it may be inherited.

If the function, i.e., the value of the expression `(g-value object slot)`, is `NIL`, nothing happens and `kr-send` simply returns `NIL`. Otherwise, the function is invoked and the value(s) it returns are returned by `kr-send`.

kr:call-prototype-method **&rest** *arguments*   [Macro]

This macro can be used inside an object's method to invoke the method attached to the object's prototype. It can only be used inside object methods. If a prototype of the current object (i.e, the one which supplied the method currently being executed) also defines a method by the same name, the prototype's method is invoked with *arguments* as the list of arguments. For example,

```
(define-method :notify a (object level)
  ;; Execute object-specific code:
  ;; ...
  ;; Now invoke :notify method from prototype, if any:
  (call-prototype-method object level))))

(kr-send a :notify a 10)
```

First of all, `kr-send` invokes the method defined locally by object `a`. Since the method itself contains a call to `call-prototype-method`, the hierarchy is searched for a prototype of object `a` which also defines the `:notify` method. If one exists, that method is invoked.

A method is free to supply a prototype method with any parameters it wants; this can be accomplished simply by using different values in the call to `call-prototype-method`. In the example above, for instance, we could have written `(call-prototype-method object (+ level 1))`. It is customary, however, to invoke `call-prototype-method` with exactly the same parameters as the original call.

Note that the name of the original object and the message name are not specified in `call-prototype-method`. KR automatically provides the right values.

kr:apply-prototype-method &rest *args*   [Macro]
The macro `apply-prototype-method` is similar to `call-prototype-method`, but the method defined by the prototype is invoked using `apply` rather than `funcall`. This macro may be useful for methods that take **&rest** arguments.

kr:method-trace *object message-name*   [Macro]
This macro can be used to trace method execution. Trace information is printed every time an instance of the *object* is sent the message named *message-name*. Since this expands into a call to the primitive macro `trace`, the Lisp expression `(untrace)` may be used later to eliminate trace information.
Example:

```
(method-trace box-object :print)
```

4.5.8 Reader Macros

A reader macro is defined by default for the `#k<...>` notation, which is produced by the functions `ps` and `gv` when the variable `kr::*print-as-structure*` is non-NIL. 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, as discussed previously. 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 The Type-Checking System

KR supports complete type-checking for slots. Any slot in any object can 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 section [creating-types], page 113). 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.

The KR type-checking mechanism is independent of the lisp type system. Even if a type is defined with lisp's `deftype`, another corresponding type must be defined with KR's `def-kr-type`. The two types may have the same name. The important thing is that the new type must be registered with KR's type system.

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.

4.6.1 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.[3] 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

---

[3] 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`.

type to be 'my-type, which may be more convenient and easier to maintain in the long run.

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")
```

4.6.2 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`.

4.6.3 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.

4.6.4 Retrieving the Predicate Expression

When types are named, `g-type` returns just the name of the type, rather than its associated expression. Sometimes it is useful to retrieve the predicate of the type associated with the type name. The following function serves this purpose:

kr:get-type-definition *type-name*   [Function]
> Given a symbol which names a KR type (i.e., a named type defined with `def-kr-type`), this function returns the type expression that was used to define the type. If no such expression is found, the function returns NIL.

4.6.5 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.

4.6.6 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.

4.6.7 System-Defined Types

The following type predicate can be used to declare types:

kr:is-a-p *prototype*   [Type Predicate]
> This is a 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 `kr: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`   [Garnet Type]
   Any value satisfies this type.

`kr-boolean`   [Garnet Type]
   Same as `t`, but specifically intended for slots which take a nil or non-nil value, often used as boolean variables.

`null`   [Garnet Type]
   Only the value nil satisfies this type.

`string`   [Garnet Type]
   Strings satisfy this type.

`keyword`   [Garnet Type]
   All Lisp keywords satisfy this type.

`integer`   [Garnet Type]
   All integers (fixnums and bignums) satisfy this type.

`number`   [Garnet Type]
   This type includes all numbers: integers, floating point, complex numbers, and fractions.

`list`   [Garnet Type]
   Any list satisfies this type.

`cons`   [Garnet Type]
   Any cons cell (lists and dotted pairs) satisfies this type.

`schema`   [Garnet Type]
   Any non-destroyed KR object satisfies this type.

Garnet also defines many non-basic types, which are typically used by many objects throughout the system. 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*`)`   [Garnet Type]
`'(integer `*0 1*`)`   [Garnet Type]
`'(integer `*0*`)`   [Garnet Type]
`'(integer `*1*`)`   [Garnet Type]

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

The following non-basic types are named, and have associated documentation strings. Users can reference these types anywhere in Garnet programs. To access each type's own documentation string, use `get-type-documentation`.

`known-as-type`   [Garnet Type]
> A keyword (this type is used in the `:known-as` slot)

`aggregate`   [Garnet Type]
> An instance of `opal:aggregate`

`aggregate-or-nil`   [Garnet Type]
> Either an instance of `opal:aggregate` or nil

`bitmap`   [Garnet Type]
> An instance of `opal:bitmap`

`bitmap-or-nil`   [Garnet Type]
> Either an instance of `opal:bitmap` or nil

`color`   [Garnet Type]
> An instance of `opal:color`

`color-or-nil`   [Garnet Type]
> Either an instance of `opal:color` or nil

`font`   [Garnet Type]
> Either an instance of `opal:font` or `opal:font-from-file`

`font-family`   [Garnet Type]
    One of `:fixed`, `:serif`, or `:sans-serif`

`font-face`   [Garnet Type]
    One of `:roman`, `:bold`, `:italic`, or `:bold-italic`

`font-size`   [Garnet Type]
    One of `:small`, `:medium`, `:large`, or `:very-large`

`filling-style`   [Garnet Type]
    An instance of `opal:filling-style`

`filling-style-or-nil`   [Garnet Type]
    Either an instance of `opal:filling-style` or `nil`

`line-style`   [Garnet Type]
    An instance of `opal:line-style`

`line-style-or-nil`   [Garnet Type]
    Either an instance of `opal:line-style` or nil

`inter-window-type`   [Garnet Type]
    A single `inter:interactor-window`, or a list of windows, or t, or nil.

`window`   [Garnet Type]
    An instance of `inter:interactor-window`

`window-or-nil`   [Garnet Type]
    Either an instance of `inter:interactor-window` or nil

`fill-style`   [Garnet Type]
    One of `:solid`, `:stippled`, or `:opaque-stippled`

`draw-function`   [Garnet Type]
    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`   [Garnet Type]
    One of `:left`, `:center`, or `:right`

`v-align`   [Garnet Type]
    One of `:top`, `:center`, or `:bottom`

`direction`   [Garnet Type]
    Either `:vertical` or `:horizontal`

`direction-or-nil`   [Garnet Type]
    Either `:vertical`, `:horizontal`, or nil

`items-type`   [Garnet Type]
    List of items: (`"Label2"...`)

```
accelerators-type                                          [Garnet Type]
     List of lists: ((#\r "alt-r" #\meta-r)...)

filename-type                                              [Garnet Type]
     A string that represents a pathname

priority-level                                             [Garnet Type]
     An instance of inter:priority-level
```

4.7 Functional Interface: Additional Topics

This section describes features of KR that are seldom needed by casual Garnet users. These features are useful for large application programs, especially ones which manipulate constraints directly, or for application programs which use the more advanced knowledge representation features of KR.

4.7.1 Schema Manipulation

kr:create-schema *object-name* **&rest** *slot-definitions*   [Macro]
     This macro creates and returns a new object named *object-name*. It is much more primitive than create-instance, since it does not copy down formulas from a prototype and does not call the :initialize method.

     If *object-name* is nil, an unnamed object is created and returned. If *object-name* is a symbol, a special variable by that name is created and bound to the new object. The *slot-definitions*, if present, are used to create initial slots and values for the object. Each slot definition should be a list whose first element is the name of a slot, and whose second element is the value for that slot.

     create-schema understands the :override keyword and the :name-prefix keyword; see [create-options], page 137, for more details.

     Examples:

```
(create-schema 'rectangle-3 (:is-a box-object) (:x 70))
(create-schema 'rectangle-3 :override (:y 12))   ; add a slot
(create-schema nil (:is-a my-graphical-object))
```

kr:create-prototype *object* **&rest** *slot-definitions*   [Macro]
     This macro is slightly more primitive than create-instance. Unlike create-instance, it does not allow a prototype to be specified directly. Moreover, it does not automatically send the :initialize message to the newly created *object*. Like create-instance, it copies formulas from any prototype into the newly created *object*.

     The following two examples are roughly equivalent:

```
(create-instance nil box-object (:x 12))

;;; The hard way to do the same thing
(let ((a (create-prototype nil
   (:is-a box-object) (:x 12))))
  (kr-send a :initialize a))
```

Most applications will find create-instance much more convenient. The only case when `create-prototype` should be used is when it is important that the `:initialize` message *not* be sent to an object at creation time.

`Create-prototype` also understands the `:override` keyword and the `:name-prefix` keyword; see [create-options], page 137, for more details.

kr:destroy-schema *object*   [Function]
Destroys the *object*. Returns `t` if the object was destroyed, `nil` if it did not exist. This function takes care of properly removing all constraint dependencies to and from the *object*. Any formula installed on any slot of the *object* is also destroyed.

Usually, Garnet users do not call this function directly. Instead, they use (`opal:destroy object`), which performs all necessary clean-up operations and eventually calls `destroy-schema`.

kr:destroy-slot *object slot*   [Function]
Destroys the *slot* from the *object*. The value previously stored in the slot, if there was one, is lost. All constraints to and from *object* are modified accordingly. The invalidate demon is run on the slot before it is destroyed, ensuring that any changes caused by this action become visible to formulas that depend on the slot. Using *destroy-slot* on slots that are declared constant gives a continuable error. Continuing from the error causes the slot to be destroyed anyway. This behavior can be overridden by using the macro `with-constants-disabled`.

kr:name-for-schema *object*   [Function]
Given a *object*, this function returns its name as a string. The special notation #*k*<> is never used, i.e., the name is the actual name of the object. The return value should never be modified by the calling program.

4.7.2 Uniform Declaration Syntax

One syntax can be used for all kinds of declarations associated with slots in an object. 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. (For details on the type-checking mechanism, see Chapter [type-system], page 112.)

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. (See section [constant-slots], page 125.)

`:ignored-slots` - The slots that follow will not be printed by the function `ps`. (See
section [print-control-slots], page 139.)

`:local-only-slots` - The values that follow should be lists of the form *(slot-name copy-
down-p)*. The *slot-name* specifies the name of a slot which should be treated as local-only,
i.e., should not be inherited by the object's instances. If *copy-down-p* is NIL, the slot will
have value NIL in the instances. Otherwise, the value from the object will be copied down
when instances are created and marked as local; this prevents further inheritance, even if
the value in the prototype is changed. (See section [local-only], page 136.)

`:maybe-constant` - Specifies the list of slots that can be made constant in this object's
instances simply by specifying the special value T. (See section [constant-slots], page 125.)

`: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. (See section [print-control-slots], page 139.)

`:type` - Introduces type declarations for one or more slots. (See chapter [type-system],
page 112.)

`:update-slots` - The list of update slots for the object, i.e., the slots that should trigger
the `:invalidate-demon` when modified. (See section [update-slots], page 132.)

4.7.3 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. For example, it is 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

```
: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 section [constant-slots], page 125, for an elaborate discussion of constant slots.

4.7.4 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*).

4.7.5 Relations and Slots

KR supports special slots called *relations*. Relations serve two purposes: allowing inheritance, and automatically creating inverse connections. In addition to a handful of predefined relations, application programs can create new relations as needed via the function `create-relation` (see below).

Slots such as `:is-a`, which enable knowledge to be inherited from other parts of a network, are called *inheritance relations*. Inheritance along such relations proceeds depth-first and may include any number of steps. The search terminates if a value is found, or if no other object can be reached.

Any relation, including user-defined ones, may also be declared to have an inverse relation . If this is the case, KR automatically generates an inverse link any time the relation is used to connect one object to another. Imagine, for instance, that we defined `:part-of` to be a relation having `:has-parts` as its inverse. Adding object A to the slot `:part-of` of object B would automatically add B to the slot `:has-parts` of object A, thereby creating a reverse link.

KR automatically maintains all relations and inverse relations, and the application programmer does not have to worry about them. In the example above, if slot `:part-of` in object B is deleted, the value B is also removed from the slot `:has-parts` of object A. The same would happen is object B is deleted. This ensures that the state of the system is consistent at any point in time, independent of the particular sequence of operations.

The following functions handle user-defined relations and slots:

`kr:create-relation` *name inherits-p* **&rest** *inverses*   [Macro]

> Declares the slot *name* to be a relation. The new relation will have *inverses* (a possibly empty list of slot names) as its inverse relations. If *inherits-p* is non-`nil`, *name* becomes a relation with inheritance, and values may be inherited through it.
>
> The following form defines the non-inheritance relation `:has-parts` and its two inverses, `:part-of` and `:subsystem-of`:
>
>   (create-relation :has-parts NIL :part-of :subsystem-of)

`kr:relation-p` *thing*   [Macro]

> This predicate returns `nil` if *thing* is not a relation, or a non-`nil` value if it is the name of a relation slot.
> Examples:
>
>   (relation-p :is-a) ==> non-NIL value
>   (relation-p :color) ==> NIL

`kr:has-slot-p` *object slot*   [Function]

> A predicate that returns `t` if the *object* contains a slot named *slot*, `nil` otherwise. Note that *slot* must be local to the *object*; inherited slots are not considered. Examples:
>
>   (has-slot-p rectangle-1 :is-a) ==> T
>   (has-slot-p rectangle-1 :thickness) ==> NIL   ; not local

kr:doslots *slot-var object* **&optional** *inherited* **&rest** *body*   [Macro]

    Iterates the *body* over all the slots of the *object*. The *slot-var* is bound to each slot in turn. The *body* is executed purely for side effects, and `doslots` returns `nil`. Example:

```
(doslots (slot rectangle-1)
  (format t "Slot ~S has value ~A~%"
          slot (gv rectangle-1 slot)))
;; prints out:
Slot :Y has value 20
Slot :X has value 10
Slot :IS-A has value #k<BOX-OBJECT>
```

    By default, `doslots` only iterates over the local slots of *object*. But if the *inherited* parameter is T, then all slots that have been inherited from the *object*'s prototype will be iterated over as well. Note: Only those slots that have actually been inherited will be included in the list of inherited slots. If they are merely defined in the prototype and have not been `gv`'d in the instance, then they will not be included in the iteration list. See the description of the function `ps` in section [print-control-slots], page 139, for a way to display all the slots that could possibly be inherited by the object.

4.7.6 Constraint Maintenance

These functions are concerned with the constraint maintenance part of KR.

kr:change-formula *object slot form*   [Function]

    If the *slot* in *object* contains a formula, the formula is modified to contain the *form* as its new function. `change-formula` works properly on any formula, regardless of whether the old function was local or inherited from another formula. If formula inheritance is involved, this function makes sure that all the links are modified as appropriate. If the *slot* does not contain a formula, nothing happens.

    Note that this function cannot be used to install a fixed value on a slot where a formula used to be; `change-formula` only modifies the expression within a formula.

kr:recompute-formula *object slot*   [Function]

    This function can be called to force a formula to be recalculated. It may be used in situations where a formula depends on values which are outside of KR (such as application data, for example). The formula stored in the *slot* of the *object* is recalculated. Formulas which depend on the *slot*, if any, are then marked invalid.

kr:mark-as-changed *object slot*   [Function]

    This function may be used to trigger constraint propagation for a *object* whose *slot* has been modified by means other than `s-value`. Some applications may need to use destructive operations on the value in a slot, and then notify the system that certain values were changed. `mark-as-changed` is used for this purpose.

kr:copy-formula *formula*   [Function]

    This function returns a copy of the given *formula*, which should be a formula object. The copy shares the same prototype with the *formula*, and its initial value is the current cached value of the *formula*.

`kr:move-formula` *from-object from-slot to-object to-slot*   [Function]
> This function takes a formula from a slot in an object and moves it to another slot
> in another object. This function is needed because one cannot move a formula from
> one slot to another simply by storing the formula in some temporary variable (this
> creates potentially serious problems with formula dependencies).

`kr::make-into-o-formula` *formula* **&optional** *compile-p*   [Function]
> This function 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. If *compile-p* is non-NIL,
> the *formula*'s expression is compiled in the process.

`kr:g-cached-value` *object slot*   [Function]
> This function is similar to `gv` if the *slot* contains an ordinary value. If the *slot* contains
> a formula, however, the cached value of the formula is returned even if the formula is
> invalid; the formula itself is never re-evaluated. Only advanced applications may need
> this function, which in some cases returns out-of-date values and therefore should be
> used with care.

`kr:destroy-constraint` *object slot*   [Function]
> If the *slot* of the *object* contains a formula, the constraint is removed and replaced
> by the current value of the formula. The formula is discarded and all dependencies
> are updated. Dependent formulas are notified that the formula has been replaced by
> the formula's value, even if the actual value does not change. If the *slot* contains an
> ordinary value, this function has no effect.
>
> Note that the expression (`s-value object slot (gv object slot)`) cannot be used
> to simulate `destroy-constraint`. This is because `s-value` does not remove a formula
> when it sets a slot to an ordinary value, and thus the expression above would simply
> set the cached value of the formula without removing the formula itself.

`kr::with-dependencies-disabled` **&body** *body*   [Macro]
> This macro 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`. 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.

4.8 Constant Formulas

It is possible to declare that certain slots are constant, and cause all formulas that only
depend on constant slots to be eliminated automatically. The main advantage of this
approach is that it reduces storage and execution time.

A slot in an object can be declared constant at object creation time. This guarantees
that the application program will never change the value of the slot after the object is
created. When a formula is evaluated for the first time, KR checks whether it depends
exclusively on constant slots. If this is the case, the formula is eliminated and its storage

is reused. The slot on which the formula was originally installed takes the value that was computed by the formula.

A slot can become constant in one of three ways. First, the slot may be declared constant explicitly. This is done by listing the name of the slot in the `:constant` slot of an object (see below for more details), or calling `declare-constant` on the slot after its object has already been created. For example, adding the following code to `create-instance` for object A will cause slots `:left` and `:top` to be declared constant in object A: `(:constant '(:left :top))`. Note that it is possible for the value of the `:constant` slot to be computed by a formula, which is evaluated once at object creation time.

Second, a slot may become constant because it is declared constant in the object's prototype. In the example above, if object B is created with A as its prototype, slots `:left` and `:top` will be declared constant in B, even if they are not explicitly mentioned in object B's `:constant` slot.

Third, a slot may become constant because it contains a formula which depends exclusively on constant slots. After the formula is removed, the slot on which it was installed is declared constant. Thus, constants propagate recursively through formulas.[4] If you cannot figure out why a formula is not being eliminated, the function `garnet-debug:why-not-constant` and related functions in the Debugging Tools Reference Manual may be useful.

To facilitate the creation of the list of constant slots for an object, the syntax of the `:constant` slot is extended as follows. First, a prototype may specify a list of all the slots that its instances may choose to declare constant. This is done by specifying a list of slot names in the prototype, using the slot `:maybe-constant`. When this is done in the prototype, an instance may choose to declare all of those slots constants by simply adding the value `t` to its `:constant` slot. Note that `t` does *not* mean that *all* slots are constant; it only means that all slots in the `:maybe-constant` list become constant.

It is also possible for the instance to add more constant slots as necessary. Consider the following example:

```
(create-instance 'proto nil (:maybe-constant '(:left :x1 :x2 :width)))
(create-instance 'inst proto (:constant '(:top :height t)))
```

No slot is declared constant in the prototype, i.e., object PROTO, because the `:maybe-constant` slot does not act on the object itself. However, because object INST includes the value `t` in its `:constant` slot, the list of constant slots in the instance is the union of the slots that are declared constant locally and the slots named in the `:maybe-constant` slot of the prototype. Therefore, the following slots are constant in INST: `:left`, `:top`, `:width`, `:height`, `:x1`, and `:x2`.

The slot `:maybe-constant` is typically used in prototypes to specify the list of all the parameters of the instances, i.e., the slots that an instance may customize to obtain gadgets with the desired appearance. Consider, for example, the prototype of a gadget. If the application is such that a gadget instance will never be changed after it is created, the application programmer may simply specify `(:constant '(T))`. This informs the system

---

[4] In the most elegant programming style, a minimum number of constants will be declared in an object, and formulas will be allowed to become constant because of their dependencies on the constant slots (rather than bluntly declaring the formulas constant). This is certaintly not a requirement of programming with constants, however.

that all parameters declared by the creator of the prototype are, in fact, constant, and formulas that depend on them can be eliminated once the gadget is created. All of the standard objects and gadgets supply a `:maybe-constant` slot.

The syntax of the `:constant` slot also allows certain slots that appeared in the `:maybe-constant` list to be explicitly excluded from the constant slots in an object. This can be done by using the marker `:except` in the `:constant` slot. The slots following this marker are removed from the list that was specified by the prototype. If a slot was not mentioned in the prototype's `:maybe-constant` slot, the `:except` marker has no effect on the slot. The following is a comprehensive example of the syntax of the `:constant` slot:

```
(create-instance 'inst-2 proto
    (:constant '(:top :height t :except :width :x2)))
```

As a result, these slots are declared constant in object inst-2: `:left`, `:top`, `:height`, and `:x1`.

It is an error to set slots that have been declared constant. This can happen in three cases: a slot may be set using `s-value` after having been declared constant, a call to `create-instance` may redefine in the instance a slot that was declared constant in the prototype, or `destroy-slot` may be used. In all cases, a continuable error is signaled. Note that this behavior can be overridden by wrapping the code in the macro `with-constants-disabled` (see below).

kr:declare-constant *object slot*   [Function]

> The function `declare-constant` may be used to declare slots constant in an object after creation time. The function takes an object and a slot, which is declared constant. The behavior is the same as if the slot had been declared in the `:constant` slot at instance creation time, although of course the change does not affect formulas which have already been evaluated. The `:constant` slot of the object is modified accordingly: the new slot is added, and it is removed from the `:except` portion if it was originally declared there. As a special case, if the second argument is `t` all the slots that appear in the slot `:maybe-constant` (typically inherited from a prototype) are declared constant. This is similar to specifying T in the `:constant` slot at instance creation time.
>
> If `declare-constant` is executed on a slot while constants are disabled (i.e., inside of a `with-constants-disabled` body), the call will have no effect and the slot will not become constant.

kr:with-constants-disabled **&body** *body*   [Macro]

> The macro `with-constants-disabled` may be used to cause all constant declarations to be temporarily ignored. During the execution of the body, no error is given when slots are set that are declared constant. Additionally, constant declarations have no effect when `create-instance` is executed inside this macro. This macro, therefore, is intended for experienced users only.
>
> Several functions in the `garnet-debug` package (loaded with Garnet by default) can be helpful in determining which slots in your application should be declared constant for maximum benefit, and can help you determine why some slots are not becoming constant. These functions are documented in the Debugging Tools Reference Manual, which starts on page `debug`:

`gd:record-from-now`	[Function]
`gd:Suggest-Constants` *object* **&key** *max* (*recompute-p* **t**) (*level* **1**)	[Function]
`gd:explain-formulas` *aggregate* **&optional** (*limit* **50**)	[Function]
eliminate-useless-p	
`gd:find-formulas` *aggregate* **&optional** (*only-totals-p* **t**) (*limit* **50**)	[Function]
`gd:count-formulas` *object*	[Function]
`gd:why-not-constant` *object slot*	[Function]

4.8.1 Efficient Path Definitions

The function `kr-path` can be used to improve the efficiency of formula access to slots that are obtained via indirect links. Inside formula expressions, macros such as `gv` are used to access a slot indirectly, traversing a number of objects until the last slot is obtained. This is sometimes called a *link* or a *path*. For example, the expression (`gvl :parent :parent :left`) will access the `:left` slot in the parent's parent. If the application program can guarantee that the intermediate path will not change, the function `kr-path` provides better performance. The expression above could be written as:

```
(gv (kr-path 0 :parent :parent) :left)
```

The call to `kr-path` computes the object's parent's parent only once, and stores the result as part of the formula. Subsequent evaluations of the formula only need to access the `:left` slot of the target object. The syntax is:

`kr:kr-path` *path-number* **&rest** *slots*   [Macro]
> The path-number is a 0-based integer which indicates the number of this path within the formula expression. In most cases, a formula contains only one call to `kr-path`, and path-number is 0. If more than one path appears in a formula expression, different numbers should be used. For example,
>
> ```
> (or (gv (kr-path 0 :parent :parent) :left)
>     (gv (kr-path 1 :alternate :parent) :left))
> ```
>
> Note that `kr-path` can only be used inside a formula expression.

4.9 Tracking Formula Dependencies

The 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.
>
> ```
> (create-instance 'a nil (:left 7))
> (create-instance 'b a (:left 14) (:top #f(+ (gvl :left) (gv a :left)))))
> (gv b :top) ; set up the dependencies
>
> (kr::i-depend-on b :top)
>     ==> ((b . :left) (a . :left))
> ```

4.10 Formula Meta-Information

It is 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 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.

4.10.1 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.

4.10.2 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.

When the function ps is given a formula, it can print associated meta-information. The
latter is printed as an object, immediately after the formula itself. For example:

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

lisp> (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"
```

4.10.3 Demons

The demon mechanism allows an application program to perform a certain action when a value is modified. This mechanism, which is totally controlled by the application program, is independent from value propagation. Regular Garnet users do not need to know the contents of this section, since Garnet already defines all appropriate demons. Garnet applications should never modify the default demons, which are defined by Opal and automatically update the graphical representation of the application's objects.

4.10.4 Overview of the Demon Mechanism

A demon is an application-defined procedural attachment to a KR schema. Demons are user-defined fragments of code which are invoked when certain actions are performed on a schema. Whenever the value of a slot in a schema is modified (either directly or as the result of value propagation), KR checks whether a demon should be invoked. This allows application programs to be notified every time a change occurs.

Two separate demons invoked at different times allow an application program to have fine control over the handling of value changes. These demons are only invoked on slots that are listed in the `:update-slots` list of a schema (see section [update-slots], page 132).

The first demon is the *invalidate demon*. This demon is invoked every time a formula is invalidated. At the time the demon is invoked, the formula has not yet been re-evaluated, and thus it contains the old cached value. This demon is contained in the `:invalidate-demon` slot of an object. This makes it possible for different objects to provide customized demons to handle slot invalidation.

The second demon is the pre-set demon. It is invoked immediately before the value in a formula is actually modified, and it is passed the new value. This allows the pre-set demon to record the difference between the old and the new value, if needed. This demon is stored in the variable `kr::*pre-set-demon*`. Garnet does not use the pre-set demon.

The relationship between value propagation and demon invocation is best illustrated by showing the complete sequence of events for the invalidate demon. This is what happens when `s-value` is called to set slot `s` of schema S to value `v`:

1. if slot `s` already contains value `v`, nothing happens.

2. otherwise, if slot `s` should trigger demons, the demon is invoked. the demon is called with schema s in its *old* state, which means that slot `s` still contains its old value.

3. the change is recursively propagated. all slots whose value is a formula that depends on slot `s` are invalidated. the process is similar to the one described in step 2, but there is no check corresponding to step 1 at this point. demons are invoked normally on any slot that is modified during this phase.

4. the value of slot `s` is finally changed to `v`.

Both the invalidate demon and the pre-set demon should be functions of three arguments. The first argument is the schema which is being modified. The second argument is the name of the slot which is being modified. The third argument is always `nil` for the invalidate demon. For the pre-set demon, the third argument is the new value which is about to be installed in the slot. This allows the pre-set demon to examine both the old value (which is still in the slot) and the new value.

4.10.5 The `:update-slots` List

The KR demons are only invoked on slots that are listed in the `:update-slots` list of the schema containing them. For example, Garnet defines a particular demon that is responsible for redrawing the objects in a window as the values of their "interesting" slots change. These "interesting" slots are declared in each object's `:update-slots` declaration during `create-instance` (the declaration is usually inherited from the prototype, so that typical Garnet users will never see this declaration). The `:update-slots` list contains all the slots in an object that should cause Opal's special demon to be invoked when they are modified. When an update-slot is modified, Opal's demon will "invalidate" the (object), causing it to be redrawn during the next pass of the update algorithm.

The `:update-slots` list can only be set directly at `create-instance` time. That is, after an object is created it is no longer sufficient to modify the value of the `:update-slots` slot to change whether a slot is an update-slot or not. This is because update-slots are internally represented by a bit associated with the slot, which is set during the `create-instance` call. Instead of setting the `:update-slots` slot, you must call the function:

`kr::add-update-slot` *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.

4.10.6 Examples of Demons

The following example shows how to define the invalidate demon for an object, and how the demon is invoked.

```
;; Define an invalidate demon
(defun inv-demon (schema slot v)
  (declare (ignore v))     ; v is not used
  (format t
  "schema ~s, slot ~s is being invalidated.~%"
  schema slot))

(create-schema 'a (:left 10)
  (:top (o-formula (1+ (gvl :left))))
  (:update-slots '(:top))
  (:invalidate-demon 'inv-demon))

(gv a :top)  ==> 11
(s-value a :left 1)
;; prints out:
schema #k<A>, slot :TOP is being invalidated.
(gv A :top)  ==> 2
```

4.10.7 Enabling and Disabling Demons

kr:With-Demons-Disabled &body *body*   [Macro]

>   The *body* of this macro is executed with demons disabled. Constraints are propagated as usual, but demons are not invoked.

>   This macro is often useful when making temporary changes to schemata which have un update demon. This happens, for instance, when a program is changing graphical objects but does not want to display the changes to the user, or when some of the intermediate states would be illegal and would cause an error if demons were to run. Objects may be freely modified inside the *body* of this macro without interference from the demons.

>   **kr:With-Demon-Disabled** *demon* **&body** *body*macro

This is similar to `with-demons-disabled`, except that it allows a specific demon to be disabled. Normally, when `with-demons-disabled` is used, all demons are disabled. This macro allows all demons except a specific one to execute normally; only the specific demon is disabled.

The forms in the *body* are executed, but the given *demon* is not invoked. For example, the following will selectively disable the invalidate demon provided by object FOO:

```
(with-demon-disabled (gv FOO :invalidate-demon)
   (s-value FOO :left 100))
```

While FOO's own demon is not executed, formulas in other objects which depend on FOO's `:left` slot will be invalidated, and the corresponding invalidate demons will be invoked normally.

>   **kr:With-Demon-Enabled** *demon* **&body** *body*macro

This macro enables a particular demon if it had been disabled, either explicitly or with `with-demons-disabled`.

4.10.8 Multiple Inheritance

KR supports multiple inheritance : a schema may inherit values from more than one direct ancestor. This can be accomplished in two ways. The first way is simply to connect the schema to more than one ancestor schema through a relation. The relation slot, in other words, may contain a list of slots. When performing inheritance, KR searches each ancestor slot in turn until a value is found.

The second way to achieve multiple inheritance is by using more than one relation with inheritance. Any schema may have several slots defined as relations with inheritance; in this case, all relations are searched in turn until a value is found. The two mechanisms may be combined, of course.

Application programs should not rely on the order in which KR searches different relations. The particular order is implementation-dependent.

4.10.9 Inheritance: Implementation Notes

KR uses a mechanism which enables inheritance to behave in the dynamic fashion describe above and, at the same time, to provide extremely efficient performance. This mechanism is named *eager inheritance* .

Eager inheritance works as follows. The first time the value of a slot is requested, but the value is not present locally, the value is obtained by inheritance as described above. At this point, however, the value is also copied into the local schema (and in any intervening schema, if necessary) with a special marker which indicates that the value was inherited.

The second time the value is requested, inheritance is no longer required and the value is immediately found locally. This makes successive accesses to inherited values much faster, and causes inheritance to be essentially as efficient as local values, no matter how many levels of inheritance were originally used.

It is vital that inherited values which were copied down into children schemata be kept up to date. Any change in the upper portions of the schema hierarchy might change what values can be inherited by the lower levels, and inherited values which were copied down must be modified. KR performs this task immediately when a value which was inherited is changed, thus justifying the term *eager inheritance*. This technique ensures minimal overhead for both access and update of inherited values, and provides superior performance for the inheritance mechanism.

4.10.10 Local Values

This group contains functions which deal with local values in a slot. Some of these functions do not treat formulas as special objects, and thus can be used to access formulas stored in a slot (remember that functions like `gv`, for example, return the *value* of a formula, rather than the formula object itself).

`kr:get-value` *object slot*   [Macro]

> Returns the value in the *slot* from *object*. If the slot is empty or not present, it returns `nil`. Inheritance may be used when looking for a value. Given a slot that contains a formula, `get-value` returns the formula itself, rather than its value. Therefore, its use is limited to applications that manipulate formulas explicitly.

`get-values` *object slot*   [Macro]

> This macro returns a list of all the values in the *slot* of the *object*. If the *slot* is empty or not present, it returns `nil`. Inheritance may be used when looking for values. This macro does not deal with constraints, i.e., it does not cause formulas to be evaluated. Examples:

```
(get-values my-graphical-object :is-a-inv) ==>
 (#k<BOX-OBJECT>)
(get-values box-object :is-a-inv) ==>
 (#k<RECTANGLE-2> #k<RECTANGLE-1>)
```

> Since `get-values` does not deal with constraints, `dovalues` (see below) is the preferred way to access all values in a slot. An additional advantage is that the expression

```
(dovalues (item object slot) ...)
```

> is potentially more efficient than the equivalent idiom

```
(dolist (item (get-values object slot))  ...)
```

> which may create garbage in some situations.

> A `setf` form for `get-values` is defined for `get-values` and expands into a call to `set-values`.

set-values *object slot values*   [Function]

> This function stores a list of values in the *slot* of the *object*. The entire list may subsequently be retrieved with `get-values`, or the first value may be retrieved with `g-value`.

dovalues (*variable object slot* **&key** *local result formulas in-formula*)   [Macro]
> **&rest** *body*
>
> dovalues executes the *body* with the *variable* bound in turn to each value in the *slot* of the *object*. The *body* is executed purely for side effects, and DOVALUES normally returns `nil`; if the keyword argument `:result` is specified, however, the given value is returned. The *body* of `dovalues` should never alter the contents of the *slot*, since this may cause unpredictable results.
>
> If `:local` (default `nil`) is non-`nil`, dovalues only considers local values; otherwise, it iterates over inherited values if no local values are present. If `:formulas` is `T` (the default), any value which is expressed by a formula is computed and returned; otherwise, the formula itself is returned. The latter is only useful for more advanced applications. Examples:
>
> ```
>     (set-values rectangle-1 :vertices '(3 6 72))
>
>     (dovalues (v rectangle-1 :vertices)
>        (format t "rectangle-1 has vertex ~S~%" v))
>     ;; prints out:
>     rectangle-1 has vertex 3
>     rectangle-1 has vertex 6
>     rectangle-1 has vertex 72
>
>     (s-value-n rectangle-1 :vertices 2
>        (o-formula (+ (gvl :vertices) 15)))
>     (dovalues (v rectangle-1 :vertices)
>        (format t "rectangle-1 has vertex ~S~%" v))
>     ;; prints out:
>     rectangle-1 has vertex 3
>     rectangle-1 has vertex 6
>     rectangle-1 has vertex 18
>
>     ;;; example of :formulas nil
>     (dovalues (v rectangle-1 :vertices :formulas nil)
>        (format t "rectangle-1 has vertex ~S~%" v))
>     rectangle-1 has vertex 3
>     rectangle-1 has vertex 6
>     rectangle-1 has vertex #k<F2297>
> ```
>
> `dovalues` may also be used inside formulas; in this case, `:in-formula` should be set to `T`. The surrounding formula is then re-evaluated when any of the values in the *slot* is changed, or whenever a value is added or deleted. `dovalues` with a non-`nil` `:in-formula` option, therefore, behaves more like `gv` than like `g-value`. When `dovalues` is used in this fashion, the keyword `:self` may be used to stand for the object to which the formula is attached.

The following is an example of a formula which uses `dovalues` and is re-evaluated when one of the values in the `:components` slot changes:

```
(o-formula (let ((is-odd NIL))
    (dovalues (value :SELF :components
       :in-formula T)
     (if (odd value) (setf is-odd T)))
    is-odd))
```

kr:get-local-value *object slot*   [Macro]
   Returns the value in the *slot* from *object*. If the slot is empty or not present, it returns `nil`. Inheritance is not used, and only local values are considered. Given a slot that contains a formula, `get-local-value` returns the formula itself, rather than the formula's value. Therefore, use of this macro is limited to applications that manipulate formulas explicitly.

get-local-values *object slot*   [Macro]
   Similar to get-values, but only local slots are examined and inheritance is never used. Examples:

```
(get-values rectangle-1 :thickness) ==> (1)
(get-local-values rectangle-1 :thickness) ==> NIL ; not local
```

   This macro does not deal with constraints, i.e., it never causes formulas to be evaluated.

kr:g-local-value *object slot* **&rest** *other-slots*   [Macro]
   This macro is very similar to `g-value`, except that it only considers local values. Inheritance is never used when looking for a value.

kr:gv-local *object slot* **&rest** *more-slots*   [Macro]
   This macro is similar to `gv`, except that it only considers local values, and it never returns an inherited value. `gv-local` should be used in situations where it is important to only retrieve values that are local to the *object*.

append-value *object slot value*   [Function]
   This function adds the *value* to the end of the list of values in the *slot* of the *object*.

delete-value-n *object slot position*   [Function]
   This function deletes the *position*-th value from the *slot* of the *object*. *position* is a 0-based non-negative integer. This function does not deal with constraints properly, and should not be used when the *slot* may contain formulas.

4.10.11 Local-only Slots

There are cases when certain slots in an object should not be inherited by any instance of the object. An example of this situation might be a slot which is used as a unique identifier; clearly, the slot should never be inherited, or else errors will occur. Such slots are called *local-only slots*.

   This effect can be achieved in KR by listing the names of all such slots in the prototype object. The names are listed in the `:local-only-slots` declaration (the general declaration

syntax is discussed in section [uniform-syntax], page 120). This declaration should contain a list of two-element sub-lists. The first element in each sub-list specifies the name of a local-only slot. The second element can be `t` or `nil`.

the value `nil` specifies that the local-only slot is always initialized to `nil` in any instance which does not define it explicitly. The value `t`, on the other hand, specifies that the current value of the local-only slot in the prototype will be used to initialize the slot in the instance. The value, however, is physically copied down into the instance, and thus inheritance is no longer used for that instance. Modifying the value in the prototype, in particular, will have no effect on the instance. This second option is used more rarely than `nil`.

Note that none of the above applies to slots whose value (in the prototype) is a formula. Slots which contain formulas are always inherited, independent of whether the slot is listed in `:local-only-slots`.

4.10.12 Schema Creation Options

Two special keywords can be used in the macros that create schemata. These options are recognized by `create-instance`, `create-schema`, and `create-prototype`. They :

`:override` *object-name*   [Keyword]

> If the *object-name* in one of the object-creating macros names an existing object, that object is normally deleted, together with its instances, and replaced by a brand new object. The default behavior may be modified by using the keyword `:override` as part of the *slot-definitions*. This keyword causes the existing object to be modified in place and contain the union of its previous slots and those specified by `create-schema`. Previous slots that are not mentioned in the call retain whatever values they had before the operation. For example,

```
(create-schema 'rectangle-1 :override (:color :magenta))
```

> adds a slot to the object RECTANGLE-1 if it already exists. Without the `:override` keyword, this would have destroyed the object and created a new one with a single slot.

`:name-prefix` *string*   [Keyword]

> The keyword `:name-prefix` may be used to specify a name prefix for unnamed objects. Unnamed objects are normally named after the object they are an instance of; this option allows a specific string to be used as the name prefix. The option, if specified, should be immediately followed by a string, which is used as the prefix.
>
> Example:

```
(create-schema nil :name-prefix "ORANGE"
  (:left 34)) ==> #k<ORANGE-2261>
```

4.10.13 Print Control

This section describes the slots that control what portions of an object are printed, and how they are printed. The need for fine control over printing arises, for example, when certain slots contain very large data structures that take a long time to print.

The print control slots are taken from the object which is specified as the *print-control* in the complicated form of `ps`, described below. In many cases, the slots are actually inherited by the object being printed.

kr:ps *object* **&key** *types-p all-p* (*control* **t**) (*inherit* **nil**) (*indent* **0**)   [Function]
         (*stream* ***standard-output***)

This form of **ps** prints the contents of the *object*, and allows fine control over what to print and how. A possible behavior is to print out all slots and all values in *object*; this happens when the *control* object is **nil**. It is possible, however, to cause **ps** to ignore certain slots and to specify that others should be printed in a given order. It is also possible to limit the number of elements printed for list values, thus preventing annoyingly long lists of values.

The function **ps** can print 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.

Supplying a non-NIL value for the *all-p* parameter will cause **ps** to print out all slots of the *object*, including slots that do not currently have any value. The default for *all-p* is **nil**.

The value of *control* should be one of four things:

nil   which means that the *object* is printed in its entirety.

t   the default, which means that the *object* itself is used as the control object. In most cases, the control slots are inherited from an ancestor of the *object*. All Opal prototypes, for example, define appropriate slots which reduce the amount of information that is shown by **ps**.

object   where *object* is used directly as the control object.

:default   indicates that the KR-supplied default print control object should be used. The name of the default print control object is **print-schema-control** , an object in the KR package. This default object limits the length of lists that are printed by **ps** to a maximum of ten for ordinary slots, and five for the **:is-a-inv** slot.

If the *inherit* option is **nil** (the default), only local slots are printed. Otherwise, all inheritable values from all prototypes of *object* are inherited and printed; inherited values are clearly indicated in the printout. As discussed in Chapter [object-oriented-prog], page 102, formulas are not copied down from prototypes until they are requested by **gv** or **g-value**. Formulas that have not yet been copied down will not be shown by **ps**, unless the *inherit* parameter is non-nil.

The **:indent** option is only used by debugging code which needs to specify an indentation level. This option is not needed by regular application programs.

ps prints slots whose value is a formula in a special way. Besides the name of the formula, the current cached value of the formula is printed in parentheses, followed by **t** if the cache is valid or **nil** otherwise. Example:

```
(create-schema 'a
  (:left 10) (:right (o-formula (+ (gvl :left) 25))))
(gv a :right)  ==> 35

(ps a)
;; prints out:
```

```
#k<A>
   :DEPENDED-SLOTS =  (:LEFT #k<F2285>)
   :RIGHT =  #k<F2285>(35 . T)
   :LEFT =  10

(s-value a :left 50)
(ps a)
;; prints out:
#k<A>
   :DEPENDED-SLOTS =  (:LEFT #k<F2285>)
   :RIGHT =  #k<F2285>(35 . NIL)
   :LEFT =  50
```

The cached value is not correct, of course, but it will be recomputed as soon as its value is requested because formula `F2285` is marked invalid.

The function `ps` prints the expression of a formula, when given the formula as argument. A formula is printed with three pieces of information: the expression, the cached value (which is printed as before), and the object and slot on which the formula is installed.

If *stream* is specified, it is used for printing to a stream other than standard output.

4.10.14 Print Control Slots

If a control object is specified in a call to `ps`, it should contain (or inherit) six special slots. These slots determine what `ps` does. The meaning of the print control slots is as follows:

- `:sorted-slots` contains a list of names of slots that should be printed before all other slots, in the desired order.
- `:ignored-slots` contains a list of names of slots that should not be printed. A summary printed at the end of the object indicates which slots were ignored.
- `:global-limit-values` contains an integer, the maximum number of elements that should be printed for each list that is a value for a slot. If a list contains more than that many elements, ellipsis are printed after the given number to indicate that not all elements of the list were actually displayed.
- `:limit-values` allows the same control on a slot-by-slot basis. It should contain lists of the form (`slot number`). If a slot name appears in one of these lists, the number specified there is used instead of the one specified in `:global-limit-values`.
- `:print-as-structure` can be `t`, in which case the #k<> notation is used when printing object names, or `nil`, in which case only pure object names are printed.
- `:print-slots` is a list of the slots that are printed as part of the #k<> notation. It is possible to cause `ps` to print a few slots from each object, inside the #k<> printed representation; this may make it easier to identify different schemata. `:print-slots` should contain a list of the names of the slots which should be printed this way. Note that this option has no effect if schema names are not being printed with the #k<> notation.

The following is a rather comprehensive example of fine control over what `ps` prints.

```
; Use top level of the hierarchy to control printing.
```

```
(create-schema 'top-object
  (:ignored-slots :internal :width))

(create-schema 'colored-thing (:color :blue) (:x 10)
  (:is-a top-object) (:width 12.5) (:y 20)
  (:internal "some information"))

(dotimes (i 20) (create-instance NIL COLORED-THING))
```

Using ps with a null *control* prints out the whole contents of the schema:

```
(ps colored-thing :control nil)
;; prints out:
#k<COLORED-THING>
  :IS-A-INV =  #k<COLORED-THING-2265>
    #k<COLORED-THING-2266> #k<COLORED-THING-2267>
    #k<COLORED-THING-2268> #k<COLORED-THING-2269>
    #k<COLORED-THING-2270> #k<COLORED-THING-2271>
    #k<COLORED-THING-2272> #k<COLORED-THING-2273>
    #k<COLORED-THING-2274> #k<COLORED-THING-2275>
    #k<COLORED-THING-2276> #k<COLORED-THING-2277>
    #k<COLORED-THING-2278> #k<COLORED-THING-2279>
    #k<COLORED-THING-2280> #k<COLORED-THING-2281>
    #k<COLORED-THING-2282> #k<COLORED-THING-2283>
    #k<COLORED-THING-2284>
  :INTERNAL =  "Some information"
  :Y =  20
  :X =  10
  :COLOR =  :BLUE
  :WIDTH =  12.5
  :IS-A =  #k<TOP-OBJECT>
```

Using the system-supplied default control object reduces the clutter in the `:is-a-inv` slot, and also eliminates printing of schemata with the special #k<> convention:

```
(ps colored-thing :control :default)
COLORED-THING
  :WIDTH =  12.5
  :IS-A-INV =  COLORED-THING-2265 COLORED-THING-2266
    COLORED-THING-2267 COLORED-THING-2268
    COLORED-THING-2269 ...
  :INTERNAL =  "Some information"
  :Y =  20
  :X =  10
  :COLOR =  :BLUE
  :IS-A =  TOP-OBJECT
```

We can make things even better by using the object itself to inherit the control slots. We add sorting information and a global limit to the number of elements to be printed for

each list. We do this at the highest level in the hierarchy, so that every object can inherit
the information:

```
(s-value top-object :global-limit-values 3)
(s-value top-object :sorted-slots
  '(:is-a :color :x :y))

(ps colored-thing)
;; prints out:
COLORED-THING
  :IS-A =  TOP-OBJECT
  :COLOR =  :BLUE
  :X =  10
  :Y =  20
  :IS-A-INV =  COLORED-THING-2265 COLORED-THING-2266
     COLORED-THING-2267 ...
  List of ignored slots:  WIDTH INTERNAL
```

4.10.15 Slot Printing Functions

It is 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.

4.10.16 Control Variables

The following variable can be set globally to achieve the same effect as the slot `:print-as-structure` described above:

kr::`*print-as-structure*`   [Special Variable]
> This variable may be used to determine whether schema names are printed with the notation `#k<name>` (the default) or simply as `name`. The former notation is more perspicuous, since it makes it immediately clear which objects are KR schemata. The second notation is more compact, and is obtained by setting kr::`*print-as-structure*` to `nil`.
>
> In addition to kr::`*print-as-structure*`, other special variables can be used to control the behavior of the system. The following variables are used to control what debugging information is printed. The default settings are such that very little debugging information is printed.

kr::`*print-new-instances*`   [Special Variable]
> This variable controls whether a notification is printed when `create-schema` or create-instance are compiled from a file. The message is printed when kr::*PRINT-NEW-INSTANCES* is `t` (the default), and may be useful to determine how far into a file compilation has progressed. Setting this variable to `nil` turns off the notification.

kr::`*warning-on-null-link*`   [Special Variable]
> This variable controls whether a notification is printed when a null link is encountered during the evaluation of a formula. When the variable is `nil` (the default), the stale value of the formula is simply reused without any warning. Setting the variable to `t` causes a notification describing the situation to be printed; the formula then returns the stale value, as usual.

kr::`*warning-on-circularity*`   [Special Variable]
> This variable controls whether a notification is printed when a circularity is detected during formula evaluation. When the variable is `nil` (the default), no warnings are generated. Setting the variable to `t` causes a notification describing the situation to be printed.

kr::`*warning-on-create-schema*`   [Special Variable]
> This variable controls whether a notification is printed when `create-instance` creates an object that has the same name as an old object, and the old object is destroyed. If `t` (the default), then a warning will be printed when an object is redefined.

kr::`*warning-on-evaluation*`   [Special Variable]
> This variable controls whether a warning is printed whenever a formula is evaluated. If its value is non-`nil`, then a warning will describe the object, slot, and name of any formula that is evaluated. This can be useful for debugging.

`kr::*store-lambdas*`   [Special Variable]

> The variable `kr::*store-lambdas*` (default [No value for "t"]) may be set to `nil`
> to prevent the expression of a formula from being stored in the formula itself. This
> produces smaller run-time programs, but because the expression is lost it may be
> impossible to dump a set of objects to a file using `opal:write-gadget`.

4.11 An Example

This section develops a more comprehensive example than the ones so far, and highlights
the operations with which most users of the system should be familiar. Note that this
example does not use graphical operations at all; refer to the Opal manual for examples of
graphical applications.

We will first construct a simple example of constraints and show how constraints work.
The example uses constraints to compute the equivalence between a temperature expressed
in degrees Celsius and in degrees Fahrenheit. This first part also illustrates how KR deals
with circular chains of constraints.

The second part of the example shows simple object-oriented programming techniques,
and illustrates many of the dynamic capabilities of KR. Note that this example is purely
indicative of a certain way to program in KR, and different programming styles would be
possible even for such a simple task.

4.11.1 The Degrees Schema

First of all, we will create the `degrees` schema as a demonstration of constraints in KR.
This is a schema with two slots, namely, `:celsius` and `:fahrenheit`. The schema can be
created with the following call to `create-schema`:

```
(create-schema 'DEGREES
  (:fahrenheit (o-formula (+ (* (gvl :celsius) 9/5) 32)
                          32))
  (:celsius (o-formula (* (- (gvl :fahrenheit) 32) 5/9)
                       0)))
;; and now:
(gv DEGREES :celsius)    ==> 0
(gv DEGREES :fahrenheit) ==> 32
```

Each of the two slots contains a formula. The formula in the `:celsius` slot, for instance,
indicates that the value is computed from the value in the `:fahrenheit` slot, using the
appropriate expression. The initial value, moreover, is 32. The formula in the `:fahrenheit`
slot, similarly, is constrained to be a function of the value in the `:celsius` slot and is
initialized with the value 0.

It is clear that this example involves a circular chain of constraints. The value of
`:celsius` depends on the value of `:fahrenheit`, which itself depends on the value of
`:celsius`. This circularity, however, is not a problem for KR. The system is able to detect
such circularities and reacts appropriately by stopping value propagation when necessary.

Consider, for instance, setting the value of the `:celsius` slot:

```
(s-value DEGREES :celsius 20)
(gv DEGREES :celsius)          ==> 20
(gv DEGREES :fahrenheit)       ==> 68
```

As the example shows, KR propagates the change to the `:fahrenheit` slot, which is given the correct value. Similarly, if we modify the value in the `:fahrenheit` slot, we have correct propagation in the opposite direction:

```
(s-value DEGREES :fahrenheit 212)
(gv DEGREES :celsius)          ==> 100
(gv DEGREES :fahrenheit)       ==> 212
```

4.11.2 The Thermometer Example

Let us now build an example of a thermometer from which one can read the temperature in both degrees Celsius and Fahrenheit, and show a more extensive application of constraints. This example also shows the role of inheritance in object-oriented programming, and a simple method combination.

We begin with `temperature-device`, a simple prototype which contains a formula to translate degrees Celsius into Fahrenheit (the formula is the same we used in the previous example) and a `:print` method which prints out both values:

```
(create-schema 'TEMPERATURE-DEVICE
  (:fahrenheit
    (o-formula (+ (* (gvl :celsius) 9/5) 32) 32)))


(define-method :print TEMPERATURE-DEVICE (schema)
  (format t "Current temperature: ~,1F C (~,1F F)~%"
  (gv schema :celsius)
  (gv schema :fahrenheit)))
```

We now create two objects to hold the current temperature outdoors and indoors, and we create the schema `thermometer`,

which will be the basic building block for other thermometers:

```
(create-schema 'OUTSIDE
  (:celsius 10))

(create-schema 'INSIDE
  (:celsius 21))

(create-instance 'THERMOMETER TEMPERATURE-DEVICE
  (:celsius (o-formula (gvl :location :celsius))))
```

Note that `thermometer` can act as a prototype, since it provides a formula which constrains the value of the `:celsius` slot to follow the value of the `:celsius` slot of a particular location. Thermometer schemata created as instances of `thermometer` will then simply track the value of temperature at the location with which they are associated. Note that instances of `thermometer` inherit the `:print` method from `temperature-device`.

```
(create-instance 'TH1 THERMOMETER
```

```
    (:location outside))

  (create-instance 'TH2 THERMOMETER
   (:location inside))

  (kr-send TH2 :print TH2)
  ;; prints out:
  Current temperature: 21.0 C (69.8 F)

  (kr-send TH1 :print TH1)
  ;; prints out:
  Current temperature: 10.0 C (50.0 F)
```

Since the temperature in the `outside` schema is 10, and thermometer `th1` is associated with `outside`, it prints out the current temperature outside. Changing the slot `:location` of `th1` to `inside` would automatically change the temperature reading, because of the dependency built into the formula in that slot.

We now want to specialize the `thermometer` in order to provide a new kind of thermometer that keeps track of minimum and maximum temperature, as well as the current temperature. We do this by creating an instance, `min-max-thermometer`, which inherits all the features of `thermometer` and defines two new formulas for computing minimum and maximum temperatures. Note the initial values in the formulas. Also, we create an instance of `min-max-thermometer` named `min-max`, and send it the `:print` message.

```
    (create-instance 'MIN-MAX-THERMOMETER THERMOMETER
      (:min (o-formula (min (gvl :min)
            (gvl :location :celsius))
       100))
      (:max (o-formula (max (gvl :max)
            (gvl :location :celsius))
       -100)))

    (create-instance 'MIN-MAX MIN-MAX-THERMOMETER
      (:location outside))

  (kr-send MIN-MAX :print MIN-MAX)
  ;; prints out:
  Current temperature: 10.0 C (50.0 F)
```

The `:print` method inherited from `temperature-device` is not sufficient for our present purpose, since it does not show minimum and maximum temperatures. We thus specialize the `:print` method, but we still use the default `:print` method to print out the current values. Let us specialize the method, print out the current status, change the temperature outside a few times, and then print out the status again:

```
    (define-method :print MIN-MAX-THERMOMETER (schema)
      ;; print out temperature, as before
      (call-prototype-method schema)
      ;; print out minimum and maximum readings.
      (format t "Minimum and maximum: ~,1F  ~,1F~%"
```

```
   (gv schema :min)
   (gv schema :max)))

(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 10.0 C (50.0 F)
Minimum and maximum: 10.0  10.0

(s-value OUTSIDE :celsius 14)
(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 14.0 C (57.2 F)
Minimum and maximum: 10.0  14.0

(s-value OUTSIDE :celsius 12)
(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 12.0 C (53.6 F)
Minimum and maximum: 10.0  14.0
```

Note that the :fahrenheit slot in any of these schemata can be accessed normally, and the constraints keep it up to date at all times:

```
(gv MIN-MAX :fahrenheit)  ==> 268/5 (53.6)
```

Finally, we can add a method to reset the minimum and maximum temperature, in order to start a new reading. This is shown in the next fragment of code:

```
(define-method :reset MIN-MAX-THERMOMETER (schema)
  (s-value schema :min (gv schema :celsius))
  (s-value schema :max (gv schema :celsius)))

(kr-send MIN-MAX :reset MIN-MAX)  ; reset min, max

(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 12.0 C (53.6 F)
Minimum and maximum: 12.0  12.0

(s-value OUTSIDE :celsius 14)

(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 14.0 C (57.2 F)
Minimum and maximum: 12.0  14.0
```

Other choices of programming style would have been possible, ranging from entirely object-oriented (i.e., without using constraints at all) to entirely demon-based.

4.12 Summary

KR provides excellent performance and three powerful paradigms: object-oriented programming, knowledge representation, and constraint maintenance. The system is designed for high performance and has a very simple program interface, which makes it easy to learn and easy to use.

The object-oriented programming component of KR is based on the prototype-instance paradigm, which is more flexible than the class-instance paradigm. Prototypes are simply objects from which other objects (called instances) may inherit values or methods. This relationship is completely dynamic, and an object can be made an instance of a different prototype as needed. Object methods are implemented as procedural attachments which are stored in an object's slots. Methods are inherited through the usual mechanism.

The knowledge representation component of KR offers multiple inheritance and user-defined relations. This component provides completely dynamic specification of a network's characteristics: inheritance, for example, is determined through user-specified relations, which the user may modify at run-time as needed. The performance of this component is very good and compares favorably with that of basic Lisp data structures. Inheritance, in particular, is efficient enough to provide the basic building block across a wide variety of application programs.

The constraint maintenance component of KR provides integrated, efficient constraint maintenance and is implemented through formulas, i.e, expressions which compute the value of a slot based on the values in other slots. Constraint maintenance uses lazy evaluation and value caching to yield excellent performance in a completely transparent way. Constraint maintenance is totally integrated with the rest of the system and can be used even without any knowledge of its internal details. The same access functions, in particular, work on both regular values and on values which are constrained by formulas.

In spite of its power, KR is small and simple. This makes it easy to maintain and extend as needed, and also makes it ideally suited for experimentation on efficient knowledge representation. The system is entirely written in portable Common Lisp and can run efficiently on any machine which supports the language. These features make KR an attractive foundation for a number of applications which use a combination of frame-based knowledge representation, object-oriented programming, and constraint maintenance.

5 Opal: The Garnet Graphical Object System

by Andrew Mickish, Brad A. Myers, David Kosbie, Richard McDaniel, Edward Pervin, Matthew Goldberg

10 May 2020

5.1 Abstract

This chapter is a refence for the graphical object system used by the Garnet project, which is called Opal. "Opal" stands for the **O**bject **P**rogramming **A**ggregate **L**ayer. Opal makes it very simple to create and manipulate graphical objects. In particular, Opal automatically handles object redrawing when properties of objects are changed.

5.2 Introduction

This document is the reference chapter for the Opal graphical object system. Opal, which stands for the **O**bject **P**rogramming **A**ggregate **L**ayer, is being developed as part of the Garnet project (garnet). The goal of Opal is to make it easy to create and edit graphical objects. To this end, Opal provides default values for all of the properties of objects, so simple objects can be drawn by setting only a few parameters. If an object is changed, Opal automatically handles refreshing the screen and redrawing that object and any other objects that may overlap it. The algorithm used to handle the automatic update is documented in *Vander Zanden 89*. Objects in Opal can be connected together using *constraints*, which are relations among objects that are declared once and automatically maintained by the system. An example of a constraint is that a line must stay attached to a rectangle. Constraints are discussed in Section 3.1 [Garnet Tutorial], page 41, and ⟨undefined⟩ [KR chapter], page ⟨undefined⟩.

Opal is built on top of the `gem` module, which is the **G**raphics and **E**vents **M**odule that refers to machine-specific functions. `gem` provides an interface to both X Windows and the Macintosh QuickDraw system, so applications implemented with Opal objects and functions will run on either platform without modification.

Opal is known to work in virtually any Common Lisp environment on many different machines (See Section 8.31.4 [Overview], page 482). Opal will also work with any window manager on top of X11, such as `uwm`, `twm`, `awm`, etc. Additionally, Opal provides support for color and gray-scale displays.

Within the Garnet toolkit, Opal forms an intermediary layer. It uses facilities provided by the KR object and constraint system *Giuse 89*, and provides graphical objects that comprise the higher level gadgets. To use Opal, the programmer should be familiar with the ideas of objects and constraints presented in the Section 2.1 [On-line Tour Through Garnet], page 23, and Section 3.1 [Garnet Tutorial], page 41. Opal does not handle any input from the keyboard or mouse. That is handled by the separate *Interactors* module. On top of Opal is also the *Aggregadgets* module which makes it significantly easier to create groups of objects. A collection of pre-defined interaction techniques, such as menus, scroll bars, buttons, and sliders, is provided in the Garnet Gadget set which, of course, use *Opal*, *Interactors*, and *Aggregadgets*.

The highest level of Garnet, built using the toolkit, contains the graphical construction tools that allow significant parts of application graphics to be created without programming.

The most sophisticated tool is *Lapidary*. When *Lapidary* is used, the programmer should rarely need to write code that calls *Opal* or any other part of the toolkit.

5.3 Overview of Opal

5.3.1 Basic Concepts

The important concepts in Opal are *windows*, *objects*, and *aggregates*.

X11 and *Macintosh QuickDraw* both allow you to create windows on the screen. In *X* they are called "*drawables*", and in *QuickDraw* they are called "views". An *Opal* window is a schema that contains pointers to these machine-specific structures. Like in *X11* and *QuickDraw*, Opal windows can be nested inside other windows (to form "sub-windows"). Windows clip all graphics so they do not extend outside the window's borders. Also, each window forms a new coordinate system with $(0,0)$ in the upper left corner. The coordinate system is one-to-one with the pixels on the screen (each pixel is one unit of the coordinate system). Garnet windows are discussed fully in section [windows], page 209.

The basics of object-oriented programming are beyond the scope of this chapter. The *objects* in *Opal* use the *KR* object system *Giuse 89*, and therefore operate as a prototype-instance model. This means that each object can serve as a prototype (like a class) for any further instances; there is (almost) no distinction between classes and instances. Each graphic primitive in *Opal* is implemented as an object. When the programmer wants to cause something to be displayed in *Opal*, it is necessary to create instances of these graphical objects. Each instance remembers its properties so it can be redrawn automatically if the window needs to be refreshed or if objects change.

An *aggregate* is a special kind of Opal object that holds a collection of other objects. Aggregates can hold any kind of graphic object including other aggregates, but an object can only be in one aggregate at a time. Therefore, aggregates form a pure hierarchy. The objects that are in an aggregate are called *components* of that aggregate, and the aggregate is called the *parent* of each of the components. Each window has associated with it a top-level aggregate. All objects that are displayed in the window must be reachable by going through the components of this aggregate (recursively for any number of levels, in case any of the components are aggregates themselves).

The prototype inheritance hierarchy for all graphical objects in *Opal* is shown in Figure [ObjectSchemata], page 150.

```
OPAL::WINDOW
:left 0
:top 0
:width 355
:height 277

INTER:INTERACTOR-WINDOW

OPAL:TEXT
:width #k<F51>
:height #k<F52>
:string ""
:font opal:default-font
:fill-background-p NIL
:justification :left

OPAL:BITMAP
:width #k<F32>
:height #k<F31>
:filling-style opal:default-filling-style

OPAL:PIXMAP
OPAL:HOURGLASS-CURSOR
OPAL:ARROW-CURSOR

OPAL:GRAPHICAL-OBJECT
:width 20
:height 20
:draw-function :copy
:line-style opal:default-line-style
:filling-style NIL
:select-outline-only NIL
:hit-threshold 0

OPAL:MULTIPOINT
:point-list (18 10 ...)
:left #k<F30>
:top #k<F29>
:width #k<F28>
:height #k<F27>

OPAL:POLYLINE

OPAL:ARROWHEAD
:from-x 0
:from-y 0
:head-x 0
:head-y 0
:length 10
:diameter 10
:open-p T

OPAL:VIEW-OBJECT
:left 0
:top 0
:width 0
:height 0
:visible #k<F16>

OPAL:ARC
:angle1 0
:angle2 (/ PI 4)

OPAL:CIRCLE
OPAL:OVAL

OPAL:RECTANGLE

OPAL:ROUNDTANGLE
:radius :small

OPAL:LINE
:x1 0   :x2 0
:y1 0   :y2 0
:left #k<F25>
:top #k<F24>
:width #k<F23>
:height #k<F24>
:line-p T

OPAL:MULTIFONT-TEXT
:initial-text ""
:word-wrap-p NIL
:text-width 300
:fill-background-p T
:draw-function :copy
:show-marks NIL

OPAL:AGGRELIST

OPAL:AGGREGATE
:left #k<F21>
:top #k<F20>
:width #k<F19>
:height #k<F18>

OPAL:AGGREGADGET       OPAL:AGGREGRAPH
```
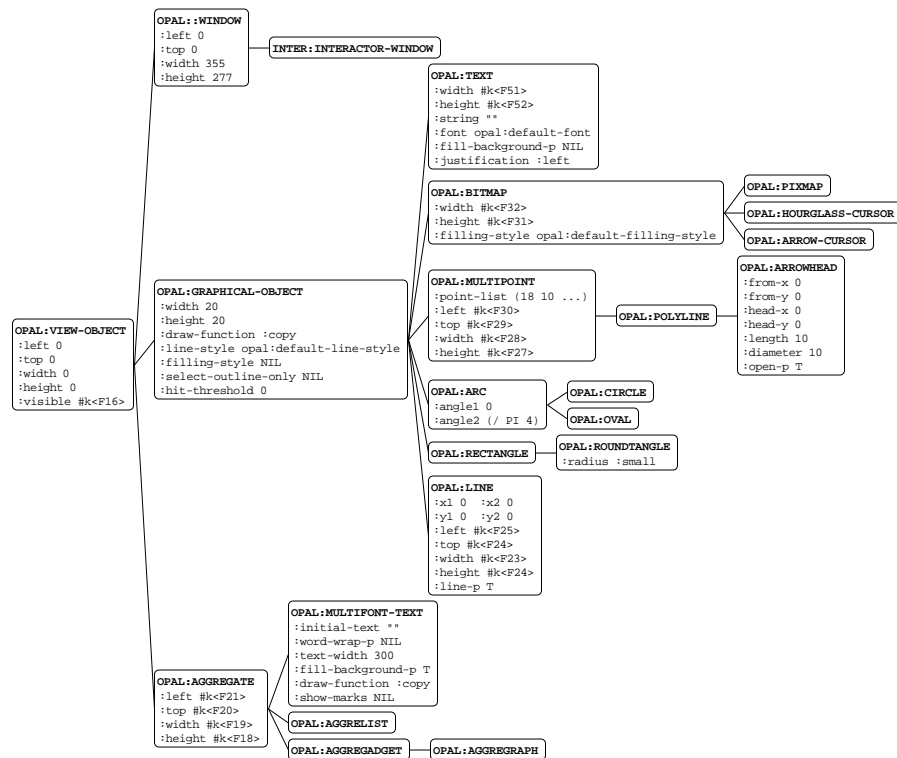
Figure 5.1: The objects in Opal and their slots. Each object also inherits slots from its prototype (the object to its left). The default values for the slots are shown. Those with values like #k<F21> have formulas in them (See Section 3.1 [Garnet Tutorial], page 41, and ⟨undefined⟩ [KR chapter], page ⟨undefined⟩).

5.3.2 The Opal Package

Once Garnet is loaded, all the graphical objects reside in the opal package. We recommend that programmers explicitly reference names from the opal package, for example: opal:rectangle, but you can also get complete access to all exported symbols by doing a (use-package :opal). All of the symbols referenced in this document are exported from opal, unless otherwise stated.

5.3.3 Simple Displays

An important goal of Opal is to make it significantly easier to create pictures, hiding most of the complexity of the X11 and QuickDraw graphics models. Therefore, there are appropriate defaults for all properties of objects (such as the color, line-thickness, etc.). These only need to be set if the user desires to. All of the complexity of the X11 and QuickDraw graphics packages is available to the Opal user, but it is hidden so that you do not need to deal with it unless it is necessary to your task.

To get the string "Hello world" displayed on the screen (and refreshed automatically if the window is covered and uncovered), you only need the following simple program:

```
(use-package :kr)

;; Create a small window at the upper left corner of the screen
(create-instance 'win inter:interactor-window
  (:left 10)(:top 10)
  (:width 200)(:height 50))

;; create an aggregate for the window
(s-value win :aggregate (create-instance 'agg opal:aggregate))

;; create the string
(create-instance 'hello opal:text
  (:left 10)(:top 20)
  (:string "Hello World"))

(opal:add-component agg hello) ; add the string to the aggregate

(opal:update win) ; cause the window and string to be displayed
```

Opal also strives to make it easy to change the picture. To change the x position of the rectangle only requires setting the value of the :left slot; Opal handles the refresh:

```
(s-value hello :left 50)  ; change the position

(opal:update win) ; cause the change to be visible
```

Note that the programmer never calls *"draw"* or *"erase"* methods on objects. This is a significant difference from other graphical object systems. Opal causes the objects to be drawn and erased at the appropriate times automatically.

⟨undefined⟩ [Specific Graphical Objects], page ⟨undefined⟩, and Figure 5.4 present all the kinds of objects available in Opal.

5.3.4 Object Visibility

Objects are visible if and only if their :visible slot is non-nil and they are a component of a visible aggregate that (recursively) is attached to a window. (Aggregates are discussed in the chapter [aggregates], page 202.) Therefore, to make a single object **in**visible, its :visible slot can be set to nil. To make it visible again, it is only necessary to set the :visible slot to t. Alternatively, the object can be removed from its aggregate to make it invisible.

Of course an object with a non-nil :visible slot in a visible aggregate hierarchy might be completely obscured behind another object so it cannot be seen.

Every object has a default formula in its :visible slot that depends on the visibility of the its *parent* (the *parent* is the aggregate that it is in). Therefore, to make an entire aggregate and all its components invisible, it is only necessary to set the :visible slot of the aggregate. All the components will become invisible (in this case, it is important that the components have the default formula in their :visible slot).

If you provide a specific value or formula for the `:visible` slot to override the default formula, it is important that this value be `nil` if the object's parent aggregate is not visible. Otherwise, routines such as `point-in-gob` may report that a point is inside the object, even though the object is invisible.

For example, if you want the `:visible` slot of an object to depend on its own `:selected` slot, you should additionally constrain it to depend on the visibility of its parent:

```
(s-value obj :visible (o-formula (if (gvl :parent :visible)
                                     (gvl :selected))))
```

5.3.5 View Objects

At the top of the class hierarchy is the class `opal:view-object`.

```
(create-instance 'opal:view-object nil
  (:left 0)
  (:top 0)
  (:width 0)
  (:height 0)
  (:visible (o-formula ...))
  ...)
```

Each view object has a bounding box as defined by the left, top corner and a width and height. The `:left`, `:top`, `:width`, and `:height` slots describe the bounding box for the object. Coordinates are given as non-negative fixnums, so any formulas must apply `floor` or `round` to all values that could generate floating point or ratio values. In particular, be careful using `"/"` for division, because that generates ratios or floats which are not legal values.

With the exception of windows, coordinates of objects are relative to the window in which the object appears. (If the window in which an object appears has borders, then the coordinates of the object are relative to the *inner* edges of the borders.) Windows coordinates are given in the coordinate system of the parent of the window, or in the case of top level windows, given in screen coordinates.

5.3.6 Read-Only Slots

There are many slots in graphical objects, windows, and interactors that are set internally by Garnet and should never be set by users. For example, the `:parent`, `:window`, and `:components` slots of graphical objects are set automatically whenever the objects are added to an aggregate using `opal:add-component`, and should not be set chapterly.

All public slots that are intended to be read-only are labeled as such in their object's definitions. Internal slots of an object (used for data or calculations) that are not documented should be considered read-only. Setting these slots "temporarily" or during initialization can lead to insidious errors at run-time.

5.3.7 Different Common Lisps

Running Opal under different implementations of Common Lisp should be almost the same. The differences in the locations of files, such the Opal binary files, and the cursors, bitmaps and fonts, are all handled in the top level `garnet-loader` file, which defines variables for the locations of the files.

An important difference among Lisp interpreters is the `main-event-loop`. In CMU Common Lisp, there is a process running in the background that allows interactors to always run with automatic refresh of Garnet windows.[1] In Allegro, Lucid, and LispWorks, Garnet creates its own `main-event-loop` process in the background that does the same thing. Some Lisp interpreters have problems running this process in the background, and you may have to call `inter:main-event-loop` by hand in order to run the interactors. Consult the Interactors chapter for directions on how to control the `main-event-loop` process.

5.4 Slots of All Graphical Objects

This section discusses properties shared by all graphical objects.

```
(create-instance 'opal:Graphical-Object opal:view-object
  (:left 0)    (:top 0)
  (:width 20) (:height 20)
  (:line-style opal:default-line-style)
  (:filling-style nil)
  (:draw-function :copy)
  (:select-outline-only nil)
  (:hit-threshold 0)
  ...)
```

5.4.1 Left, top, width and height

Graphical objects are objects with graphical properties that can be displayed in Garnet windows. They inherit the `:left`, `:top`, `:width` and `:height` slots from `view-objects`, of course.

5.4.2 Line style and filling style

The `:line-style` and `:filling-style` slots hold instances of the `opal:line-style` prototype and the `opal:filling-style` prototype, respectively. These objects parameterize the drawing of graphical objects. Graphical objects with a `:line-style` of `:filling-style` of `:line-style` and `:filling-style` control various parameters of the outline and filling when the object is drawn. Appropriate values for the `:line-style` and `:filling-style` slots are described below in the chapter Section 5.6.1 [Graphic Qualities], page 160.

5.4.3 Drawing function

The value of the `:draw-function` slot determines how the object being drawn will affect the graphics already in the window. For example, even though a line may be "black", it could cause objects that it covers to be "whited-out" if it is drawn with a `:clear` draw-function. A list of all allowed values for the `:draw-function` slot is included in Figure Figure 5.2.

Every time an object is displayed in a window, its drawn bits interact with the bits of the pixels already in the window. The way the object's bits (the source bits) interact

---

[1] Automatic refresh while an interactor is running is different from updating a window after you chapterly make a change with `s-value`. Unless changes are made by the interactors, you will still have to call `opal:update` to see the graphics change.

with the window's current bits (the destination bits) depends on the draw function. The `:draw-function` is the bitwise function to use in calculating the resulting bits. Opal insures that black pixels pretend to be "1" and white pixels pretend to be "0" for the purposes of the drawing functions (independent of the values of how the actual display works). Therefore, when using the colors black and white, you can rely on `:or` to always add to the picture and make it more black, and `:and` to take things away from the picture and make it more white.

Results of draw-functions on colors other than black and white tend to be random. This is because X11 and Mac QuickDraw initialize the colormap with colors stored in an arbitrary order, and a color's index is unlikely to be the same between Garnet sessions. So performing a logical operation on two particular colors will yield a different resulting color in different Garnet sessions.

One of the most useful draw functions is `:xor`, which occurs frequently in feedback objects. If a black rectangle is XOR'ed over another object, the region under the rectangle will appear in inverse video. This technique is used in the `gg:text-button`, and many other standard Garnet gadgets.

A fundamental limitation of the PostScript language prevents it from rendering draw functions properly. If `opal:make-ps-file` (see chapter [printing], page 218) is used to generate a PostScript file from a Garnet window, the draw functions used in the window will be ignored in the printed image. Usually the graphics in the window can be reimplemented without using draw-functions to get the same effect, so that the picture generated by `opal:make-ps-file` matches the window exactly.

draw-function	function
:clear	0
:set	1
:copy	src
:no-op	dst
:copy-inverted	(not src)
:invert	(not dst)
:or	src or dst
:and	src and dst
:xor	src xor dst
:equiv	(not src) xor dst
:nand	(not src) or (not dst)
:nor	(not src) and (not dst)
:and-inverted	(not src) and dst
:and-reverse	src and (not dst)
:or-inverted	(not src) or dst
:or-reverse	src or (not dst)

Figure 5.2: Allowed values for the :draw-function slot and their logical.

5.4.4 select-outline-only, hit-threshold, and pretend-to-be-leaf

The :select-outline-only, :hit-threshold, :pretend-to-be-leaf, and :visible slots are used by functions which search for objects given a rectangular region or an (x, y) coordinate (see sections [rect-regions], page 205, and [querying-children], page 204). If the :select-outline-only slot is non-only report hits only on or near the outline of the object. Otherwise, the object will be sensitive over the entire region (inside and on the outline). The :select-outline-only slot defaults to nil.

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. The default value of `:hit-threshold` is 0.

Note: it is often necessary to set the `:hit-threshold` slot of all aggregates *above* a target object; if an event occurs "outside" of an aggregate, then the `point-in-gob` methods will not check the components of that aggregate. The function `opal:set-aggregate-hit-threshold` (see section [agg-class], page 202) can simplify this procedure.

When an aggregate's `:pretend-to-be-leaf` slot contains the value `leaf-objects-in-rectangle` will treat that aggregate as a leaf object (even though the aggregate has components). This might be useful in searching for a button aggregate in an aggrelist of buttons.

5.5 Methods on All `view-objects`

There are a number of methods defined on all subclasses of `opal:view-object`. This section describes these methods and other accessors defined for all graphical objects.

5.5.1 Standard Functions

The various slots in objects, like `:left`, `:top`, `:width`, `:height`, `:visible`, etc. can be set and accessed using the standard `s-value` and `gv` functions and macros. Some additional functions are provided for convenience in accessing and setting the size and position slots. Some slots of objects should not be set (although they can be accessed). This includes the `:left`, `:top`, `:width,` and `:height` of lines and polylines (since they are computed from the end points), and the components of aggregates (use the `add-component` and `remove-component` functions).

`opal:point-in-gob` *graphical-object x y*   [Method on `view-object`]
    This routine determines whether the point (x, y) is inside the graphical object ("gob" stands for graphical object). This uses an object-specific method, and is dependent on the setting of the `:select-outline-only` and `:hit-threshold` slots in the object as described above.

    The `:point-in-gob` methods for `opal:polyline` and `opal:arrowhead` actually check whether the point is inside the polygon, rather than just inside the polygon's bounding box. Additionally, the `:hit-full-interior-p` slot of a polygon controls which algorithm is used to determine if a point is inside it (see section ⟨undefined⟩ [polyline], page ⟨undefined⟩). If an object's `:visible` slot is `nil`, then `point-in-gob` will always return `nil` for that object.

`graphical-object` **&optional** *erase*   [Method on `opal:destroy`]
    This causes the object to be removed from an aggregate (if it is in one), and the storage for the object is deallocated. You can `destroy` any kind of object, including windows. If you destroy a window, all objects inside of it are automatically destroyed. Similarly, if you destroy an aggregate, all objects in it are destroyed (recursively). When you destroy an object, it is automatically removed from any aggregates it might be in and erased from the screen. If destroying the object causes you to go into

the debugger (usually due to illegal values in some slots), you might try passing in the *erase* parameter as `nil` to cause Opal to not erase the object from the window. The default for *erase* is `t`.

Often, it is not necessary to destroy individual objects because they are destroyed automatically when the window they are in is destroyed.

graphical-object *angle* **&optional** *center-x center-y*   [Method on `opal:rotate`]
The `rotate` method rotates *graphical-object* around (*center-x*, *center-y*)] by *angle* radians. It does this by changing the values of the controlling points (using `s-value`) for the object (e.g., the values for `:x1`, `:y1`, `:x2`, and `:y2` for lines). Therefore, it is a bad idea to call `rotate` when there are formulas in these slots. If *center-x* or *center-y* are not specified, then the geometric center of the object (as calculated by using the center of its bounding box) is used. Certain objects can't be rotated, namely Ovals, Arcs, Roundtangles, and Text. A rectangle that is rotated becomes a polygon and remains one even if it is rotated back into its original position.

5.5.2 Extended Accessor Functions

The following macros, functions and `setf` methods are defined to make it easier to access the slots of graphical objects.

When set, the first set of functions below only change the position of the graphical object; the width and height remain the same. The following are both accessors and valid place arguments for `setf`. These use `s-value` and `g-value` so they should not be used inside of formulas, use the `gv-`*xxx* forms below instead inside of formulas.

`opal:bottom` *graphical-object*	[Function]
`opal:right` *graphical-object*	[Function]
`opal:center-x` *graphical-object*	[Function]
`opal:center-y` *graphical-object*	[Function]

To use one of these in a `setf`, the form is

```
(setf (opal:bottom obj) new-value)
```

In contrast to the above accessors, the four below when set change the size of the object. For example, changing the top-side of an object changes the top and height of the object; the bottom does not change.

`opal:top-side` *graphical-object value*	[Macro]
`opal:left-side` *graphical-object value*	[Macro]
`opal:bottom-side` *graphical-object value*	[Macro]
`opal:right-side` *graphical-object value*	[Macro]

Opal also provides the following accessor functions which set up dependencies and should only be used inside of formulas. For more information on using formulas, see the example section and the KR document. These should not be used outside of formulas.

`opal:gv-bottom` *graphical-object*	[Function]
`opal:gv-right` *graphical-object*	[Function]
`opal:gv-center-x` *graphical-object*	[Function]

`opal:gv-center-y` *graphical-object*   [Function]

   The following functions should be used in the `:left` and `:top` slots of objects, respectively. The first returns the value for `:left` such that (`gv-center-x :self`) equals
(`gv-center-x object`).

`opal:gv-center-x-is-center-of` *object*   [Function]
`opal:gv-center-y-is-center-of` *object*   [Function]

   In more concrete terms, if you had two objects OBJ1 and OBJ2, and you wanted to
   constrain the `:left` of OBJ1 so that the centers of OBJ1 and OBJ2 were the same,
   you would say:

```
        (s-value obj1 :left (o-formula (opal:gv-center-x-is-center-of obj2)))
```

   The next group of functions are for accessing multiple slots simultaneously. These are
not `setf`'able.

`opal:center` *graphical-object* (*declare* (*values center-x center-y*))   [Function]
`opal:set-center` *graphical-object center-x center-y*   [Function]
`opal:bounding-box` *graphical-object*   [Function]
      (declare (values left top width height))

`opal:set-bounding-box` *graphical-object left top width height*   [Function]
`opal:set-position` *graphical-object left top*   [Function]
`opal:set-size` *graphical-object width height*   [Function]

5.6 Graphic Qualities

Objects that are instances of class `opal:graphic-quality` are used to specify a number of
related drawing qualities at one time. The `:line-style` and `:filling-style` slots present
in all graphical objects hold instances of `opal:line-style` and `opal:filling-style` objects. The `opal:line-style` object controls many parameters about how a graphical object's outline is displayed. Likewise, the `opal:filling-style` object controls how the filling
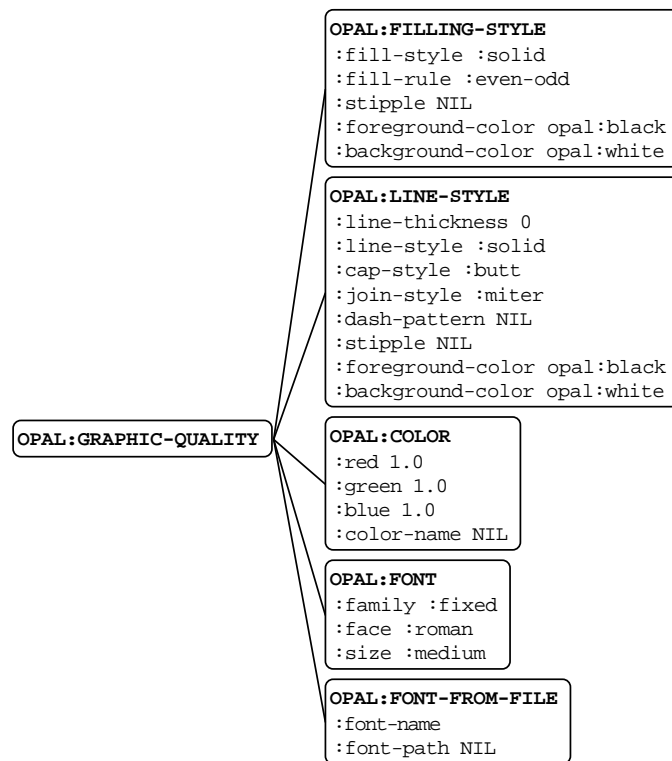of objects are displayed. Figure 14.25 shows the graphic qualities provided by Opal.

```
┌─────────────────────────────────────┐
│ OPAL:FILLING-STYLE                   │
│ :fill-style :solid                   │
│ :fill-rule :even-odd                 │
│ :stipple NIL                         │
│ :foreground-color opal:black         │
│ :background-color opal:white         │
└─────────────────────────────────────┘
┌─────────────────────────────────────┐
│ OPAL:LINE-STYLE                      │
│ :line-thickness 0                    │
│ :line-style :solid                   │
│ :cap-style :butt                     │
│ :join-style :miter                   │
│ :dash-pattern NIL                    │
│ :stipple NIL                         │
│ :foreground-color opal:black         │
│ :background-color opal:white         │
└─────────────────────────────────────┘
┌─────────────────────────────────────┐
│ OPAL:COLOR                           │
│ :red 1.0                             │
│ :green 1.0                           │
│ :blue 1.0                            │
│ :color-name NIL                      │
└─────────────────────────────────────┘
┌─────────────────────────────────────┐
│ OPAL:FONT                            │
│ :family :fixed                       │
│ :face :roman                         │
│ :size :medium                        │
└─────────────────────────────────────┘
┌─────────────────────────────────────┐
│ OPAL:FONT-FROM-FILE                  │
│ :font-name                           │
│ :font-path NIL                       │
└─────────────────────────────────────┘

OPAL:GRAPHIC-QUALITY
```

Figure 5.3: The graphic qualities that can be applied to objects.

The properties controlled by the `opal:line-style`, `opal:filling-style`, and `opal:font` objects are similar to PostScript's graphics state (described in section 4.3 in the PostScript Language Reference chapter) or the XLIB graphics context (described in the X Window System Protocol Manual). The Opal design is simpler since there

are appropriate defaults for all values and you only have to set the ones you are interested in. The `:line-style` slot in graphical objects holds an object that contains all relevant information to parameterize the drawing of lines and outlines. Similarly, the `:filling-style` controls the insides of objects. The `:font` slot appears only in text and related objects, and controls the font used in drawing the string.

Note: Although the properties of these graphic qualities can be changed after they are created, for example to make a font change to be italic, Garnet will not notice the change because the font object itself is still the same (i.e., the value of the `:font` slot has not changed). Therefore, line-styles, filling-styles and fonts should be considered read-only after they are created. You can make as many as you want and put them in objects, but if you want to change the property of an object, insert a *new* line-style, filling-style, or font object rather than changing the slots of the style or font itself. If a set of objects should share a changeable graphics quality, then put a formula into each object that calculates which graphic quality to use, so they will all change references together, rather than sharing a pointer to a single graphic quality object that is changed.

5.6.1 Color

A graphical quality called `opal:color` exists which is defined as:

```
(create-instance 'opal:color opal:graphic-quality
   (:constant '(:color-p))
   ;; Set during initialization according to the display - t if color, nil otherwise
   (:color-p ...)
   (:red 1.0)
   (:green 1.0)
   (:blue 1.0)
   (:color-name nil))
```

The following colors are exported from Opal. They are instances of `opal:color` with the appropriate values for their `:red`, `:green`, and `:blue` slots as shown:

```
opal:red    --- (:red 1.0)  (:green 0.0)  (:blue 0.0)
opal:green --- (:red 0.0)  (:green 1.0)  (:blue 0.0)
opal:blue   --- (:red 0.0)  (:green 0.0)  (:blue 1.0)
opal:yellow --- (:red 1.0)  (:green 1.0)  (:blue 0.0)
opal:purple --- (:red 1.0)  (:green 0.0)  (:blue 1.0)
opal:cyan   --- (:red 0.0)  (:green 1.0)  (:blue 1.0)
opal:orange --- (:red 0.75) (:green 0.25) (:blue 0.0)
opal:white --- (:red 1.0)  (:green 1.0)  (:blue 1.0)
opal:black --- (:red 0.0)  (:green 0.0)  (:blue 0.0)
```

The following objects are also instances of `opal:color`, with RGB values chosen to correspond to standard Motif colors:

```
opal:motif-gray
opal:motif-blue
opal:motif-green
opal:motif-orange
opal:motif-light-gray
opal:motif-light-blue
opal:motif-light-green
opal:motif-light-orange
```

Users can create any color they want by creating an object of type `opal:color`, and setting the `:red`, `:green` and `:blue` slots to be any real number between 0.0 and 1.0.

An `opal:color` can also be created using the `:color-name` slot instead of the `:red`, `:green,` and `:blue` slots. The `:color-name` slot takes a string such as "*pink*" or atom such as *'pink*. These names are looked up by the Xserver, and the appropriate color will be returned. Usually the list of allowed color names is stored in the file `/usr/misc/lib/rgb.txt` or `/usr/misc/.X11/lib/rgb.txt` or `/usr/lib/X11/rgb.txt`. However, if the Xserver does not find the color, an error will be raised. There is apparently no way to ask X whether it understands a color name. Thus, code that uses the `:color-name` slot may not be portable across machines. Note that the `:red, :green,` and `:blue` slots of the color are set automatically in color objects defined with names.

For example:

```
(create-instance 'fun-color opal:color (:color-name "papaya whip"))
```

The `:color-p` slot of `opal:color` is automatically set to `t` or `nil` depending on whether or not your screen is color or black-and-white (it is also `t` if the screen is gray-scale). This should not be set by hand. The Motif widget set contains formulas that change their display mode based on the value of `:color-p`.

5.6.2 `line-style` Class

```
(create-instance 'opal:line-style opal:graphic-quality
  (:maybe-constant '(:line-thickness :cap-style :join-style :line-style :dash-pattern
      :foreground-color :background-color :stipple))
  (:line-thickness 0)
  (:cap-style :butt)
  (:join-style :miter)
  (:line-style :solid)
  (:foreground-color opal:black)
  (:background-color opal:white)
  (:dash-pattern nil)
  (:stipple nil))

(create-instance 'opal:default-line-style opal:line-style
    (:constant t))
```

Before you read the sordid details below about what all these slots mean, be aware that most applications will just use the default line styles provided.

The following line-styles (except `opal:no-line`) are all instances of `opal:line-style`, with particular values for their `:line-thickness`, `:line-style`, or `:dash-pattern` slots. Except as noted, they are identical to `opal:default-line-style`. All of them are black.

`opal:No-Line`
          (nil)

`opal:thin-line`
          same as `opal:default-line-style`

`opal:line-0`
          same as `opal:default-line-style`

`opal:line-1`
          `:line-thickness` = 1

`opal:line-2`
          `:line-thickness` = 2

`opal:line-4`
          `:line-thickness` = 4

`opal:line-8`
          `:line-thickness` = 8

`opal:dotted-line`
          `:line-style` = :dash, and `:dash-pattern` = '(1 1)

`opal:dashed-line`
          `:line-style` = :dash, and `:dash-pattern` = '(4 4)

The following line-styles are all identical to `opal:default-line-style`, except that their `:foreground-color` slot is set with the appropriate instance of `opal:color`. For example, the `:foreground-color` slot of `opal:red-line` is set to `opal:red`.

`opal:red-line`
`opal:green-line`
`opal:blue-line`
`opal:yellow-line`
`opal:purple-line`
`opal:cyan-line`
`opal:orange-line`
`opal:white-line`

For each of the predefined line-styles above, you may not customize any of the normal parameters described below. These line-styles have been created with their `:constant` slot set to `t` for efficiency, which prohibits the overriding of the default values. You may use these line-styles as values of any `:line-style` slot, but you may not create customized instances of them. Instead, to create a thick red line-style, for example, you should create your own instance of `opal:line-style` with appropriate values for `:line-thickness`, `:foreground-color`, etc. See the examples at the end of this section.

The `:line-thickness` slot holds the integer line thickness in pixels. There may be a subtle difference between lines with thickness zero and lines with thickness one. Zero

thickness lines are free to use a device dependent line drawing algorithm, and therefore may be less aesthetically pleasing. They are also probably drawn much more efficiently. Lines with thickness one are drawn using the same algorithm with which all the thick lines are drawn. For this reason, a thickness zero line parallel to a thick line may not be as aesthetically pleasing as a line with thickness one.

For objects of the types `opal:rectangle`, `opal:roundtangle`, `opal:circle` and `opal:oval`, increasing the `:line-thickness` of the `:line-style` will not increase the `:width` or `:height` of the object; the object will stay the same size, but the solid black boundary of the object will extend *inwards* to occupy more of the object. On the other hand, increasing the `:line-thickness` of the `:line-style` of objects of the types `opal:line`, `opal:polyline` and `opal:arrowhead` will increase the objects' `:width` and `:height`; for these objects the thickness will extend *outward* on *both sides* of the line or arc.

The `:cap-style` slot (which is ignored by the Mac) describes how the endpoints of line segments are drawn:

Line :capstyle	Description
:butt	Square at the endpoint (perpendicular to the slope of the line) with no projection beyond.
:not-last	Equivalent to :butt, except that for :line-thickness 0 or 1 the final endpoint is not drawn.
:round	A circular arc with the diameter equal to the :line-thickness centered on the endpoint.
:projecting	Square at the end, but the path continues beyond the endpoint for a distance equal to half of the :line-thickness.

The `:join-style` slot (which is ignored by the Mac) describes how corners (where multiple lines come together) are drawn for thick lines as part of poly-line, polygon, or rectangle kinds of objects. This does not affect individual lines (instances of `opal:line`) that are part of an aggregate, even if they happen to have the same endpoints.

Line :join-style Result	
:miter	
	The outer edges of the two lines extend to meet at an angle.
:round	A circular arc with a diameter equal to the :line-thickness is drawn centered on the join point.
:bevel	
:butt	endpoint styles, with the triangular notch filled.

The `:foreground-color` slot contains an object of type `opal:color` which specifies the color in which the line will appear on a color screen. The default value is `opal:black`.

The `:background-color` slot contains an object of type `opal:color` which specifies the color of the "off" dashes of double-dash lines will appear on a color screen (see below). The default value is `opal:white`. It also specifies the color of the bounding box of a text object whose `:fill-background-p` slot is set to `t`.

The contents of the `:line-style` slot declare whether the line is solid or dashed. Valid values are `:solid`, `:dash` or `:double-dash`. With `:dash` only the on dashes are drawn, and nothing is drawn in the off dashes. With `:double-dash`, both on and off dashes are drawn; the on dashes are drawn with the foreground color (usually black) and the off dashes are drawn with the background color (usually white).

The `:dash-pattern` slot holds an (optionally empty) list of numbers corresponding to the pattern used when drawing dashes. Each pair of elements in the list refers to an on and an off dash. The numbers are pixel lengths for each dash. Thus a `:dash-pattern` of (1 1 1 1 3 1) is a typical dot-dot-dash line. A list with an odd number of elements is equivalent to the list being appended to itself. Thus, the dash pattern (3 2 1) is equivalent to (3 2 1 3 2 1).

Since Mac QuickDraw does not support drawing real dashed lines, Garnet simulates dashed lines on the Mac by drawing lines with a stippled pattern. There is only one stipple pattern available for this simulation, so lines whose `:line-style` is `:dash` or `:double-dash` have the same gray stipple. The `:dash-pattern` slot is ignored on the Mac. You can supply your own stipple for this simulation in the `:stipple` slot of the `line-style` object (see below).

The `:stipple` slot holds either `nil` or a `opal:bitmap` object with which the line is to be stippled. The `:foreground-color` of the line-style will be used for the "dark" pixels in the stipple pattern, and the `:background-color` will be used for the "light" pixels.

Some examples:

```
; black line of thickness 2 pixels
opal:line-2

; black line of thickness 30 pixels
(create-instance 'thickline opal:line-style (:line-thickness 30))

; gray line of thickness 5 pixels
(create-instance 'grayline opal:line-style
  (:line-thickness 5)
  (:stipple (create-instance nil opal:bitmap
              (:image (opal:halftone-image 50))))) ; 50% gray

; dot-dot-dash line, thickness 1
(create-instance 'dotdotdashline opal:line-style
  (:line-style :dash)
  (:dash-pattern '(1 1 1 1 3 1)))
```

5.6.3 Filling-Styles

```
(create-instance 'opal:Filling-Style opal:graphic-quality
```

```
    (:foreground-color opal:black)
    (:background-color opal:white)
    (:fill-style :solid)    ;; Transparent or opaque. See section ⟨undefined⟩
  [other-stipple-slots], page ⟨undefined⟩.
    (:fill-rule :even-odd)  ;; For self-intersecting polygons. See section ⟨un-
  defined⟩ [other-stipple-slots], page ⟨undefined⟩.
    (:stipple nil))              ;; The pattern. See section [custom-stipple],
  page 166.

    (create-instance 'opal:Default-Filling-Style opal:filling-style)
```

Before you read all the sordid details below about what all these slots mean, be aware that most applications will just use the default filling styles provided. There are two basic types of filling-styles: those that rely on stipple patterns to control their shades of gray, and those that are solid colors.

Stippled Filling-Styles

Stippled filling-styles rely on their patterns to control their color shades. The `:stipple` slot controls the mixing of the `:foreground-color` and `:background-color` colors, which default to `opal:black` and `opal:white`, respectively. Thus, the default stippled filling-styles are shades of gray, but other colors may be used as well. Here is a list of pre-defined stippled filling-styles:

`opal:no-fill`
    nil

`opal:black-fill`
    same as `opal:default-filling-style`

`opal:gray-fill`
    same as (`opal:halftone` 50)

`opal:light-gray-fill`
    same as (`opal:halftone` 25)

`opal:dark-gray-fill`
    same as (`opal:halftone` 75)

`opal:diamond-fill`
    a special pattern, defined with

`opal:make-filling-style`
    See section ⟨undefined⟩ [fancy-stipple], page ⟨undefined⟩.

Solid Filling-Styles

The second set of filling-styles are solid colors, and do not rely on stipples. For these filling-styles, the `:foreground-color` slot of the object is set with the corresponding instance of `opal:color`. For example, the `:foreground-color` slot of `opal:red-fill` is set with `opal:red`. Otherwise, these filling-styles are all identical to `opal:default-filling-style`.

```
opal:white-fill
opal:red-fill
opal:green-fill
opal:blue-fill
opal:yellow-fill
opal:purple-fill
opal:cyan-fill
opal:orange-fill


opal:motif-gray-fill
opal:motif-blue-fill
opal:motif-green-fill
opal:motif-orange-fill
opal:motif-light-gray-fill
opal:motif-light-blue-fill
opal:motif-light-green-fill
opal:motif-light-orange-fill
```

5.6.3.1 Creating Your Own Stippled Filling-Styles

The `:stipple` slot of a `filling-style` object is used to specify patterns for mixing the foreground and background colors. The `:stipple` slot is either `nil` or an `opal:bitmap` object, whose image can be generated from the `/usr/misc/.X11/bin/bitmap` Unix program (see section [bitmap-sec], page 184). Alternatively, there is a Garnet function supplied for generating halftone bitmaps to get various gray shades.

`opal:halftone` *percentage*   [Function]

> The `halftone` function returns an `opal:filling-style` object. The *percentage* argument is used to specify the shade of the halftone (0 is white and 100 is black). Its halftone is as close as possible to the varpercentage halftone value as can be generated. Since a range of *percentage* values map onto each halftone shade, two additional functions are provided to get halftones that are guaranteed to be one shade darker or one shade lighter than a specified value.

`opal:halftone-darker` *percentage*   [Function]
`opal:halftone-lighter` *percentage*   [Function]

> The `halftone-darker` and `halftone-lighter` functions return a stippled `opal:filling-style` object that is guaranteed to be exactly one shade different than the halftone object with the specified *percentage*. With these functions you are guaranteed to get a different darker (or lighter) `filling-style` object. Currently, there are 17 different halftone shades.

> Examples of creating rectangles that are: black, 25% gray, and 33% gray are:

```
(create-instance 'BLACKRECT opal:rectangle
   (:left 10)(:top 20)(:width 50)(:height 70)
   (:filling-style opal:black-fill))
(create-instance 'LIGHTGRAYRECT opal:rectangle
   (:left 10)(:top 20)(:width 50)(:height 70)
   (:filling-style opal:light-gray-fill))
```

```
(create-instance 'ANOTHERGRAYRECT opal:rectangle
   (:left 10)(:top 20)(:width 50)(:height 70)
   (:filling-style (opal:halftone 33)))
```

5.6.3.2 Fancy Stipple Patterns

Another way to create your own customized filling styles is to use the function `opal:make-filling-style`:

opal:make-filling-style *description* **&key** *from-file-p*   [Function]
        (*foreground-color* **opal:black**) (*background-color* **opal:white**)
    The *description* can be a list of lists which represent the bit-mask of the filling style, or may be the name of a file that contains a bitmap. The *from-file-p* parameter should be T if a filename is being supplied as the *description*.

    As an example, the filling-style `opal:diamond-fill` is defined by:

```
      (setq opal:diamond-fill
            (opal:make-filling-style
             '((1 1 1 1 1 1 1 1 1)
      (1 1 1 1 0 1 1 1 1)
      (1 1 1 0 0 0 1 1 1)
      (1 1 0 0 0 0 0 1 1)
      (1 0 0 0 0 0 0 0 1)
      (1 1 0 0 0 0 0 1 1)
      (1 1 1 0 0 0 1 1 1)
      (1 1 1 1 0 1 1 1 1)
      (1 1 1 1 1 1 1 1 1)))))
```

5.6.3.3 Other Slots Affecting Stipple Patterns

The `:fill-style` slot specifies the colors used for drawing the "off" pixels in the stippled pattern of filling-styles. The "on" pixels are always drawn with the `:foreground-color` of the filling-style.

Line `:fill-style`
        Color used for "off" pixels

`:solid`   Color in `:foreground-color`

`:stippled`
        Transparent

`:opaque-stippled`
        Color in `:background-color`

    The `:fill-rule` is either `:even-odd` or `:winding`. These are used to control the filling for self-intersecting polygons. For a better description of these see any reasonable graphics textbook, or the X11 Protocol Manual.

5.6.4 Fast Redraw Objects

When an interface contains one or more objects that must be redrawn frequently, the designer may choose to define these objects as fast redraw objects. Such objects could be

feedback rectangles that indicate the current selection, or text strings which are updated after any character is typed. Fast redraw objects are redrawn with an algorithm that is much faster than the standard update procedure for refreshing Garnet windows.

However, because of certain requirements that the algorithm makes on fast redraw objects, most objects in an interface are not candidates for this procedure. Primarily, fast redraw objects cannot be covered by other objects, and they must be either drawn with *xor*, or else are guaranteed to be over only a solid background. Additionally, aggregates cannot be fast-redraw objects; only instances of `opal:graphical-object` (those with their own `:draw` methods) can be fast-redraw objects.

To define an object as a fast redraw object, the `:fast-redraw-p` slot of the object must be set to one of three allowed values – `:redraw`, `:rectangle`, or `t`. These values determine how the object should be erased from the window (so that it can be redrawn at its new position or with its new graphic qualities). The following paragraphs describe the functions and requirements of each of these values.

`:redraw`   The object will be erased by drawing it a second time with the line style and filling style defined in the slots `:fast-redraw-line-style` and `:fast-redraw-filling-style`. These styles should be defined to have the same color as the background behind the object. Additionally, these styles should have the same structure as the line and filling styles of the object. For example, if the object has a line thickness of 8, then the fast redraw line style must have a thickness of 8 also. This value may be used for objects on color screens where there is a uniform color behind the object.

`:rectangle`
              The object will be erased by drawing a rectangle over it with the filling style defined in the slot `:fast-redraw-filling-style`. This filling style should have the same color as the background behind the object. Like `:redraw`, this value assumes that there is a uniform color behind the object. However, `:rectangle` is particularly useful for complicated objects like bitmaps and text, since drawing a rectangle takes less time than drawing these intricate objects.

`t`   In this case, the object must additionally have its `:draw-function` slot set to `:xor`. This will cause the object to be XOR'ed on top of its background. To erase the object, the object is just drawn again, which will cause the two images to cancel out. This value is most useful when the background is white and the objects are black (e.g., on a monochrome screen), and can be used with a feedback object that shows selection by inverse video.

5.7 Specific Graphical Objects

This chapter describes a number of specific subclasses of the `opal:graphical-object` prototype that implement all of the graphic primitives that can be displayed, such as rectangles, lines, text strings, etc.

For all graphical objects, coordinates are specified as fixnum quantities from the top, left corner of the window. All coordinates and distances are specified in pixels.

Most of these objects can be filled with a filling style, have a border with a line-style or both. The default for closed objects is that `:filling-style` is `nil` (not filled) and the `:line-style` is `opal:default-line-style`.

Note that only the slots that are not inherited from view objects and graphic objects are shown below. In addition, of course, all of the objects shown below have the following slots (described in the previous sections):

```
(:left 0)
(:top 0)
(:width 0)
(:height 0)
(:visible (o-formula ...))
(:line-style opal:default-line-style)
(:filling-style nil)
(:draw-function :copy)
(:select-outline-only nil)
(:hit-threshold 0)
```

Most of the prototypes in this section have a list of slots in their `:maybe-constant` slot, which generally correspond to the customizable slots of the object. This is part of the *constant slots* feature of Garnet which allows advanced users to optimize their Garnet objects by reusing storage space. Consult the KR chapter for documentation about how to take advantage of constant slots.

HINT: If you want a black-filled object, set the line-style to be `nil` or else the object will take twice as long to draw (since it draws both the border and the inside).

Figure 5.4 shows examples of the basic object types in Opal.
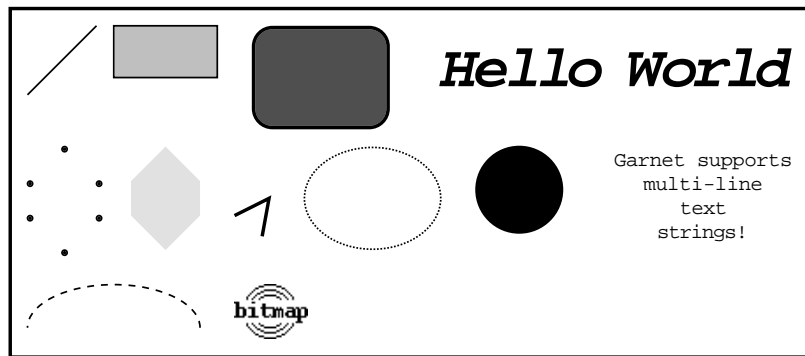
Figure 5.4: Examples of the types of objects supported by Opal: lines, rectangles, rounded rectangles, text, multipoints, polylines, arrowheads, ovals, circles, arcs, and bitmaps, with a variety of line and filling styles.

5.7.1 Line

```
(create-instance 'opal:line opal:graphical-object
  (:maybe-constant '(:x1 :y1 :x2 :y2 :line-style :visible))
  (:x1 0)
  (:y1 0)
  (:x2 0)
  (:y2 0))
```

The `opal:line` class describes an object that displays a line from $(:x1, :y1)$ to $(:x2, :y2)$. The `:left`, `:top`, `:width`, and `:height` reflect the correct bounding box for the line, but cannot be used to change the line (i.e., **do not set the** `:left`, `:top`, `:width`, **or** `:height` **slots**). Lines ignore their `:filling-style` slot.

5.7.2 Rectangles

```
(create-instance 'opal:rectangle opal:graphical-object
  (:maybe-constant '(:left :top :width :height :line-style :filling-style
      :draw-function :visible)))
```

The `opal:rectangle` class describes an object that displays a rectangle with top, left corner at $(:left, :top)$, width of `:width`, and height of `:height`.

5.7.2.1 Rounded-corner Rectangles

```
(create-instance 'opal:roundtangle opal:rectangle
  (:maybe-constant '(:left :top :width :height :radius :line-style
      :filling-style :draw-function :visible))
  (:radius 5))
```

Instances of the `opal:roundtangle` class are rectangles with rounded corners. Objects of this class are similar to rectangles, but contain an additional slot, `:radius`, which specifies the curvature of the corners. The values for this slot can be either `:small`, `:medium`, `:large`, or a numeric value interpreted as the number of pixels to be used. The keyword values do not correspond directly to pixels values, but rather compute a pixel value as a fraction of the length of the shortest side of the bounding box.

	:radius Fraction
:small	1/5
:medium	1/4
:large	1/3

Figure Figure 5.5 demonstrates the meanings of the slots of roundtangles. If the value of `:radius` is 0, the roundtangle looks just like a rectangle. If the value of `:radius` is more than half of the minimum of `:width` or `:height`, the roundtangle is drawn as if the value of `:radius` were half the minimum of `:width` and `:height`.
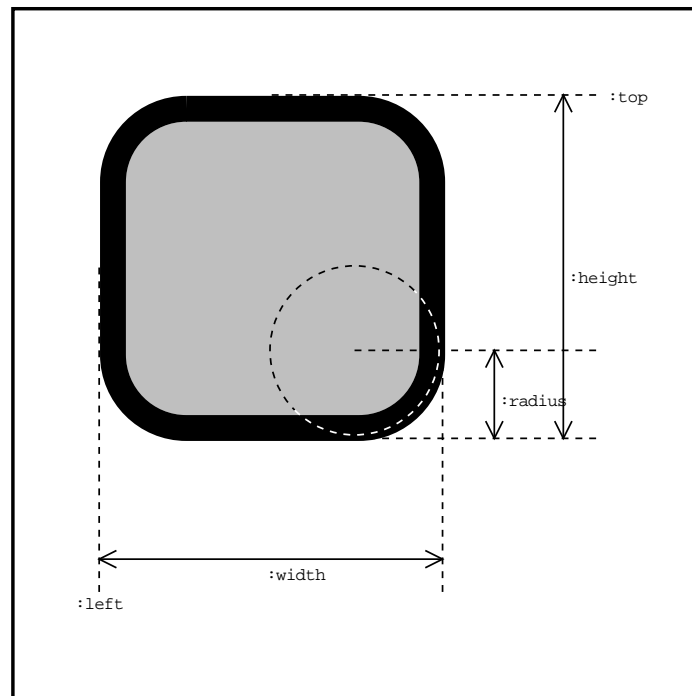
Figure 5.5: The parameters of a roundtangle.

5.7.3 Polyline and Multipoint

```
(create-instance 'opal:multipoint opal:graphical-object
  (:maybe-constant '(:point-list :line-style :filling-style :draw-function :visible))█
  (:point-list nil))

(create-instance 'opal:polyline opal:multipoint
  (:hit-full-interior-p nil))
```

The `opal:polyline` prototype provides for multi-segmented lines. Polygons can be specified by creating a polyline with the same first and last points. The point list is a flat list of values (`x-1 y-1 x-2 y-2 ... x-n y-n`). If a polyline object has a filling-style, and if the last point is not the same as the first point, then an invisible line is drawn between them, and the resulting polygon is filled.

The `:point-in-gob` method for the `opal:polyline` actually checks whether the point is inside the polygon, rather than just inside the polygon's bounding box. If the `:hit-full-interior-p` slot of a `polyline` is `nil` (the default), then the `:point-in-gob` method will use the "even-odd" rule to determine if a point is inside it. If the value of `:hit-full-interior-p` is T, the method will use the "winding" rule. The slot `:hit-threshold` has its usual functionality.

The `:left`, `:top`, `:width`, and `:height` slots reflect the correct bounding box for the polyline, but cannot be used to change the polyline (i.e., **do not set the** `:left`, `:top`, `:width`, **or** `:height` **slots**).

For example:

```
(create-instance nil opal:polyline
   (:point-list '(10 50 50 10 90 10 130 50))
   (:filling-style opal:light-gray-fill)
   (:line-style opal:line-4))
```

A multipoint is like a polyline, but only appears on the screen as a collection of discon-
nected points. The line-style and filling-style are ignored.

5.7.4 Arrowheads

```
(create-instance 'opal:arrowhead opal:polyline
  (:maybe-constant '(:line-style :filling-style :length :diameter :open-p
      :head-x :head-y :from-x :from-y :visible))
  (:head-x 0) (:head-y 0)
  (:from-x 0) (:from-y 0)
  (:connect-x (o-formula ...))  ; Read-only slot
  (:connect-y (o-formula ...))  ; Read-only slot
  (:length 10)
  (:diameter 10)
  (:open-p T)
  ...)
```

The `opal:arrowhead` class provides arrowheads. Figure [arrowfig], page 176, shows the meaning of the slots for arrowheads. The arrowhead is oriented with the point at (`:head-x`, `:head-y`) and will point away from (`:from-x`, `:from-y`). (**Note:** no line is drawn from (`:from-x`, `:from-y`) to (`:head-x`, `:head-y`); the `:from-` point is just used for reference.) The `:length` slot determines the distance (in pixels) from the point of the arrow to the base of the triangle. The `:diameter` is the distance across the base. The `:open-p` slot determines if a line is drawn across the base.

The arrowhead can have both a filling and an outline (by using the standard `:filling-style` and `:line-style` slots). Arrowhead objects also have 2 slots that describe the point at the center of the base to which one should attach other lines. This point is (`:connect-x`, `:connect-y`) and is set automatically by Opal; do not set these slots. These slots are useful if the arrow is closed (see Figure [arrowfig], page 176, below).

If you want an arrowhead connected to a line, you might want to use the **arrow-line** object (with one arrowhead) or **double-arrow-line** (with arrow-heads optionally at either or both ends) supplied in the Garnet Gadget Set *GarnetGadgetsChapter*.
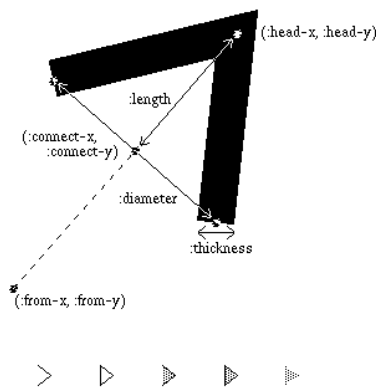
	1st	2nd	3rd	4th	5th
`:open-p:`	t	nil	t	nil	t
`:filling-style:`	nil	nil	opal:light-gray-fill		
`:line-style:`	....	opal:line-0	....	nil	

Figure 5.6: The slots that define an arrowhead. At the bottom are various arrowheads with different styles. Note that a shaft for the arrow must be drawn by the user.

5.7.5 Arcs

```
(create-instance 'opal:arc opal:graphical-object
  (:maybe-constant '(:left :top :width :height :line-style :filling-style
     :draw-function :angle1 :angle2 :visible))
  (:angle1 0)
  (:angle2 0))
```

The `opal:arc` class provides objects that are arcs, which are pieces of ovals. The arc segment is parameterized by the values of the following slots: `:left`, `:top`, `:width`, `:height`, `:angle1`, and `:angle2`.

The arc is a section of an oval centered about the point $(\mathtt{center} - \mathtt{xarc}, \mathtt{center} - \mathtt{yarc})$ calculated from the arc's `:left`, `:top`, `:width` and `:height`, with width `:width` and height `:height`. The arc runs from `:angle1` counterclockwise for a distance of `:angle2` radians. That is, `:angle1` is measured from 0 at the center right of the oval, and `:angle2` is measured from `:angle1` (`:angle2` is relative to `:angle1`).

Arcs are filled as pie pieces to the center of the oval.

For example:
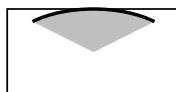
```
;; the rectangle is just for reference
(create-instance 'myrect opal:rectangle
  (:left 10)(:top 10)(:width 100)(:height 50))
(create-instance 'myarc opal:arc
  (:left 10)(:top 10)
  (:width 100)(:height 50)
  (:angle1 (/ PI 4))
  (:angle2 (/ PI 2))
```

```
(:line-style opal:line-2)
(:filling-style opal:light-gray-fill))
```

5.7.6 Ovals

```
(create-instance 'opal:Oval opal:arc)
```

Instances of the :oval class are closed arcs parameterized by the slots :left[,] :top[,] :width[, and] :height.

5.7.7 Circles

```
(create-instance 'opal:Circle opal:arc)
```

The circle is positioned at the top, leftmost part of the bounding box described with the :left. :top, :width, and :height slots. The circle drawn has diameter equal to the *minimum* of the width and height, though the effective bounding box (used by point-in-gob, for example) will still be defined by the actual values in :width and :height. Both :width and :height need to be specified.

5.7.8 Fonts and Text

5.7.8.1 Fonts

There are two different ways to get fonts from Garnet. One way is to explicitly create your own font object, and supply the object with a description of the desired font, either with family, face, and size descriptions, or with a font pathname. The other way is to use the function get-standard-font which will create a new font object for you if necessary, or return a previously created font object that you can use again.

There are two different types of font objects – one which handles the standard Garnet fonts (described by family, face, and size parameters), and one which handles fonts specified by a filename. The get-standard-font function only returns font objects that can be described with the three standard parameters. Either kind of font object may be used anywhere a *font* is called for.

Built in Fonts

```
(create-instance 'opal:font opal:graphic-quality
  (:maybe-constant '(:family :face :size))
  (:family :fixed)
  (:face :roman)
  (:size :medium)
   ...)

(create-instance 'opal:default-font opal:font
    (:constant t))
```

To use the standard Garnet fonts, create an instance of `opal:font` with your desired values for the `:family`, `:face`, and `:size` slots. Opal will automatically find the corresponding font for your display. The allowed values for each slot are as follows:

Values for `:family` can be:

`:fixed`   a fixed width font, such as Courier. All characters are the same width.

`:serif`   a variable-width font, with "serifs" on the characters, such as Times.

`:sans-serif`
            a variable-width font, with no serifs on the characters, such as Helvetica.

Values for `:face` can be a single keword or a list of the following:

            Faces available for both X windows and the Mac:

`:roman`

`:italic`

`:bold`

`:bold-italic`

Faces available for the Mac only:

`:plain`

`:condense`
`:extend`

`:outline`

`:shadow`

`:underline`

Values for `:size` can be:

            :small a small size, such as 10 points.
            :medium a normal size, such as 12 points.
            :large a large size, such as 18 points.
            :very-large a larger size, such as 24 points.

The exported `opal:default-font` object contains the font described by `:fixed`, `:roman`, and `:medium`. This object should be used when a font is required and you want to use the default values. However, since this object's slots have been made constant for efficiency, do not create instances of the `opal:default-font` object. Instead, create instances of the `opal:font` objects with customized values for the parameters, or use `get-standard-font` (explained below).

Reusing Fonts

Instead of creating a new font object every time one is needed, you may use the same font object in multiple applications. The function `get-standard-font` remembers what fonts have been created, and will return a previously created font object if a new font is needed that has a matching description. Otherwise, `get-standard-font` will allocate a new font object and return it, remembering it for later.

`opal:get-standard-font` *family face size*   [Function]

> The parameters are all the keywords that are allowed for standard fonts. For example: (`opal:get-standard-font :fixed :italic :medium`). In addition, any of the parameters can be `nil`, which means to use the defaults (`:fixed :roman :medium`). It is more efficient to use this procedure than to repeatedly allocate new font objects.
>
> Since all the font objects returned by `get-standard-font` have been declared constant for efficiency, you may not change the font descriptions after the objects have been created.
>
> Note: `get-standard-font` only remembers those fonts that were allocated by using `get-standard-font`. If a requested font matches an independently-generated font, `get-standard-font` will not know about it and will allocate a new font.

Fonts from Files

```
(create-instance 'opal:font-from-file opal:graphic-quality
  (:font-path nil)
  (:font-name "")
   ...)
```

This allows you to specify a file name to load a font from.

X11 keeps a set of font directories, called the current "Font Path". You can see what directories are on the font path by typing `xset q` to the Unix shell, and you can add and remove directories from the font path by using the `xset fp+` and `xset fp-` commands.

If the `:font-path` slot of a `:font-from-file` is a string which is a directory, Opal pushes that directory onto the X font path and then looks up the font. If the font name is somewhere on the path already, you can let the `:font-path` slot be `nil`. You can usually access fonts in the standard system font area (often `/usr/misc/.X11/lib/fonts/`) without specifying a path name.

For example, for the font `vgi-25.snf` in the default directory, use:

```
(create-instance nil opal:font-from-file
  (:font-name "vgi-25"))
```

If the font was not in the default font path, then use something like:

```
(create-instance nil opal:font-from-file
  (:font-path "/usr/misc/.X11/lib/fonts/75dpi/")
  (:font-name "vgi-25"))
```

The font name `"vgi-25"` is looked up in a special file in the font directory called `fonts.dir`. This file contains a long list of fonts with the file name of the font on the left and the name for the server to use on the right. For example, the entry corresponding to `opal:default-font` may look like this:

```
courier12.pcf            -adobe-courier-medium-r-normal--17-120-100-100-m-100-iso8859-1|
```

On some displays, this font lookup may not proceed smoothly, and you may have to supply the long `"-adobe-..."` name as the value of `:font-name` instead of the more convenient `"courier12"`. Garnet internally builds these names for the standard fonts, so font name lookup should never be a problem for them.

Opal:Cursor-Font

```
(create-instance 'opal:cursor-font opal:font-from-file
  (:constant t)
  (:font-name "cursor"))
```

The `opal:cursor-font` object accesses 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 [the-cursor-slot], page 212).

Functions on Fonts

`opal:string-width` *font-obj string* **&key** (*start* **0**) *end*   [Function]
`opal:string-height` *font-obj string* **&key** (*actual-heightp* **nil**)   [Function]

> The function `string-width` takes a font object (which can be a `font` or a `font-from-file`) and a Lisp string, and returns the width in pixels of that string written in that font. The *start* and *end* parameters allow you to specify the beginning and ending indices of the portion of *string* that you want to measure.
>
> The function `string-height` takes a font (or font-from-file) and a Lisp string, and returns the height in pixels of that string written in that font. There is an optional keyword parameter *actual-heightp* which defaults to `nil`, and has exactly the same effect on the return value of `string-height` that the `:actual-heightp` slot of an `opal:text` object has on the value of the `:height` slot of that `opal:text` object (see section [actualheightp], page 183).

5.7.8.2 Text

```
(create-instance 'opal:text opal:graphical-object
  (:maybe-constant '(:left :top :string :font :actual-heightp :line-style :visible))
  (:string "")
  (:font opal:default-font)
  (:actual-heightp nil)
  (:justification :left)
  (:fill-background-p nil)
  (:line-style opal:default-line-style)
  (:cursor-index nil))
```

Instances of the `opal:text` class appear as a horizontal string of glyphs in a certain font. The `:string` slot holds the string to be displayed, and can contain multiple lines. The `:font` slot specifies a font object as described in the previous section (an instance of `opal:font` or `opal:font-from-file`).

The `:line-style` slot can control the color of the object, and can hold any instance of `opal:line-style`, such as `opal:red-line`. The `:foreground-color` slot of the `line-style` object determines the color of the text. When the `:fill-background-p` slot is T, then the background of each glyph of the text is drawn with the color in the

:background-color slot of the line-style. If the :fill-background-p slot is nil, then the background is unaffected.

The :justification slot can take one of the three values :left, :center, or :right, and tells whether the multiple-line string is left-, center-, or right-justified. The default value is :left.

A vertical bar cursor before the :cursor-indexth character. If :cursor-index is 0, the cursor is at the left of the string, and if it is >= the length of the string, then it is at the right of the string. If :cursor-index is nil, then the cursor is turned off. The :cursor-index slot is set by the inter:text-interactor during text editing.

> opal:Get-Cursor-Index *string-obj x y*[function], page 90

This function returns the appropriate cursor-index for the (x,y) location in the string. It assumes that the string is displayed on the screen. This is useful for getting the position in the string when the user presses over it with the mouse.

The slot :actual-heightp determines whether the height of the string is the actual height of the characters used, or the maximum height of the font. This will make a difference in variable size fonts if you have boxes around the characters or if you are using a cursor (see section [text], page 182). The default (nil) means that the height of the font is used so all strings that are drawn with the same font will have the same height.

The :width and :height slots reflect the correct width and height for the string, but cannot be used to change the size (i.e., **do not set the :width or :height slots**).

5.7.8.3 Scrolling Text Objects

When an opal:text or opal:multifont-text object is used inside a scrolling-window, there is an option that allows the window to scroll automatically whenever the cursor is moved out of the top or bottom of the visible region. To use this feature, two things need to be done:

> The :scrolling-window slot of the text object must contain the scrolling window object.
>
> The text object must also have its :auto-scroll-p slot set to T.

NOTE: Auto scroll is NOT the same as word wrap. If the cursor is moved out of the right edge of the window, auto-scroll will not do anything.

For an example of how the auto-scroll feature works, look at the code for Demo-Multifont. Try the demo with the :auto-scroll-p slot of the object demo-multifont::text1 set to both T and nil.

Auto scroll does not keep track of changes in family, font, size, or when a segment is cut or pasted. The :auto-scroll method has to be invoked explicitly in such cases, using the following method:

text-obj   [Method on gg:auto-scroll]
> For examples of calling gg:auto-scroll explicitly, look at the menu functions in Demo-Multifont.

5.7.9 Bitmaps

```
(create-instance 'opal:bitmap opal:graphical-object
  (:maybe-constant '(:left :top :image :filling-style :visible))
  (:image nil)
  (:filling-style opal:default-filling-style)
  ...)
```

On the Mac, and in the usual case with X11, the `:image` slot contains a machine-dependent structure generated by the function `opal:read-image` (see below). Under X11, there are a variety of other CLX image objects that can be stored in this slot (consult your CLX Manual for details on images).

Bitmaps can be any size. Opal provides a function to read in a bitmap image from a file:

opal:Read-Image *file-name*[function], page 90

The `read-image` function reads a bitmap image from *file-name* which is stored in the default X11 ".bm" file format. Files of this format may be generated by using the Unix program `/usr/misc/.X11/bin/bitmap`.

The `:filling-style` slot can contain any instance of `opal:filling-style`. If the `:fill-style` of the bitmap's `:filling-style` is `:solid` or `:opaque-stippled`, then the bitmap will appear with that filling-style's foreground-color and background-color. If, however, the `:fill-style` of the filling-style is `:stippled`, then the bitmap will appear with the filling-style's `:foreground-color`, but its background will be transparent. For example, the following code creates a bitmap which will be drawn with a red and white stipple (because white is the default `:background-color` of `opal:filling-style`):

```
(create-instance 'red-arrow opal:arrow-cursor
  (:filling-style (create-instance nil opal:filling-style
                      (:foreground-color opal:red)
                      (:fill-style :stippled))))
```

There are several functions supplied for generating halftone images, which can then be supplied to the `:image` slot of a bitmap object. These functions are used to create the filling styles returned by the `halftone` function (section [halftone], page 166).

opal:Halftone-Image *percentage*[function], page 90

The `halftone-image` function returns a image for use in the `:image` slot of a bitmap object. The *percentage* argument is used to specify the shade of the halftone (0 is white and 100 black). This image is as close as possible to the *percentage* halftone value as can be generated. Since a range of *percentage* values map onto each halftone image, two additional functions are provided to get images that are guaranteed to be one shade different or one shade lighter than a specified value.

opal:Halftone-Image-Darker *percentage*[function], page 90

opal:Halftone-Image-Lighter *percentage*[function], page 90

The `halftone-image-darker` and `halftone-image-lighter` functions return a halftone that is guaranteed to be exactly one shade darker than the halftone with the specified

percentage. With these functions you are guaranteed to get a different darker (or lighter) image. Currently, there are 17 different halftone shades.

The `:width`[`, and`] `:height` slots reflect the correct width and height for the bitmap, but cannot be used to change the size (i.e., **do not set the** `:width` **or** `:height` slots)).

5.7.10 Pixmaps

```
(create-instance 'opal:pixmap opal:bitmap
  (:image nil)
  (:line-style opal:default-line-style)
  (:pixarray (o-formula (if (gvl :image)
    (gem:image-to-array (gv-local :self :window)
(gvl :image))))))
  ...)
```

This object is similar to the `opal:bitmap` object, except that it handles images which use more than one bit per pixel.

`opal:bitmap`, in conjunction with the function `opal:read-xpm-file` (see below).

The `:pixarray` slot contains an array of colormap indices. This is useful if you want to manipulate a pixmap directly, as in the demo "demo-pixmap".

The `:width`[`, and`] `:height` slots reflect the correct width and height for the pixmap, but cannot be used to change the size (i.e., **do not set the** `:width` **or** `:height` [slots]).

5.7.10.1 Creating a pixmap

The following routine can be used to create an image for a pixmap.

　　　`opal:Read-XPM-File` *pathname* [No value for ''function'']

The argument *pathname* should be the name of a file containing a C pixmap image. `Read-xpm-file` returns an X-specific or Mac-specific object, which then should be put in the `:image` slot of an `opal:pixmap`. The file *pathname* containing the C pixmap image should be in the *xpm* format. Please refer to the X Window System documentation for more details about that format.

The function `read-xpm-file` will read pixmaps in the XPM1 or XPM2 format. Files in these formats are produced by the program `ppmtoxpm` and the OpenLook `IconEditor` utility. The `ppm` collection of utilities are useful for converting one format into another. If you do not have them, you can store Unix utilities.

In Unix, to convert the contents of a color window into an *xpm* format file, you can use programs such as `xwd`, `xwdtopnm`, `ppmtoxpm`, etc. For example, inside a Unix shell, type:

```
xwd > foo.xwd
```

When the cursor changes to a plus, click on the window you want to dump. Then type:

```
xwdtopnm foo.xwd > foo.ppm
ppmtoxpm foo.ppm > foo.xpm
```

This will create a file named "foo.xpm". Finally, in Garnet, type:

```
(create-instance 'FOO opal:pixmap
   (:image (opal:read-xpm-file "foo.xpm")))
```

Here are two more routines that can be used to create images for pixmaps.

> opal:**Create-Pixmap-Image** *width height* &optional *color* [Function], page 720■

This creates a solid color pixmap image. If you wanted to create a pixmap whose image was, say, a 20x30 blue rectangle, you would say:

```
(create-instance 'BLUE-PIXMAP opal:pixmap
    (:image (opal:create-pixmap-image 20 30 opal:blue)))
```

If no color is given, the color defaults to white.

> opal:**Window-To-Pixmap-Image** *window* &key *left top width height* [Function],■
> page 720

This creates an image containing the contents of a Garnet window, within a rectangular region specified by the values *left*, *top*, *width*, and *height*. Left and top default to 0. *Width* and *height* default to the values of the :**width** and :**height** slots of the window, respectively.

5.7.10.2 Storing a pixmap

> opal:**Write-XPM-File** *pixmap pathname* &key *(xpm-format :xpm1)* [Function], page 720

This function writes the :**image** of a pixmap object into a C pixmap file whose name is *pathname*. **Write-xpm-file** will write pixmap files in either XPM1 or XPM2 format, depending on the value of the *xpm-format* key, which may be either :**xpm1** or :**xpm2**. By default, the function generates files in XPM1 format, which can be read by the **xpmtoppm** utility.

5.8 Multifont

```
(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)
    (:draw-function :copy)
    (:show-marks nil))
```

The opal:**multifont-text** object is designed to allow users to create more complicated editing applications. The object is similar to the opal:**text** object 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 [Interactors for Multifont Text], page 194). The font object and the two interactors are combined into the `multifont-gadget` gadget for convenience (See [A Multifont Text Gadget], page 201).

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 indentation and paren-matching features of lisp mode.

5.8.1 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.

```
(create-instance 'opal:multifont-text opal:aggregate
   ...
   (:initial-text "here is my example string.")
   ...
```

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 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 [Using `view-objects` as Text], page 193).
- a mark, in the form `(list :mark sticky-left name info)` (See [Using Marks], page 194).

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

```
'((,(cons "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 5.7. Details on using fonts, colors, marks, and graphical objects are given in section [Functions on Multifont Text], page 189.

```
;; 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")))
```

```
An example string
with multiple lines

An example string
with multiple lines

Here is my example string.


Here is a circle:◯

Here is yellow and red text

The ˃(parentheses)˂ are marked
```
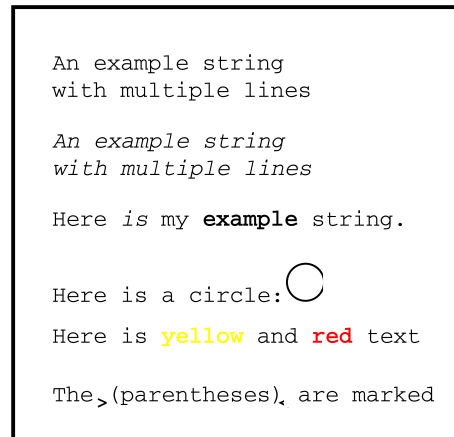
Figure 5.7: Examples of the multifont-text object

5.8.2 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.

5.8.2.1 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-num*   [Function]
        char-num
> 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.

5.8.2.2 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-num*   [Function]
> *char-num*

> 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*   [Function]
italic bold

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*   [Function]
background-color

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`.

5.8.2.3 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 func-
>   tion `get-string` returns the contents as a single string with `#\newline`s 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.

5.8.3 Adding and Editing Text

opal:**add-char** *text-obj char* **&optional** *font foreground-color*   [Function]
>   *background-color lisp-mode-p*
opal:**insert-string** *text-obj string* **&optional** *font foreground-color*   [Function]
>   *background-color*
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
>   chapterly 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 work-
>   ing 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-num start-char-num*   [Function]
        end-line-num end-char-num
`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.

5.8.3.1 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*.

5.8.3.2 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.

5.8.3.3 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-p* 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.

5.8.4 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 chapter 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` (See [A Multifont Text Gadget], page 201) 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.

5.8.4.1 Multifont Text Interactor

```
(create-instance 'inter:multifont-text-interactor inter:text-interactor
   (:window nil)
   (:edit-func #'inter::multifont-text-edit-string)

   ;; For the following three slots, See [Lisp Mode], page 199
   (:lisp-mode-p nil)
   (:match-parens-p nil)
   (:match-obj ...)

   ;; See Section 6.20.1 [text-interactor], page 284
   (:button-outside-stop? t)

   (:drag-through-selection? t)

   ;; (lambda (inter text-obj))
   (:stop-action #'inter::multifont-text-int-stop-action)

   (:after-cursor-moves-func nil))
```

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 RET. 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 keys names below are the familiar Unix/Emacs compose and function keys.

`C-f`, `C-b`, `C-d`, `C-h`
        Move or delete character: forward, backwards, delete forwards, delete backwards

`LEFT`, `RIGHT`
          backwards, forwards

`M-f`, `M-b`, `M-d`, `M-h`
          same but by words

`C-p`, `C-n`   previous line, next line

`UP`, `DOWN`   previous line, next line

`C-,` or `HOME`
          beginning of document

`C-.` or `END`
          end of document

`C-a`   beginning of line

`C-e`   end of line

`C-k`   kill line

`C-u`   delete entire string

`C-w`, `CUT`   delete selection

`M-w`, `COPY`   copy selection to interactor cut buffer

`C-c`   copy entire string to X11 cut buffer

`C-y`, `PASTE`
          yank interactor cut buffer or X11 cut buffer into string

`C-Y`, `C-PASTE`
          yank X11 buffer

`M-y`, `C-PASTE`
          yank interactor cut buffer

The following key combinations extend the selection while moving:

`C-LEFT`, `C-RIGHT`
          prev, next char selecting

`C-LEFT`, `C-RIGHT`
          prev, next word selecting

`C-UP`, `C-DOWN`
          up-line, down-line selecting

`C-HOME`, `C-END`
          beginning, end of string selecting

`C-*`   select all

C-M-*key* Lisp Stuff If You Have Lisp Mode on (See Below)

C-M-b, *C-M-LEFT*
 prev lisp expression

C-M-f, *C-M-RIGHT*
 next lisp expression

C-M-h, *C-M-BS*, *C-M-DELETE*
 delete prev s-expr

C-M-d delete next s-expr

C-SHIFT-*key* Is for Font Stuff

C-SHIFT-B
 toggle bold

C-SHIFT-I
 toggle italic

C-SHIFT-F
 fixed font (courier)

C-SHIFT-T
 times font (serif)

C-SHIFT-H
 helvetica font (sans-serif)

C-SHIFT-<
 smaller font

C-SHIFT->
 bigger font

C-1, *C-2*, *C-3*, *C-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 See Section 6.20.1 [`text-interactor`], page 284.

5.8.4.2 Focus Multifont Text Interactor

```
(create-instance 'inter:focus-multifont-textinter inter:interactor
  (:window nil)
  (:obj-to-change nil)
  (:stop-event nil)
  (:lisp-mode-p nil)
  (:match-parens-p nil)
  (:match-obj ...)
  ;; (lambda (inter obj final-event final-string x y))
  (:final-function nil)
  ;; (lambda (inter text-obj))
```

```
(:after-cursor-moves-func nil))
```

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 RET), 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.

5.8.4.3 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.

5.8.4.4 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 "create-instance" 2 4)

(indent "let" 1 4)

(indent "do" 2 -1)

(indent "cond" 0)

(indent "define-method" 3 4)
```

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

C-M-f, *C-M-RIGHT*
> skip forward lisp expression

C-M-b, *C-M-LEFT*
> skip backward lisp expression

C-M-d   delete lisp expression

C-M-h, *C-M-BS*
> delete previous lisp expression

Also helpful in lisp mode is setting the `:match-parens-p` of the interactors to T. 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*   [Function]
> *new-foreground-color new-background-color*

`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.

5.8.5 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 `opal:multifont-text` is the same as for `opal:text`, See [Scrolling Text Objects], page 183,

5.8.6 After Cursor Moves

To support lisp-mode, there is a 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.

5.8.7 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))
```

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 `RET`), 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`).

5.9 Aggregate objects

Aggregate objects hold a collection of other graphical objects (possibly including other aggregates). The objects in an aggregate are called its *components* and the aggregate is the *parent* of each component. An aggregate itself has no filling or border, although it does have a left, top, width and height.

Note: When you create an aggregate and add components to it, creating an instance of that aggregate afterwards does *not* create instances of the children. If you use Aggregadgets instead, then you *do* get copies of all the components. Aggregadgets also provide a convenient syntax for defining the components. Therefore, it is often more appropriate to use Aggregadgets than aggregates. See the Aggregadgets chapter *AggregadgetsChapter*.

5.9.1 Class Description

```
(create-instance 'opal:Aggregate opal:view-object
  (:components nil)
  (:hit-threshold 0)
  (:overlapping T))
```

The `:components` slot holds a list of the graphical objects that are components of the aggregate. *This slot should not be set directly but rather changed using* `add-component` *and* `remove-component` *(section [addremsection], page 203).* The covering (which is the ordering among children) in the aggregate is determined by the order of components in the `:components` slot. **The list of components is stored from bottommost to topmost.** This slot cannot be set directly.

```
opal:Set-Aggregate-Hit-Threshold agg[function], page 90
```

As is the case with graphical objects, the `:hit-threshold` slot of an aggregate controls the sensitivity of the `point-in-gob` methods to hits that are near to that aggregate. The value of the `:hit-threshold` slot defaults to 0, but calling `set-aggregate-hit-threshold` sets the `:hit-threshold` of an aggregate to be the maximum of all its components.

The `:overlapping` slot is used as a hint to the aggregate as to whether its components overlap. This property allows the aggregate to redraw it's components more efficiently. You can set the `:overlapping` slot to `nil` when you know that the first level children of this aggregate will never overlap each other on the screen. *Currently, this slot is not used, but it may be in the future.*

Aggregates have a bounding box, which, by default, is calculated from the sizes and positions of all its children. If you want to have the position or size of the children depend on that of the parent, it is important to provide an explicit value for the position or size of the aggregate, and then provide formulas in the components that depend on the aggregate's values. Be careful to avoid circularities: either the aggregate should depend on the sizes and positions of the children (which is the default) **or** the children should depend on the parent. These cannot be easily mixed in a single aggregate. It is important that the size and position of the aggregate correctly reflect the bounding box of all its components, or else the redisplay and selection routines will not work correctly.

5.9.2 Insertion and Removal of Graphical Objects

opal:**add-component** *aggregate graphical-object* [[:*where*]   [Function]
     position[*locator*]]
>   The method `add-component` adds *graphical-object* to *aggregate*. The *position* and
>   *locator* arguments can be used to adjust the placement/covering of *graphical-object*
>   with respect to the rest of the components of *aggregate*.

There are five legal values for *position*; these are: :`front`, :`back`, :`behind`, :`in-front`,
and :`at`. Putting an object at the :`front` means that it is not covered by any other objects
in this aggregate, and at the :`back`, it is covered by all other objects in this aggregate.
Positioning *graphical-object* at either :`front` or :`back` requires no value for *locator*, as these
are unique locations. If position is either :`behind` or :`in-front` then the value of *locator*
should be a graphical object already in the component list of the aggregate, in which case
graphical-object is placed with respect to *locator*. In the final case, with *position* being :`at`,
graphical-object is placed at the *locator*th position in the component list, where 0 means at
the head of the list (the back of the screen).

If none are supplied, then the new object is in front of all previous objects. The :`where`
keyword is optional before the locators, so all of the following are legal calls:

```
(opal:add-component agg newobj :where :back)
(opal:add-component agg newobj :back)
(opal:add-component agg newobj)        ; adds newobj at the :front
(opal:add-component agg newobj :behind otherobj)
(opal:add-component agg newobj :at 4)
```

Objects cannot belong to more than one aggregate. Attempting to add a component of
one aggregate to a second aggregate will cause Opal to signal an error. If the *locator* for
:`behind` or :`in-front` is not a component of the aggregate Opal will also signal an error.

opal:**add-components** *aggregate* **&rest** *graphical-objects*   [Function]
>   This function adds multiple components to an aggregate. Calling this function is
>   equivalent to:
>
>    ```
>    (dolist (gob (list graphical-object[*]))
>       (add-component aggregate gob))
>    ```
>
>   An example of using `add-components` is:
>
>    ```
>    (opal:add-components agg obj1 obj2 myrect myarc)
>    ```
>
>   Note that this has the effect of placing the list of graphical objects from back to front
>   in *aggregate* since it inserts each new object with the default :`where` :`front`.

aggregate *graphical-object*   [Method on opal:**remove-component**]
>   The `remove-component` method removes the *graphical-object* from *aggregate*. If *ag-*
>   *gregate* is connected to a window, then *graphical-object* will be erased when the
>   window next has an update message ([Methods and Functions on Window Objects],
>   page 216) sent to it.

opal:**remove-components** *aggregate* **&rest** *graphical-object*   [Function]
>   Removes all the listed components from *aggregate*.

`opal:move-component` *aggregate graphical-object* [[:where]   [Function]
      position[*locator*]]
> `move-component` is used to change the drawing order of objects in an aggregate, and
> therefore change their covering (since the order of objects in an aggregate determines
> their drawing order). For example, this function can be used to move an object to the
> front or back. The object should already be in the aggregate, and it is moved to be at
> the position specified. It is like a `remove-component` followed by an `add-component`
> except that it is more efficient. The parameters are the same as `add-component`.

5.9.3 Application of functions to components

There are two methods defined on aggregates to apply functions to some subset of the aggregate's components. The methods work on either the direct components of the aggregate or all objects that are either direct or indirect components of the aggregate.

`opal:do-components` *aggregate function* **&key** *type self*   [Function]
> The `do-components` method applies *function* to all components of *aggregate* in back-
> to-front order. The *function* should take one argument which will be the component.
> If a type is specified, the function is only applied to components that are of that type.
> If the call specifies `:self` to be `t` (the default is `nil`), and the aggregate is of the
> specified type, then the function is applied to *aggregate* after being applied to all of
> the components.
>
> The *function* must be non-destructive, since it will be applied to the components list
> of *aggregate*, not to a copy of the components list. For instance, *function* cannot
> call *remove-component* on the components. If you want to use a *function* that is
> destructive, you must make a copy of the components list and call dolist yourself.

`opal:do-all-components` *aggregate function* **&key** *type self*   [Function]
> The `do-all-components` method works similarly to `do-components`, except that in
> the case that a component is an aggregate, `do-all-components` is first called recur-
> sively on the component aggregate and then applied to the component aggregate itself.
> `self` determines whether to call the function on the top level aggregate (default=`nil`)
> after all components.

5.9.4 Finding Objects Under a Given Point

`opal:point-to-component` *aggregate x y* **&key** *type*   [Function]
`opal:point-to-leaf` *aggregate x y* **&key** *type*   [Function]
> `Point-to-component` queries the aggregate for the first generation children at point
> (*x*,*y*). The value of *type* can limit the search to graphical objects of a specific type.
> This function returns the topmost object at the specified point (*x*,*y*).
>
> `Point-to-leaf` is similar except that the query continues to the deepest children in
> the aggregate hierarchy (the leaves of the tree). Sometimes you will want an aggregate
> to be treated as a leaf in this search, like a button aggregate in a collection of button
> aggregates. In this case, you should set the aggregate that should be treated like a
> leaf. The search will not proceed through the components of such an aggregate, but
> will return the aggregate itself.
>
> The *type* slot can be either `t` (the default), a type, or a list of types. If *type* is
> specified as an atom, only objects that are of that *type* will be tested. If *type* is

specified as a list, only objects whose type belongs to that list will be tested. The
value t for *type* will match all objects. If the *type* is specified for a `point-to-leaf`
call, and the `type` is a kind of aggregate, then the search will stop when an aggregate
of that type (or types) is found at the specified (x,y) location, rather than going all
the way to the leaves. For example:

```
(create-instance 'myaggtype opal:aggregate)
(create-instance 'myagg myaggtype)
(create-instance top-agg opal:aggregate)
(opal:add-component top-agg myagg)

(create-instance obj1 ...)
(create-instance obj2 ...)
(opal:add-components myagg obj1 obj2)

;; will return obj1, obj2, or nil
(opal:point-to-leaf top-agg x y)
;; will return myagg or nil
(opal:point-to-leaf top-agg x y :type myaggtype)
```

`Point-to-leaf` and `point-to-component` always use the function `point-in-gob` on
the components.

5.9.5 Finding objects inside rectangular regions

opal:Components-In-Rectangle *aggregate top left bottom right* &key *type in-*■
tersect[function], page 90

opal:Leaf-Objects-In-Rectangle *aggregate top left bottom right* &key *type in-*■
tersect[function], page 90

opal:Obj-In-Rectangle *object top left bottom right* &key *type inter-*
sect[function], page 90

The routine `components-in-rectangle` queries the aggregate for the first generation
children that intersect the rectangle bounded by *top*, *left*, *bottom*, and *right*. If *intersect*
is `nil`, then the components which are returned must be completely inside the rectangle,
whereas if *intersect* is non-`nil` (the default), then the components need only intersect the
rectangle. The value of *type* can limit the search to graphical objects of a specific type.

`Leaf-objects-in-rectangle` is similar except that the query continues to the deepest
children in the aggregate hierarchy (the leaves of the tree). Sometimes you will want an
aggregate to be treated as a leaf in this search, like a button aggregate in an aggregate of
buttons. In this case, you should set the aggregate that should be treated like a leaf. The
search will not proceed through the components of such an aggregate, but will return the
aggregate itself.

`Obj-in-rectangle` tells whether the bounding box of *object* intersects the rectangle
bounded by *top*, *left*, *width* and *height*. If *intersect* is non-`nil` (the default) then *object*
need only intersect the rectangle, whereas if *intersect* is `nil` then *object* must lie completely
inside the rectangle. If *type* is not `t` (the default) then *object* must be of type *type*.

5.10 Virtual-Aggregates

Virtual-aggregates are used when you are going to create a very large number of objects
(e.g., 300 to 50,000) all of which are fairly similar. For example, they are useful for points in
a scatter plot, squares in a "fat-bits" bitmap editor, line segments in a map, etc. The virtual
aggregate *pretends* to provide an object for each element, but actually doesn't. This can
save an enormous amount of memory and time, while still providing an interface consistent
with the rest of Garnet.

The primary restriction is that there cannot be references or constraints from external
objects *to* or *from* any of the elements of the virtual-aggregate. Typically, all the constraints
will be internal to each object displayed, and all the properties will be determined by the
values in the `:items` array.

The interface is similar to *aggrelists*. The programmer provides an item-prototype, used
for all the elements, and an (optional) items list to form the initial value. To be more
efficient, the items list is actually an array for virtual-aggregates. The item-prototype can
be an arbitrary object or aggregadget structure, and can use whatever formulas are desired
to calculate the appropriate display based on the corresponding value of the items list and
the object's rank in the item's list.

We have implemented two styles of virtual-aggregates, with a third style in planning.
The first style is for arbitrary overlapping objects, and is described below. The second style
is for non-overlapping 2-D arrays of objects, such as bitmap-editor tiles.

The third style is like the first, for arbitrary overlapping objects. However, unlike the
first style, it would use more sophisticated techniques for computing the overlapping of
objects, rather than using linear search like the first style. For example, it might use quad
trees or whatever.

So far, we have implemented the first and second style only. Examples of using these
virtual-aggregates are in demo-circle for the first style and demo-array for the second.

5.10.1 Virtual-Aggregates Slots

A virtual-aggregate is a graphical object, with its own `:draw`, `:point-to-component`,
`:add-item`, and `:remove-item` methods. It is defined as:

```
(create-instance 'opal:virtual-aggregate opal:graphical-object
   ...
  (:item-prototype ...)  ;; you must provide this
  (:point-in-item ...)   ;; you must provide this
  (:item-array ...)      ;; you may provide this
  (:dummy-item ...)
  )
```

For example, in demo-circle the virtual-aggregate is:

```
(create-instance nil opal:virtual-aggregate
   (:item-prototype my-circle)
   (:point-in-item #'my-point-in-circle))
```

Here are the slots you must provide for a virtual-aggregate.

:ITEM-PROTOTYPE

In the :item-prototype slot, you put the Garnet object of your choice (primitive object or

aggregadget). You must, however, have formulas in your :item-prototype object that depend on its :item-values and/or :rank slot. The :rank is set with the object's rank in the :items array. The :item-values is set with the appropriate data from the :item-array. For instance, in demo-circle, the item-prototype is:

```
(create-instance 'MY-CIRCLE opal:circle
   (:filling-style (o-formula (fourth (gvl :item-values))))
   (:radius (o-formula (third (gvl :item-values))))
   (:left (o-formula (- (first (gvl :item-values)) (gvl :radius))))
   (:top (o-formula (- (second (gvl :item-values)) (gvl :radius))))
   (:width (o-formula (* 2 (gvl :radius))))
   (:height (o-formula (gvl :width))))
```

In this case the :item-values slot contains a list of four numbers: the x and y coordinates of the center of the circle, the radius of the circle, and an Opal color. For your item-prototype, the format for the item-values data can be anything you like, and you don't have to set the :item-values slot yourself: Opal will do that for you.

:POINT-IN-ITEM

This slot contains a function of the form

```
(lambda (virtual-aggregate item-values x y) ...)
```

which returns t or nil depending on whether the point <x,y> lies within an :item-prototype object with :item-values item-values. Typically, you will be able to compute this function efficiently based on your knowledge of the how the objects will look. For instance, in demo-circle, the :point-in-item slots contains:

```
(lambda (virtual-aggregate item-values x y)
  (<= (+ (expt (- x (first item-values)) 2)
         (expt (- y (second item-values)) 2))
      (expt (third item-values) 2)))
```

:ITEM-ARRAY

This is a slot you *may*, but need not provide. If you don't provide one, then all of the items will be added using the add-item function, below. :item-array contains either a 1-dimensional array of item-values, ordered from back to front on your display, or a 2-dimensional array. So for the demo-circle example, it will look something like:

```
#((304 212 12 #k<RED-FILL>)
  (88 64 11 #k<GREEN-FILL>)
  ...)
```

The array may have nils in it. Each nil represents a gap in this items list.

5.10.2 Two-dimensional virtual-aggregates

You can create a virtual-aggregate whose :item-array is a *two* dimensional array. The formulas in the :dummy-item of the aggregate must depend on two slots :rank1 and :rank2 instead of the single slot :rank. This is useful for non-overlapping tables, such as bitmap editors (fat-bits), spreadsheets, etc. See the example in demo-array.

5.10.3 Manipulating the Virtual-Aggregate

These are the routines exported by Opal that you can use to manipulate the item array:

`opal:Add-Item` *a-virtual-aggregate* *item-values*⟨undefined⟩ `[method]`, `page` ⟨un▮
`defined`⟩

This adds a new item to the `:item-array` of *a-virtual-aggregate*. *Item-values* is a list
containing the values for an `:item-values` slot of the item-prototype. `Add-item` returns
the rank into the `:item-array` where the new item was inserted. The `:item-array` must
be one-dimensional.

`opal:Remove-Item` *a-virtual-aggregate* *rank*⟨undefined⟩ `[method]`, `page` ⟨un
`defined`⟩

This removes an item from the `:item-array` of *a-virtual-aggregate*. Actually, it puts
a `nil` in the `:item-array` (it does not compress the array). The `:item-array` must be
one-dimensional.

`opal:Change-Item` *a-virtual-aggregate* *new-item* *rank* `&optional` *rank2*⟨un
`defined`⟩ `[method]`, `page` ⟨undefined⟩

This changes the *rank*'th entry of the `:item-array` of the virtual-aggregate to be *new-
item*. (It also marks that item to be redrawn at the next update). To manipulate a
two-dimensional array, use *rank* and *rank2* as the two indices. Note: you have to use this
function and cannot directly modify the items array after the virtual-aggregate has been
displayed.

`opal:Point-To-Rank` *a-virtual-aggregate* *x* *y*⟨undefined⟩ `[method]`, `page` ⟨un▮
`defined`⟩

Returns the rank of the front-most item in the virtual-aggregate that contains point
<x,y>. (This is why you had to supply `:point-in-item`.) The virtual-aggregate must be
one-dimensional.

`opal:Point-To-Component` *a-virtual-aggregate* *x* *y*⟨undefined⟩ `[method]`, `page` ⟨un▮
`defined`⟩

This is like `point-to-rank`, but it returns an actual Opal object. However, the object is
actually a dummy object with the appropriate value placed in its `:item-values` and `:rank`
slots. So you cannot call Point-to-component twice and hope to hold on the first value.
(The virtual-aggregate must be one-dimensional.)

`opal:Recalulate-Virtual-Aggregate-Bboxes` *a-virtual-aggregate*`[function]`,
`page 90`

The purpose of this routine is to re-initialize all the bounding boxes of the items of
the virtual-aggregate. This would come in handy if, for instance, you created a virtual-
aggregate whose items depended for their position on the position of the virtual-aggregate
itself. After you changed the `:left` or `:top` of the virtual-aggregate, you would call
`recalculate-virtual-aggregate-bboxes` to re-calculate the bounding boxes of the items.

There is a macro for performing operations iteratively on elements of a 2-dimensional
virtual-aggregate:

`opal:Do-In-Clip-Rect` (*var1* *var2* *a-virtual-aggregate* *clip-rect*) `&body` *body*⟨un▮
`defined`⟩ `[macro]`, `page` ⟨undefined⟩

The variables *var1* and *var2* take on all values for which the item with `:rank1` = *var1*
and `:rank2` = *var2* intersect the clip-rectangle *clip-rect*. The *clip-rect* is a list of left, top,
width, and height – the kind of argument that is returned from a two-point-interactor.

As an example, consider the following code borrowed from demo-array:

```
(defun Whiten-Rectangle (dum clip-rect)
  (declare (ignore dum))
  (do-in-clip-rect (index-1 index-2 the-array clip-rect)
    (change-item the-array 1 index-1 index-2)))

(create-instance 'WHITER inter:two-point-interactor
  (:start-event :leftdown)
  (:continuous T)
  (:start-where '(:in ,The-Array))
  (:window w)
  (:feedback-obj FEED-RECT)
  (:final-function #'Whiten-Rectangle))
```

The-array is a 2-dimensional virtual-aggregate. The routine `Whiten-Rectangle` performs `opal:change-item` on every element of the-array that is inside the clip-rect (the second argument to the `:final-function` of a two-point interactor is always a rectangle).

This is a macro for performing operations iteratively on elements of a 2-dimensional virtual-aggregate. The variables *var1* and *var2* take on all values for which the item with `:rank1` = *var1* and `:rank2` = *var2* intersect the clip-rectangle clip-rect. The clip-rect is a list of left, top, width, and height – the kind of argument that is returned from a two-point-interactor.

5.11 Windows

Graphical objects can only display themselves in a *window*.

```
(create-instance 'inter:Interactor-Window opal::window
  (:maybe-constant '(:left :top :width :height :visible))
  (:left 0)
  (:top 0)
  (:width 355)
  (:height 277)
  (:border-width 2)
  (:left-border-width ...) (:top-border-width ...)     ;; Read-only slots ─
- Do not set!
  (:right-border-width ...) (:bottom-border-width ...) ;; See section [border─
widths], page 212.
  (:max-width nil) (:max-height nil)
  (:min-width nil) (:min-height nil)
  (:cursor opal:Arrow-Pair)     ;; Shape of the pointer in this window. (See sec─
tion [window-cursors], page 212).
  (:position-by-hand nil)
  (:title "Opal N")
  (:omit-title-bar-p nil)
  (:icon-title "Opal N")
  (:icon-bitmap nil)
  (:draw-on-children nil)
```

```
      (:background-color nil)
      (:double-buffered-p nil)
      (:save-under nil)
      (:aggregate nil)
      (:parent nil)
      (:visible ...)
      (:modal-p nil)                    ;; Whether to suspend input while visible. See the In-
    teractors chapter.
      (:in-progress nil)                ;; Read by opal:update-all. See section [quarantine-
    slot], page 215.
      ...)
```

Caveats:

Garnet windows will not appear on the screen until they are updated, by calling the functions `opal:update` or `opal:update-all`. These functions will also cause all of the graphics in the window to be brought up-to-date.

Windows are not usually used as prototypes for other windows. If a window is created with its `:visible` slot set to T, then it should be expected to appear on the screen (even if `opal:update` is not explicitly called on it). When similar windows need to be generated, it is recommended that a function be written (like at the end of the Tutorial) that will return the window instances.

The `:left`, `:top`, `:width`, and `:height` slots of the window control its position and dimensions. These slots can be set using `s-value` to change the window's size and position (which will take affect after the next `update` call). If the user changes the size or position of a window using the window manager (e.g., using the mouse), this will *usually* be reflected in the values for these slots.[2] Some special issues involving the position and dimensions of Garnet windows when adorned with window manager title bars are discussed in section [border-widths], page 212.

If you create a window with values in its `:max-width`, `:max-height`, `:min-width`, and `:min-height`, then the window manager will make sure the user doesn't change the window's size to be outside of those ranges. However, you can still `s-value` the `:width` and `:height` of `win` to be any value. The slots `:max-width` and `:max-height` can only be set at creation time. Furthermore, due to peculiarities in X windows, you must set *both* `:max-width` and `:max-height` to be non-`nil` at creation time to have any effect. The slots `:min-width` and `:min-height` behave in the analogous manner.

The `:title` slot contains a string specifying the title of the Garnet window. The default title is "Opal *N*", where *N* starts at 1, and increments each time a new window is created in that Lisp.

The `:omit-title-bar-p` slot tells whether or not the Garnet window should have a title bar. If the slot has value `nil` (the default), and the window manager permits it, then the window will have a title bar; otherwise the window will not have a title bar.

The `:icon-title` slot contains a string specifying the icon title of the window. The default icon title is the same as the `:title`. This is the string that gets displayed when a window is iconified.

---

[2] There are bugs in some window managers that make this difficult or impossible.

You may set the icon of a window to be an arbitrary bitmap by setting its `:icon-bitmap` slot. The value should be a filename which specifies the location of a bitmap file.

In the rare case when you want to have graphics drawn on a parent window appear over the enclosed (child) windows, you can set the `:draw-on-children` of the parent to be non-`nil`. Then any objects that belong to that window will appear on top of the window's subwindows (rather than being hidden by the subwindows). Note: Because of the inability to redraw the graphics in the window and the subwindows simultaneously, objects that will appear over the subwindows must be fast-redraw objects drawn with `:xor` (see section ⟨undefined⟩ [fast-redraw-objects], page ⟨undefined⟩).

The `:background-color` slot of an `inter:interactor-window` can be set to be any `opal:color`. The window will then appear with that as its background color. This is more efficient than putting a rectangle behind all the objects.

When the `:double-buffered-p` slot is T, then an exact copy of the window will be maintained internally by Garnet. Then, when the graphics in the window change, the change occurs first in the copy, and then the changed region is transferred as a pixmap to the original window. This has the potential to reduce flicker in the redrawing of the window. By default, windows do not use this feature because of the extra memory required by the internal buffer.

When the `:save-under` slot is T, then Garnet internally stores the contents of the screen under the window. If the window is made invisible, then Garnet does not have to redraw any Garnet windows under it, because the image can simply be redrawn from the saved contents. This option is used in the `menubar` and `option-button` gadgets.

The `:aggregate` slot specifies an aggregate object to hold all the objects to be displayed in the window. Each window must contain exactly one aggregate in this slot, and all objects in the window should be put into this aggregate. This slot should be set after the window is created, not during the `create-instance` call. This will ensure that the proper demons are running when the slot is set. **Performance hint: specify the top, left, width and height of this aggregate to be formulas depending on the window, rather than using the default formulas, which depend on all of the objects in the aggregate**.

The `:visible` slot specifies if the window is currently visible on the screen or not. In X terminology, this determines if the window is mapped or not. You can set the `:visible` slot at any time to change the visibility (which will take effect after an `update` call).

If you create a window and set the `:position-by-hand` slot to be T, then when you call `opal:update` the first time, the cursor on your screen will change to a prompt asking you where to position the window, and the initial values of `:left` and `:top` will be ignored.

If a window is created with a window object in its `:parent` slot, then the new window will be a sub-window of the parent window. Each window sets up its own coordinate system, so the `:left` and `:top` of the subwindow will be with respect to the parent window. **The parent window must be updated before the subwindow is created.** Using `nil` for the `:parent` makes the window be at the top level. Only top-level windows can be manipulated by the window manager (i.e, by using the mouse).

5.11.1 Window Positioning

When top-level windows first become visible, their `:left` and `:top` slots may change values slightly to accomodate the title bars added by the window manager. When you create a

regular top-level window with a `:top` of 100, for example, the inside edge of the window will appear at 100. The window manager frame of the window (the outside edge) will appear a little higher, depending on the window manager, but somewhere around 25 pixels higher. The window manager then notifies Garnet that this frame has been added by changing the `:top` of the window to 75. The drawable region of the window remains at 100.

When the `:top` of the window is changed (via `s-value`) after it is visible, then it is the outside edge of the window that is being changed, which is the top of the frame. You can always determine the height of the window's title bar in the `:top-border-width` slot (see section [border-widths], page 212). There are corresponding slots for `:left-`, `:right-`, and `:bottom-border-width`. All of these slots are read-only, and are set by Garnet according to your window manager.

When stacking windows in a cascading arrangement, it is sufficient to be consistent in setting their positions either before or after updating them. If the two kinds of position-setting strategies need to be mixed, then the `:top-border-width` of the windows that have already been made visible should be taken into account, versus those that have never been updated.

5.11.2 Border Widths

There are two different meanings of "border widths" in windows. One involves the user-settable thickness of subwindows, and the other kind involves *read-only* widths that are determined by the window manager:

> **Subwindow Border Width** - The `:border-width` slot affects the width of the border on a subwindow. Setting the `:border-width` slot of a subwindow to 0 during its `create-instance` call will cause the window to have no border at all, but setting it to a value larger than the default usually has no effect. Currently, the border width cannot be changed after the window is created.

> **Window Manager Frame Widths** - After a window has been created, the `:left-border-width`, `:right-border-width`, `:top-border-width`, and `:bottom-border-width` slots tell what thicknesses the left, right, top, and bottom borders of the windows actually have. These slots are set by the window manager, and should **not** be set by Garnet users.

5.11.3 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 functions and objects 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 [with-hourglass-sec], page 214, describe the high-level functions that allow you to change the cursor with a single function call.

5.11.3.1 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.[3]

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 `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:

index-1 *index-2*   [Slot Syntax on :cursor]

> 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.

5.11.3.2 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")))
```

---

[3] 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.

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

5.11.3.3 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.

5.11.4 Update Quarantine Slot

A "quarantine slot" named `:in-progress` 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.

Execute `(garnet-load "demos:demo-multiwin")` and `(demo-multiwin:do-go)`.

Artificially create an error situation by executing

```
(kr:with-types-disabled
 (kr:s-value demo-multiwin::OBJ1 :left 'x))
```

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.

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.

After you give OBJ1's `:left` slot a reasonable value and do a **total** 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`.

5.11.5 Windows on other Displays

An important feature of the X window manager is that it allows you to run a process on one machine and have its window appear on another machine. Opal provides a simple way to do this, although many commands have to be given to the Unix Shell.

Let's suppose that you want to run Opal on a machine named `OpalMachine.cs.edu` and you want the windows to appear on a machine named `WindowMachine.cs.edu` (of course you will substitute your own full machine names). Assuming you are sitting at `WindowMachine.cs.edu`, perform the following steps before starting Garnet:

Create an extra Xterm (shell) window and use it to telnet to `OpalMachine.cs.edu` and then log in.

TODO: Slime makes this approach inconvenient, as you have to change the way that emacs launches Slime to set the environment variable before you start lisp. An easy approach is to use uiop, the cross platform utility functions provided by ASDF. If you had a second display running on local machine you could specify it using the following command:

```
(setf (uiop:getenv "DISPLAY") ":1")
```

Then launch Garnet as usual. Unfortunately this is propably not 100% reliable accross platforms. Idealy the display that you want to connect to would be part of an initialization parameter you could pass.

Type the following to `OpalMachine.cs.edu` to tell Opal where the windows should go:

```
setenv DISPLAY WindowMachine.cs.edu:0.0
```

Now go to another Xterm (shell) window on `WindowMachine.cs.edu` and type the following to allow `OpalMachine.cs.edu` to talk to X:

```
xhost + OpalMachine.cs.edu
```

Now go back to the telnet window, and start Lisp and load Garnet and any programs. All windows will now appear on `WindowMachine.cs.edu`.

The exported variables `opal:*screen-width*` and `opal:*screen-height*` contain the width and height of the screen of the machine you are using. Do not set these variables yourself.

5.11.6 Methods and Functions on Window Objects

There are a number of functions that work on window objects, in addition to the methods described in this section. All of the extended accessor functions (`bottom`, `left-side`, `set-center`, etc.) described in section [Extended-accessors], page 157, also work on windows.

`opal:update` *window* **&optional** *total-p*   [Method on `Window`]
> The `update` method updates the image in *window* to reflect changes to the objects contained inside its aggregate. If *total-p* is a non-`nil` value, then the window is erased, and all the components of the window's aggregate are redrawn. This is useful for when the window is exposed or when something is messed up in the window (e.g., after a bug). The default for *total-p* is `nil`, so the window only redraws the changed portions. `update` must be called on a newly-created window before it will be visible. Updating a window also causes its subwindows to be updated.
>
> If `update` crashes into the debugger, this is usually because there is an object with an illegal value attached to the window. In this case, the debugging function `garnet-debug:fix-up-window` is very useful; see the Debugging chapter.

`opal:destroy` *window*   [Method on `Window`]
> The `destroy` method unmaps and destroys the X window, destroys the *window* object, and calls destroy on the window's aggregate and the window's subwindows.

opal:update-all &optional *total-p*   [Function]
> been created but never `updated` (so they are not yet visible). 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.

opal:clean-up *how-to*   [Function]
> This function is useful when debugging for deleting the windows created using Opal. It can delete windows in various ways:

how-to	**Result**])
`:orphans-only` Destroy all orphaned garnet windows. Orphans are described below.	
`:opal` Destroy all garnet windows by calling `xlib:destroy-window` or `ccl:window-close` on orphaned CLX "drawables" and Mac "views", and `opal:destroy` on non-orphaned windows.	
`:opal-set-agg-to-nil` Same as above, but before calling `opal:destroy`, set the aggregate to `nil` so it won't get destroyed as well.	
`:clx` Destroy all Garnet windows by calling `xlib:destroy-window` or `ccl:window-close`. Does not call the `:destroy` method on the window or its aggregate.	

A window is "orphaned" when the Opal name is no longer attached to the CLX drawable or Mac view. This can happen, for example, if you create an instance of a window object, update it, then create another instance of a window with the same name, and update it as well. Then the first window will not be erased and will be orphaned.

The default is `orphans-only`. Another useful value is `:opal`. The other options are mainly useful when attempts to use these fail due to bugs. See also the function `Fix-Up-Window` in the Garnet Debugging chapter *Garnetdebugchapter*.

opal:`convert-coordinates` *win1 x1 y1* **&optional** *win2* (*declare*   [Function]
      (**values** *x2 y2*))
This function converts the coordinates *x1* and *y1* which are in window `win1`'s coordinate system to be in `win2`'s. Either window can be `nil`, in which case the screen is used.

opal:`get-x-cut-buffer` *window*   [Function]
opal:`set-x-cut-buffer` *window newstring*   [Function]
These manipulate the window manager's cut buffer. `get-x-cut-buffer` returns the string that is in the X cut buffer, and `set-x-cut-buffer` sets the string in the X cut buffer.

opal:`raise-window` *window*   [Function]
opal:`lower-window` *window*   [Function]
opal:`iconify-window` *window*   [Function]
opal:`deiconify-window` *window*   [Function]
`Raise-window` moves a window to the front of the screen, so that it is not covered by any other window. `Lower-window` moves a window to the back of the screen. `Iconify-window` changes the window into an icon, and `deiconify-window` changes it back to a window.

5.12 Printing Garnet Windows

The function `make-ps-file` is used to generate a PostScript file for Garnet windows. This file can then be sent directly to any PostScript printer. The file is in "Encapsulated PostScript" format, so that it can also be included in other documents, such as Scribe, LaTeX and FrameMaker on Unix, and Pagemaker on Macintoshes.

The PostScript files generated by this function will produce pictures that are prettier, have much smaller file sizes, and work better in color than those produced by the window utilities like `xwd` and `xpr`. However, a limitation of PostScript is that it is not possible to print with XOR. It is usually possible to change the implementation of Garnet objects or hand-edit the generated PostScript file to simulate the XOR draw function.

By default, the contents of the window and all subwindows are reproduced exactly as on the screen, with the image scaled and centered on the output page. Other options (see the `clip-p` parameter) allow this function to be used to output the entire contents of a window (not just what is on the screen), so it can be used to do the printing for application data that might be in a scrolling-window, for example. This is used in the demo `demo-arith`.

opal:`make-ps-file` *window-or-window-list filename* **&key** *position-x*   [Function]
      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
The only two required parameters to `make-ps-file` are the Garnet window to be printed and the name of the file in which to store the PostScript output. The *window-*

or-window-list parameter may be either a single window or a list of windows. When multiple windows are printed, the space between the windows is filled with the color specified by *background-color*.

The optional arguments affect the position and appearance of the picture:

`position-x`
> Either `:left`, `:center`, or `:right`. Determines the position of the picture on the page horizontally. Ignored if a value is supplied for *left*. Default is `:center`.

`position-y`
> Either `:top`, `:center`, or `:bottom`. Determines the position of the picture on the page vertically. Ignored if a value is supplied for *top*. Default is `:center`.

`left-margin, right-margin, top-margin, bottom-margin`
> These parameters specify the minimum distance (in points) from the corresponding edge of the page to the rendered image. All four values default to 72, which is one inch in PostScript.

`left, top` The distance (in points) from the left and top margins (offsets from *left-margin* and *top-margin*) to the rendered image. The defaults are `nil`, in which case the values of *position-x* and *position-y* are used instead.

`scale-x, scale-y`
> Horizontal and vertical scaling for the image. The default is `nil`, which will ensure that the image fits within the specified margins (the scaling will be the same for vertical and horizontal).

`landscape-p`
> If `nil` (the default) then the top of the picture will be parallel to the short side of the page (portrait). If T, then the picture will be rotated 90 degrees, with the top of the picture parallel to the long side of the page.

`subwindows-p`
> Whether to include the subwindows of the specified window in the image. Default is T.

`borders-p`
> Whether to draw the outline of the window (and subwindows, if any). The allowed values are `t`, `nil`, `:generic`, and `:motif`. The default value of `:motif` gives your image a simulated Motif window manager frame, like the picutres in the Gilt Reference chapter. The value of `:generic` puts a plain black frame around your printed image, with the title of the window centered in the title bar. The value `t` gives the image a thin black border, and `nil` yields no border at all.

`clip-p`   How to clip the objects in the window. Allowed values are:
> - `t` This is the default, which means that the printed picture will look like the screen image. If the graphics inside the window extend outside the borders of the window, then they will be clipped in the printed image.

- **nil** This value causes the window in the printed image to be the same size as the top-level aggregate, whether it is larger or smaller than the actual window. That is, if the window is too small to show all of the objects in its aggregate, then the printed window will be enlarged to show all of the objects. Conversely, if the top-level aggregate is smaller than the dimensions of the window on the screen, then the printed window will be "shrink wrapped" around the objects.
- **(`left top width height`)** A list of screen-relative coordinates that describe absolute pixel positions for the printed 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 chapterly 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.

`color-p`   Whether to generate a file that will print out the real colors of the window's objects (T), or pretend that all the colors are black (automatically produce half-tones for colors, so usually T will work even for color pictures printed on black and white printers.) **Note:** Pixmaps print in full color when they are being displayed on a color screen and the *color-p* parameter is T. 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 using a *color-p* argument of `nil`.

`background-color`

When *window-or-window-list* is a list of windows, the space between the windows will be filled with this color. The value of this parameter may be any Opal color. The default is `opal:white`.

`paper-size`

This 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.

`title, creator, for`

These parameters should take strings to be printed in the header comments of the PostScript file. These comments are sometimes used to print user information on the header sheets of printer output. The default *title* is based on the window's title. The default *creator* is Garnet, and the default *for* is "".

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.

5.13 Saving and Restoring

Opal includes the ability to save and restore Garnet core images. The function `opal:make-image`, described below, can be used to automate the process of closing the connection to the display server and generating a core file. Low-level details are provided below also, in case you need more control over the saving process.

5.13.1 Saving Lisp Images

`opal:make-image` *filename* **&key** *quit* (*verbose* **t**) (*gc* **t**) **&rest**   [Function]
         other-args

> 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.
>
> The *other-args* parameter is supplied to accomodate the miscellaneous parameters of each lisp vendor's image-saving function. For example, with Allegro's `dumplisp` command, you can supply the keywords `:libfile` and `:flush-source-info?`. Since `opal:make-image` calls `dumplisp` for Allegro, you can supply the extra parameters to `opal:make-image` and they will be passed on to `dumplisp`. Therefore, it is not necessary to call your lisp's image-saving function chapterly; you can always pass the additional desired parameters to `opal:make-image`.
>
> 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`. Some lisps (like Allegro) allow you to restart the saved image just by executing the binary file, while others (like CMUCL) require that the binary file is passed as an argument when the standard lisp image is executed. Consult your lisp's reference chapter for instructions on restarting your saved image.

5.13.2 Saving Lisp Images Manually in X11

It recommended that you use `opal:make-image` whenever possible to save images of lisp. In particular, restarted images of MCL containing Garnet that were created by other means will probably not work right, due to the skipping of initialization steps that would have been performed automatically if the image had been saved with `opal:make-image`.

When you do not want to use the function `opal:make-image` to generate an executable lisp image, and instead want to perform the saving procedure chapterly, you can use the functions `opal:disconnect-garnet` and `opal:reconnect-garnet`, along with your implementation-dependent function for saving lisp images.

       `opal:Disconnect-Garnet[function], page 90`

       `opal:Reconnect-Garnet &optional` *display-name* `[function], page 90`

Before saving a core image of Garnet, you must first close all connections to the X server by calling `opal:disconnect-garnet`. All windows which are currently visible will disappear (but will reappear when `opal:reconnect-garnet` is executed).

While the connection to the X server is closed, you may save a core image of Garnet by calling the appropriate Lisp command. In Lucid Lisp the command is `(disksave)`, in Allegro Lisp it is `(excl:dumplisp)`, and in CMU Common Lisp it is `(ext:save-lisp)`. Consult your Common Lisp chapter to find the disk save command for your version of Common Lisp, as well as how to start up a saved Lisp core.

It is usually convenient to specify `opal:reconnect-garnet` as the *restart-function* during your save of lisp. For example, the following instruction will cause `opal:reconnect-garnet` to be invoked in Allegro lisp whenever the saved lisp is restarted:

```
(excl:dumplisp :name "garnet-image" :restart-function #'opal:reconnect-garnet)
```

Otherwise, you will need to call `opal:reconnect-garnet` chapterly when the lisp image is restarted in order to restore the connection to the server and make all Garnet windows visible again.

If the *display-name* parameter to `opal:reconnect-garnet` is specified, it should be the name of a machine (e.g., "ecp.garnet.cs.cmu.edu"). If not specified, *display-name* defaults to the current machine.

5.14 Utility Functions

5.14.1 Executing Unix Commands

`opal:shell-exec` *command*   [Function]

The 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.

In Lucid, CMUCL, and LispWorks, the shell spawned by `opal:shell-exec` is `/bin/sh`. In Allegro and CLISP, the shell is the user's default. Executing this function in other lisps, including MCL, causes an error (please let the Garnet group know how to enhance this function to run in your lisp).

5.14.2 Testing Operating System Directories

This 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/"`. If your lisp is running on a Unix system, 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. On the Mac, a lisp-specific directory command is executed.

5.15 Aggregadgets and Interactors

The *Aggregadgets* module makes it much easier to create instances of an aggregate and all its components. With an aggregadget, you only have to define the aggregate and its components once, and then when you create an instance, it creates all of the components automatically. Aggregadgets also allow lists of items to be created by simply giving a single prototype for all the list elements, and a controlling value that the list iterates through. Aggregadgets are described in their own chapter *AggregadgetsChapter*.

Interactors are used to handle all input from the user. Interactor objects control input and perform actions on Opal graphical objects. There are high-level interactor objects to handle all the common forms of mouse and keyboard input. Interactors are described in their own chapter *InterChapter*.

Together Opal and Interactors should hide all details of X and QuickDraw from the programmer. There should never be a need to reference any symbols in `xlib` or `ccl`.

5.16 Creating New Graphical Objects

An interesting feature of object-oriented programming in Garnet is that users are expected to create new objects only by combining existing objects, not by writing new methods. Therefore, you should only need to use Aggregadgets to create new kinds of graphical objects. It should never be necessary to create a new `:draw` method, for example.

If for some reason, a new kind of primitive object is desired (for example, a spline or some other primitive not currently supplied by X11), then contact the Garnet group for information about how this can be done. Due to the complexities of X11, Mac QuickDraw, and automatic update and redrawing of objects in Opal, it is not particularly easy to create new primitives.

```
****************************************************************
@b{BAM: I am not sure this section is correct.}


Most users of Opal will only use the pre-defined graphical objects, and
will combine them into aggregates and use formulas to attach them together.
It will be rare to create new kinds of graphical objects. This should only
be needed when new primitives are available, such as splines.


This chapter discusses how to create new types of graphical objects, should
that be necessary.


@section Internal slots in graphical objects
@node Internal slots in graphical objects


There are numerous extra slots in all graphical objects that are used
internally by Opal. This section will attempt to describe these slots and
their potential uses when designing new graphical objects.


@subsection :update-slots
@node :update-slots
@anchor{opaldemons}
```

```
@cindex{update-slots}
The @code{:update-slots} slot contains an association list of all slots in
the object that affect the output picture from the object.
For example:

@example
* (gv opal:arc :update-slots)
((:VISIBLE) (:LINE-STYLE) (:FILLING-STYLE) (:DRAW-FUNCTION) (:LEFT)
 (:TOP) (:WIDTH) (:HEIGHT) (:ANGLE1) (:ANGLE2))
@end example

If any of the values of these objects slots in an instance of an
@code{opal:arc} object change, the instance will need to be redrawn at the
next window update.

@cindex{demons}
Anytime a slot on the @code{:update-slots} list is changed (either with
@code{s-value} or by a formula being invalidated) the KR's invalidate demon
is called with the object, the slot, and the slot's value on the association
list.

Opal doesn't use the second value of the association pair, so it should be
left as @code{nil}.

When creating an object that is a specialized instance of a prototype
object, one should inherit all the slots on the @code{:update-slots}
list, and then add any others as necessary. Commonly this is done by
something of the form:

@example
(create-instance 'opal:arc opal:graphical-object
  ...
  (:update-slots
   (append (gv opal:graphical-object :update-slots)
           '((:left) (:top) (:width) (:height)
              (:angle1) (:angle2))))
  ...)
@end example

By doing this, you insure that all the necessary slots are inherited, and
add any new slots as necessary.

@subsection :drawable
@node :drawable

@cindex{drawable}
```

The @code{:drawable} slot contains a structure that is the CLX drawable
object that the
object is to display itself into when it is sent a @code{draw} message. This
object may not be the physical window that the object is to be displayed
into, it may be a pixmap that is double buffered onto the screen somewhere
in the update algorithm. All objects should trust the value in this slot,
even though is may not correspond to the drawable of their window. This
slot may not contain a value until the object (or one of its parents) is
placed in a Garnet window.

@subsection :display-info
@node :display-info
@cindex{display-info}

The @code{:display-info} slot holds information used by many of the CLX
primitives for
computation, and drawing. Once an object is placed in a window, this slot
contains an opal structure:

@example

(defstruct (display-info
            (:print-function display-info-printer))
  display
  screen
  root-window
  default-gcontext)

@end example

The form @code{(display-info-@emph{xxx} (gv @emph{object} :display-info))}
returns the @emph{xxx} structure from @emph{object}'s @code{:display-info} slot.
These fields are useful as follows:

@cindex{display-info-display}
@cindex{display}
@cindex{display-info-screen}
@cindex{screen}
@cindex{display-info-root-window}
@cindex{root-window}
@cindex{display-info-default-gcontext}
@cindex{default-gcontext}
@begin[itemize]

@code{display} is the CLX structure corresponding to the current display
connection to the X server. This is used in calls that affect or query the
server directly, such as @t<xlib:open-font>, @code{xlib:display-force-output},

and @t<xlib:global-pointer-position>.

@code{screen} is the CLX structure containing information about the window's
screen. This is used most often with structure accessors to get
information on values for the screen's white and black pixels, width and
height in pixels or millimeters.

@code{root-window} is the CLX window that corresponds to
@code{(xlib:screen-root @emph{screen})} for use in calls to
@code{xlib:create-window}.

@code{default-context} is a CLX graphical context structure used all drawing
requests. Opal maintains a cache on this object, so it should not be
changed. It is acceptable to use this structure in an
@code{xlib:with-gcontext} form when it is necessary to modify a gcontext
outside the bounds of @code{with-filling-styles @r[and]with-line-styles}.

@end[itemize]

@subsection :x-tiles
@node :x-tiles

@cindex{x-tiles}
The @code{x-tiles} slot contains a formula that computes a pixmap for use in drawing▮
tiled lines, or pattern filled regions. The formula evaluates to a cons
cell the car of which is the pixmap to use for tiling lines, and the cdr of
which is a pixmap to use when drawing fillings. These are computed from
values in the object's @code{:line-style @r[and] :filling-style} slots.

@subsection :x-draw-function
@node :x-draw-function
@cindex{x-draw-function}

This slot contains a formula that is used to compute the CLX drawing
function from the @code{:draw-function} slot. It probably won't ever be
necessary to change the formula in this slot.

@section Methods on all graphical objects
@node Methods on all graphical objects

The following methods are defined on all graphical objects and may be
specialized upon when creating new classes of graphical objects.

@cindex{draw}
@example
opal:Draw @emph{graphical-object}@ref{method}
@end example

The @code{draw} method on a graphical object causes the object to display
itself in the window of its aggregate. This is only called by the update
methods, never directly by users of Opal.

@cindex{initialize}
@example
initialize @emph{graphical-object}@ref{method}
@end example

This method is called immediately after an instance of an object is
created. It is passed the new object as its only argument.

@example
opal:Point-In-Gob @emph{graphical-object x y}@ref{method}
@end example

This method should be provided for all new objects. It is used by
@code{point-to-component} and @code{point-to-leaf} to query an object
for a hit. The method should return @code{t} if the object is under the
point @math{(x, y)}. This function should also take into account the
values in the @code{:hit-threshold}, @code{:select-outline-only}, and
@code{:visible} slots of the object, as described in section
@ref{stdfuncs}. Objects that are not visible should return @code{nil}.

@cindex{fix-properties}
@example
opal::Fix-Properties @emph{graphical-object changed-slots}@ref{method}
@end example

This method is called on aggregates and windows at the time when the update
algorithm passes them during an update. The method is called with an
object, and a list of slots that have changed since it was last called.
This function is often useful for calling functions that cannot
easily be put into formulas.

Currently @code{fix-properties} is not called on graphical objects, but this
functionality can be added to Opal by talking to the maintainer.

@section Draw Methods
@node Draw Methods

There are several things that are worthy of note when working on @code{draw}
methods for new objects.

Objects @emph{must} draw entirely within their bounding box. The redisplay
algorithm will not work properly if things are drawn outside of their

bounding boxes.

There are two macros for use in writing draw methods that prepare a
gcontext from the gcontext cache that is appropriate for drawing outlines
or fillings as described by the values in the @code{:line-style} and
@code{:filling-style} slots of the object.

@cindex{with-filling-styles}

@defmac with-filling-styles (variable graphical-object) body

This form executes the forms inside @emph{body} with @emph{variable} bound to a█
CLX gcontext structure suitable for drawing the filling of an object with
respect to @emph{graphical-object}'s filling style object in the slot
@code{:filling-style}.

@end defmac

@cindex{with-line-styles}

@defmac with-line-styles (variable graphical-object) body

This form executes the forms inside @emph{body} with @emph{variable} bound to a█
CLX gcontext structure suitable for drawing the outline of an object with
respect to @emph{graphical-object}'s line style object in the slot
@code{:line-style}.

These forms are commonly used like this:

```
@example
(define-method :draw opal:polyline (polyline)
  (let ((point-list (gv polyline :point-list))
        (drawable (gv polyline :window :drawable)))
    (with-filling-styles (gcontext polyline)
      (xlib:draw-lines drawable gcontext
                       point-list :fill-p t))
    (with-line-styles (gcontext polyline)
      (xlib:draw-lines drawable gcontext
                       point-list))))
@end example
```

@end defmac

@anchor{creating-new-gobs}
****

⟨undefined⟩ [References], page ⟨undefined⟩,

6 Interactors: Encapsulating Mouse and Keyboard Behaviors

by Brad A. Myers, James A. Landay, Andrew Mickish

10 May 2020

6.1 Abstract

This document describes a set of objects which encapsulate mouse and keyboard behaviors. The motivation is to separate the complexities of input device handling from the other parts of the user interface. We have tried to identify some common mouse and keyboard behaviors and implement them in a separate place. There are only a small number of interactor types, but they are parameterized in a way that will support a wide range of different interaction techniques. These interactors form the basis for all interaction in the Garnet system.

6.2 Introduction

This document is the reference chapter for the *Interactors* system, which is part of the Garnet User Interface Development System *Myers, 1989a*. The Interactors module is responsible for handling all of the input from the user. Currently, this includes handling the mouse and keyboard.

The design of the Interactors is based on the observation that there are only a few kinds of behaviors that are typically used in graphical user interfaces. Examples of these behaviors are selecting one of a set (as in a menu), moving or growing with the mouse, accepting keyboard typing, etc. Currently, in Garnet, there are only nine types of interactive behavior, but these are all that is necessary for the interfaces that Garnet supports. These behaviors are provided in interactor objects. When the programmer wants to make a graphical object (created using Opal; the Garnet graphics package) respond to input, an interactor object is created and attached to the graphical object. In general, the graphics and behavior objects are created and maintained separately, in order to make the code easier to create and maintain.

This technique of having objects respond to inputs is quite novel, and different from the normal method in other graphical object systems. In others, each type of object is responsible for accepting a stream of mouse and keyboard events and managing the behavior. Here, the interactors handle the events internally, and cause the graphical objects to behave in the desired way.

The Interactors, like the rest of Garnet, are implemented in Common Lisp for X11 and Macintosh QuickDraw. Interactors are set up to work with the Opal graphics package and the KR object and constraint systems, which are all part of Garnet.

The motivation and an overview of the Interactors system is described in more detail in conference papers *Myers, 1989b*, *Myers, 1990*.

Often, interactors will be included in the definition of Aggregadgets. See Section 7.4.1 [Aggregadgets], page 321, for a description of how this works.

6.3 Advantages of Interactors

The design for interactors makes creating graphical interfaces easier. Other advantages of the interactors are that:

- They are entirely "look" independent; any graphics can be attached to a particular "feel."

- They allow the details of the behavior of objects to be separated from the application and from the graphics, which has long been a goal of user interface software design.

- They support multiple input devices operating in parallel.

- They simulate multiple processing. Different applications can be running in different windows, and the operations attached to objects in all the windows will execute whenever the mouse is pressed over them. The applications all exist in the same Common Lisp process, but the interactors insure that the events go to the correct application and that the correct procedures are called. If the application is written correctly (e.g., without global variables), multiple instantiations of the *same* application can exist in the same process.

- All of the complexities of *X11* graphics and event handling are hidden by Opal and the Interactors package. This makes Garnet much easier to use than *X11* directly, and allows applications written in Garnet to be run on anly platform that supports *X11* without modification.

6.4 Overview of Interactor Operation

The interactors sub-system resides in the `inter` package. We recommend that programmers explicitly reference names from the `inter` package, for example: `inter:menu-interactor`, but you can also get complete access to all exported symbols by doing a (`use-package :inter`). All of the symbols referenced in this document are exported.

In a typical mouse-based operation, the end user will press down on a mouse button to start the operation, move the mouse around with the button depressed, and then release to confirm the operation. For example, in a menu, the user will press down over one menu item to start the operation, move the mouse to the desired item, and then release.

Consequently, the interactors have two modes: waiting and running. An interactor is waiting for its start event (like a mouse button down) and after that, it is waiting for its stop event, after which it stops running and goes back to waiting.

In fact, interactors are somewhat more complicated because they can be aborted at any time and because there are often active regions of the screen outside of which the interactor does not operate. The full description of the operation is presented in section Section 6.6.8 [Operation], page 235.

All the interactors operate by setting specific slots in the graphic objects.[1] For example, the menu interactor sets a slot called `:selected` to show which menu item is selected, and the moving and growing interactor sets a slot called `:box`. Typically, the objects will contain constraints that tie appropriate graphical properties to these special slots. For example, a movable rectangle would typically contain the following constraints so it will follow the mouse:

```
(create-instance 'moving-rectangle opal:rectangle
   (:box '(80 20 100 150))
   (:left (o-formula (first (gvl :box)))))
```

---

[1] "Slots" are the "instance variables" of the objects.

```
(:top (o-formula (second (gvl :box))))
(:width (o-formula (third (gvl :box))))
(:height (o-formula (fourth (gvl :box)))))
```

The initial size and position for the rectangle are in the `:box` slot. When an interactor changes the box slot, the `:left, :top, :width,` and `:height` slots would change automatically based on constraints.

If the constraints (formulas) were *not* there, the interactor would still change the `:box` slot, *but nothing would change on the screen*, since the rectangle's display is controlled by `:left, :top, :width,` and `:height`, not by `:box`. The motivation for setting this extra slot, is to allow application-specific filtering on the values. For example, if you do not want the object to move vertically, you can simply eliminate the formula in the `:top` slot.

6.5 Simple Interactor Creation

To use interactors, you need to create *interactor-window*s for the interactors to work in (windows are fully documented in [Opal: The Garnet Graphical Object System], page 148. To create an `interactor-window`, you use the standard *KR* `create-instance` function. For example:

```
(create-instance 'mywindow inter:interactor-window
    (:left 100)(:top 10)
    (:width 400)(:height 500)
    (:title "My Window"))

(opal:update mywindow)
```

To create interactor objects, you also use the `create-instance` function. Each interactor has a large number of optional parameters, which are described in detail in the rest of this chapter. It must be emphasized, however, that normally it is not necessary to supply very many of these. For example, the following code creates an interactor that causes the `moving-rectangle` (defined above) to move around inside `mywindow`:

```
(create-instance 'mymover inter:move-grow-interactor
    (:start-where (list :in moving-rectangle))
    (:window mywindow))
```

This interactor will use the default start and stop events, which are the left mouse button down and up respectively. All the other aspects of the behavior also will use their default values (as described below).

Several implementations of lisp allow interactors to run automatically (see section [The Main Event Loop], page 234). If you are *not* running in CMU, LispWorks, Allegro, Lucid, or MCL Commonlisp, then you need to execute the following function to make the interactor run:

```
(inter:main-event-loop)
```

This function does not exit, so you have to type `^C` (or whatever your operating system break character is) to the Lisp window when you are finished (or press the `F1` key (or whatever your Garnet break key is, section [`main-event-loop`], page 234)).

As another example, here is a complete, minimal "goodbye world" program, that creates
a window with a button that causes the window to go away (created from scratch, without
using any predefined gadgets).

```
;;; using the kr package, but no others, is the "garnet style"
(use-package "kr")
;;; first create the graphics; see the opal chapter for explanations
(create-instance 'mywindow inter:interactor-window
   (:left 100)(:top 10)
   (:width 125)(:height 25)
   (:title "my window"))
(s-value mywindow :aggregate (create-instance 'myagg opal:aggregate))

(create-instance 'mytext opal:text
   (:string "goodbye world")
   (:left 2)(:top 5))
(opal:add-component myagg mytext)
(opal:update mywindow)

;;; now add the interactor
(create-instance nil inter:button-interactor
   (:window mywindow)
   (:start-where (list :in mytext))
   (:continuous nil) ; happen immediately on the downpress
   (:final-function #'(lambda (inter final-obj-over)
                        (opal:destroy mywindow)
                        ;; the next line is needed unless you are running cmu lisp
;; or you are running the main-event-loop process in the
                        ;; background in allegro, lucid, or lispworks
#-(or cmu allegro lucid lispworks) (inter:exit-main-event-loop)))
;;; if not cmu lisp, or if not running the background main-event-loop pro-
cess in
;;; allegro, lispworks, or lucid lisp, then the following is needed to run the interac
#-(or cmu allegro lucid lispworks) (inter:main-event-loop)
```

6.6 Overview of the Section

This section is organized as follows. Section [The Main Event Loop], page 234, discusses the
`main-event-loop`, which allows you to run interactors while automatically updating the
appearance of the windows. Section [Operation], page 235, describes how interactors work
in detail. Section [Mouse and Keyboard Accelerators], page 249, describes the definition
and operation of global accelerators. Section [Slots of all Interactors], page 250, lists all the
slots that are common to all interactors. section [Specific Interactors], page 254, describes
all the interactors that are provided. Section [Transcripts], page 302, describes how to
make transcripts of events. Finally, section [Advanced Features], page 303, describes some
advanced features.

Normally, you will not need most of the information in this chapter. To make an object
respond to the mouse, look in section [Specific Interactors], page 254, to find the interactor

you need, then check its introduction to see how to set up the constraints in your graphical objects so that they will respond to the interactor, and to see what parameters of the interactor you need to set. you can usually ignore the advanced customization sections.

6.6.1 the Main Event Loop

CMU Common Lisp *McDonald, 1987* supports sending events to the appropriate windows internally. Therefore, under cmu Common Lisp, the interactors begin to run immediately when they are created, and run continuously until they are terminated. While they are running, you can still type commands to the lisp listener (the `read-eval-print` loop).

To get the same effect on other lisps, Garnet uses the multiple process mechanism of Lucid, Allegro, Lispworks, and MCL Common Lisps. You usually do not need to worry about the information in this section if you are using CMU, Allegro, Lucid, or MCL Common Lisp, but you will probably need to go through an initialization phase for multiprocessing in Lispworks (See [Lispworks], page 12.)

Note: `main-event-loop` also handles Opal window refreshing, so graphical objects will not be redrawn automatically in other lisps unless this function is executing.

6.6.2 `main-event-loop`

Under other Common Lisps (like AKCL and CLISP), you need to explicitly start and stop the main loop that listens for X11 events. it is always ok to call the `main-event-loop` function, because it does nothing if it is not needed. Therefore, after all the objects and interactors have been created, and after the `opal:update` call has been made, you must call the `inter:main-event-loop` procedure. This loops waiting and handling X11 events until explicitly stopped by typing `^c` (or whatever is your operating system break character) to the lisp listener window, or until you hit the garnet break key while the mouse is in a garnet window. This is defined by the global variable `inter:*garnet-break-key*`, and is bound to `F1` by default. you can simply setf `inter:*garnet-break-key*` to some other character if you want to use `f1` for something else.

The other way for a program to exit `main-event-loop` is for it to call the procedure `inter:exit-main-event-loop`. Typically, `inter:main-event-loop` will be called at the end of your set up routine, and `inter:exit-main-event-loop` will be called from your quit routine, as in the example of section [eventloopexample], page 233.

`inter:main-event-loop` **&optional** *inter-window*   [Function]
`inter:exit-main-event-loop`   [Function]
>   The optional window to `main-event-loop` is used to tell which display to use. If not
>   supplied, it uses the default opal display. You only need to supply a parameter if you
>   have a single lisp process talking to multiple displays.

6.6.3 `main-event-loop` Process

By default, garnet spawns a background process in Allegro, Lucid, and Lispworks, which will run the interactor's main-event-loop while simultaneously allowing you to use the ordinary lisp listener. This means that you can use the lisp listener without having to hit the garnet break key (usually `f1`).

Some programs seem to have trouble with this process. If your system doesn't work, try killing the main-event-loop process and executing (`inter:main-event-loop`) explicitly. in

MCL, the background process is controlled by MCL itself, and cannot be killed. However, you might be able to break out of an infinite loop (or otherwise get MCL's attention) by executing the abort command (control-comma) or the reset command (control-period).

6.6.4 Launching and Killing the `main-event-loop-process`

`opal:launch-main-event-loop-process`   [Function]
`opal:kill-main-event-loop-process`   [Function]
These are the top-level functions used for starting and stopping the `main-event-loop` process. You may need to call `launch-main-event-loop-process` if the process is killed explicitly or if the process crashes due to a bug.

While the main-event-loop background process is running, calling (`inter:main-event-loop`), hitting the garnet break key, and calling `launch-main-event-loop-process` all have no effect.

You can kill the background main-event-loop process by executing `kill-main-event-loop-process`, but normally you should not have to, even if you encounter an error and are thrown in the debugger. If you call it when the main-event-loop process is not running, there is no effect.

`launch-main-event-loop-process` and `kill-main-event-loop-process` belong to the opal package because `opal:reconnect-garnet` and `opal:disconnect-garnet` need to call them.

6.6.5 `launch-process-p`

In the `garnet-loader`, there is a switch called `user::launch-process-p` which tells whether or not garnet should automatically call `launch-main-event-loop-process` at load time. You can edit the `garnet-loader` to change the default value of this variable, or you can `setf` the variable before loading `garnet-loader`.

6.6.6 `main-event-loop-process-running-p`

`opal:main-event-loop-process-running-p`   [Function]
This function tells you whether the parallel `main-event-loop` process is running, and is not in the debugger.

6.6.7 Operation

6.6.8 Creating and Destroying

For interactors to be used, they must operate on objects that appear in Garnet windows. The `inter:interactor-window` prototype is described in the Opal Chapter. To create an interactor window, use:

`create-instance` *name inter:interactor-window* (`slot value`)(`slot`   [Function]
    `value`)...
This creates an interactor window named *name* (which will usually be a quoted symbol like '`mywindow` or `nil`). If *name* is `nil`, then a system-supplied name is used. This returns the new window. The `:left`, `:top`, `:width`, and `:height` (and other

parameters) are given just as for all objects. Note that the window is not visible
("mapped") until an `opal:update` call is made on it:

>   `(opal:update an-interactor-window)`

To create an interactor, use:

>   `(create-instance name Inter:InteractorType (slot value) (slot value)...)`■

This creates an interactor named *name* (which can be `nil` if a system-supplied name is
desired) that is an instance of *InteractorType* (which will be one of the specific types
described in section [Specific Interactors], page 254, such as `button-interactor`,
`menu-interactor`, etc. The slots and values are the other parameters to the new
interactor, as described in the rest of this chapter. The create-instance call returns
the interactor.

>   `opal:Destroy an-interactor &optional (erase T)` [No value for ''method'']■

>   `opal:Destroy an-interactor-window` [No value for ''method'']

Invoking this method destroys an interactor or window. If *erase* is T, then the inter-
actor is aborted and deallocated. If *erase* is `nil`, it is just destroyed. Use `nil` when
the window the interactor is in is going to be destroyed anyway. Normally, it is not
necessary to call this on interactors since they are destroyed automatically when the
window they are associated with is destroyed.

Invoking this method on a window destroys the window, all objects in it, and all
interactors associated with it.

6.7 Continuous

Interactors can either be *continuous* or not. A continuous interactor operates between a
start and stop event. For example, a Move-Grow interactor might start the object following
the mouse when the left button goes down, and continue to move the object until the button
is released. When the button is released, the interactor will stop, and the object will stay in
the final place. Similarly, a menu interactor can be continuous to show the current selection
while the mouse is moving, but only make the final selection and do the associated action
when the button is released.

The programmer might want other interactors to operate only once at the time the
start-event happens. For example, a non-continuous `Button-Interactor` can be used to
execute some action when the `delete` key is hit on the keyboard.

The `:continuous` slot of an interactor controls whether the interactor is continuous or
not. The default is T.

Many interactors will do reasonable things for both values of `:continuous`. For example,
a continuous `button-interactor` would allow allow the user to press down on the graphical
button, and then move the mouse around. It would only execute the action if the mouse
button is released over the graphical button. This is the way Macintosh buttons work. A
non-continuous button would simply execute as soon as the mouse-button was hit over the
graphical button, and not wait for the release.

6.8 Feedback

When an interactor is continuous, there is usually some feedback to show the user what is happening. For example, when an object is being moved with the mouse, the object usually moves around following the mouse. Sometimes, it is desirable that the actual object not move, but rather that a special *feedback object* follows the mouse, and then the real object moves only when the interaction is complete.

The interactors support this through the use of the `:feedback-obj` slot. If a graphical object is supplied as the value of this slot, then the interactor will modify this object while it is running, and only modify the "real" object when the interaction is complete (section [where], page 240, discusses how the interactor finds the "real" object). If no value is supplied in this slot (or if `nil` is specified), then the interactor will modify the actual object while it is running. In either case, the operation can still be aborted, since the interactor saves enough state to return the objects to their initial configuration if the user requests an abort.

Typically, the feedback object will need the same kinds of constraints as the real object, in order to follow the mouse. For example, a feedback object for a `Move-Grow-Interactor` would need formulas to the `:box` slot. The sections on the various specific interactors discuss the slots that the interactors set in the feedback and real objects.

6.9 Events

An interactor will start running when its *start event* occurs and continue to run until a *stop event* occurs. There may also be an *abort event* that will prematurely cause it to exit and restore the status as if it had not started.

An "event" is usually a transition of a mouse button or keyboard key. Interactors provide a lot of flexibility as to the kinds of events that can be used for start, stop and abort.

6.9.1 Keyboard and Mouse Events

Events can be a mouse button down or up transition, or any keyboard key. The names for the mouse buttons are `:leftdown`, `:middledown`, and `:rightdown` (simulating multiple mouse buttons on the Mac is discussed in section [mac-keys], page 237). Keyboard keys are named by their Common Lisp character, such as `#\g`, `#\a`, etc. Note that `#\g` is lower-case "g" and `#\G` is upper case "G" (shift-g).

When specifying shift keys on keyboard events, it is important to be careful about the "\". For example, `:control-g` is *upper* case "G" and `:control-\g` is *lower* case "g" (note the extra "\"). You may also use the form `:|CONTROL-g|`, which is equivalent to `:control-\g` (when using vertical bars, you must put the CONTROL in upper-case). It is not legal to use the `shift` modifier with keyboard keys.

Events can also be specified in a more generic manner using `:any-leftdown`, `:any-middledown`, `:any-rightdown`, `:any-leftup`, `:any-middleup`, `:any-rightup`, `:any-mousedown`, `:any-mouseup`, and `:any-keyboard`. For these, the event will be accepted no matter what modifier keys are down.

6.9.2 "Middledown" and "Rightdown" on the Mac

To simulate the three-button mouse on the Macintosh, we use keyboard keys in place of the buttons. By default, the keys are `F13`, `F14`, and `F15` for the left, middle, and right

mouse buttons, respectively. The real mouse button is also mapped to `:leftdown`, so you can specify mouse events as usual on the Mac (e.g., `:rightdown`). The Overview section at the beginning of this chapter provides instructions for customizing the keys that simulate the mouse buttons, and provides instructions for a small utility that changes the keys to be used from function keys to arrow keys.

6.9.3 Modifiers (Shift, Control, Meta)

Various modifier keys can be specified for the event. The valid prefixes are `shift`, `control`, and `meta`. For example, `:control-meta-leftdown` will only be true when the left mouse button goes down while both the Control and Meta keys are held down. When using a conglomerate keyword like `:shift-meta-middleup`, the order in which the prefixes are listed matters. The required order for the prefixes is: `shift`, `control`, `meta`. For instance, `:shift-control-leftdown` is legal; `:control-shift-leftdown` is not.

As with MCL itself, the Option key is the "Meta" modifier on the Mac. There is no way to access the Mac's Command key through Garnet.

6.9.4 Window Enter and Leave Events

Sometimes it is useful to know when the cursor is inside the window. Garnet has the ability to generate 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 (gv inter :window)
  :background-color
                            ; :start-char is described in section [specialslots],
  page 310
  (if (eq :window-enter (gv inter :start-char))
      opal:red
      opal:white)))))
```

6.9.5 Double-Clicking

Garnet also supports double-clicking of the mouse buttons. When the variable `inter:*Double-Click-Time*` has a non-NIL value, then it is the time in milleseconds of how fast clicks must be to be considered double-clicking. By default, double clicking is enabled with a time of 250 milleseconds. When the user double-clicks, Garnet first reports the first press and release, and then a `:double-xxx` press and then a *regular* release. For example, the events that will be reported on a double-click of the left button are: `:leftdown :leftup :double-leftdown :leftup`. Note that the normal `-up` events are used. You can use the normal `:shift, :control,` and `:meta` modifiers in the usual order, before the `double-`. For example: `:shift-control-double-middledown`. If you specify the start-event of a continuous interactor to use a `:double-` form, then the correct stop event will be generated automatically. If you have both single and double click interactors, then you should be careful that it is OK for the single click one to run before the double-click one.

If you want to handle triple-clicks, quadruple-clicks, etc., then you have to count the clicks yourself. Garnet will continue to return `:double-xxx` as long as the clicks are fast enough. When the user pauses too long, there will be a regular `:xxxdown` in between. Therefore, for triple click, the events will be: `:leftdown, :leftup, :double-leftdown, :leftup, :double-leftdown, :leftup` whereas for double-click-pause-click, the events will be: `:leftdown, :leftup, :double-leftdown, :leftup, :leftdown, :leftup`.

6.9.6 Function Keys, Arrows Keys, and Others

The various special keys on the keyboard use special keywords. For example, `:uparrow`, `:delete`, `:F9`, etc. The prefixes are added in the same way as for mouse buttons (e.g., `:control-F3`). The arrow keys are almost always named `:uparrow`, `:downarrow`, `:leftarrow`, and `:rightarrow` (and so there are no bindings for `:R8` (uparrow), `:R10` (leftarrow), `:R12` (rightarrow), and `:R14` (downarrow) on the Sun keyboard). On the Mac, some users prefer to change their arrow keys to generate mouse events (see section [mac-keys], page 237). To see what the Lisp character is for an event, turn on event tracing using `(Inter:Trace-Inter :event)` and then type the key in some interactor window, as described in the Garnet Debugging chapter. If you have keys on your keyboard that are not handled by Garnet, it is easy to add them. See the section on "Keyboard Keys" in the Overview Chapter, and then please send the bindings to `garnet@cs.cmu.edu` so we can add them to future versions of Garnet.

You can control whether Garnet raises an error when an undefined keyboard key is hit. The default for `inter::*ignore-undefined-keys*` is T, which means that the keys are simply ignored. If you set this variable to NIL, then an error will be raised if you hit a key with no definition.

6.9.7 Multiple Events

The event specification can also be a set of events, with an optional exception list. In this case, the event descriptor is a list, rather than a single event. If there are exceptions, these should be at the end of the list after the keyword `:except`. For example, the following lists are legal values when an event is called for (as in the `:start-event` slot):

```
(:any-leftdown :any-rightdown)
(:any-mousedown #\RETURN)   ;; any mouse button down or the RETURN key
```

```
(:any-mousedown :except :leftdown :shift-leftdown)
(:any-keyboard :any-rightdown :except #\b #\a #\r)
```

6.9.8 Special Values T and `nil`

Finally, the event specification can be T or `nil`. T matches any event and NIL matches no event. Therefore, if `nil` is used for the `:start-event`, then the interactor will never start by itself (which can be useful for interactors that are explicitly started by a programmer). If T is used for the `:start-event`, the interactor will start immediately when it is created, rather than waiting for an event. Similarly, if `stop-event` is `nil`, the interactor will never stop by itself.

6.10 Values for the "Where" slots

6.10.1 Introduction

In addition to specifying what events cause interactors to start and stop, you must also specify *where* the mouse should be when the interaction starts using the slot `:start-where`. The format for the "where" arguments is usually a list with a keyword at the front, and an object afterwards. For example, (`:in myrect`). These lists can be conveniently created either using `list` or back-quote:

```
(:start-where (list :in MYRECT))
(:start-where `(:in ,MYRECT))
```

For the backquote version, be sure to put a comma before the object names.

The "where" specification often serves two purposes: it specifies where the interaction should start and what object the interaction should work on.

Unlike some other systems, the Interactors in Garnet will work on any of a set of objects. For example, a single menu interactor will handle all the items of the menu, and a moving interactor will move any of a set of objects. Typically, the object to be operated on is chosen by the user when the start event happens. For example, the move interactor may move the object that the mouse is pressed down over. This one object continues to move until the mouse is released.

Some of the interactors have an optional parameter called `:obj-to-change`, where you can specify a different object to operate on than the one returned by the `:start-where` specification.

One thing to be careful about is that some slots of the *graphical objects themselves* affect how they are picked, in particular, the `:hit-threshold`, `:select-outline-only`, and `:pretend-to-be-leaf` slots. See section [hitthreshold], page 245.

6.10.2 Running-where

There are actually two "where" arguments to each interactor. One is the place where the mouse should be for the interaction to start (`:start-where`). The other is the active area for the interaction (`:running-where`). The default value for the running-where slot is usually the same as the start-where slot. As an example of when you might want them to be different, with an object that moves with the mouse, you might want to start moving when the press was over the object itself (so `:start-where` might be (`:in MY-OBJ`)) but

continue moving while the mouse is anywhere over the background (so `:running-where` might be (`:in MY-BACKGROUND-OBJ`)).

6.10.3 Kinds of "where"

There are a few basic kinds of "where" values.

Single object:
> These operate on a single object and check if the mouse is inside of it.

Element of an aggregate:
> These check if the object is an element of an aggregate. Aggregadgets and Aggrelists will also work since they are subclasses of aggregate.

Element of a list:
> The list is stored as the value of a slot of some object.

The last two kinds have a number of varieties:

Immediate child vs. leaf:
> Sometimes it is convenient to ask if the mouse is over a "leaf" object. This is one of the basic types (rectangle, line, etc.). This is useful because aggregates often contain extra white-space (the bounding box of an aggregate includes all of its children, and all the space in between). Asking for the mouse to be over a leaf insures that the mouse is actually over a visible object.

Return immediate child or leaf:
> If you want the user to have to press on a leaf object, you may still want the interactor to operate on the top level object. Suppose that the movable objects in your system are aggregates containing a line with an arrowhead and a label. The user must press on one of the objects directly (so you want leaf), but the interactor should move the entire aggregate, not just the line. In this case, you would use one of the forms that checks the leaf but returns the element.

Or none Sometimes, you might want to know when the user presses over no objects, for example to turn off selection. The "or-none" option returns the object normally if you press on it, but if you press on no object, then it returns the special value `:none`.

Finally, there is a **custom** method that allows you to specify your own procedure to use.

6.10.4 Type Parameter

After the specification of the object, an optional `:type` parameter allows the objects to be further discriminated by type. For example, you can look for only the `lines` in an aggregate using `'(:element-of ,MYAGG :type ,opal:line)`. Note the comma in front of `opal:line`.

The type parameter can either be a single type, as shown above, or a list of types. In this case, the object must be one of the types listed (the "or" of the types). For example

    '(:element-of ,MYAGG :type (,opal:circle ,opal:rectangle))

will match any element of `myagg` that is either a circle or a rectangle.

Normally, the `leaf` versions of the functions below only return primitive (leaf) elements. However, if the `:type` parameter is given and it matches an interior (aggregate) object, then

that object is checked and returned instead of a leaf. For example, if an object is defined as follows:

```
(create-instance 'MYAGGTYPE opal:aggregate)

(create-instance 'TOP-AGG opal:aggregate)

(create-instance 'A1 MYAGGTYPE)
(create-instance 'A2 MYAGGTYPE)
(opal:add-components TOP-AGG A1 A2)

;; now add some things to A1 and A2
```

Then, the description `(:leaf-element-of ,TOP-AGG :type ,MYAGGTYPE)` will return `A1` or `A2` rather than the leaf elements of `A1` or `A2`.

Another way to prevent the search from going all the way to the actual leaf objects is to set the `:pretend-to-be-leaf` slot of an intermediate object. Note that the `:pretend-to-be-leaf` slot is set in the Opal objects, not in the interactor, and it is more fully explained in the Opal chapter.

6.10.5 Custom

The `:custom` option for the `:start-where` field can be used to set up your own search method. The format is:

```
(list :custom obj #'function-name arg1 arg2 ...)
```

There can be any number of arguments supplied, even zero. The function specified is then called for each event that passes the event test. The calling sequence for the function is:

```
(lambda (obj an-interactor event arg1 arg2 ...))
```

The arguments are the values in the `-where` list, along with the interactor itself, and an event. The `event` is a Garnet event structure, defined in section Section 14.22 [Events], page 696. This function should return `nil` if the event does not pass (e.g., if it is outside the object), or else the object that the interactor should start over (which will usually be obj itself or some child of obj). The implementor of this function should call `opal:point-to-leaf`, or whatever other method is desired. The function is also required to check whether the event occurred in the same window as the object.

For example, if the interactor is in an aggregadget, and we need a custom checking function which takes the aggregadget and a special parameter accessed from the aggregadget, the following could be used:

```
;;; First define the testing function
(defun Check-If-Mouse-In-Obj (obj inter event param)
  (if (and (eq (gv obj :window)(inter:event-window event))   ; have to check window
      (> (inter:event-x event) (gv obj :left))
      .....)
        obj  ; then return object
        NIL)) ; else return NIL
```

```
(create-instance NIL opal:aggregadget
  ... ; various fields
 (:parameter-val 34)
 (:parts '((....)))
 (:interactors
  '((:start-it ,Inter:Button-Interactor
 ... ; all the usual fields
 (:start-where
  ,(o-formula (list :custom (gvl :operates-on)
     #'Check-If-Mouse-In-Obj
     (gvl :operates-on :parameter-val))))))))
```

6.10.6 Full List of Options for Where

All of the options for the where fields are concatenated together to form long keyword names as follows:

T   anywhere. This always succeeds. (The T is not in a list.) T for the `:start-where` means the interactor starts whenever the start-event happens, and T for the `:running-where` means the interactor runs until the stop event no matter where the mouse goes.

NIL   nowhere. This never passes the test. This is useful for interactors that you want to start explicitly using `Start-Interactor` (section [startinteractor], page 309).

`(:in <obj>)`
> message to the object to ask if it contains the mouse position.

`(:in-box <obj>)`
> might be different from `:in` the object since some objects have special tests for inside. For example, lines test for the position to be near the line. `:In-box` may also be more efficient than `:in`.

`(:full-object-in <obj>)`
> object being moved is inside the object specified here (if a circle is being moved with its center connected on the mouse, this will make sure that all of the circle is inside the `<obj>` specified here). [Not implemented yet.]

`(:in-but-not-on <agg>)`
> inside the bounding rectangle of `<agg>`, but not over any of the children of `<agg>`.

`(:element-of <agg> [:type <objtype>])`
> element of the aggregate `<agg>`. If the `:type` keyword is specified, then it searches the components of `<agg>` for an element of the specified type under the mouse. This uses the Opal message `point-to-component` on the aggregate.

`(:leaf-element-of <agg> [:type <objtype>])`
> over any leaf object of the aggregate `<agg>`. If the `:type` keyword is specified, then it searches down the hierarchy from `<agg>` for an element of the specified type under the mouse. This uses the Opal message `point-to-leaf` on the aggregate.

`(:element-of-or-none <agg> [:type<objtype>])`

> the mouse is over `<agg>`. If there is an object at the mouse, then it is returned (as with `:element-of`). If there is no object, then the special value `:none` is returned. If the mouse is not over the aggregate, then `nil` is returned. This uses the Opal message `point-to-component` on the aggregate.

`(:leaf-element-of-or-none <agg> [:type <objtype>])`

> returns leaf children like `:leaf-element-of`. If there is an object at the mouse, then it is returned. If there is no object, then the special value `:none` is returned. If the mouse is not over the aggregate, then `nil` is returned. This uses the Opal message `point-to-leaf` on the aggregate.

`(:list-element-of <obj> <slot> [:type<objtype>])`

> `<obj>` should be a list. Goes through the list to find the object under the mouse. Uses `gv` to get the list, so the contents of the slot can be a formula that computes the list. If the `:type` keyword is specified, then it searches the list for an element of the specified type. This uses the Opal message `point-in-gob` on each element of the list.

`(:list-leaf-element-of <obj> <slot> [:type<objtype>])`

> one of the objects is an aggregate, then returns its leaf element. The contents of the `<slot>` of `<obj>` should be a list. Goes through the list to find the object under the mouse. Uses `point-in-gob` if the object is *not* an aggregate, and uses `point-to-leaf` if it is an aggregate.

`(:list-element-of-or-none <obj> <slot> [:type<objtype>])`

> except if the event isn't over an object, then returns the special value `:none`. Note that this never returns `nil`.

`(:list-leaf-element-of-or-none <obj> <slot> [:type<objtype>])`

> except if the event isn't over an object, then returns the special value `:none`. Note that this never returns `nil`.

`(:check-leaf-but-return-element <agg> [:type<objtype>])`

> This is like `:leaf-element-of` except when an object is found, the immediate component of `<agg>` is returned instead of the leaf element. If the `:type` keyword is specified, then it searches the list for an element of the specified type. This choice is useful, for example, when the top level aggregate contains aggregates (or aggregadgets) that mostly contain lines, and the programmer wants the user to have to select on the lines, but still have the interactor affect the aggregate.

`(:list-check-leaf-but-return-element <obj> <slot> [:type<objtype>])`

> `:list-leaf-element-of`, except that it returns the element from the list itself if a leaf element is hit.

`(:check-leaf-but-return-element-or-none <agg> [:type<objtype>])`

> This is like `:check-leaf-but-return-element` except that if no child is under the event, but the event is inside the aggregate, then `:none` is returned.

`(:list-check-leaf-but-return-element-or-none <agg> [:type<objtype>])`

> This is like `:list-check-leaf-but-return-element` except that if nothing is found, `:none` is returned instead of `nil`.

```
(:custom <obj> 'function-name arg1 arg2 ....)
```
Use a programmer-defined method to search for the object. See section [customwhere], page 242.

6.10.7 Same Object

A special value for the object can be used when the specification is in the `:running-where` slot. Using `*` means "in the object that the interactor started over." For example, if the start-where is `(:element-of <agg>)`, a running-where of `'(:in *)` would refer to whatever object of the `<agg>` the interactor started over. This `*` form cannot be used for the `:start-where`.

6.10.8 Outside while running

While the interactor is running, the mouse might be moved outside the area specified by the `:running-where` slot. The value of the interactor slot `:outside` determines what happens in this case. When `:outside` is `nil`, which is the default, the interaction is temporarily turned off until the mouse moves back inside. This typically will make the feedback be invisible. In this case, if the user gives the stop event while outside, the interactor will be aborted. For example, for a menu, the `:running-where` will usually be `(:element-of MENU-AGG)` (same as the `:start-where`). If the user moves outside of the menu while the mouse button is depressed, the feedback will go off, and the mouse button is released outside, then no menu operation is executed. This is a convenient way to allow the user to abort an interaction once it has started.

On the other hand, if you want the interactor to just save the last legal, inside value, specify `:outside` as `:last`. In this case, if the user stops while outside, the last legal value is used.

If you want there to be no area that is outside (so moving everywhere is legal), then simply set `:running-where` to T, in which case the `:outside` slot is ignored.

6.10.9 Thresholds, Outlines, and Leaves

Three slots of Opal objects are useful for controlling the "where" for interactors. These are `:hit-threshold`, `:select-outline-only`, and `:pretend-to-be-leaf`. If you set the `:select-outline-only` slot of an Opal object (note: *not* in the interactor) to T, then all the "where" forms (except `:in-box`) will only notice the object when the mouse is directly over the outline. The `:hit-threshold` slot of Opal objects determines how close to the line or outline you must be (note that you usually have to set the `:hit-threshold` slot of the aggregate as well as for the individual objects.) See the Opal chapter for more information on these slots.

An important thing to note is that if you are using one of the `-leaf` forms, you need to set the `:hit-threshold` slot of *all the aggregates* all the way down to the leaf from the aggregate you put in the `-where` slot. This is needed if the object happens to be at the edge of the aggregate (otherwise, the press will not be considered inside the aggregate).

The `:pretend-to-be-leaf` slot is used when you want an interactor to treat an aggregate as a leaf (without it, only the components of an aggregate are candidates to be leaves). When you set the `:pretend-to-be-leaf` slot of an aggregate to T (note: not in the interactor), then the search for a leaf will terminate when the aggregate is reached, and the aggregate will be returned as the current object.

6.11 Details of the Operation

Each interactor runs through a standard set of states as it is running. First, it starts off *waiting* for the start-event to happen over the start-where. Once this occurs, the interactor is *running* until the stop-event or abort-event happens, when it goes back to waiting. While it is running, the mouse might move *outside* the active area (determined by `:running-where`), and later move *back inside*. Alternatively, the stop or abort events might happen while the mouse is still outside. These state changes are implemented as a simple state machine inside each interactor.

At each state transition, as well as continuously while the interactor is running, special interactor-specific routines are called to do the actual work of the interactor. These routines are supplied with each interactor, although the programmer is allowed to replace the routines to achieve customizations that would otherwise not be possible. The specifics of what the default routines do, and the parameters if the programmer wants to override them are discussed in section [Specific Interactors], page 254.

The following table and figure illustrate the working of the state machine and when the various procedures are called.

> If the interactor is not *active*, then it waits until a program explicitly sets the interactor to be active (see section [active], page 306).
>
> If active, the interactor waits in the start state for the start-event to happen while the mouse is over the specified start-where area.
>
> When that event happens, if the interactor is *not* "continuous" (defined in section [continuous], page 236), then it executes the Stop-action and returns to waiting for the start-event. If the interactor is continuous, then it does all of the following steps:
>
>> First, the interactor calls the Start-action and goes into the running state.
>>
>> In the running state, it continually calls the running-action routine while the mouse is in the running-where area. Typically, the running-action is called for each incremental mouse movement (so the running-action routine is not called when the mouse is not moving).
>>
>> If the mouse goes *outside* the running-where area, then outside-action is called once.
>>
>> If the mouse returns from outside running-where to be back inside, then the back-inside-action is called once.
>>
>> If the abort-event ever happens, then the abort-action is called and the state changes back to the start state.
>>
>> If the stop-event occurs while the mouse is inside running-where, then the stop-action is called and the state returns to start.
>>
>> If the stop-event occurs while the mouse is *outside*, then if the `:outside` field has the value `:last`, the the stop-action is called with the last legal value. If `:outside` is `nil`, then the abort-action is called. In either case, the state returns to start. Note: if `:outside = :last`, and there is no abort-event, then there is no way to abort an interaction once it has started.

If a program changes the active state to `nil` (not active) and the interactor is running or outside, the interactor is immediately aborted (so the abort-action is called), and the

interactor waits for a program to make it active again, at which point it is in the start state. (If the interactor was in the start state when it became inactive, it simply waits until it becomes active again.) This transition is not shown in the following figure. Section [active], page 306, discusses making an interactor in-active.
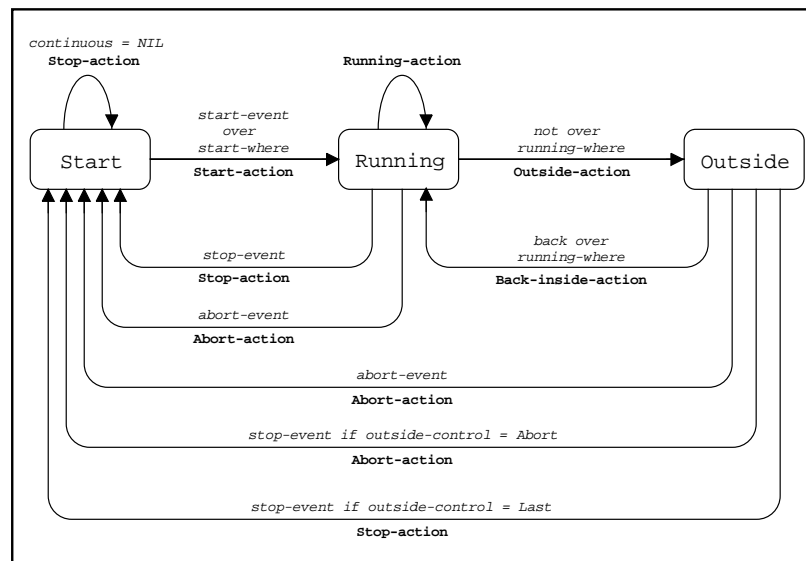
Figure 6.1: Each Interactor runs the same state machine to control its operation. The **start-event**, **stop-event** and **abort-event** can be specified (see section ⟨undefined⟩ [events], page ⟨undefined⟩), as can the various **-action** procedures (section [customroutines], page 312). Where the mouse should be for the Interactor to start (**start-where**), and where it should run (**running-where**) can also be supplied as parameters (sections [startwhere], page 240, and [runningwhere], page 240). The **outside-control** parameter determines whether the interaction is aborted when the user moves outside, or whether the last legal value is used (section [runningwhere], page 240). There are default values for all parameters, so the programmer does not have to specify them. In addition to the transitions shown, Interactors can be aborted by the application at any time.

6.12 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 the Debugging Reference chapter. 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], page 90

inter:Add-Window-Accelerator *win key fn* &key *replace-existing? first?* [function], page 90

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],
page 90
inter:Remove-Window-Accelerator win key &key remove-all? first? [function],▉
page 90
```

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], page 90
inter:Clear-Window-Accelerators win [function], page 90

inter:Default-Global-Accelerators ;; sets up the default accelerators [function],▉
page 90
```

6.13 Slots of All Interactors

This section lists all the slots common to all interactors. Most of these have been explained in the previous sections. The slots a programmer is most likely to want to change are listed first. Some specific interactor types have additional slots, and these are described in their sections.

The various `-action` procedures are used by the individual interactors to determine their behavior. *You will rarely need to set these slots.* See section [customroutines], page 312, for how to use the `-action` slots.

The following field *must* be supplied:

`:start-where`

   - where the mouse should be for this interactor to start working. Valid values for `where` are described in section [where], page 240.

   The following fields are optional. If they are not supplied, then the default value is used, as described below. Note that supplying `nil` is *not* the same as not supplying a value (since not supplying a value means to use the default, and `nil` often means to not do something).

`:window`   - the window that the interactor should be connected to. Usually this is supplied as a single window, but other options are possible for interactors that operate on multiple windows. See section [multiwindow], page 311.

`:start-event`

   - the event that causes the interactor to start working. The default value is `:leftdown`. `nil` means the interactor never starts by itself (see [startinteractor], page 309). Using `T` means no event, which means that the interactor is operating whenever the mouse is over `:start-where`. The full syntax for event specification is described in section [eventspec], page 237.

`:continuous`

   - if this is T, then the interactor operates continuously from start-event until stop-event. If it is `nil`, then the interactor operates exactly once when start-event happens. The default value is `T`. See section [continuous], page 236, for more explanation.

:stop-event

> - This is not used if :continuous is nil. If :continuous is T, :stop-event is the event that the interaction should stop on. If not supplied, and the start-event is a mouse down event (such as :leftdown), then the default :stop-event is the corresponding up event (e.g. :leftup). If start-event is a keyboard key, the default stop event is #\RETURN. If the :start-event is a list or a special form like :any-mousedown, then the default :stop-event is calculated based on the actual start event used. You only need to define stop-event if you want some other behavior (e.g. starting on :leftdown and stopping on the next :leftdown so you must click twice). The form for stop-events is the same as for start-events (see section ⟨undefined⟩ [events], page ⟨undefined⟩). T means no event, so the interactor never stops (unless it is turned off using ChangeActive).

:feedback-obj

> - If supplied, then this is the object to be used to show the feedback while the interaction is running. If nil, then typically the object itself will be modified. The default value is nil. See the descriptions of the specific interactors for more information.

:running-where

> - Describes where the interaction should operate if it is continuous. The default is usually to use the same value as start-where. Running-where will sometimes need to be different from start-where, however. For example, with an object that moves with the mouse, you might want to start moving when the press was over the object itself. See section [where], page 240, for a complete discussion of this field.

:outside - Determines what to do when the mouse goes outside of running-where. Legal values are :last, which means to use the last value before the mouse went outside, or nil which means to return to the original value (before the interaction started). The default value is nil. See section ⟨undefined⟩ [outside], page ⟨undefined⟩, for more explanation.

:abort-event

> - This is an event that causes the interaction to terminate prematurely. If abort-event is nil, then there is no separate event to cause aborts. The default value is nil. The form for abort-events is the same as for start-events (see section ⟨undefined⟩ [events], page ⟨undefined⟩).

:waiting-priority

> - This determines the priority of the interactor while waiting for the start event to happen. See section [priorities], page 303, for a description of priority levels.

:running-priority

> - This determines the priority of the interactor while it is running (waiting for the stop event to happen). See section [priorities], page 303, for a description of priority levels.

:final-function

> - This function is called after the interactor is complete. The programmer might supply a function here to cause the application to notice the users actions. The

particular form for the parameters to this function is specific to the particular type of the interactor.

:stop-action

- This procedure is called once when the :stop-event happens, or if the interactor is *not* continuous, then this procedure is called once when the :start-event happens. The form for the arguments is specific to the particular interactor sub-class. Specifying nil means do no action. Normally, the stop-action procedure (as well as the start-action, running-action, etc. below is *not* provided by the programmer, but rather inherited. These functions provide the default behavior, such as turning on and off the feedback object. In particular the default stop-action calls the final-function. See section [customroutines], page 312.

:start-action

- The action to take place when start-event happens when the mouse is over start-where and continuous is T (if continuous is nil, then stop-action is called when the start-event happens). The form for the arguments is specific to the particular interactor sub-class. Specifying nil means do no action. See section [customroutines], page 312.

:running-action

- A procedure to be called as the interaction is running. This is called repeatedly (typically for each incremental mouse movement) while the mouse is inside :running-where and between when :start-event and :stop-event happen. The form for the arguments is specific to the particular interactor sub-class. Specifying nil means do no action. See section [customroutines], page 312.

:abort-action

- This procedure is called when the interaction is aborted, either by :abort-event or :stop-event while outside. The form for the arguments is specific to the particular interactor sub-class. Specifying nil means do no action. See section [customroutines], page 312.

:outside-action

- This procedure is called once each time the mouse goes from inside :running-where to being outside. It is *not* called repeatedly while outside (so it is different from :running-action). The form for the arguments is specific to the particular interactor sub-class. Specifying nil means do no action. See section [customroutines], page 312.

:back-inside-action

- This is called once each time the mouse goes from outside :running-where to being inside. Note that :running-action is *not* usually called on this point. The form for the arguments is specific to the particular interactor sub-class. Specifying nil means do no action. See section [customroutines], page 312.

:active   - Normally, an interactor is active (willing to accept its start event) from the time it is created until it is destroyed. However, it is sometimes convenient to make an interactor inactive, so it does not look for any events, for example, to have different modes in the interface. This can be achieved by setting the active

field of the interactor. If the interactor is running, setting :active to nil causes it to abort, and if the interactor is not running, then this just keeps it from starting. This field can be set and changed at any time either using s-value or by having a formula in this slot, but it is safest to use the Change-active procedure, since this guarantees that the interactor will be aborted immediately if it is running. Otherwise, if it is running when the active field changes to nil, then it will abort the next time there is an event (e.g., when the mouse moves). See section [active], page 306, for more information.

:self-deactivate

- Normally, interactors are always active. If this field is T however, the interactor will become inactive after it runs once (it will set its own :active slot to nil). The interactor will then not run again until the :active field is explicitly set to T. If this field is used, it is probably a bad idea to have a formula in the :active slot.

```
************************* NIY **********************************
@code{:exception-p}
@cindex{exception}
- a function to determine whether the current value is illegal. This
serves as a temporary way to change what is specified by the @code{-where}
parameters. Illegal
values add one extra state to the state machine of the figure; the
interactor goes into the exception state when the mouse goes over an
illegal item (as determined by this function), and leaves the exception
state when the mouse goes over a legal item or goes outside. @b{Note: Is
it sufficient to just use the actions for outside and back-inside for
exceptions, or do we need the following?}
There are two procedures added for exceptions: @code{:over-illegal-item} and
@code{:leave-illegal-item}, see below.
Parameters to the exception-p procedure
are @code{(an-interactor objUnderMouse)} and it should
return T or @code{nil}. Returning @code{T} means that the value is an exception (
illegal), and this is treated as if the mouse went outside. Returning @code{nil}
means the value is OK. The default is for there to be no illegal values.
The default function for this parameter (use this by supplying T) checks the
slot @code{:illegal} in the object under the mouse.
@B{This is not implemented yet.}

@code{:over-illegal-item}
@cindex{over-illegal-item}
- Called when go over an illegal item with parameters @code{(an-interactor
illegal-obj)}. Default action is to call the @code{:outside-action} procedure.
@B{This is not implemented yet.}

@code{:leave-illegal-item}
@cindex{leave-illegal-item}
- Called when no longer over an illegal item with parameters @code{(an-interactor
```

```
illegal-obj)}. Default action is to do nothing.
Note that if the mouse moves from one illegal item to another,
the @code{:leave-} and @code{:over-} procedures will be called. Also unlike█
back-inside, the @code{:running-action} procedure is called on the new object█
the mouse goes over. @B{This is not implemented yet.}

@code{:pop-up}
@cindex{pop-up}
- If non-NIL, then this interactor controls something that should pop-up█
(become visible) when the interactor starts, and become invisible when the█
interactor completes.
If @code{nil}, then object that the interactor refers to should be
visible. If non-NIL, then the value of pop-up should be a procedure to█
cause the object to be displayed and erased. It is called with
@code{(an-interactor visible-or-erase)}, where @code{visible-or-erase} is T to█
make it visible, and @code{nil} to erase it.
The default procedure is to set
the object's @code{:visible} slot to T, and to @code{nil} for
stop-action. If would be an error to have start-where be the object if it█
was a pop-up, since it would not be visible to press on. The default value█
for this parameter is @code{nil}. @B{This is not implemented yet.}
************************** NIY ***********************************
```

6.14 Specific Interactors

This section describes the specific interactors that have been defined. Below is a list of the interactors, and then the following sections describe them in more detail. There are also several interactors defined for the `multifont-text` object. These are described in the Opal chapter.

`Inter:Menu-Interactor`
> - to handle menu items, where the mouse can choose among a set of items. Useful for menus, etc.

`inter:button-interactor`
> - to choose a particular button. The difference from menus is that when the mouse moves away, the item is deselected, rather than having a different item selected. Useful for sets of buttons like "radio buttons" and "check boxes", and also for single, stand-alone buttons. This can also be used just to select an object by making `:continuous` be `nil`.

`inter:move-grow-interactor`
> - move or change the size of an object or one of a set of objects using the mouse. There may be feedback to show how the object moves or grows, or the object itself may change with the mouse. If defined over a set of objects, then the interactor gets the object to change from where the interaction starts. Useful for scroll bars, horizontal and vertical gauges, and for moving and changing the size of application objects in a graphics editor. It can change the bounding box for the objects or the end points for a line.

`inter:two-point-interactor`
> \- This is used when there is no original object to modify, but one or two new points are desired. A rubber-band feedback object (usually a rubber-band line or rectangle) will typically be drawn based on the points specified.

`inter:angle-interactor`
> \- Useful for getting the angle the mouse moves from around some point. This can be used for circular gauges or for "stirring motions" for rotating.

`inter:text-interactor`
> \- Used to input a small edited string of text. The text can be one line or multi-line.

`inter:gesture-interactor`
> \- Used to recognize single-path gestures drawn with the mouse.

`inter:animator-interactor`
> \- This interactor causes a function to be executed at regular intervals, allowing rapid updating of graphics for animation.

The following interactors are planned but not implemented yet.

`Inter:Trace-Interactor`
> \- This returns all of the points the mouse goes through between `start-event` and `stop-event`. This is useful for inking in a drawing program. Although this isn't implemented yet, it is trivial to use a gesture interactor with a `:classifier` of `nil`.

`inter:multi-point-interactor`
> \- This is used when there is no original object to modify, but more than 2 new points are desired. This is separate from the `two-point-interactor` because the way the points are stored is usually different, and the stopping conditions are much more complicated for multi-points. **Not implemented yet. However, there is a gadget in the gadget set that will do most of this. See** `garnet-gadgets:polyline-creator`.

6.15 Menu-Interactor

```
(create-instance 'inter:Menu-Interactor inter:interactor
  ;; Slots common to all interactors (see section [Slots of All Interac-
tors], page 250)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)
  (:stop-event NIL)
  (:running-where NIL)
  (:outside NIL)
  (:abort-event :control-\g)
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
```

```
    (:active T)
    (:self-deactivate NIL)

    ; Slots specific to the menu-interactor (discussed in this section)
    (:final-function NIL)       ; (lambda (inter final-obj-over))
    (:how-set :set)             ; How to select new items (toggle selection, etc.)█
    (:feedback-obj NIL)         ; Optional interim feedback object. The in-
  ter will set this object's :obj-over slot.
    (:final-feedback-obj NIL)   ; The optional object to indicate the final selection█
    (:slots-to-set              ; Names of slots to set in the objects
     '(:interim-selected        [;   '(<interim-selected-slot-name-in-obj>]
       :selected                ;       <selected-slot-name-in-obj>]
       :selected)               ;       <selected-slot-name-in-aggregate>]
    (:final-feed-inuse NIL)     ; Read-only slot. A list of final feedback ob-█
  jects (section [menufinalfeedbackobj], page 259)

    ; Advanced feature:  Read-only slots.
    ; See section [specialslots], page 310, for details about these slots.
    (:first-obj-over NIL)       ; Read-only slot. The object returned from the start-█
  where.
    (:current-window NIL)       ; Read-only slot. The window of the last (or cur-█
  rent) event.
    (:start-char NIL)           ; Read-only slot. The character or keyword of the start ev

    ; Advanced feature: Customizable action routines.
    ; See sections [Slots of All Interactors], page 250, and ⟨undefined⟩ [menu-█
  customaction], page ⟨undefined⟩, for details about functions in these slots.█
    (:start-action ...)         ; (lambda (inter first-obj-under-mouse))
    (:running-action ...)       ; (lambda (inter prev-obj-over new-obj-over))█
    (:stop-action ...)          ; (lambda (inter final-obj-over))
    (:abort-action ...)         ; (lambda (inter last-obj-over))
    (:outside-action ...)       ; (lambda (inter outside-control prev-obj-over))█
    (:back-inside-action ...) ; (lambda (inter outside-control prev-obj-over new-█
  obj-over))
    ...)
```

(Note: If you just want to use a pre-defined menu, it may be sufficient to use one of the menu objects in the Garnet Gadget Set.)

The menu interactor is used (not surprisingly) mostly for menus. There is typically some feedback to show where the mouse is while the interactor is running. This is called the *interim feedback*. A separate kind of feedback might be used to show the final object selected. This is called the *final feedback*.

Unlike button interactors (see section [buttoninter], page 262), Menu-interactors allow the user to move from one item to another while the interactor is running. For example, the user can press over one menu item, move the mouse to another menu item, and release, and the second item is the one that is selected.

There are a number of examples of the use of menu interactors below. Other examples can be found in the `menu` gadget in the Garnet Gadget Set, and in the file `demo-menu.lisp`.

6.15.1 Default Operation

This section describes how the menu interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section [Menucustomaction], page 313.

The menu interactor provides many different ways to control how the feedback graphics are controlled. In all of these, the interactor sets special slots in objects, and the graphics must have formulas that depend on these slots.

6.15.2 Interim Feedback

To signify the object that the mouse is over as *interim feedback* (while the interactor is running), menu-interactors set two different slots. If there is a feedback object supplied in the `:feedback-obj` slot of the interactor, then the `:obj-over` slot of the feedback object is set to the current menu item object. Also, the `:interim-selected` slot of the current menu item is set to T, and the `:interim-selected` slots of all other items are set to `nil`. Note: there is always at most one interim-selected object, independent of the value of the `:how-set` slot.

This supports two different ways to handle interim feedback:

A single feedback object.

This object should be supplied in the `:feedback-obj` slot of the interactor. The `:obj-over` slot of this object is set to the menu item that the feedback should appear over, or `nil` if there is no object. The following is an example of a typical reverse-video black rectangle as a feedback object:

```
(create-instance 'FEEDBACK-RECT opal:rectangle
   (:obj-over NIL) ; set by the interactor
   (:visible (o-formula (gvl :obj-over))) ; this rectangle is visible
                                          ; only if over something
   (:left (o-formula (gvl :obj-over :left)))
   (:top (o-formula (gvl :obj-over :top)))
   (:width (o-formula (gvl :obj-over :width)))
   (:height (o-formula (gvl :obj-over :height)))
   (:fast-redraw-p T)
   (:draw-function :xor)
   (:filling-style opal:black-fill)
   (:line-style NIL))
```

The interactor to use it would be something like:

```
(create-instance 'SELECT-INTER Inter:Menu-Interactor
   (:start-where '(:element-of ,ITEMSAGG))
   (:feedback-obj FEEDBACK-RECT)
   (:window MYWINDOW))
```

The items that can be chosen are elements of an aggregate named ITEMSAGG.

Multiple feedback objects.

In this case, each item of the menu might have its own feedback object, or else some property of that menu item object might change as the mouse moves over it. Here, you would have formulas that depended on the :interim-selected slot of the menu item.

If there are separate objects associated with each menu item that will be the interim feedback, then their visibility slot can simply be tied to the :interim-selected slot. An example using an Aggregadget which is the item-prototype for an AggreList (see the Aggregadgets chapter) with an embedded interactor is:

```
(create-instance 'MYMENU opal:aggrelist
   (:items '("One" "Two" "Three"))
   (:item-prototype
    '(opal:aggregadget
      (:width ,(o-formula (gvl :str :width)))
      (:height ,(o-formula (gvl :str :height)))
      (:my-item ,(o-formula (nth (gvl :rank) (gvl :parent :items))))
      (:parts
       '((:str ,opal:text
               (:string ,(o-formula (gvl :parent :my-item)))
               (:left ,(o-formula (gvl :parent :left)))
               (:top ,(o-formula (gvl :parent :top))))
         (:interim-feed ,opal:rectangle
          ;; The next slot causes the feedback to go on at the right time
                        (:visible ,(o-formula (gvl :parent :interim-selected)))
                        (:left ,(o-formula (gvl :parent :left)))
                        (:top ,(o-formula (gvl :parent :top)))
                        (:width ,(o-formula (gvl :parent :width)))
                        (:height ,(o-formula (gvl :parent :height)))
                        (:fast-redraw-p T)
                        (:draw-function :xor)
                        (:filling-style ,opal:black-fill)
                        (:line-style NIL))))))
   (:interactors
    '((:inter ,Inter:Menu-Interactor
             (:start-where ,(o-formula (list :element-of (gvl :operates-on))))
             (:window ,MYWINDOW)))))
```

6.15.3 Final Feedback

For some menus, the application just wants to know which item was selected, and there is no graphics to show the final selection. In other cases, there should be *final feedback* graphics to show the object the mouse ends up on.

The Menu-Interactor supplies three ways to have graphics (or applications) depend on the final selection. Both the :selected slot of the individual item and the :selected slot of the aggregate the items are in are set. The item's :selected slot is set with t or nil, as appropriate, and the aggregate's :selected slot is set with the particular item(s) selected. The number of items that are allowed to be selected is controlled by the :how-set slot of the interactor, as described in section [menuhowset], page 260.

Note that the aggregate's `:selected` slot often contains a list of object names, but the `:selected` slot in the individual items will always contain `t` or `nil`. The programmer is responsible for setting up constraints so that the appropriate final feedback is shown based on the `:selected` field.

If there is no aggregate (because `:start-where` is something like `(:in xxx)` rather than something like `(:element-of xxx)`), then the slot of the object is set with T or NIL. If the the `:start-where` is one of the "list" styles (e.g. `(:list-element-of obj slot)`, then the `:selected` slot of the object the list is stored in (here, `obj`) is set as if that was the aggregate.

The third way to show the final feedback is to use the `:final-feedback-obj` slot, which is described in the next section.

6.15.4 Final Feedback Objects

The `:feedback-obj` slot can be used for the object to show the interim-feedback, and the `:final-feedback-obj` slot can be used to hold the object to show the final feedback. Garnet will set the `:obj-over` slot of this object to the object that the interactor finishes on. If the `:how-set` field of the interactor is one of the `:list-*` options, then there might be *multiple* final feedback objects needed to show all the objects selected. In this case, the interactor creates instances of object in the `:final-feedback-obj` slot. Therefore, this object should *not* be an *aggregate*; it must be an *aggregadget* instead (or it can be a single Opal object, such as a rectangle, circle, polyline, etc.). Furthermore, the final-feedback object itself should not be a `:part` of an aggregadget, since you are not allowed to add new objects to an aggregadget with parts.

The `:final-feedback-obj` slot may 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.

If the start-where is one of the `...-or-none` forms, then whenever the user presses in the background, the final feedback objects are all turned off.

For examples of the use of final-feedback-objects, see MENU1 (the month menu) or MENU2 (the day-of-the-week menu) in `demo-menu.lisp`.

Useful Functions

In order to help with final feedback objects, there are a number of additional, useful functions. To get the final-feedback objects currently being displayed by an interactor, you can use:

    `inter:Return-Final-Selection-Objs` *inter*`[function], page 90`

If you want to reference the current final feedback objects in a *formula*, however, then you should access the `:final-feed-inuse` slot of the menu interactor. This slot contains a list of the final feedback objects that are in use. *Do not set this slot.* This might be useful

if you wanted to use the final feedback objects as the start objects for another interactor (e.g, one to move the object selected by a final-feedback object):

```
(create-instance NIL Inter:Move-Grow-Interactor
   ;; start when press on a final-feedback object of SELECT-INTER
   (:start-where (formula '(list :list-element-of
                                  ',SELECT-INTER :final-feed-inuse)))
   ;; actually move the object which the feedback objects are over.
   (:obj-to-change (o-formula (gvl :first-obj-over :obj-over)))
   ..... ; all the other slots
   )
```

If a program wants to make an object be selected, it can call:

> inter:SelectObj *inter obj*[function], page 90

which will cause the object to become selected. This uses the `:how-set` slot of the interactor to decide whether to deselect the other objects (whether single or multiple objects can be selected). The `:selected` slots of the object and the aggregate are set, and the final-feedback objects are handled appropriately. To de-select an object, use:

> inter:DeSelectObj *inter obj*[function], page 90

6.15.5 Items Selected

The menu interactor will automatically handle control over the *number* of items selected. A slot of the interactor (`:how-set`) determines whether a single item can be selected or multiple items. In addition, this slot also determines how this interactor will affect the selected items. For example, if multiple items can be selected, the most common option is for the interactor to "toggle" the selection (so if the item under the mouse was selected, it becomes de-selected, and if it was not selected, then it becomes selected). Another design might use two interactors: one to select items when the left button is pressed, and another to de-select items when the right button is pressed. The `:how-set` slot provides for all these options.

In particular, the legal values for the `:how-set` slot are:

`:set`   - Select the final item. One item is selectable at a time. The aggregate's `:selected` slot is set with this object. The item's `:selected` slot is set with T.

`:clear`   - De-select the final item. At most one item is selectable at a time. The aggregate's `:selected` slot is set to `nil`. (If some item other than the final item used to be selected, then that other item becomes de-selected. I.e., using `:clear` always causes there to be no selected items.) The item's `:selected` slot is set to `nil`. (This choice for how-set is mainly useful when the menu item contains a single item that can be turned on and off by different interactors, e.g., left button turns it on and right button turns it off. With a set of menu items, `:set` is usually more appropriate.)

`:toggle`   - Select if not selected, clear if selected. At most one item is selectable at a time. This means that if there are a set of objects and you select the object that used to be selected, then there becomes no objects selected. (This is mainly useful when there is a single button that can be turned on and off by one interactor,

each press changes the state. With a set of menu items, `:list-toggle` or `:set` is usually more appropriate. However, this option could be used with a set of items if you wanted to allow the user to make there be *no* selection.)

`:list-add`

- If not in list of selected items, then add it. Multiple items are selectable at a time. The item is added to the aggregate's `:selected` slot using `pushnew`. The item's `:selected` slot is set with T.

`:list-remove`

- If in list of selected items, then remove it. Multiple items selectable at a time. The item is removed from the aggregate's `:selected` slot. The item's `:selected` slot is set with `nil`.

`:list-toggle`

- If in list of selected items, then remove it, otherwise add it. Multiple items are selectable at a time. The item is removed or added to the aggregate's `:selected` slot. The item's `selected` slot is set with `t` or `nil`.

<a number> - Increment the `:selected` slot of the item by that amount (which can be negative). The aggregate's `:selected` slot is set to this object. The value of the item's selected slot should be a number.

<a list of two numbers>: `(inc mod)` - Increment the `:selected` slot of the item by the `car` of the list, modulus the `cadr` of the list. The aggregate's `:selected` slot is set to this object. The value of the item's selected slot should be a number.

The default value for `:how-set` for menus is `:set`, so one item is selected at a time.

6.15.6 Application Notification

To have an application notice the effect of the menu-interactor, you can simply have some slot of some object in the application contain a formula that depends on the aggregate's `:selected` slot.

Alternatively, the programmer can provide a function to be called when the interactor is complete by putting the function in the `:final-function` slot. This function is called with the following arguments:

```
(lambda (an-interactor final-obj-over))
```

6.15.7 Normal Operation

If the value of `:continuous` is T, then when the start event happens, the interim feedback is turned on, as described in section [menuinterimfeedback], page 257. If the mouse moves to a different menu item, the interim feedback is changed to that item. If the mouse moves outside, the interim feedback is turned off, unless `:outside` is `:last` (see section ⟨undefined⟩ [outside], page ⟨undefined⟩). If the interactor aborts, the interim feedback is turned off. When the stop event happens, the interim feedback is turned off, and the final `:selected` slots are set as described in section [menufinalfeedback], page 258, based on the value of the `:how-set` parameter (section [menuhowset], page 260), then the `:obj-over` field of the final-feedback-obj is set to the final selection (possibly after creating a new final

feedback object, if necessary), as described in section [menufinalfeedbackobj], page 259. Then the final-function (if any) is called (section [menufinalfunc], page 261).

If the interactor is *not* continuous, when the start event happens, the `:selected` slots are set based on the value of the `:how-set` parameter, the `:obj-over` slot of the final-feedback-obj is set, and then the final-function is called.

6.15.8 Slots-To-Set

The button and menu interactors by default set the `:selected` and `:interim-selected` slots of objects. This sometimes results in a conflict when two interactors are attached to the same object. Therefore, the `:slots-to-set` slot has been provided in which you may specify what slot names should be used. Note: it is very important that once an interactor is started, the slot names for it should never change.

The `:slots-to-set` slot takes a list of three values:

```
(<interim-selected-slot-name-in-obj>
 <selected-slot-name-in-obj>
 <selected-slot-name-in-aggregate> )
```

The default value is (`:interim-selected :selected :selected`). If NIL is supplied for any slot name, then that slot isn't set by the interactor.

The slots in the object are set with `t` or `nil`, and the slot in the aggregate is set with the selected object or a list of the selected objects.

6.16 Button-Interactor

```
(create-instance 'inter:Button-Interactor inter:interactor
 ;; Slots common to all interactors (see section [Slots of All Interac-
tors], page 250)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)
  (:stop-event NIL)
  (:running-where '(:in *))
  (:outside NIL)
  (:abort-event :control-\g)
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
  (:active T)
  (:self-deactivate NIL)

  ; Slots common to the menu-interactor and the button-interactor (see sec-
tion ⟨undefined⟩ [menuinter], page ⟨undefined⟩)
  (:final-function NIL)       ; (lambda (inter final-obj-over))
  (:how-set :list-toggle)     ; How to select new items (toggle selection, etc.)
  (:feedback-obj NIL)         ; Optional interim feedback object. The in-
ter will set this object's :obj-over slot.
  (:final-feedback-obj NIL)   ; The optional object to indicate the final selection
```

```
  (:slots-to-set              ; Names of slots to set in the objects
   '(:interim-selected        [;   '(<interim-selected-slot-name-in-obj>]
     :selected                ;      <selected-slot-name-in-obj>]
     :selected)               ;      <selected-slot-name-in-aggregate>]
  (:final-feed-inuse NIL)     ; Read-only slot. A list of final feedback ob-
jects (section [menufinalfeedbackobj], page 259)

  ; Slots specific to the button-interactor (discussed in this section)
  (:timer-repeat-p NIL)         ; when T, then does timer
  (:timer-initial-wait 0.75)    ; time in seconds
  (:timer-repeat-wait 0.05)     ; time in seconds

  ; Advanced feature:  Read-only slots.
  ; See section [specialslots], page 310, for details about these slots.
  (:first-obj-over NIL)       ; Read-only slot. The object returned from the start-
where.
  (:current-window NIL)       ; Read-only slot. The window of the last (or cur-
rent) event.
  (:start-char NIL)           ; Read-only slot. The character or keyword of the start ev

  ; Advanced feature: Customizable action routines.
  ; See sections [Slots of All Interactors], page 250, and [buttoncustomaction],
  page 313, for details about functions in these slots.
  (:start-action ...)         ; (lambda (inter obj-under-mouse))
  (:stop-action ...)          ; (lambda (inter final-obj-over))
  (:abort-action ...)         ; (lambda (inter last-obj-over))
  (:outside-action ...)       ; (lambda (inter last-obj-over))
  (:back-inside-action ...) ; (lambda (inter new-obj-over))
...)
```

(Note: If you just want to use a pre-defined set of buttons, it may be sufficient to use the radio buttons or x-button objects from the Garnet Gadget Set.

The button interactor is used (not surprisingly) mostly for buttons. There is typically some feedback to show where the mouse is while the interactor is running. This is called the *interim feedback*. A separate kind of feedback might be used to show the final object selected. This is called the *final feedback*.

Unlike menu interactors (see section ⟨undefined⟩ [menuinter], page ⟨undefined⟩), Button-interactors do not allow the user to move from one item to another while the interactor is running. For example, if there are a group of buttons, and the user presses over one button, moving to a different button in the set does *not* cause the other button to become selected. Only the first button that the user presses over can be selected. This is similar to the way radio buttons and check boxes work on the Macintosh.

There are a number of examples of the use of button interactors below. Other examples can be found in the demos for the `radio-button` and `x-button` gadgets in the Garnet Gadget Set, and in the file `demo-grow.lisp`.

6.16.1 Default Operation

The button interactor works very similar to the menu interactor (section ⟨undefined⟩ [menuinter], page ⟨undefined⟩). This section describes how the button interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section [buttoncustomaction], page 313.

The button interactor provides the same ways to control the feedback as the menu interactor.

****

6.16.2 Interim Feedback

As with menus, button-interactors set both the `:obj-over` slot of the object in the `:feedback-obj` slot, and the `:interim-selected` slot of the current button item. The `:obj-over` slot is set with the object that is under the mouse or `nil` if none, and the `:interim-selected` slot is set with T or `nil`. See section [menuinterimfeedback], page 257, for more information.

6.16.3 Final Feedback

The final feedback for buttons works the same way as for menus: Both the `:selected` slot of the individual item and the `:selected` slot of the aggregate the items are in are set, and the `:obj-over` slot of the object in the `:final-feedback-obj` slot (if any) is set. The item's `:selected` slot is set with `t` or `nil`, as appropriate, and the aggregate's `:selected` slot is set with the name(s) of the particular item(s) selected.

For more information, see sections [menufinalfeedback], page 258, and [menufinalfeedbackobj], page 259.

6.16.4 Items Selected

As with Menus, the button interactor will automatically handle control over the *number* of items selected. A slot of the interactor (`:how-set`) determines whether a single item can be selected or multiple items. In addition, this slot also determines how this interactor will affect the selected items.

The legal values for `:how-set` are exactly the same as for menu (see section [menuhowset], page 260: `:set`, `:clear`, `:toggle`, `:list-add`, `:list-remove`, `:list-toggle`, a number, or a list of two numbers).

The default for buttons is `:list-toggle`, however.

6.16.5 Application Notification

As with menus, to have an application notice the effect of the button-interactor, you can simply have some slot of some object in the application contain a formula that depends on the aggregate's `:selected` slot.

Alternatively, the programmer can provide a function to be called when the interactor is complete by putting the function in the `:final-function` slot. This function is called with the following arguments:

```
(lambda (an-interactor final-obj-over))
```

6.16.6 Normal Operation

****

If the value of `:continuous` is T, then when the start event happens, the interim feedback is turned on, as described in section [buttoninterimfeedback], page 264. If the mouse moves away from the item it starts on, the interim feedback goes off. If the mouse moves back, the interim feedback goes back on. If the interactor aborts, the interim feedback is turned off. When the stop event happens, the interim feedback is turned off. If the mouse is over the item that the interactor started on, the final `:selected` slots are set as described in section [buttonfinalfeedback], page 264, based on the value of the `:how-set` parameter (section [buttonhowset], page 264), then the `:obj-over` field of the final-feedback-obj is set to the final selection (possibly after creating a new final feedback object, if necessary), as described in section [menufinalfeedbackobj], page 259. Then the final-function (if any) is called (section [buttonfinalfunc], page 264). Otherwise, when the stop event happens, the interactor aborts.

The `:last` parameter is ignored by button interactors.

If the interactor is *not* continuous, when the start event happens, the `:selected` slots are set based on the value of the `:how-set` parameter, the `:obj-over` slot of the final-feedback-obj is set, and then the final-function is called.

The `:slots-to-set` slot can be used to change the name of the slots that are set, as described in section [slots-to-set], page 262.

6.16.7 Auto-Repeat for Buttons

The `button-interactor` can auto-repeat the `:final-function`. *Note: This only works for Allegro, LispWorks, and Lucid lisps (including Sun and HP CL); not for CMU CL, AKCL, etc.*

If `:timer-repeat-p` is non-NIL, then after the interactor starts, if the mouse button is held down more than `:timer-initial-wait` seconds, then every `:timer-repeat-wait` seconds, the `:final-function` is called and the appropriate slot (usually `:selected`) is set into the object the interactor is operating over (this might be useful, for example, if the `:how-set` was an integer to cause the value of the `:selected` slot to increment each time).

The various scroll bar and slider gadgets use this feature to cause the arrows to auto repeat.

6.16.8 Examples

6.16.9 Single button

The button in this example is not continuous, and does not have a final feedback; it just causes a value to be incremented.

```
(create-instance 'ARROW-INC opal:aggregadget
  (:parts
   '((:arrow ,opal:polyline
           (:selected 10)
           (:point-list (20 40 20 30 10 30 25 15 40 30 30 30 30 40 20 40)))
     (:label ,opal:text
           (:left 17)(:top 50)
```

```
                      (:string ,(o-formula (prin1-to-string
               (gvl :parent :arrow :selected)))))))))
              (:interactors
               '((:incrementor ,Inter:Button-Interactor
                            (:continuous NIL)
                            (:start-where ,(o-formula (list :in (gvl :operates-on :arrow)))))█
                            (:window ,MYWINDOW)
                            (:how-set 3)))))  ; increment by 3
```

6.16.10 Single button with a changing label

Here we have an object whose label changes every time the mouse is pressed over it. It
cycles through a set of labels. This interactor is not continuous, so the action happens
immediately on the down-press and there is no feedback object.

```
      (create-instance 'CYCLE-STRING opal:aggregadget
        (:parts
         '((:label ,opal:text
                    (:left 10)(:top 80)
                    (:selected 0)
                    (:choices ("USA" "Japan" "Mexico" "Canada"))
                    (:string ,(o-formula (nth (gvl :selected) (gvl :choices)))))))█
        (:interactors
         '((:incrementor ,Inter:Button-Interactor
                  (:continuous NIL)
          (:start-where
            ,(o-formula (list :in (gvl :operates-on :label))))
          (:window ,MYWINDOW)
          ;; use a list of 2 numbers and interactor will do MOD
          (:how-set
                          ,(o-formula (list 1 (length (gvl :operates-on
                                                  :label :choices)))))))))))█
```

6.17 Move-Grow-Interactor

```
      (create-instance 'inter:Move-Grow-Interactor inter:interactor
        ;; Slots common to all interactors (see section [Slots of All Interac-
      tors], page 250)
        (:start-where NIL)
        (:window NIL)
        (:start-event :leftdown)
        (:continuous T)
        (:stop-event NIL)
        (:running-where NIL)
        (:outside NIL)
        (:abort-event :control-\g)
        (:waiting-priority normal-priority-level)
        (:running-priority running-priority-level)
        (:active T)
```

```
          (:self-deactivate NIL)

          ; Slots specific to the move-grow-interactor (discussed in this section)█
          (:final-function NIL)        ; (lambda (inter obj-being-changed final-points))█
          (:line-p NIL)                ; If NIL, set :box slot of object. If T, set :points slot│
          (:grow-p NIL)                ; If T, grow the object instead of move it
          (:obj-to-change NIL)         [; The object to move or grow (usually this is automatica
                                       ; returned from the start-where)
          (:attach-point :where-hit)   ; Where the mouse will attach to the object
          (:min-width 0)               ; Minimum width for any object being grown
          (:mih-height 0)              ; Minimum height for any object being grown
          (:min-length NIL)            ; Minimum length of any line being grown
          (:feedback-obj NIL)          ; Optional interim feedback object. The in-
     ter will set this object's :obj-over slot
                                       ; and either its :box or :points slot.
          (:slots-to-set :box)         ; Names of slots to set in the objects. Note: :box = :poi
     cause of :line-p slot.
          (:input-filter NIL)          ; Used for gridding

          ; Advanced feature:  Read-only slots.
          ; See section [specialslots], page 310, for details about these slots.
          (:first-obj-over NIL)      ; Read-only slot. The object returned from the start-█
     where.
          (:current-window NIL)      ; Read-only slot. The window of the last (or cur-█
     rent) event.
          (:start-char NIL)          ; Read-only slot. The character or keyword of the start ev

          ; Advanced feature: Customizable action routines.
          ; See sections [Slots of All Interactors], page 250, and [movegrowcustomaction],█
     page 314, for details about functions in these slots.
          (:start-action ...)        ; (lambda (inter obj-being-changed first-points))█
          (:running-action ...)      ; (lambda (inter obj-being-changed new-points))█
          (:stop-action ...)         ; (lambda (inter obj-being-changed final-points))█
          (:abort-action ...)        ; (lambda (inter obj-being-changed))
          (:outside-action ...)      ; (lambda (inter outside-control obj-being-
     changed))
          (:back-inside-action ...) ; (lambda (inter outside-control obj-being-
     changed new-inside-points))
          ...)
```

This is used to move or change the size of an object or one of a set of objects with the mouse. This is quite a flexible interactor and will handle many different behaviors including: moving the indicator in a slider, changing the size of a bar in a thermometer, changing the size of a rectangle in a graphics editor, changing the position of a circle, and changing an end-point of a line.

The interactor can either be permanently tied to a particular graphics object, or it will get the object from where the mouse is when the interaction starts. There may be a

feedback object to show where the object will be moved or changed to, or the object itself may change with the mouse.

There are a number of examples of the use of move-grow-interactors below. Other examples can be found in sections [movegrowexample1], page 306, [movegrowexample2], page 311, and [movegrowexample3], page 313, in the `graphics-selection` gadget in the Garnet Gadget Set, and in the files `demo-grow.lisp`, `demo-moveline.lisp`, `demo-scrollbar.lisp` and `demo-manyobjs.lisp`.

6.17.1 Default Operation

This section describes how the `move-grow-interactor` works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section [movegrowcustomaction], page 314.

The feedback object (if any) *and* the object being edited are modified indirectly, by setting slots called `:box` or `:points`. The programmer must provide constraints between these slots and the `:left`, `:top`, `:width`, and `:height` slots or the `:x1`, `:y1`, `:x2`, and `:y2` slots (as appropriate). For example, a rectangle that can be moved and changed size with the mouse might have the following definition:

```
(create-instance 'MOVING-RECTANGLE opal:rectangle
   (:box (list 0 0 10 10))  ; some initial values (x, y, width, height)
   (:left (o-formula (first (gvl :box))))
   (:top (o-formula (second (gvl :box))))
   (:width (o-formula (third (gvl :box))))
   (:height (o-formula (fourth (gvl :box)))))
```

A movable line could be defined as:

```
(create-instance 'MOVING-LINE opal:line
   (:points (list 0 0 10 10))  ; some initial values (x1 y1 x2 y2)
   (:x1 (o-formula (first (gvl :points))))
   (:y1 (o-formula (second (gvl :points))))
   (:x2 (o-formula (third (gvl :points))))
   (:y2 (o-formula (fourth (gvl :points)))))
```

The slot `:line-p` tells the interactor whether to change the `:box` slot or the `:points` slot. If `:line-p` is `nil` (the default), then the interactor changes the object by setting its `:box` slot to a list containing the new values for (left, top, width, height). If T, then the interactor changes the object by setting its `:points` slot to a list containing the new values for (x1, y1, x2, y2). (These are the same slots as used for `two-point-interactor`, section [twopoint], page 274).

This allows the object to perform any desired filtering on the values before they are used in the real `:left` `:top` `:width` `:height` or `:x1` `:y1` `:x2` `:y2` slots. For example, a scroll bar might be defined as follows:

```
(create-instance 'MYSCROLLER opal:aggregadget
   (:parts
    '((:outline ,opal:rectangle
 (:left 100)(:top 10)(:width 20)(:height 200))
       (:indicator ,opal:rectangle
  (:box (52 12 16 16)) ;; only the second value is used
```

```
(:left ,(o-formula (+ 2 (gvl :parent :outline :left))))
;; Clip-And-Map clips the first parameter to keep it
;;  between the other two parameters, see section [clipandmap], page 273
(:top ,(o-formula
(Clip-And-Map (second (gvl :box))
  12  ; Top of outline + 2
  192 ; Bottom of outline - indicator height - 2
  )))
(:width 16)(:height 16)
(:filling-style ,opal:gray-fill)
(:line-style NIL)
(:fast-redraw-p T)
(:draw-function :xor))))
 (:interactors
  '((:move-indicator ,Inter:Move-Grow-Interactor
     (:start-where
      ,(o-formula (list :in (gvl :operates-on :indicator))))
     (:window ,(o-formula (gvl :operates-on :window)))))))
```

This interactor will either change the position of the object (if `:grow-p` is `nil`) or the size. For lines, (if `:line-p` is T), "growing" means changing a single end point to follow the mouse while the other stays fixed, and moving means changing both end points to follow the mouse so that the line keeps the same length and slope.

Since an object's size can change from the left and top, in addition to from the right and bottom, and since objects are defined to by their left, top, width and height, this interactor may have to change any of the left, top, width and height fields when changing an object's size. For example, to change the size of an object from the left (so that the left moves and the right side stays fixed), both the `:left` and `:width` fields must be set. Therefore, by default, this interactor sets a `:box` field containing 4 values. When the interactor is used for moving an object, the last two values of the `:box` slot are set with the original width and height of the object. Similarly, when setting the `:points` slot, all of the values are set, even though only two of them will change.

When the interaction is running, either the object itself or a separate *feedback* object can follow the mouse. If a feedback object is used, it should be specified in the `:feedback-obj` slot of the interactor, and it will need the same kinds of formulas on `:box` or `:points` as the actual object. If the object itself should change, then `:feedback-obj` should be `nil`. If there is a feedback object, the interactor also sets its `:obj-over` field to the actual object that is being moved. This can be used, for example, to control the visibility of the feedback object or its size.

The object being changed is either gotten from the `:obj-to-change` slot of the interactor, or if that is `nil`, then from the object returned from `:start-where`. If the interactor is to work over multiple objects, then `:obj-to-change` should be `nil`, and `:start-where` will be one of the forms that returns one of a set of objects (e.g., `:element-of`).

6.17.2 attach-point

the `:attach-point` slot of interactors controls where the mouse will attach to the object. the legal choices depend on `:line-p`.

if `:line-p` is t (so the end-point of the line is changing), and the object is being grown, then legal choices are:

```
1: change the first endpoint of the line (x1, y1).
2: change the second endpoint of the line (x2, y2).
:where-hit: change which-ever end point is nearest the
            initial press.
```

if `:line-p` is t and the object is being moved, then legal choices are:

```
1: attach mouse to the first endpoint.
2: attach mouse to the second endpoint.
:center: attach mouse to the center of the line.
:where-hit: attach mouse where pressed on the line.
```

if `:line-p` is `nil` (so the bounding box is changing, either moving or growing) the choices are:

```
:N – Top

:S – Bottom
:E – Right
:W – Left

:NE – Top, right
:NW – Top, left
:SE – Bottom, right
:SW – Bottom, left
:center – Center
:where-hit – The mouse attaches to the object
            wherever the mouse was first pressed inside the object.
```

The default value is `:where-hit` since this works for both `:line-p` T and `nil`.

If growing and `:attach-point` is `:where-hit`, the object grows from the nearest side or corner (the object is implicitly divided into 9 regions). If the press is in the center, the object grows from the `:NW` corner.

The value set into the `:box` slot by this interactor is always the correct value for the top, left corner, no matter what the value of attach-point (the interactor does the conversion for you). Note that the conversion is done based on the `:left`, `:top`, `:width` and `:height` of the actual object being changed; not based on the feedback object. Therefore, if there is a separate feedback object, either the feedback object should be the same size as the object being changed, or `:attach-point` should be `:NW`. **Possible future enhancement: allow a list of points, and pick the closest one to the mouse.**

6.17.3 Running where

Normally, the default value for `:running-where` is the same as `:start-where`, but for the move-grow-interactor, the default `:running-where` is T, to allow the mouse to go anywhere.

6.17.4 Extra Parameters

The extra parameters are:

`:line-p`   - This slot determines whether the object's bounding box or line end points are set. If `:line-p` is `nil`, then the `:box` slot is set to a list containing (left

top width height) and if `:line-p` is T, then the `:points` slot is set with a list containing (x1 y1 x2 y2). The default is `nil`.

`:grow-p`   - This slot determines whether the object moves or changes size. The default is `nil`, which means to move. Non-NIL means to change size.

`:obj-to-change`
          - If an object is supplied as this parameter, then the interactor changes that object. Otherwise, the interactor changes the object returned from `:start-where`. If the interactor should change one of a set of objects, then `:obj-to-change` should be `nil` and `:start-where` should be be a form that will return the object to change. The reason that there may need to be a separate object passed as the `:obj-to-change` is that sometimes the interactor cannot get the object to be changed from the `:where` fields. For example, the programmer may want to have a scroll bar indicator changed whenever the user presses over the background. The object in the `:obj-to-change` field may be different from the one in the `:feedback-obj` since the object in the `:feedback-obj` field is used as the interim feedback.

`:attach-point`
          - This tells where the mouse will attach to the object. Values are `1`, `2`, `:center` or `:where-hit` if `:line-p` is T, or `:N`, `:S`, `:E`, `:W`, `:NW`, `:NE`, `:SW`, `:SE`, `:center`, or `:where-hit` if `:line-p` is `nil`. The default value is `:where-hit`. See section [attachpoint], page 269, for a full explanation.

`:min-width`
          - The `:min-width` and `:min-height` fields determine the minimum legal width and height of the object if `:line-p` is `nil` and `:grow-p` is T. Default is 0. If `:min-width` or `:min-height` is `nil`, then there is no minimum width or height. In this case, the width and height of the object may become negative values which causes an error (so this is not recommended). Unlike the `two-point-interactor` (section [twopoint], page 274), there are no `:flip-if-change-side` or `:abort-if-too-small` slots for the `move-grow-interactor`.

`:min-height`
          - See `:min-width`.

`:min-length`
          - If `:line-p` is T, this specifies the minimum length for lines. The default is `nil`, for no minimum. This slot is ignored if `:line-p` is `nil`.

`:input-filter`
          - Used to support gridding. See section [gridding], page 272,

6.17.5 Application Notification

Often, it is not necessary to have the application notified of the result of a move-grow-interactor, if you only want the object to move around. Otherwise, you can have constraints in the application to the various slots of the object being changed.

Alternatively, the programmer can provide a function to be called when the interactor is complete by putting the function in the `:final-function` slot. This function is called with the following arguments:

```
(lambda (an-interactor object-being-changed final-points))
```

Final-points is a list of four values, either the left, top, width and height if `:line-p` is `nil`, or x1, y1, x2, and y2 if `:line-p` is T.

6.17.6 Normal Operation

If the value of `:continuous` is T, then when the start event happens, the interactor determines the object to be changed as either the value of the `:obj-to-change` slot, or if that is `nil`, then the object returned from the `:start-where`. The `:obj-over` slot of the object in the `:feedback-obj` slot of the interactor is set to the object being changed. Then, for every mouse movement until the stop event happens, the interactor sets either the `:box` slot or the `:points` slot (depending on the value of `:line-p`) based on a calculation that depends on the values in the minimum slots and `:attach-point`. The object that is modified while running is either the feedback object if it exists or the object being changed if there is no feedback object.

If the mouse goes outside of `:running-where`, then if `:outside` is `:last`, nothing happens until the mouse comes back inside or the stop or abort events happen (the object stays at its last legal inside value). If `:outside` is `nil`, then the feedback object's `:obj-over` slot is set to `nil` (so there should be a formula in the feedback object's `:visible` slot that depends on `:obj-over`). If there is no feedback object and the mouse goes outside, then the object being changed is returned to its original size and position (before the interactor started).

If the abort event happens, then the feedback object's `:obj-over` slot is set to `nil`, or if there is no feedback object, then the object being changed is returned to its original size and position (before the interactor started).

When the stop event happens, the feedback object's `:obj-over` slot is set to `nil`, and the `:box` or `:points` slot of the actual object are set with the last value, and the final-function (if any) is called.

If the interactor is *not* continuous, when the start event happens, the `:box` or `:points` slot of the actual object are set with the initial value, and the final-function (if any) is called. This is probably not very useful.

6.17.7 Gridding

The `move-grow-interactor` supports arbitrary gridding of the values. The slot `:input-filter` can take any of the following values:

- `NIL` - for no filtering. This is the default.
- a number - grid by that amount in both X and Y with the origin at the upper left corner of the window.
- a list of four numbers: (`xmod xorigin ymod yorigin`) to allow non-uniform gridding with a specific origin.
- a function of the form (`lambda(inter x y) ...`) which returns (`values gridx gridy`). This allows arbitrary filtering of the values, including application-specific gravity to interesting points of other objects, snap-dragging, etc.

6.17.8 Setting Slots

The `move-grow-interactor` by default sets either the `:box` or `:points` slots of objects (depending on whether it was a rectangle or line-type object). We discovered that there were a large number of formulas that simply copied the values out of these lists. Therefore, in the current version, you can ask the `move-grow-interactor` to directly set the slots of objects, if you don't need any filtering on the values. If you want to use `Clip-and-Map` or other filtering, you should still use the `:box` slot. The slot `:slots-to-set` can be supplied to determine which slots to set. The values can be:

`:box`   if line-p object, then sets the `:points` slot, otherwise sets the `:box` slot.

`:points`   same as `:box`. Note that the interactor ignores the actual value put in `:slots-to-set` and decides which to use based on the value of the `:line-p` slot of the object.

`a list of four T's and nils (representing (:left :top :width :height)`
             or `(:x1 :y1 :x2 :y2))` In this case, the interactor sets the slots of the object that have T's and doesn't set the slots that are `nil`. For example, if `:slots-to-set` is `(T T nil nil)`, then the interactor will set the `:x1` and `:y1` slots of objects that are `:line-p`, and the `:left` and `:top` slots of all other objects.

`a list of four slot names or nils`
             In this case, the values are set into the specified slots of the object. Any NILs mean that slot isn't set. The specified slots are used whether the object is `:line-p` or not. This can be used to map the four values into new slots.

6.17.9 Useful Function: Clip-And-Map

It is often useful to take the value returned by the mouse and clip it within a range. The function `Clip-And-Map` is provided by the interactors package to help with this:

> `inter:Clip-And-Map val  val-1  val-2 &optional target-val-1  target-val-2`[function]`, page 90`

If *target-val-1* or *target-val-2* is `nil` or not supplied, then this function just clips *val* to be between *val-1* and *val-2* (inclusive).

If *target-val-1* and *target-val-2* are supplied, then this function clips *val* to be in the range *val-1* to *val-2*, and then then scales and translates the value (using linear-interpolation) to be between *target-val-1* and *target-val-2*.

Target-val-1 and *target-val-2* should be integers, but *val*, *val-1* and *val-2* can be any kind of numbers. *Val-1* can either be less or greater than *val-2* and *target-val-1* can be less or greater than *target-val-2*.

Examples:

```
(clip-and-map 5 0 10) => 5
(clip-and-map 5 10 0) => 5
(clip-and-map -5 0 10) => 0
(clip-and-map 40 0 10) => 10
(clip-and-map 5 0 10 100 200) => 150
(clip-and-map -5 0 10 100 200) => 100
(clip-and-map 0.3 0.0 1.0 0 100) => 30
```

```
(clip-and-map 5 20 0 100 200) => 175
```

```
;; Formula to put in the :percent slot of a moving scroll bar indicator.
;; Clip the moving indicator position to be between the top and bottom of█
;; the slider-shell (minus the height of the indicator to keep it inside),█
;; and then map the value to be between 0 and 100.
(formula '(Clip-and-Map (second (gvl :box))
(gv ',SLIDER-SHELL :top)
(- (gv-bottom ',SLIDER-SHELL) (gvl :height) 2)
0 100))
```

6.18 Two-Point-Interactor

```
(create-instance 'inter:Two-Point-Interactor inter:interactor
  ;; Slots common to all interactors (see section [Slots of All Interac-
tors], page 250)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)
  (:stop-event NIL)
  (:running-where NIL)
  (:outside NIL)
  (:abort-event :control-\g)
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
  (:active T)
  (:self-deactivate NIL)

  ; Slots specific to the two-point-interactor (discussed in this section)█
  (:final-function NIL)        ; (lambda (inter final-point-list))
  (:line-p NIL)                ; Whether to set the :box or :points slot of the feedback
obj
  (:min-width 0)               ; Minimum width for new rectangular region
  (:mih-height 0)              ; Minimum height for new rectangular region
  (:min-length NIL)            ; Minimum length for new line
  (:abort-if-too-small NIL)    ; Whether to draw feedback and execute fi-
nal function when the selected region
                               ; is smaller than the minimum
  (:feedback-obj NIL)          ; Optional interim feedback object. The in-
ter will set this object's :visible slot
                               ; and its :points or :box slot.
  (:flip-if-change-side T)     ; Whether to flip origin of rectangle when appropriate█
  (:input-filter NIL)          ; Used for gridding (see section [gridding],█
page 272)

  ; Advanced feature:  Read-only slots.
```

```
    ; See section [specialslots], page 310, for details about these slots.
    (:first-obj-over NIL)      ; Read-only slot. The object returned from the start-█
  where.
    (:current-window NIL)      ; Read-only slot. The window of the last (or cur-█
  rent) event.
    (:start-char NIL)          ; Read-only slot. The character or keyword of the start ev

    ; Advanced feature: Customizable action routines.
    ; See sections [Slots of All Interactors], page 250, and [twopcustomaction],█
  page 315, for details about functions in these slots.
    (:start-action ...)        ; (lambda (inter first-points))
    (:running-action ...)      ; (lambda (inter new-points))
    (:stop-action ...)         ; (lambda (inter final-points))
    (:abort-action ...)        ; (lambda (inter))
    (:outside-action ...)      ; (lambda (inter outside-control))
    (:back-inside-action ...) ; (lambda (inter outside-control new-inside-
  points))
    ...)
```

The Two-Point-interactor is used to enter one or two new points, when there is no existing object to change. For example, this interactor might be used when creating a new rectangle or line. If the new object needs to be defined by more than two points (for example for polygons), then you would probably use the `multi-point-interactor` instead, except that it is not implemented yet.

Since lines and rectangles are defined differently, there are two modes for this interactor, determined by the `:line-p` slot. If `:line-p` is `nil`, then rectangle mode is used, so the new object is defined by its left, top, width, and height. If `:line-p` is T, then the object is defined by two points: x1, y1, and x2, y2. Both of these are stored as a list of four values.

As a convenience, this interactor will handle clipping of the values. A minimum size can be supplied, and the object will not be smaller than this.

While the interactor is running, a feedback object, supplied in the `:feedback-obj` slot is usually modified to show where the new object will be. When the interaction is complete, however, there is no existing object to modify, so this interactor cannot just set an object field with the final value, like most other interactors. Therefore, the `final-function` (section [twopapplnotif], page 277) will usually need to be used for this interactor.

There are a number of examples of the use of two-point-interactors below, and another in section ⟨undefined⟩ [twopselectexample], page ⟨undefined⟩. Other examples can be found in the file `demo-twop.lisp`.

6.18.1 Default Operation

This section describes how the two-point interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section [twopcustomaction], page 315.

Just as for move-grow-interactors (section ⟨undefined⟩ [movegrowinter], page ⟨undefined⟩), the feedback object (if any) is modified indirectly, by setting slots called `:box` or `:points`. The programmer must provide constraints between the `:left`, `:top`, `:width`,

and :height slots or the :x1, :y1, :x2, and :y2 slots (as appropriate). The examples in section ⟨undefined⟩ [movegrowinter], page ⟨undefined⟩, show how to define constraints for the feedback object.

The slot :line-p tells the interactor whether to change the :box slot or the :points slot in the feedback object. If :line-p is nil (the default), then the interactor changes the object by setting its :box slot to a list containing the new values for (left, top, width, height). If T, then the interactor changes the object by setting its :points slot to a list containing the new values for (x1, y1, x2, y2). (These are the same slots as used for move-grow-interactor).

6.18.2 Minimum sizes

The two-point interactor will automatically keep objects the same or bigger than a specified size. There are two different mechanisms: one if :line-p is NIL (so the object is defined by its :box), and another if :line-p is T.

In both modes, the slot :abort-if-too-small determines what happens if the size is smaller than the defined minimum. The default is nil, which means to create the object with the minimum size. If :abort-if-too-small is T, however, then the feedback object will disappear if the size is too small, and if the mouse is released, the final-function will be called with an error value (NIL) so the application will know not to create the object.

If :line-p is nil, the slots :min-width and :min-height define the minimum size of the object. If both of these are not set, zero is used as the minimum size (the two-point-interactor will not let the width or height get to be less than zero). If the user moves the mouse to the left or above of the original point, the parameter :flip-if-change-side determines what happens. If :flip-if-change-side is T (the default), then the box will still be drawn from the initial point to the current mouse position, and the box will be flipped. The values put into the :box slot will always be the correct left, top, width and height. If :flip-if-change-side is nil, then the box will peg at its minimum value.

If :line-p is T, the slot :min-length determines the minimum length. This length is the actual distance along the line, and the line will extend from its start point through the current mouse position for the minimum length. If not supplied, then the minimum will be zero. The :min-width, :min-height and :flip-if-change-side slots are ignored for lines.

6.18.3 Extra Parameters

The extra parameters are:

:line-p   - If T, the :points slot of the feedback object is set with the list (x1 y1 x2 y2). If nil, the :box slot of the feedback object is set with the list (left top width height). The values in the list passed to the final-function is also determined by :line-p. The default is nil (rectangle mode).

:min-width

       - The :min-width and :min-height fields determine the minimum legal width and height of the rectangle or other object if :line-p is nil. Default is nil, which means use 0. Both min-width and min-height must be non-NIL for this to take effect. :min-width and :min-height are ignored if :line-p is non-NIL (see :min-length).

`:min-height`
> - See `:min-width`.

`:min-length`
> - If `:line-p` is non-NIL, then `:min-width` and `:min-height` are ignored, and the `:min-length` slot is used instead. This slot determines the minimum allowable length for a line (in pixels). If `nil` (the default), then there is no minimum length.

`:abort-if-too-small`
> - If this is `nil` (the default), then if the size is smaller than the minimum, then the size is made bigger to be the minimum (this works for both `:line-p` T and `nil`). If `:abort-if-too-small` is T, then instead, no object is created and no feedback is shown if the size is smaller than `:min-width` and `:min-height` or `:min-length`.

`:flip-if-change-side`
> - This only applies if `:line-p` is `nil` (rectangle mode). If `:flip-if-change-side` is T (the default), then if the user moves to the top or left of the original point, the rectangle will be "flipped" so its top or left is the new point and the width and height is based on the original point. If `:flip-if-change-side` is `nil`, then the original point is always the top-left, and if the mouse goes above or to the left of that, then the minimum legal width or height is used.

`:input-filter`
> - Used to support gridding. See section [gridding], page 272.

6.18.4 Application Notification

Unlike with other interactors, it is usually necessary to have an application function called with the result of the two-point-interactor. The function is put into the `:final-function` slot of the interactor, and is called with the following arguments:

```
(lambda (an-interactor final-point-list))
```

The `final-point-list` will either be a list of the left top width, and height or the x and y of two points, depending on the setting of the `:line-p` slot. If the `:abort-if-too-small` slot is set (section [Minimumsizes], page 276), then the `final-point-list` will be `nil` if the user tries to create an object that is too small.

Therefore, the function should check to see if `final-point-list` is `nil`, and if so, not create the object. If you want to access the points anyway, the original point is available as the `:first-x` and `:first-y` slots of the interactor, and the final point is available in the `*Current-Event*` as described in section ⟨undefined⟩ [twopselectexample], page ⟨undefined⟩.

IMPORTANT NOTE: When creating an object using `final-point-list`**, the elements of the list should be accessed individually (e.g,** `(first final-point-list)` `(second final-point-list)` **etc.) or else the list should be copied (**`copy-list final-point-list`**) before they are used in any object slots, since to avoid consing, the interactor reuses the same list. Examples:**

```
(defun Create-New-Object1 (an-interactor points-list)
  (when points-list
```

```
        (create-instance NIL opal:rectangle
           (:left (first points-list)) ; access the values in
           (:top (second points-list)) ; the list individually
           (:width (third points-list))
           (:height (fourth points-list)))))
```
OR
```
  (defun Create-New-Object2 (an-interactor points-list)
    (when points-list
      (create-instance NIL opal:rectangle
         (:box (copy-list points-list)) ; copy the list
         (:left (first box))
         (:top (second box))
         (:width (third box))
         (:height (fourth box)))))
```

6.18.5 Normal Operation

If the value of `:continuous` is T, then when the start event happens, if `:abort-if-too-small` is non-NIL, then nothing happens until the mouse moves so that the size is big enough. Otherwise, if `:line-p` is nil, then the `:visible` slot of the `:feedback-obj` is set to T, and its `:box` or `:points` slot is set with the correct values for the minimum size rectangle or line. As the mouse moves, the `:box` or `:points` slot is set with the current size (or minimum size). If the size gets to be less than the minimum and `:abort-if-too-small` is non-NIL, then the `:visible` field of the feedback object is set to `nil`, and it is set to T again when the size gets equal or bigger than the minimum.

If the mouse goes outside of `:running-where`, then if `:outside` is `:last`, nothing happens until the mouse comes back inside or the stop or abort events happen (the object stays at its last legal inside value). If `:outside` is `nil`, then the feedback object's `:visible` slot is set to `nil`.

If the abort event happens, then the feedback object's `:visible` slot is set to `nil`.

When the stop event happens, the feedback object's `:visible` slot is set to `nil` and the final-function is called.

If the value of `:continuous` is `nil`, then the final-function is called immediately on the start event with the `final-point-list` parameter as `nil` if `:abort-if-too-small` is non-NIL, or else a list calculated based on the minimum size.

6.18.6 Examples

6.18.7 Creating New Objects

Create a rectangle when the middle button is pressed down, and a line when the right button is pressed.

```
    (defun Create-New-Object (an-interactor points-list)
      (when points-list
        (let (obj)
          (if (gv an-interactor :line-p)
      ;; then create a line
      (setq obj (create-instance NIL opal:line
```

```
                              (:x1 (first points-list))
                              (:y1 (second points-list))
                              (:x2 (third points-list))
                              (:y2 (fourth points-list))))
   ;; else create a rectangle
   (setq obj (create-instance NIL opal:rectangle
                              (:left (first points-list))
                              (:top (second points-list))
                              (:width (third points-list))
                              (:height (fourth points-list)))))
       (opal:add-components MYAGG obj)
       obj)))


(create-instance 'CREATERECT Inter:Two-Point-Interactor
   (:window MYWINDOW)
   (:start-event :middledown)
   (:start-where T)
   (:final-function #'Create-New-Object)
   (:feedback-obj MOVING-RECTANGLE) ; section [howobjsdefined], page 268
   (:min-width 20)
   (:min-height 20))
(create-instance 'CREATELINE Inter:Two-Point-Interactor
   (:window MYWINDOW)
   (:start-event :rightdown)
   (:start-where T)
   (:final-function #'Create-New-Object)
   (:feedback-obj MOVING-LINE) ; section [howobjsdefined], page 268
   (:line-p T)
   (:min-length 20))
```

6.19 Angle-Interactor

```
(create-instance 'inter:Angle-Interactor inter:interactor
   ;; Slots common to all interactors (see section [Slots of All Interac-
   tors], page 250)
   (:start-where NIL)
   (:window NIL)
   (:start-event :leftdown)
   (:continuous T)
   (:stop-event NIL)
   (:running-where NIL)
   (:outside NIL)
   (:abort-event :control-\g)
   (:waiting-priority normal-priority-level)
   (:running-priority running-priority-level)
   (:active T)
   (:self-deactivate NIL)
```

```
      ; Slots specific to the move-grow-interactor (discussed in this section)█
      (:final-function NIL)        ; (lambda (inter obj-being-rotated final-angle))█
      (:obj-to-change NIL)         [; The object to change the :angle slot of (if NIL, then
                                   ; the object returned from the start-where)
      (:feedback-obj NIL)          ; Optional interim feedback object. The in-
  ter will set this object's :obj-over slot
                                   ; and its :angle slot during interim selection█
    (:center-of-rotation NIL)  ; A list (x y) indicating a coordinate around which the o
  jects will be rotated.
                                   ; If NIL, the center of the object is used

    ; Advanced feature:  Read-only slots.
    ; See section [specialslots], page 310, for details about these slots.
    (:first-obj-over NIL)      ; Read-only slot. The object returned from the start-█
  where.
    (:current-window NIL)      ; Read-only slot. The window of the last (or cur-█
  rent) event.
    (:start-char NIL)          ; Read-only slot. The character or keyword of the start ev

    ; Advanced feature: Customizable action routines.
    ; See sections [Slots of All Interactors], page 250, and [anglecustomaction],█
  page 315, for details about functions in these slots.
    (:start-action ...)        ; (lambda (inter obj-being-rotated first-angle))█
    (:running-action ...)      ; (lambda (inter obj-being-rotated new-angle angle-█
  delta))
    (:stop-action ...)         ; (lambda (inter obj-being-rotated final-angle angle-█
  delta))
    (:abort-action ...)        ; (lambda (inter obj-being-rotated))
    (:outside-action ...)      ; (lambda (inter outside-control obj-being-
  rotated))
    (:back-inside-action ...) ; (lambda (inter outside-control obj-being-
  rotated new-angle))
    ...)
```

This is used to measure the angle the mouse moves around a point. It can be used for circular gauges, for rotating objects, or for "stirring motions" for objects.

It operates very much like the `move-grow-interactor` and has interim and final feedback that work much the same way.

The interactor can either be permanently tied to a particular graphics object, or it will get the object from where the mouse is when the interaction starts. There may be a feedback object to show where the object will be moved or changed to, or the object itself may change with the mouse.

There is an example of the use of the angle-interactor below. Other examples can be found in the `Gauge` gadget in the Garnet Gadget Set, and in the files `demo-angle.lisp` and `demo-clock.lisp`.

6.19.1 Default Operation

This section describes how the angle interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section [anglecustomaction], page 315.

The feedback object (if any) *and* the object being edited are modified indirectly, by setting a slot called `:angle`. The programmer must provide constraints to this slot. If there is a feedback object, the interactor also sets its `:obj-over` field to the actual object that is being moved. This can be used, for example, to control the visibility of the feedback object or its size.

The angle slot is set with a value in radians measured counter-clockwise from the far right. Therefore, straight up is `(/ PI 2.0)`, straight left is `PI`, and straight down is `(* PI 1.5)`.

The object being changed is either gotten from the `:obj-to-change` slot of the interactor, or if that is `nil`, then from the object returned from `:start-where`.

The interactor needs to be told where the center of rotation should be. The slot `:center-of-rotation` can contain a point as a list of `(x y)`. If `:center-of-rotation` is `nil` (the default), then the center of the object being rotated is used.

For example, a line that can be rotated around an endpoint might have the following definition:

```
(create-instance 'ROTATING-LINE opal:line
   (:angle (/ PI 4))  ; initial value = 45 degrees up
   (:line-length 50)
   (:x1 70)
   (:y1 170)
   (:x2 (o-formula (+ (gvl :x1)
                      (round (* (gvl :line-length)
(cos (gvl :angle)))))))
   (:y2 (o-formula (- (gvl :y1)
      (round (* (gvl :line-length)
(sin (gvl :angle)))))))

(create-instance 'MYROTATOR Inter:Angle-Interactor
   (:start-where T)
   (:obj-to-change ROTATING-LINE)
   (:center-of-rotation (o-formula (list (gvl :obj-to-change :x1)
 (gvl :obj-to-change :y1))))
   (:window MYWINDOW))
```

6.19.2 Extra Parameters

The extra parameters are:

`:obj-to-change`

  - If an object is supplied here, then the interactor modifies the `:angle` slot of that object. If `:obj-to-change` is `nil`, then the interactor operates on whatever is returned from `:start-where`. The default value is `nil`.

`:center-of-rotation`
>   - This is the center of rotation for the interaction. It should be a list of (`x y`).
>   If `nil`, then the center of the real object being rotated (note: *not* the feedback
>   object) is used. The default value is `nil`.

6.19.3 Application Notification

Often, it is not necessary to have the application notified of the result of a angle-interactor,
if you only want the object to rotate around. Otherwise, you can have constraints in the
application to the `:angle` slot.

   Alternatively, the programmer can provide a function to be called when the interactor
is complete by putting the function in the `:final-function` slot. This function is called
with the following arguments:

```
(lambda (an-interactor object-being-rotated final-angle))
```

6.19.4 Normal Operation

If the value of `:continuous` is T, then when the start event happens, the interactor deter-
mines the object to be changed as either the value of the `:obj-to-change` slot, or if that
is `nil`, then the object returned from the `:start-where`. The `:obj-over` slot of the object
in the `:feedback-obj` slot of the interactor is set to the object being changed. Then, for
every mouse movement until the stop event happens, the interactor sets the `:angle` slot.
The object that is modified while running is either the feedback object if it exists or the
object being changed if there is no feedback object.

   If the mouse goes outside of `:running-where`, then if `:outside` is `:last`, nothing hap-
pens until the mouse comes back inside or the stop or abort events happen (the object stays
at its last legal inside value). If `:outside` is `nil`, then the feedback object's `:obj-over` slot
is set to `nil`. If there is no feedback object and the mouse goes outside, then the object
being changed is returned to its original angle (before the interactor started).

   If the abort event happens, then the feedback object's `:obj-over` slot is set to `nil`, or
if there is no feedback object, then the object being rotated is returned to its original angle
(before the interactor started).

   When the stop event happens, the feedback object's `:obj-over` slot is set to `nil`, and
the `:angle` slot of the actual object is set with the last value, and the final-function (if any)
is called.

   If the interactor is *not* continuous, when the start event happens, the `:angle` slot of the
actual object is set with the initial value, and the final-function (if any) is called.

6.20 `text-interactor`

```
(create-instance 'inter:text-interactor inter:interactor
  ;; Slots common to all interactors (see section [Slots of All Interac-
tors], page 250)
  (:start-where nil)
  (:window nil)
  (:start-event :leftdown)
  (:continuous t)
  (:stop-event nil)
```

```
    (:running-where t)
    (:outside nil)
    (:abort-event '(:control-\g :control-g))
    (:waiting-priority normal-priority-level)
    (:running-priority running-priority-level)
    (:active t)
    (:self-deactivate nil)

    ; Slots specific to the text-interactor (discussed in this section)
    (:final-function nil)       ; (lambda (inter obj-being-edited final-event final-█
string x y))
    (:obj-to-change nil)        [; The object to change the :string slot of (if nil, then
                                ; the object returned from the start-where)

    (:feedback-obj nil)         ; Optional interim feedback object. The in-
ter will set this object's :string, :cursor-index,
                                ; :obj-over, and :box slots
    (:cursor-where-press T)     ; Whether to position the cursor under the mouse or at th
    (:input-filter nil)         ; Used for gridding (see section [gridding],█
page 272)
    (:button-outside-stop? T)   ; Whether a click outside the string should stop edit-█
ing (see section [extra-text-parameters], page 286)

    ; Advanced feature:  Read-only slots.
    ; See section [specialslots], page 310, for details about these slots.
    (:first-obj-over nil)     ; Read-only slot. The object returned from the start-█
where.
    (:current-window nil)     ; Read-only slot. The window of the last (or cur-█
rent) event.
    (:start-char nil)         ; Read-only slot. The character or keyword of the start ev

    ; Advanced feature: Customizable action routines.
    ; See sections [Slots of All Interactors], page 250, and [textcustomaction],█
page 317, for details about functions in these slots.
    (:start-action ...)       ; (lambda (inter new-obj-over start-event))
    (:running-action ...)     ; (lambda (inter obj-over event))
    (:stop-action ...)        ; (lambda (inter obj-over stop-event))
    (:abort-action ...)       ; (lambda (inter obj-over abort-event))
    (:outside-action ...)     ; (lambda (inter obj-over))
    (:back-inside-action ...) ; (lambda (inter obj-over event))
...)
```

If you want to use multi-font, multi-line text objects, you will probably want to use the special interactors defined for them, which are described in the Opal chapter.

The text-interactor will input a one-line or multi-line string of text, while allowing editing on the string. A fairly complete set of editing operations is supported, and the programmer or user can add new ones or change the bindings of the default operations.

The intention is that this be used for string entry in text forms, for file names, object names, numbers, labels for pictures, etc. The strings can be in any font, but the entire string must be in the same font. More complex editing capabilities are clearly possible, but not implemented here.

Text-interactors work on `opal:text` objects. The interactor can either be permanently tied to a particular text object, or it will get the object from where the mouse is when the interaction starts. There may be a feedback object to show the edits, with the final object changed only when the editing is complete, or else the object itself can be edited. (Feedback objects are actually not very useful for text-interactors.) Both the feedback and the main object should be an `opal:text` object.

There is an example of the use of the text-interactor below. Other examples can be found in the top type-in area in the `v-slider` gadget in the Garnet Gadget Set, and in the file `demo-text.lisp`.

6.20.1 Default Editing Commands

There is a default set of editing commands provided with text interactors. These are based on the EMACS command set. The programmer change this and can create his own mappings and functions (see section [keytranslations], page 289). In the following, keys like `"insert"` and `"home"` are the specially labeled keys on the IBM/RT or Sun keyboard. If your keyboard has some keys you would like to work as editing commands, see section [eventspec], page 237.

`^h, delete, backspace`
        delete previous character.

`^w, ^backspace, ^delete`
        delete previous word.

`^d`   delete next character.

`^u`   delete entire string.

`^k`   delete to end of line.

`^b, left-arrow`
        go back one character.

`^f, right-arrow`
        go forward one character.

`^n, down-arrow`
        go vertically down one line (for multi-line strings).

`^p, up-arrow`
        go vertically up one line (for multi-line strings).

`^<, ^comma, home`
        go to the beginning of the string.

`^>, ^period, end`
        go to the end of the string.

`^a`   go to beginning of the current line (different from `^<` for multi-line strings).

`^e`   go to end of the current line (different from `^>` for multi-line strings).

`^y, insert`
          insert the contents of the cut buffer into the string at the current point.

`^c`   copy the current string to the cut buffer.

`enter, return, ^j, ^J`
          Add a new line.

`^o`   Insert a new line after the cursor.

`any mouse button down inside the string`
          move the cursor to the specified point. This only works if the `:cursor-where-press` slot of the interactor is non-NIL.

In addition, the numeric keypad is mapped to normal numbers and symbols.

Note: if you manage to get an illegal character into the string, the string will only be displayed up to the first illegal character. The rest will be invisible (but still in the `:string` slot).

The interactor's `:stop-event` and `:abort-event` override the above operations. For example, if the `:stop-event` is `:any-mousedown`, then when you press in the string, editing will stop rather than causing the cursor to move. Similarly, if `#\RETURN` is the `:stop-event`, then it cannot be inserted into the string for a multi-line string, and if `:control-\c` is the `:abort-event`, it cannot be used to copy to the X cut buffer. Therefore, you need to pick the `:stop-event` and `:abort-event` appropriately, or change the bindings (see section [keytranslations], page 289)

6.20.2 Default Operation

This section describes how the text interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section [textcustomaction], page 317.

Unlike other interactors, the feedback object (if any) and the object being edited are modified directly, by setting the `:string` and `:cursor-index` fields (that control the value displayed and the position of the cursor in the string). If there is a feedback object, the interactor also sets the first two values of its `:box` field to be the position where the start event happened. This might be used to put the feedback object at the mouse position when the user presses to start a new string.

In general, feedback objects are mainly useful when you want to create new strings as a result of the event.

The object being changed is either gotten from the `:obj-to-change` slot of the interactor, or if that is `nil`, then from the object returned from `:start-where`.

6.20.3 Multi-line text strings

The default stop event for text interactors is `#\RETURN`, which is fine for one-line strings, but does not work for multi-line strings. For those, you probably want to specify a stop event as something like `:any-mousedown` so that `#\RETURN`s can be typed into the string (actually, the character in the string that makes it go to the next line is #\NEWLINE; the interactor maps the return key to #\NEWLINE). Also `:any-mousedown` would be a bad

choice for the stop event if you wanted to allow the mouse to be used for changing the text insert cursor position.

Note that the stop event is *not* edited into the string.

The `:outside` slot is ignored.

The default `:running-where` is T for text-interactors.

6.20.4 Extra Parameters

The extra parameters are:

`:obj-to-change`
> If an object is supplied here, then the interactor modifies the

`:string` and `:cursor-index`
> slots of that object. If `:obj-to-change` is `nil`, then the interactor operates on whatever is returned from `:start-where`. The default value is `nil`.

`:cursor-where-press`
> If this slot is non-NIL, then the initial position of the text editing cursor is underneath the mouse cursor (i.e, the user begins editing the string on the character under where the mouse was pressed). This is the default. If `:cursor-where-press` is specified as `nil`, however, the cursor always starts at the end of the string. This slot also controls whether the mouse is allowed to move the cursor while the string is being edited. If `:cursor-where-press` is `nil`, then mouse presses are ignored while editing (unless they are the `:stop-` or `:abort-` events), otherwise, presses can be used to move the cursor.

`:input-filter`
> Used to support gridding. See section [gridding], page 272.

`:button-outside-stop?`
> Whether a mouse click *outside* the string should stop 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 feature is enabled, but you can turn it off by setting the `:button-outside-stop?` parameter to `nil`.

6.20.5 Application Notification

Often, it is not necessary to have the application notified of the result of a text-interactor, if you only want the string object to be changed, it will happen automatically.

Alternatively, the programmer can provide a function to be called when the interactor is complete by putting the function in the `:final-function` slot. This function is called with the following arguments:

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

The definition of the type for `final-event` is in section Section 14.22 [Events], page 696. (It is a Lisp structure containing the particular key hit.) The `final-string` is the final value for the entire string. *It is important that you copy the string (with* `copy-seq`*) before using it, since it will be shared with the feedback object.* The `x` and `y` parameters are the *initial*

positions put into the feedback object's `:box` slot (which might be used as the position of the new object). The values of x and y are *filtered* values computed via the `:input-filter` given to the interactor (see section [gridding], page 272).

6.20.6 Normal Operation

If the value of `:continuous` is T, then when the start event happens, if there is a feedback object, then its `:box` slot is set to the position of the start-event, and its `:obj-over` slot is set to `:obj-to-change` or the result of the `:start-where`. Its `:cursor-index` is set to the position of the start-event (if `:cursor-where-press` is T) or to the end of the string (so the cursor becomes visible). If there is no `:feedback-obj`, then the `:obj-to-change` or if that is `nil`, then the object returned from `:start-where` has its cursor turned on at the appropriate place. If the start event was a keyboard character, it is then edited into the string. Therefore, you can have a text interactor start on `:any-keyboard` and have the first character typed entered into the string.

Then, for every subsequent keyboard down-press, the key is either entered into the string, or if it is an editing command, then it is performed.

If the mouse goes outside of `:running-where`, then the cursor is turned off, and it is turned back on when the mouse goes back inside. Events other than the stop event and the abort event are ignored until the mouse goes back inside. Note: this is usually not used because `:running-where` is usually T for text-interactors. If it is desirable to only edit while the mouse is over the object, then `:running-where` can be specified as `'(:in *)` which means that the interactor will work only when the mouse is over the object it started over.

If the abort event happens, then the feedback object's `:string` is set with its initial value, its `:cursor-index` is set to `nil`, and its `:obj-over` is set to `nil`. If there is no feedback object, then the main object's `:string` is set to its original value and its `:cursor-index` is set to `nil`.

When the stop event happens, if there is a feedback object, then its `:visible` slot is set with `nil`, the main object is set with feedback object's `:string`, and the `:cursor-index` is set to `nil`. If there is no feedback object, then the `:cursor-index` of the main object is set to `nil`. Note that the stop event is *not* edited into the string. Finally, the final-function (if any) is called.

If the interactor is *not* continuous, when the start event happens, the actions described above for the stop event are done.

6.20.7 Useful Functions

> `inter:Insert-Text-Into-String` *text-obj new-string* &optional *(move-back-cursor 0)*[function], page 90

The function `Insert-Text-Into-String` inserts a string *new-string* into an `opal:text` object *text-obj* at the current cursor point. This can be used even while the text-interactor is running. For example, an on-screen button might insert some text into a string. After the text is inserted, the cursor is moved to the end of the new text, minus the optional offset *move-back-cursor* (which should be a non-negative integer). For example, to insert the string `"(+ foo 5)"` and leave the cursor between the `"5"` and the `""`], you could call:

> `(Insert-Text-Into-String MYTEXT "(+ foo 5)" 1)`

6.20.8 Examples

6.20.9 Editing a particular string

This creates an aggregadget containing a single-line text object and an interactor to edit it
when the right mouse button is pressed.

```
(create-instance 'EDITABLE-STRING opal:aggregadget
   (:left 10)
   (:top 200)
   (:parts
    '((:txt ,opal:text
    (:left ,(o-formula (gvl :parent :left)))
    (:top ,(o-formula (gvl :parent :top)))
    (:string "Hello World")))) ; default initial value
   (:interactors
    '((:editor ,Inter:Text-Interactor
       (:start-where ,(o-formula (list :in (gvl :operates-on :txt))))
       (:window ,(o-formula (gvl :operates-on :window)))
       (:stop-event (:any-mousedown #\RETURN)) ; either
       (:start-event :rightdown)))))
```

6.20.10 Editing an existing or new string

Here, the right button will create a new multi-line string object when the user presses on
the background, and it will edit an existing object if the user presses on top of it, as in
Macintosh MacDraw.

Note: This uses a formula in the `:feedback-obj` slot that depends on the `:first-obj-`
`over` slot of the interactor. This slot, which holds the object the interactor starts over, is
explained in section [specialslots], page 310.

```
(create-instance 'THE-FEEDBACK-OBJ opal:text
   (:string "")
   (:visible NIL)
   (:left (formula '(first (gvl :box))))
   (:top (formula '(second (gvl :box)))))

;;; Assume there is a top level aggregate in the window called top-agg.
;;; Create an aggregate to hold all the strings. This aggregate must have a fixed
;;;    size so user can press inside even when it does not contain any objects.
(create-instance 'OBJECT-AGG opal:aggregate
   (:left 0)(:top 0)
   (:width (o-formula (gvl :window :width)))
   (:height (o-formula (gvl :window :height))))

(opal:add-components TOP-AGG THE-FEEDBACK-OBJ OBJECT-AGG)
(opal:update MYWINDOW)

(create-instance 'CREATE-OR-EDIT Inter:Text-Interactor
    (:feedback-obj (o-formula
```

```
       (if (eq :none (gvl :first-obj-over))
  ;then create a new object, so use feedback-obj
  THE-FEEDBACK-OBJ
  ;else use object returned by mouse
  NIL)))
       (:start-where '(:element-of-or-none ,OBJECT-AGG))
       (:window MYWINDOW)
       (:start-event :any-rightdown)
       (:stop-event '(:any-mousedown :control-\j)) ; either one stops
       (:final-function
        #'(lambda (an-interactor obj-being-edited stop-event final-string x y)▮
    (declare (ignore an-interactor stop-event))
    (when (eq :none obj-being-edited)
      ;; then create a new string and add to aggregate.
      ;; Note that it is important to copy the string.
      (let ((new-str (create-instance NIL opal:text
        (:string (copy-seq final-string))
        (:left x)(:top y))))
        (opal:add-component OBJECT-AGG new-str)
        (s-value THE-FEEDBACK-OBJ :string "") ; so starts empty next time
        )))))
```

6.20.11 Key Translation Tables

The programmer can change the bindings of keyboard keys to editing operations, and even add new editing operations in a straightforward manner.

Each text interactor can have its own *key translation table*. The default table is stored in the top-level `Text-Interactor` object, and so text-interactor instances will inherit it automatically. If you want to change the bindings, you need to use `Bind-key`, `Unbind-key`, `Unbind-All-Keys`, or `Set-Default-Key-Translations` (these functions are defined below).

If you want to change the binding for all of your text interactors, you can edit the bindings of the top-level `Text-Interactor` object. If you want a binding to be different for a particular interactor instance, just modify the table for that instance. What happens in this case is that the inherited table is copied first, and then modified. That way, other interactors that also inherit from the default table will not be affected. This copy is performed automatically by the first call to one of these functions.

Bindings can be changed while the interactor is running, and they will take effect immediately.

    `inter:Bind-Key` *key val an-interactor*[function], page 90

`Bind-Key` sets the binding for *key* to be *val* for *an-interactor*. The *key* can either be a Lisp character (like `:control-\t`) or a special keyword that is returned when a key is hit (like `:uparrow`). If the key used to have some other binding, the old binding is removed. It is fine to bind multiple keys to the same value, however (e.g., both `^p` and `:uparrow` are bound to `:upline`).

The second parameter (*val*) can be any one of the following four forms:

A character to map into. This allows special keys to map to regular keys. So, for example, you can have `:super-4` map to `#\4`.

A string. This allows the key to act like an abbreviation and expand into a string. For example, `(inter:bind-key :F2 "long string" MYINTER)` will insert "long string" whenever F2 is hit. Unfortunately, the string must be constant and cannot, for example, be computed by a formula.

One of the built-in editing operations which are keywords. These are implemented internally to the text interactor, but the user can decide which key(s) causes them to happen. The keywords that are available are:

> `:prev-char` - move cursor to previous character.
>
> `:next-char` - move cursor to next character.
>
> `:up-line` - move cursor up one line.
>
> `:down-line` - move cursor down one line.
>
> `:delete-prev-char` - delete character to left of cursor.
>
> `:delete-prev-word` - delete word to left of cursor.
>
> `:delete-next-char` - delete character to right of cursor.
>
> `:kill-line` - delete to end of line.
>
> `:insert-lf-after` - add new line after cursor.
>
> `:delete-string` - delete entire string.
>
> `:beginning-of-string` - move cursor to beginning of string.
>
> `:beginning-of-line` - move cursor to beginning of line.
>
> `:end-of-string` - move cursor to end of string.
>
> `:end-of-line` - move cursor to end of line.
>
> `:copy-to-X-cut-buffer` - copy entire string to cut buffer.
>
> `:copy-from-X-cut-buffer` - insert cut buffer at current cursor.

For example, `(inter:bind-key :F4 :upline MYINTER)` will make the F4 key move the cursor up one line.

A function that performs an edit. The function should be of the following form:

```
(lambda (an-interactor text-obj event))
```

The interactor will be the text-interactor, the text object is the one being edited, and the event is an Interactor event structure (see section ⟨undefined⟩ [events], page ⟨undefined⟩). Note: *not* a lisp character; the character is a field in the event. This function can do arbitrary manipulations of the `:string` slot and the `:cursor-index` slot of the `text-obj`. For example, the following code could be used to implement the "swap previous two character" operation from EMACS:

```
;; first define the function
(defun flip (inter str event) ; swap the two characters to the left of the cursor
  (let ((index (gv str :cursor-index)) ; get the old values
(s (gv str :string)))
    (when (> index 1)  ; make sure there are 2 chars to the left of the cursor
      (let ((oldsecondchar (elt s (1- index)))) ; do the swap in place in the str
```

```
      (setf (elt s (1- index)) (elt s (- index 2)))
      (setf (elt s (- index 2)) oldsecondchar)
      (mark-as-changed str :string ))))) ; since we modified the string value▉
      ; of the object in place, we have to let KR know
      ; it has been modified.
      ;; now bind it to ^t for a particular text-interactor called my-text-inter.▉
      (inter:bind-key :control-\t #'flip MY-TEXT-INTER) ; lower case t
```

The function `Unbind-Key` removes the binding of *key* for *an-interactor*. All keys that are not bound to something either insert themselves into the string (if they are printable characters), or else cause the interactor to beep when typed.

    inter:Unbind-Key *key an-interactor*[function], page 90

    inter:Unbind-All-Keys *an-interactor*[function], page 90

`Unbind-All-Keys` unbinds all keys for *an-interactor*. This would usually be followed by binding some of the keys in a different way.

    inter:Set-Default-Key-Translations *an-interactor*[function], page 90

This sets up *an-interactor* with the default key bindings presented in section [defaulteditingcommands], page 284. This might be useful to restore an interactor after the other functions above were used to change the bindings.

6.20.12 Editing Function

If you need even more flexibility than changing the key translations offers, then you can override the entire editing function, which is implemented as a method. Simply set the `:edit-func` slot of the text interactor with a function as follows:

    lambda (an-interactor string-object event)

It is expected to perform the modifications of the string-object based on the `event`, which is a Garnet event structure (section ⟨undefined⟩ [events], page ⟨undefined⟩).

6.21 Gesture-Interactor

```
(create-instance 'inter:Gesture-Interactor inter:interactor
  ;; Slots common to all interactors (see section [Slots of All Interac-
tors], page 250)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)                   ; Must be T for gesture-interactor
  (:stop-event NIL)
  (:running-where T)
  (:outside NIL)
  (:abort-event '(:control-\g :control-g))
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
  (:active T)
  (:self-deactivate NIL)
```

```
      ; Slots specific to the gesture-interactor (discussed in this section)
      (:final-function NIL)            ; (lambda (inter first-obj-over gesture-
   name attribs points-array nap dist))
      (:classifier NIL)               ; classifier to use
      (:show-trace T)                 ; show trace of gesture?
      (:min-non-ambig-prob nil)       ; non-ambiguity probability
      (:max-dist-to-mean nil)         ; distance to class mean
      (:went-outside NIL)             ; Read-only slot. Set in outside action function

      ; Advanced feature:  Read-only slots.
      ; See section [specialslots], page 310, for details about these slots.
      (:first-obj-over NIL)     ; Read-only slot. The object returned from the start-
   where.
      (:current-window NIL)     ; Read-only slot. The window of the last (or cur-
   rent) event.
      (:start-char NIL)         ; Read-only slot. The character or keyword of the start ev

      ; Advanced feature: Customizable action routines.
      ; See sections [Slots of All Interactors], page 250, and [gestcustomaction],
   page 317, for details about functions in these slots.
      (:start-action ...)       ; (lambda (inter obj-under-mouse point))
      (:running-action ...)     ; (lambda (inter new-obj-over point))
      (:stop-action ...)        ; (lambda (inter final-obj-over point))
      (:abort-action ...)       ; (lambda (inter))
      (:outside-action ...)     ; (lambda (inter prev-obj-over))
      (:back-inside-action ...) ; (lambda (inter new-obj-over))
   ...)
```

The Gesture-interactor is used to recognize single-path gestures that are drawn with the mouse. For example, this interactor might be used to allow the user to create circles and rectangles by drawing an ellipse for a circle and an "L" shape for a rectangle with the mouse. A *classifier* will be created for these two gestures. A "classifier" is a data structure that holds the information the gesture interactor needs to differentiate the gestures. Classifiers are created by using a special training program to give several examples of each kind of gesture that will be recognized. For instance, you might use Agate (section [agate], page 297), the Garnet gesture trainer, to give 15 examples of the ellipses and 15 of the "L" shape. Each gesture is named with a keyword (here, :circle and :rectangle might be used). Then, the classifier will be written to a file. The gesture interactor will then read this file and know how to recognize the specified gestures.

The classification algorithm is based on Rubine's gesture recognition algorithm *rubine, rubinesiggraph*. It uses a statistical technique.

There is one example of the gesture-interactor below. Other examples can be found in the files demo-arith.lisp and demo-gesture.lisp.

Unlike other interactors, Gestures are not automatically loaded when you load Garnet. To load gestures, use:

```
   (load Garnet-Gesture-Loader)
```

6.21.1 Default Operation

This section describes how the gesture-interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section [gestcustomaction], page 317.

The interactor is used by specifying a classifier to use and a `final-function` ([gestapplnotif], page 294) to call with the result of the classification.

The `:classifier` slot should be set to the value of a gesture classifier. Classifiers trained and saved by Agate can be read with `inter:gest-classifier-read`. The `:final-function` slot should be set to a function to call with the result of the gesture classification.

Since the programmer may or not want the trace of the gesture to be shown, there are two drawing modes for the interactor, determined by the `:show-trace` slot. If `:show-trace` is non-NIL (the default), then the points making up the gesture will be displayed as the gesture is drawn and erased when it is finished.

6.21.2 Rejecting Gestures

If the gesture-interactor is unable to classify the gesture, it will call the `final-function` with a value of `nil` for the classified gesture name. Often, the gesture will be ambiguous, in that it is similar to more than one known gesture. By setting the `:min-non-ambig-prob` slot, the programmer can specify the minimum non-ambiguous probability below which gestures will be rejected. Empirically, a value of .95 has been found to be a reasonable value for a number of gesture sets *rubine*.

Also, the gesture may be an outlier, different from any of the expected gestures. An approximation of the Mahalanobis distance from the features of the given gesture to the features of the gesture it was classified as gives a good indication of this. By setting the `:max-dist-to-mean` slot, the programmer can specify the maximum distance above which gestures will be rejected. Rubine shows that a value of 60 (for our feature set) is a good compromise between accepting obvious outliers and rejecting reasonable gestures.

NIL for either parameter means that that kind of checking is not performed.

6.21.3 Extra Parameters

The extra parameters are:

`:classifier`
> - This field determines which classifier to use when recognizing gestures. If `nil` (the default), the gesture-interactor will call the `final-function` with a result of `nil`.

`:show-trace`
> - If non-NIL (the default), the points making up the gesture are displayed in the supplied interactor window as the gesture is drawn. If `nil`, no points are displayed.

`:min-non-ambig-prob`
> - This field determines the minimum non-ambiguous probability below which gestures will be rejected. The default value of `nil` causes the interactor to not make this calculation and pass `nil` as the `nap` parameter to `final-function`.

`:max-dist-to-mean`
> - This field determines the maximum distance to the classified gesture from the given gesture. Any gesture with a distance above this value will be rejected. The default value of `nil` causes the interactor to not make this calculation and pass `nil` as the `dist` parameter to `final-function`.

6.21.4 Application Notification

Like the two-point-interactor, it is always necessary to have an application function called with the result of the gesture-interactor. The function is put into the `:final-function` slot of the interactor, and is called with the following arguments:

```
(lambda (an-interactor first-obj-over gesture-name attribs points-array nap dist))
```

The `gesture-name` will be set to the name the drawn gesture was recognized as. These names are stored in the classifier as keyword parameters (e.g., `:circle`). If the gesture could not be recognized this will be set to `nil`.

The `attribs` will be set to a structure of gesture attributes that may be useful to the application. For example, the bounding box of the gesture is one of these attributes. The following function calls can be used to access these attributes:

```
(gest-attributes-startx attribs)          ; first point
(gest-attributes-starty attribs)

(gest-attributes-initial-sin attribs)     ; initial angle to the x axis
(gest-attributes-initial-cos attribs)

(gest-attributes-dx2 attribs)             ; differences: endx - prevx
(gest-attributes-dy2 attribs)             ;                    endy - prevy
(gest-attributes-magsq2 attribs)          ; (dx2 * dx2) + (dy2 * dy2)

(gest-attributes-endx attribs)            ; last point
(gest-attributes-endy attribs)

(gest-attributes-minx attribs)            ; bounding box
(gest-attributes-maxx attribs)
(gest-attributes-miny attribs)
(gest-attributes-maxy attribs)

(gest-attributes-path-r attribs)          ; total path length (in rads)
(gest-attributes-path-th attribs)         ; total rotation (in rads)
(gest-attributes-abs-th attribs)          ; sum of absolute values of path angles
(gest-attributes-sharpness attribs)       ; sum of non-linear function of ab-
solute values
                                          ; of path angles counting acute an-
gles heavier
```

The `points-array` will be set to an array (of the form [x1 y1 x2 y2...]) containing the points in the gesture. This array can be used along with a `nil` classifier to use the gesture-interactor as a trace-interactor. A trace-interactor returns all the points the mouse

goes through between the `start-event` and the `stop-event`. This is useful for inking in a drawing program.

IMPORTANT NOTE: The elements of the `attribs` structure and the `points-array` should be accessed individually (e.g., `(gest-attributes-minx attribs)` `(aref points-array 0)` etc.) or else they should be copied (e.g., `(copy-gest-attributes attribs)` `(copy-seq points-array)`) before they are used in any object slots. This is necessary because the interactor reuses the `attribs` structure and the `points-array` in order to avoid extra memory allocation.

If `:min-non-ambig-prob` is not `nil`, the `nap` parameter will be set to the calculated non-ambiguous probability of the entered gesture.

If `:max-dist-to-mean` is not `nil`, the `dist` parameter will be set to the calculated distance of the entered gesture from the classification.

6.21.5 Normal Operation

When the start event happens, if `:show-trace` is non-NIL, a trace following the mouse pointer will be displayed. If `:show-trace` is `nil`, no trace will be seen.

If the mouse goes outside of `:running-where`, then the system will beep and if `:show-trace` is non-NIL, the trace will be erased.

If the abort event happens and if `:show-trace` is non-NIL, the trace will be erased.

When the stop event happens, if `:show-trace` is non-NIL, the trace will be erased. Then, the final-function is called with the result of classifying the given gesture with the classifier supplied in the `:classifier` slot.

An error will be generated if the `:continuous` slot is anything other than T, the default.

6.21.6 Example - Creating new Objects

Create a rectangle when an "L" shape is drawn and create a circle when a circle is drawn.

```
; load the gesture interactor, unless already loaded (Garnet does NOT load the gesture
interactor by default)
(defvar DEMO-GESTURE-INIT
    (load Garnet-Gesture-Loader))


; handle-gesture is called by the gesture interactor after it classifies a gesture
(defun Handle-Gesture (inter first-obj-over gesture-name attribs
                        points-array nap dist)
    (declare (ignore inter first-obj-over points-array nap dist))
    (case gesture-name
        (:CIRCLE
            ; create a circle with the same "radius" as the gesture and with the same
per left of the gesture
            (opal:add-components SHAPE-AGG
                (create-instance nil opal:circle
                    (:left (inter:gest-attributes-minx attribs))
                    (:top (inter:gest-attributes-miny attribs))
                    (:width (- (inter:gest-attributes-maxx attribs)
```

```
                                         (inter:gest-attributes-minx attribs)))
                       (:height (- (inter:gest-attributes-maxx attribs)
                                   (inter:gest-attributes-minx attribs)))))
          )
          (:RECTANGLE
              ; create a rectangle with the same height and width as the ges-
ture and with the same upper left of the gesture
              (opal:add-components SHAPE-AGG
                   (create-instance nil opal:rectangle
                       (:left (inter:gest-attributes-minx attribs))
                       (:top (inter:gest-attributes-miny attribs))
                       (:width (- (inter:gest-attributes-maxx attribs)
                                  (inter:gest-attributes-minx attribs)))
                       (:height (- (inter:gest-attributes-maxy attribs)
                                   (inter:gest-attributes-miny attribs)))))
          )
          (otherwise
              (format T "Can not handle this gesture ...~%~%")
          )
     )
     (opal:update TOP-WIN)
)

; create top-level window
(create-instance 'TOP-WIN inter:interactor-window
   (:left 750) (:top 80) (:width 520) (:height 400)
)

; create the top level aggregate in the window
(s-value TOP-WIN :aggregate (create-instance 'TOP-AGG opal:aggregate))

; create an aggregate to hold the shapes we will create
(create-instance 'SHAPE-AGG opal:aggregate)
(opal:add-components TOP-AGG SHAPE-AGG)
(opal:update TOP-WIN)

; create a gesture interactor that will allow us to create circles and rectangles
(create-instance 'GESTURE-INTER inter:gesture-interactor
   (:window TOP-WIN)
   (:start-where (list :in TOP-WIN))
   (:running-where (list :in TOP-WIN))
   (:start-event :any-mousedown)
   (:classifier (inter:gest-classifier-read
                     (merge-pathnames "demo-gesture.classifier"
                           #+cmu "gesture-data:"
                           #-cmu user::Garnet-Gesture-Data-Pathname)))
   (:final-function #'Handle-Gesture)
```

```
      (:min-non-ambig-prob .95)
      (:max-dist-to-mean 60)
  )
```

6.21.7 Agate

Agate is a Garnet application that is used to train gestures for use with the gesture inter-
actor. Agate stands for **A G**esture-recognizer **A**nd **T**rainer by **E**xample. Agate is in the
`gesture` subdirectory, and can be loaded using (`garnet-load "gestures:agate"`). Then
type (`agate:do-go`) to begin.
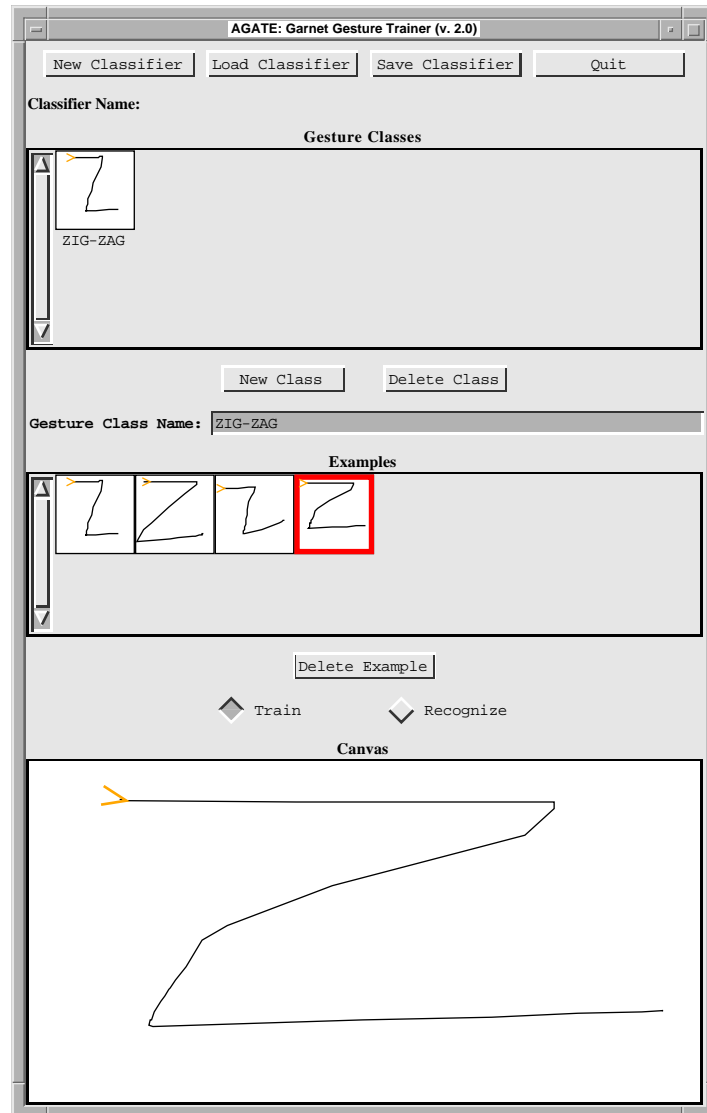
Figure 6.2: An example session with the Agate gesture trainer

6.21.8 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 section [rejecting-gestures], page 293).

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.

6.21.9 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*[function],■
> page 90
>
> *initial-classifier initial-examples initial-gesture-*
> *name final-function*

`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 five parameters:

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?

6.21.10 Gesture Demos

There are two demos that show how gestures can be used in an application. Demo-gesture allows you to draw rough approximations of circles and rectangles, which become perfect shapes in the window. Demo-unidraw is a gesture-based text editor which uses a gesture shorthand for entering characters. Both of these demos are discussed in the Demos section of this chapter, starting on page ⟨undefined⟩ [demos-first-page], page ⟨undefined⟩.

6.22 Animator-Interactor

```
(create-instance 'inter:Animator-Interactor inter:interactor
  ;; Slots common to all interactors (see section [Slots of All Interac-
tors], page 250)
  (:window nil)
  (:active T)

  ; Slots specific to the button-interactor (discussed in this section)
  (:timer-handler nil)          ; (lambda (inter))  ;; function to execute
  (:timer-repeat-wait 0.2)      ; time in seconds
...)
```

The `animator-interactor` has been implemented using the multiple process mechanism of Allegro, LispWorks, Lucid (also Sun and HP) Common Lisp. *It does not work under CMU Common Lisp, AKCL, CLISP, etc.; sorry.*

The `animator-interactor` works quite differently from other interactors. In particular, it is more procedural. You provide a function to be called at a fixed rate in the `:timer-handler` slot, and a time interval in the slot `:timer-repeat-wait` at which this function will be executed. The `:timer-handler` function takes as a parameter the animation interactor and should update the appropriate graphics.

Unlike other interactors, the animation interactor does *not* start immediately when created. You must explicitly start it operating with `inter:Start-Animator` and stop it with `inter:Stop-Animator`:

> `inter:Start-Animator` *animator-inter*[function], page 90

```
    inter:Stop-Animator animator-inter[function], page 90
```
After starting, the interactor will call the :timer-handler every :timer-repeat-wait seconds until you explicitly stop the interactor. It is OK for the :timer-handler itself to call stop-animator.

Two special-purpose animator interactors have been supplied that have built-in timer functions (so you don't have to supply the :timer-handler for these):

```
(create-instance 'inter:Animator-Bounce inter:animator-interactor
  (:x-inc 2)
  (:y-inc 2)
  (:timer-repeat-wait 0.2)  ; seconds
  (:obj-to-change nil)      ; fill this in
  ...)

(create-instance 'inter:Animator-Wrap inter:animator-interactor
  (:x-inc 2)
  (:y-inc 2)
  (:timer-repeat-wait 0.2) ; seconds
  (:obj-to-change nil)     ; fill this in
  ...)
```

Animator-bounce will move the object supplied in the :obj-to-change by :x-inc pixels in the x direction and :y-inc pixels in the y direction every :timer-repeat-wait seconds. The object is modified by directly setting its :left and :top. (Note: *not* its :box slot.) When the object comes to the edge of its window, it will bounce and change direction.

Animator-wrap moves an object the same way except that when it gets to an edge, it re-appears at the opposite edge of the window.

See the demo demo-animator for examples.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++■
         All the interactors that are not yet implemented
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++■
```

```
@section Button-Trill-Interactor
@node Button-Trill-Interactor
@emph{Performs the action once when the interactor starts, then after a
specified delay, does the action repeatedly at a particular rate until
stop-event}.

@emph{This section not yet written or implemented.}

@section Trace-Interactor
@node Trace-Interactor
@emph{This section not yet written or implemented.}

@section Multi-Point-Interactor
@node Multi-Point-Interactor
```

Stopping conditions: One of a set of other interactors wants to run, or two
points in the same place, or last point same as first point, or some other
special event, or event while outside.

@emph{This section not yet written or implemented.}

++■

6.23 Transcripts

Garnet will create a transcript of all mouse and keyboard events in a file, and allow the
file to be replayed later as if the user had executed all events again. This can be used for
demonstrations, human factors testing, and/or debugging. Using the transcript mechanism
is very easy. The procedure to start saving events is:

> `inter:Transcript-Events-To-File` *filename* *window-list*[function], page 90
> `&key` (*motion* T) (*if-exists* `:supersede`)
>   (*wait-elapsed-time* T) (*verbose* T)

Events are then written to file *filename*. The *window-list* is a list of windows that events
should be saved for. It is also allowed to be a single window. (Note: subwindows of windows
on the window list are also handled automatically, and do *not* have to be specified.) If the
`:motion` parameter is specified as `nil`, then mouse movement events are not saved to the
file, which can significantly decrease the file size. The `:if-exists` parameter is used in the
Lisp `open` command when opening a file, and takes the same values (see the Common Lisp
book). If specified as `:append`, then the new events are appended to the end of an existing
file. The transcript is a textual file, where each event has its own line.

When you are finished making the transcript, call

> `inter:Close-Transcript`[function], page 90

To replay a transcript, use:

> `inter:Transcript-Events-From-File` *filename window-list* `&key` (*wait-elapsed-*■
> *time* T)[function], page 90

The *filename* is the file to read from. *Wait-elapsed-time* determines if the replay should
wait for the correct time so the replay goes about the same speed as the original user went,
or else (if `nil`) whether the replay should just go as fast as possible. (Each event in the
transcript has a timestamp in it.)

It is important that the *window-list* passed to `Transcript-Events-From-File` be
windows that are the same type and in the same order as the windows passed to the
`Transcript-Events-To-File` call that made the transcript. Garnet maps each event from
the transcript into the corresponding window in the specified window-list. The windows
do not have to be in the same places (all events are window-relative), however.

A typical example would be to create a bunch of windows, call `Transcript-Events-`
`To-File`, do some operations, call `Close-Transcript`, then sometime later, create new
windows the same way, then call `Transcript-Events-From-File`.

During playback, all mouse and keyboard events are ignored, except `inter:*Garnet-Break-Key*`, which is normally bound to `:F1`. This aborts the transcript playback. Window refresh events are handled while replaying, however.

6.24 Advanced Features

This chapter describes a number of special features that will help experienced Interactor users achieve some necessary effects. The features described in this chapter are:

Priorities   Interactors can be put at different priority levels, to help control which ones start and stop with events.

Modes   The priority levels and the `:active` slots can be used for local or global modes.

Events   The `event` structure that describes the user's event can be useful.

Start-interactor and Abort-Interactor
            These functions can be used to explicitly start and stop an interactor without waiting for its events.

Special slots of interactors
            There are a number of slots of interactors that are maintained by the system that can be used by programmers in formulas or custom action routines.

Multiple windows
            Interactors can be made to work over multiple windows.

Waiting for interaction to complete
            To support synchronous interfaces.

Custom Action Routines
            Some advice about how to write your own action routines, when necessary.

6.25 Priority Levels

Normally, when events arrive from the user, they are processed by *all* the interactors that are waiting for events. This means that if two interactors are waiting for the same event (e.g. `:leftdown`) they may both start if the mouse location passes both of their `:start-where`s.

The interactors do not know about object covering, so that even if an object is covered by some other object, the mouse can still be in that object. For example, you might have an interactor that starts when you press over the indicator of a scroll bar, and a different interactor that starts when you press on the background of the scroll bar. However, if these interactors both start with the same event, they will both start when the user presses on the indicator, because it is also inside the background. Priority levels can be used to solve this problem. The higher-priority interactors get to process events and run first, and if they accept the event, then lower-priority interactors can be set up so they do not run. Garnet normally uses three priority levels, but you can but you can add more priority levels for your interactors as you need them (see below).

By default, interactors wait at "normal" priority for their start event to happen, and then are elevated to a higher priority while they are running. This means that the stop event for the running interactor will not be seen by other interactors. The programmer has

full control over the priorities of interactors, however. There are two slots of interactors that control this:

`:waiting-priority`

> - the priority of the interactor while waiting for its start event. The default value is `inter:normal-priority-level`.

`:running-priority`

> - the priority of the interactor while running (waiting for the stop event). The default value is `inter:running-priority-level`.

There are a list of priority levels, each of which contains a list of interactors. The events from the user are first processed by all the interactors in the highest priority level. All the interactors at this level are given the event. After they are finished, then lower level priorities may be given the event (controlled by the `:stop-when` slot of the priority level that has just finished running, see below). Thus, all the interactors at the same priority level get to process the events that any of them get.

There is a list of priorities stored in the variable `inter:priority-level-list`. The first element of this list has the highest priority, and the second element has the second priority, etc. This list is exported so programs can use the standard list manipulation routines to modify it.

The elements of this list must be instances of `inter:priority-level`, which is a KR schema with the following slots:

`:interactors`

> - List of interactors at this priority level. This slot is maintained automatically based on the values in the interactor's `:waiting-priority` and `:running-priority` slots. **Do not set or modify this slot directly.**

`:active`   - Determines whether this priority level and all the interactors in it are active. The default value is T. For an interactor to be usable, both the interactor's `:active` slot and the priority-level's `:active` slot must be non-NIL. If this slot is `nil`, then this level is totally ignored, including its `:stop-when` field (see below). The value of the `:active` slot can be a formula, but if it changes to be `nil`, the interactors will not be automatically aborted. Use the `change-active` function to get the priority level and all its interactors to be aborted immediately (see section [active], page 306). Note: It is a really bad idea to make the `:active` slot of any *running*-priority levels be `nil`, since interactors will start but never complete.

`:stop-when`

> - This slot controls what happens after the event has been processed by the interactors at this priority level. This slot can take one of three values:

> > `:always`   - Always stop after handling this level. This means that the event is never seen by interactors at lower levels. Pushing a new priority level with `:stop-when` as `:always` on the front of `:priority-level-list` is a convenient way to set up a special mode where the interactors in the new priority level are processed and all other ones are ignored. The priority level can be popped

or de-activated (by setting its `:active` slot to `nil`) to turn this mode off.

`:if-any` - If any of the interactors at this level accept the event, then do not pass the event down to lower levels. If no interactors at this level want the event, then *do* pass it through to lower levels. This is used, for example, for the `:stop-when` of the default `running-priority-level` to keep the stop-event of a running interactor from starting a different interactor.

`nil` - If `:stop-when` is `nil`, then the events are always passed through. This might be useful if you want to control the order of interactors running, or if you want to set the `:active` slots of the priority levels independently.

`:sorted-interactors`
 - See section [sorted-sec], page 306.

Three priority levels are supplied by default. These are:

`inter:running-priority-level`
 - The highest default priority is for interactors that are running. It is defined with `:stop-when` as `:if-any`.

`inter:high-priority-level`
 - A high-priority level for use by programs. It is defined with `:stop-when` as `:if-any`.

`inter:normal-priority-level`
 - The normal priority for use by interactors that are waiting to run. `:Stop-when` is `nil`.

The initial value of `priority-level-list` is:

```
(list running-priority-level high-priority-level normal-priority-level)
```

The programmer can create new priority levels (using (`create-instance` `nil` `inter:priority-level` ...) and add them to this list (using the standard Common Lisp list manipulation routines). The new priorities can be at any level. Priorities can also be removed at any time, but *do not remove the three default priority levels*. There is nothing special about the pre-defined priorities. They are just used as the defaults for interactors that do not define a waiting and running priority. For example, it is acceptable to use the pre-defined `inter:running-priority-level` as the `:waiting-priority` for an interactor, or to use `inter:high-priority-level` as the `:running-priority` of another interactor.

It is acceptable for an interactor to use the same priority level for its `:waiting-priority` and `:running-priority`, but it is a bad idea for the `:running-priority` to be *lower* than the `:waiting-priority`. Therefore, if you create a new priority level above the `running-priority-level` and use it as the `:waiting-priority` of an interactor, be sure to create an even higher priority level for use as the `:running-priority` of the interactor (or use the same priority level as both the waiting and running priorities).

6.25.1 Example

Consider the scroll bar. The interactor that moves the indicator might have higher priority than the one that operates on the background.

```
(create-instance NIL Inter:Move-Grow-Interactor
   (:window MYWINDOW)
   (:start-where (list :in-box INDICATOR))
   (:running-where (list :in-box SLIDER-SHELL))
   (:outside :last)
   (:attach-point :center)
   (:waiting-priority inter:high-priority-level))

(create-instance NIL Inter:Move-Grow-Interactor
   (:continuous NIL)
   (:window MYWINDOW)
   (:start-event :leftdown)
   (:start-where (list :in-box SLIDER-SHELL))
   (:obj-to-change indicator)
   (:attach-point :center))
```

6.25.2 Sorted-Order Priority Levels

As an experiment, and to support the Marquise tool which is in progress, there is an alternative way to control which interactors run. You can 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.

6.26 Modes and Change-Active

In order to implement "Modes" in a user interface, you need to have interactors turn off sometimes. This can be done in several ways. Section [modal-p], page 306, below discusses how to restrict all interactor input to a single window (like a dialog box) while suspending the interactors in all other windows. Section [change-active], page 307, below discusses how to turn off particular interactors or groups of interactors.

6.26.1 Modal Windows

When the `:modal-p` slot of an `interactor-window` is T, then interaction in all other Garnet windows will be suspended until the window goes away (e.g., the user clicks an "OK" button). Any input directed to a non-modal window will cause a beep. If more than one

modal window is visible at the same time, then input can be directed at any of them (this allows stacking of modal windows). The `:modal-p` slot can be calculated by a formula. Typically, however, the `:modal-p` slot will stay T, and you will simply set the window to be visible or invisible.

The `:modal-p` slot is often used in conjunction with `wait-interaction-complete`, a function which suspends all lisp activity until `interaction-complete` is called. An example application would make a modal window visible, then call `wait-interaction-complete`. The user would be unable to interact with the rest of the interface until the modal window was addressed. Then, when the user clicks on the "OK" button in the modal window, the window becomes invisible and `interaction-complete` is called. Interaction then resumes as usual in the interface. See section [w-i-c], page 311, for a discussion of `wait-interaction-complete`.

The `error-gadget` and `query-gadget` dialog boxes use this feature exactly as in the example above. They ensure that the user responds to the error message before continuing any action in the rest of the interface. The property sheet gadget display routines and the `gilt:show-in-window` routine have an optional modal parameter which uses this feature. You may be able to implement your design using these gadgets and routines, rather than using the `:modal-p` slot explicitly.

6.26.2 Change-Active

Interactors can either be turned on and off individually using the `:active` slot in each interactor, or you can put a group of interactors together in a priority level (see section [priorities], page 303) and turn on and off the entire group using the priority level's `:active` slot.

The `:active` slot of an interactor may be `s-value`'d explicitly, causing the interactor to abort immediately. But to change the activity of a priority level, you should use the function `Change-Active`:

inter:Change-Active *an-interactor-or-priority-level new-value*[function],
page 90

This makes the interactor or priority-level be active (if *new-value* is T) or inactive (if *new-value* is `nil`). When `change-active` makes a priority level not active, then all interactors on the priority level will abort immediately. Interactors are not guaranteed to abort immediately if their priority level's `:active` slot is simply set to `nil`.

6.27 Events

Some functions, such as `Start-Interactor` (see section [startinteractor], page 309) take an "event" as a parameter. You might also want to look at an event to provide extra features.

`Inter:Event` is an interactor-defined structure (a regular Lisp structure, not a KR schema), and is not the same as the events created by the X window manager or Mac QuickDraw. Normally, programs do not need to ever look at the event structure, but it is exported from interactors in case you need it.

`Inter:Event` has the following fields:

window   - The Interactor window that the event occurred in.

char   - The Lisp character that the event corresponds to. If this is a mouse event,
            then the `Char` field will actually hold a keyword like `:leftdown`.

code   - The X11 or MCL internal code for the event.

mousep   - Whether the event is a mouse event or not.

downp   - If a mouse event, whether it is a down-transition or not.

x   - The X position of the mouse when the event happened.

y   - The Y position of the mouse when the event happened.

timestamp
            - The X11 or MCL timestamp for the event.

Each of the fields has a corresponding accessor and setf function:

```
(event-window event)     (setf (event-window event) w)
(event-char event)       (setf (event-char event) c)
(event-code event)       (setf (event-code event) c)
(event-mousep event)     (setf (event-mousep event) T)
(event-downp event)      (setf (event-downp event) T)
(event-x event)          (setf (event-x event) 0)
(event-y event)          (setf (event-y event) 0)
(event-timestamp event)  (setf (event-timestamp event) 0)
```

You can create new events (for example, to pass to the `Start-Interactor` function),
using the standard structure creation function `Make-Event`.

`inter:Make-Event &key (`*window* `NIL) (`*char* `:leftdown) (`*code* `1) (`*mousep* `T)` [function],
page 90

                        `(`*downp* `T) (`*x* `0) (`*y* `0) (`*timestamp* `0))`

The last event that was processed by the interactors system is stored in the variable
`Inter:*Current-Event*`. This is often useful for functions that need to know where the
mouse is or what actual mouse or keyboard key was hit. Note that two of the fields of this
event (window and char) are copied into the slots of the interactor (see section [specialslots],
page 310) and can be more easily accessed from there.

6.27.1 Example of using an event

The two-point interactor calls the final-function with a `nil` parameter if the rectangle is
smaller than a specified size (see section [twopapplnotif], page 277). This feature can be
used to allow the end user to pick an object under the mouse if the user presses and releases,
but to select everything inside a rectangle if the user presses and moves (in this case, moves
more than 5 pixels).

Assume the objects to be selected are stored in the aggregate `all-obj-agg`.

```
(create-instance 'SELECT-POINT-OR-BOX Inter:Two-Point-Interactor
  (:start-where T)
  (:start-event :leftdown)
  (:abort-if-too-small T)
  (:min-width 5)
  (:min-height 5)
```

```
       (:line-p NIL)
       (:flip-if-change-sides T)
       (:final-function
        #'(lambda (an-interactor final-point-list)
            (if (null final-point-list)
        ; then select object at point. Get point from
        ;   the *Current-event* structure, and use it in the
        ;   standard point-to-component routine.
        (setf selected-object
     (opal:point-to-component ALL-OBJ-AGG
      (inter:event-x inter:*Current-event*)
      (inter:event-y inter:*Current-event*)))
        ; else we have to find all objects inside the rectangle.
        ;   There is no standard function to do this.
        (setf selected-object
     (My-Find-Objs-In-Rectangle ALL-OBJ-AGG final-point-list))))))
```

6.28 Starting and Stopping Interactors Explicitly

Normally an interactor will start operating (go into the "running" state) after its start-event
happens over its start-where. However, sometimes it is useful to explicitly start an interactor
without waiting for its start event. You can do this using the function `Start-interactor`.
For example, if a menu selection should cause a sub-menu to start operating, or if after
creating a new rectangle you want to immediately start editing a text string that is the
label for that rectangle.

> inter:Start-Interactor *an-interactor* &optional (*event* T)[function], page 90█

This function does nothing if the interactor is already running or if it is not active.
If an event is passed in, then this is used as the x and y location to start with. This
may be important for selecting which object the interactor operates on, for example if the
`:start-where` of the interactor is (`:element-of <agg>`), the choice of which element is
made based on the value of x and y in the event. (See section ⟨undefined⟩ [events], page ⟨un-
defined⟩, for a description of the event). If the event parameter is T (the default), then the
last event that was processed is re-used. The event is also used to calculate the appropriate
default stop event (needed if the start-event is a list or something like `:any-mousedown` and
the stop-event is not supplied). If the event is specified as `nil` or the x and y in the event
do not pass `:start-where`, the interactor is still started, but the initial object will be `nil`,
which might be a problem (especially for button-interactors, for example). NOTE: If you
want the interactor to never start by itself, then its `:start-where` or `:start-event` can be
set to `nil`.

Examples of using `start-interactor` are in the file `demo-sequence.lisp`.

Similarly, it is sometimes useful to abort an interactor explicitly. This can be done with
the function:

> inter:Abort-Interactor *an-interactor*[function], page 90

If the interactor is running, it is aborted (as if the abort event had occurred).

`Stop-Interactor` can be called to stop an interactor as if the stop event had happened.

> inter:Stop-Interactor *an-interactor*[function], page 90

It reuses the last object the interactor was operating on, and the current event is ignored. This function is useful if you want to have the interactor stopped due to some other external action. For example, to stop a text-interactor when the user chooses a menu item, simply call stop-interactor on the text-interactor from the final-function of the menu.

6.29 Special slots of interactors

There are a number of slots of interactors that are maintained by the system that can be used by programmers in formulas or custom action routines. These are:

`:first-obj-over`
> this is set to the object that is returned from

`:start-where`
> This might be useful if you want a formula in the

`:obj-to-change`
> lot that will depend on which object is pressed on (see the examples below and in section [editstringexample], page 288). Note that if the `:start-where` is T, then

`:first-obj-over`
> ill be T, rather than an object. The value in

`:first-obj-over`
> does not change as the interactor is running (it is only set once at the beginning).

`:current-obj-over`
> this slot is set with the object that the mouse was last over (see section [menufinalfeedbackobj], page 259).

`:current-window`
> this is set with the actual window of the last (or current) input event. This might be useful for multi-window interactors (see section [multiwindow], page 311). The `:current-window` slot is set repeatedly while the interactor is running.

`:start-char`
> The Lisp character (or keyword if a mouse event) of the actual start event. This might be useful, for example, if the start event can be one of a set of things, and some parameter of the interactor depends on which one. See the example below. The value in

`:start-char`
> does not change as the interactor is running (it is only set once at the beginning).

6.29.1 Example of using the special slots

This example uses two slots of the interactor in formulas. A formula in the `:grow-p` slot determines whether to move or grow an object based on whether the user starts with a left or right mouse button (`:start-char`). A formula in the `:line-p` slot decides whether to change this object as a line or a rectangle based on whether the object started on (`:first-obj-over`) is a line or not. Similarly, a formula in the feedback slot chooses the correct type of object (line or rectangle).

The application creates a set of objects and stores them in an aggregate called `all-object-agg`.

```
(create-instance 'MOVE-OR-GROWER Inter:Move-Grow-Interactor
   (:start-event '(:leftdown :rightdown))  ; either left or right
   (:grow-p (o-formula (eq :rightdown (gvl :start-char)))) ; grow if right button█
   (:line-p (o-formula (is-a-p (gvl :first-obj-over) opal:line)))
   (:feedback-obj (o-formula
   (if (gvl :line-p)
       MY-LINE-FEEDBACK-OBJ
       MY-RECTANGLE-FEEDBACK-OBJ)))
   (:start-where '(:element-of ,ALL-OBJECT-AGG))
   (:window MYWINDOW))
```

6.30 Multiple Windows

Interactors can be made to work over multiple windows. The `:window` slot of an interactor can contain a single window (the normal case), a list of windows, or T which means all Interactor windows (this is rather inefficient). If one of the last two options is used, then the interactor will operate over all the specified windows. This means that as the interactor is running, mouse movement events are processed for all windows that are referenced. Also, when the last of the windows referenced is deleted, then the interactor is automatically destroyed.

This is mainly useful if you want to have an object move among various windows. If you want an object to track the mouse as it changes windows, however, you have to explicitly change the aggregate that the object is in as it follows the mouse, since each window has a single top-level aggregate and aggregates cannot be connected to multiple windows. You will probably need a custom `:running-action` routine to do this (see section [customroutines], page 312). This is true of the feedback object as well as the main object.

You can look at the demonstration program `demo-multiwin.lisp` to see how this might be done.

6.31 Wait-Interaction-Complete

Interactors supplies a pair of functions which can be used to suspend Lisp processing while waiting for the user to complete an action. It is a little complicated to do this at the Interactors level, but there is a convenient function for Gilt-created dialog boxes called `gilt:Show-In-Window-And-Wait` (see the Gilt chapter). Also, `garnet-gadgets:display-error-and-wait` and `garnet-gadgets:display-query-and-wait` can be used to pop up message windows and wait for the user's response (see the `error-gadget` in the Gadgets chapter).

For other applications, you can call:

`inter:Wait-Interaction-Complete &optional` *window-to-raise*`[function], page 90`█

which does not return until an interactor executes:

`inter:Interaction-Complete &optional` *val*`[function], page 90`

If a *val* is supplied, then this is returned as the value of `Inter:Wait-Interaction-Complete`. The *window-to-raise* parameter is provided to avoid a race condition that occurs

when you call update on a window and immediately call `wait-interaction-complete`. If you have problems with this function, then try supplying your window as the optional argument. `Wait-interaction-complete` will then raise your window to the top and update it for you.

Typically, `Inter:Interaction-Complete` will be called in the final-function of the interactor (or the selection-function of the gadget) that should cause a value to be returned, such as a value associated with the "OK" button of a dialog box. Note that you must use some other mechanism of interactors to make sure that only the interactors you care about are executable; `Wait-Interaction-Complete` allows *all* interactors in *all* windows to run.

6.32 Useful Procedures

The text interactor beeps (makes a sound) when you hit an illegal character. The function to cause the sound is exported as

> `inter:Beep[function], page 90`

which can be used anywhere in application code also.

The Interactors package exports the function

> `inter:Warp-Pointer window x y[function], page 90`

which moves the position of the mouse cursor to the specified point in the specified window. The result is the same as if the user had moved the mouse to position `<x,y>`.

6.33 Custom Action Routines

We have found that the interactors supply sufficient flexibility to support almost all kinds of interactive behaviors. There are many parameters that you can set in each kind of interactor, and you can use formulas to determine values for these dynamically. The `final-function` can be used for application notification if necessary.

However, sometimes a programmer may find that special actions are required for one or more of the action routines. In this case, it is easy to override the default behavior and supply your own functions. As described in section [Slots of All Interactors], page 250, the action routines are:

> :stop-action
>
> :start-action
>
> :running-action
>
> :abort-action
>
> :outside-action
>
> :back-inside-action

Each of the interactor types has its own functions supplied in each of these slots.

If you want the default behavior *in addition to* your own custom behavior, then you can use the KR function `Call-Prototype-Method` to call the standard function from your function. The parameters are the same as for your function.

For example, the `:running-action` for Move-Grow interactors is defined (in section [movegrowcustomaction], page 314) as:

> `(lambda (an-interactor object-being-changed new-points))`

so to create an interactor with a custom action as well as the default action, you might do:

```
(create-instance NIL Inter:Move-Grow-Interactor
   ... the other usual slots
   (:running-action
    #'(lambda (an-interactor object-being-changed new-points)
  (call-prototype-method an-interactor object-being-changed new-points)
  (Do-My-Custom-Stuff))))
```

The parameters to all the action procedures for all the interactor types are defined in the following sections.

6.33.1 Menu Action Routines

The parameters to the action routines of menu interactors are:

`:Start-action`

> `(lambda (an-interactor first-object-under-mouse))`

> Note that `:running-action` is not called until the mouse is moved to a different object (it is not called on this first object which is passed as `first-object-under-mouse`).

`:Running-action`

> `(lambda (an-interactor prev-obj-over new-obj-over))`

> This is called once each time the object under the mouse changes (not each time the mouse moves).

`:Outside-action`

> `(lambda (an-interactor outside-control prev-obj-over))`

> This is called when the mouse moves out of the entire menu. `Outside-Control` is simply the value of the `:outside` slot.

`:Back-inside-action`

> `(lambda (an-interactor outside-control prev-obj-over new-obj-over))`▮

> Called when the mouse was outside all items and then moved back inside. `Prev-obj-over` is the last object the mouse was over before it went outside. This is used to remove feedback from it if `:outside` is `:last`.

`:Stop-action`

> `(lambda (an-interactor final-obj-over))`

> The interactor guarantees that `:running-action` *has* been called on `final-obj-over` before the `:stop-action` procedure is called.

`:Abort-action`

> `(lambda (an-interactor last-obj-over))`

6.33.2 Button Action Routines

The parameters to the action routines of button interactors are:

> :Start-action

> `(lambda (an-interactor object-under-mouse))`

Note that `back-inside-action` is not called this first time.

:running-action This is not used by this interactor. `:Back-inside-action` and `:Outside-action` are used instead.

:back-inside-action

```
(lambda (an-interactor new-obj-over))
```

This is called each time the mouse comes back to the original object.

:outside-action

```
(lambda (an-interactor last-obj-over))
```

This is called if the mouse moves outside of `:running-where` before stop-event. The default `:running-where` is `'(:in *)` which means in the object that the interactor started on.

:stop-action

```
(lambda (an-interactor final-obj-over))
```

:abort-action

```
(lambda (an-interactor obj-over))
```

Obj-over will be the object originally pressed on, or `nil` if outside when aborted.

6.33.3 Move-Grow Action Routines

The parameters to the action routines of move-grow interactors are:

:start-action

```
(lambda (an-interactor object-being-changed first-points))
```

`First-points` is a list of the original left, top, width and height for the object, or the original X1, Y1, X2, Y2, depending on the setting of `:line-p`. The `object-being-changed` is the actual object to change, not the feedback object. Note that `:running-action` is not called on this first point; it will not be called until the mouse moves to a new point.

:running-action

```
(lambda (an-interactor object-being-changed new-points))
```

The `object-being-changed` is the actual object to change, not the feedback object.

:outside-action

```
(lambda (an-interactor outside-control object-being-changed))
```

The `object-being-changed` is the actual object to change, not the feedback object. `Outside-control` is set with the value of `:outside`.

:back-inside-action

```
(lambda (an-interactor outside-control object-being-changed new-inside-point
```

The `object-being-changed` is the actual object to change, not the feedback object. Note that the running-action procedure is not called on the point passed to this procedure.

:stop-action

```
(lambda (an-interactor object-being-changed final-points))
```

The `object-being-changed` is the actual object to change, not the feedback object. `:Running-action` was not necessarily called on the point passed to this procedure.

:abort-action

```
(lambda (an-interactor object-being-changed))
```

The `object-being-changed` is the actual object to change, not the feedback object.

6.33.4 Two-Point Action Routines

The parameters to the action routines of two-point interactors are:

:start-action

```
(lambda (an-interactor first-points))
```

The `first-points` is a list of the initial box or 2 points for the object (the form is determined by the `:line-p` parameter). If `:abort-if-too-small` is non-NIL, then `first-points` will be `nil`. Otherwise, the width and height of the object will be the `:min-width` and `:min-height` or 0 if there are no minimums. Note that `:running-action` is not called on this first point; it will not be called until the mouse moves to a new point.

:running-action

```
(lambda (an-interactor new-points))
```

`New-points` may be `nil` if `:abort-if-too-small` and the size is too small.

:outside-action

```
(lambda (an-interactor outside-control))
```

`Outside-control` is set with the value of `:outside`.

:back-inside-action

```
(lambda (an-interactor outside-control new-inside-points))
```

Note that the running-action procedure is not called on the point passed to this procedure. `New-inside-points` may be `nil` if `:abort-if-too-small` is non-NIL.

:stop-action

```
(lambda (an-interactor final-points))
```

`:Running-action` was not necessarily called on the point passed to this procedure. `Final-points` may be `nil` if `:abort-if-too-small` is non-NIL.

:abort-action

```
(lambda (an-interactor))
```

6.33.5 Angle Action Routines

In addition to the standard measure of the angle, the procedures below also provide an incremental measurement of the difference between the current and last values. This might be used if you just want to have the user give circular gestures to have something rotated. Then, you would just want to know the angle differences. An example of this is in `demo-angle.lisp`.

The parameters to the action routines of angle interactors are:

:Start-action -

    (lambda (an-interactor object-being-rotated first-angle))

The `first-angle` is the angle from directly to the right of the `:center-of-rotation` that the mouse presses. This angle is in radians. The `object-being-rotated` is the actual object to move, not the feedback object. Note that `:running-action` is not called on `first-angle`; it will not be called until the mouse moves to a new angle.

:Running-action -

    (lambda (an-interactor object-being-rotated new-angle angle-delta))▮

The `object-being-rotated` is the actual object to move, not the feedback object. `Angle-delta` is the difference between the current angle and the last angle. It will either be positive or negative, with positive being counter-clockwise. Note that it is always ambiguous which way the mouse is rotating from sampled points, and the system does not *yet* implement any hysteresis, so if the user rotates the mouse swiftly (or too close around the center point), the delta may oscillate between positive and negative values, since it will guess wrong about which way the user is going. *In the future, this could be fixed by keeping a history of the last few points and assuming the user is going in the same direction as previously.*

:Outside-action -

    (lambda (an-interactor outside-control object-being-rotated))

The `object-being-rotated` is the actual object to move, not the feedback object. `Outside-control` is set with the value of `:outside`.

:Back-inside-action -

    (lambda (an-interactor outside-control object-being-rotated new-angle))▮

The `object-being-rotated` is the actual object to move, not the feedback object. Note that the running-action procedure is not called on the point passed to this procedure. There is no `angle-delta` since it would be zero if `:outside-control` was NIL and it would probably be inaccurate for `:last` anyway.

:Stop-action -

    (lambda (an-interactor object-being-rotated final-angle angle-delta))▮

The `object-being-rotated` is the actual object to move, not the feedback object. `:Running-action` was not necessarily called on the angle passed to this procedure. `Angle-delta` is the difference from the last call to `:running-action`.

:Abort-action -

    (lambda (an-interactor object-being-rotated))

The `object-being-rotated` is the actual object to move, not the feedback object.

6.33.6 Text Action Routines

The parameters to the action routines of text interactors are:

> `:Start-action` -
>> `(lambda (an-interactor new-obj-over start-event))`

New-Obj-over is the object to edit, either `:obj-to-change` if it is supplied, or if `:obj-to-change` is `nil`, then the object returned from `:start-where`. The definition of events is in section ⟨undefined⟩ [events], page ⟨undefined⟩.

> `:Running-action` -
>> `(lambda (an-interactor obj-over event))`

> `:Outside-action` -
>> `(lambda (an-interactor obj-over))`

Often, `:running-where` will be T so that this is never called.

> `:Back-Inside-action` -
>> `(lambda (an-interactor obj-over event))`

> `:Stop-action` -
>> `(lambda (an-interactor obj-over stop-event))`

> `:Abort-action` -
>> `(lambda (an-interactor obj-over abort-event))`

6.33.7 Gesture Action Routines

The parameters to the action routines of gesture interactors are:

> `:Start-action` -
>> `(lambda (an-interactor object-under-mouse point))`

The point is the first point of the gesture.

> `:Running-action` -
>> `(lambda (an-interactor new-obj-over point))`

> `:Outside-action` -
>> `(lambda (an-interactor prev-obj-over))`

This beeps and erases the trace if `show-trace` is non-NIL. It also sets `:went-outside` to T.

> `:Back-inside-action` -
>> `(lambda (an-interactor new-obj-over))`

This currently does nothing.

> `:Stop-action` -
>> `(lambda (an-interactor final-obj-over point))`

`:Running-action` was not necessarily called on the point passed to this procedure, so it is added to `*points*`. This procedure calls `gest-classify` with the points in the trace, `*points*`, and the classifier given by `:classifier`.

> `:Abort-action` -
>> `(lambda (an-interactor))`

This erases the trace if `:show-trace` is non-NIL and `:went-outside` is `nil`.

6.33.8 Animation Action Routines

The `animator-interactor` does not use these action slots. All of the work is done by the function supplied in the `:timer-handler` slot.

6.34 Debugging

There are a number of useful functions that help the programmer debug interactor code. Since these are most useful in conjunction with the tools that help debug KR structures and Opal graphical objects, all of these are described in a separate Garnet Debugging chapter.

In summary, the functions provided include:

Interactors are KR objects so they can be printed using `kr:ps` and `hemlock-schemas`.

The `Inter:Trace-Inter` routine is useful for turning on and off tracing output that tells what interactors are running. Type `(describe 'inter:trace-inter)` for a description. *This function is only available when the* `garnet-debug` *compiling switch is on (the default).*

`(garnet-debug:ident)` will tell the name of the next event (keyboard key or mouse button) you hit.

`(garnet-debug:look-inter &optional parameter)` describes the active interactors, or a particular interactor, or the interactors that affect a particular graphic object.

`(inter:Print-Inter-Levels)` will print the names of all of the active interactors in all priority levels.

`(inter:Print-Inter-Windows)` will print the names of all the interactor windows, and `(garnet-debug:Windows)` will print all Opal and Interactor windows.

Destroying the interactor windows will normally get rid of interactors. You can use `(opal:clean-up :opal)` to delete all interactor windows.

If for some reason, an interactor is not deleted (for example, because it is not attached to a window), then

> inter:Reset-Inter-Levels &optional *level*[function], page 90

will remove *all* the existing interactors by simply resetting the queues (it does not destroy the existing interactors, but they will never be executed). If a level is specified, then only interactors on that level are destroyed. If level is `nil` (the default), then all levels are reset. This procedure should not be used in applications, only for debugging. It is pretty drastic.

===
@section Issues
@node Issues

How handle sliding out of a menu and having a sub-menu appear?

Should we increase the number of interactors and decrease the number of parameters to each?

What other specific interactors are needed?

Need trill button also?

Can we eliminate some of these interactors if we make interactors simpler?
For example, use a timer interactor and a regular button interactor
together to make a trill.

Can Menu and Button interactors be combined?

How to make the feedback object have the same properties as the real
object, e.g., with respect to constraints on movement (grids, only in X,
etc.). Want to be able to get them dynamically from the object and/or from
the particular interactor or parameters to the interactor (e.g., when press
here, only move in X) Solutions: extra parameters w/filters passed to the
interactors, have a "move-object" function (like create-function in twop)
that is passed either the feedback-obj or the real object (but this is
similar to the running-action and stop-action), somehow copy the
constraints into the feedback object (may require creating a new object
and/or new constraint objects), or leave as is so user defines own
running-action and stop-action procedures.
===

⟨undefined⟩ [References], page ⟨undefined⟩,

7 Aggregadgets, Aggrelists & Aggregraphs

Andrew Mickish, Roger B. Dannenberg, Philippe Marchal, David Kosbie, A. Bryan Loyall
10 May 2020

7.1 Abstract

Aggregadgets and aggrelists are objects used to define natural hierarchies of other objects in the Garnet system. They allow the interface designer to group graphical objects and associated behaviors into a single prototype object by declaring the structure of the components. Aggrelists are particularly useful in the creation of menu-type objects, whose components are a sequence of similar items corresponding to a list of elements. Aggrelists will automatically maintain the layout of the graphical list of objects. Aggregraphs are similarly used to create and maintain graph structures.

7.2 Aggregadgets

7.3 Accessing Aggregadgets and Aggrelists

The aggregadgets and aggrelists files are automatically loaded when the file `garnet-loader.lisp` is used to load Garnet. The `garnet-loader` file uses one loader file for both aggregadgets and aggrelists called `aggregadgets-loader.lisp`. Loading this file causes the KR, Opal, and Interactors files to be loaded also.

Aggregadgets and aggrelists reside in the `Opal` package. All identifiers in this chapter are exported from the `Opal` package unless another package name is explicitly given. These identifiers can be referenced by using the `opal` prefix, e.g. `opal:aggregadget`; the package name may be dropped if the line `(use-package "OPAL")` is executed before referring to any object in that package.

7.4 Aggregadgets

During the construction of a complicated Garnet interface, the designer will frequently be required to arrange sets of objects into groups that are easy to manipulate. These sets may have intricate dependencies among the objects, or possess a hierarchical structure that suggests a further subgrouping of the individual objects. Interactors may also be associated with the objects that should intuitively be defined along with the objects themselves.

Aggregadgets provide the designer with a straightforward method for the definition and use of sets of Garnet objects and interactors. When an aggregadget is supplied with a list of object definitions, Garnet will internally create instances of those objects and add them to the aggregadget as components. If the objects are given names, Garnet will create slots in the aggregadget which point to the objects, granting easy access to the components. Interactors that manipulate the components of the aggregadget may be similarly defined.

By creating instances of aggregadgets, the designer actually groups the objects and interactors under a single prototype (class) name. The defined prototype may be used repeatedly to create more instances of the defined group. To illustrate this feature of aggregadgets, consider the schemata shown below:

```
(create-instance 'MY-GROUP opal:aggregadget
```

```
    (:parts
            ...)         ; some group of graphical objects
    (:interactors
            ...))        ; some group of interactors

  (create-instance 'GROUP-1 MY-GROUP)

  (create-instance 'GROUP-2 MY-GROUP
    ...)                 ; definition of more slots
```

The schema MY-GROUP defines a set of associated graphical objects and interactors using an instance of the `opal:aggregadget` object. The schemata `group-1` and `group-2` are instances of the `my-group` prototype which inherit all of the parts and behaviors defined in the prototype. The `group-2` schema additionally defines new slots in the aggregadget for some special purpose.

7.4.1 How to Use Aggregadgets

In order to group a set of objects together as components of an aggregadget, the designer must define the objects in the `:parts` slot of the aggregadget.

The syntax of the `:parts` slot is a backquoted list of lists, where each inner list defines one component of the aggregadget. The definition of each component includes a keyword that will be used as a name for that part (or `nil` if the part is to be unnamed), the prototype of that part, and a set of slot definitions that customize the component from the prototype.

The aggregadget will internally convert this list of parts into components of the aggregadget, with each part named by the keyword provided (or unnamed, if the keyword is `nil`).

Everything inside the backquote that should be evaluated immediately must be preceded by a comma. Usually the following will need commas: the prototype of the component, variable names, calls to `formula` and `o-formula`, etc.

After an aggregadget is created, the designer should not refer to the `:parts` slot. Each component may be accessed by name as a slot of the aggregadget. Additionally, all components are listed in the `:components` slot just as in aggregates. in display order, that is, *from back to front.*

A short example of an aggregadget definition is shown in figure [check-mark], page 322, and the picture of this aggregadget is in figure [simple-expl-pict-ref], page 323.

```
(create-instance 'CHECK-MARK opal:aggregadget
   (:parts
    '((:left-line ,opal:line
                   (:x1 70)
                   (:y1 45)
                   (:x2 95)
                   (:y2 70))
      (:right-line ,opal:line
                   (:x1 95)
                   (:y1 70)
                   (:x2 120)
                   (:y2 30)))))
```

Figure 7.1: A simple CHECK-MARK aggregadget.
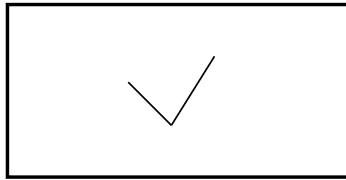
Figure 7.2: The picture of the CHECK-MARK aggregadget.

Of course, the designer may define other slots in the aggregadget besides the `:parts` slot. One convenient programming style involves the definition of several slots in the top-level aggregadget (such as `:left`, `:top`, etc.) with formulas in several components that refer to these values, thereby allowing a change in one top-level slot to propagate to all dependent

slots in the components. Slots of components may also contain formulas that refer to other components (see section [agg-dependencies], page 327).

7.4.2 Named Components

When keywords are given in the :parts list that correspond to each component, those keywords are used as names for the components. In figure [check-mark], page 322, the names are :left-line and :right-line. Since these names were supplied, the slots :left-line and :right-line are set in the CHECK-MARK aggregadget with the components themselves as values. That is, (gv CHECK-MARK :left-line) yields the actual component that was created from the :parts description.

The slot :known-as in the component is also set with the name of the component. In the example above, (gv CHECK-MARK :left-line :known-as) yields :left-line. Another way to look at these slots and objects is shown in figures [agg-ps-ref], page 324, and [part-ps-ref], page 325.

When adding a new component to an aggregadget, you can set the :known-as slot of the component with a keyword name, which will be used in the top-level aggregadget as a slot name that points directly to the new component. The example at the end of section [constants-and-aggregadgets], page 325, illustrates the idea of setting the :known-as slot.

```
lisp> (ps CHECK-MARK)

#k<CHECK-MARK>
  :RIGHT-LINE = #k<KR-DEBUG:RIGHT-LINE-226>
  :LEFT-LINE = #k<KR-DEBUG:LEFT-LINE-220>
  :COMPONENTS = #k<KR-DEBUG:LEFT-LINE-220> #k<KR-DEBUG:RIGHT-LINE-226>
  ...
  :PARTS = ((:LEFT-LINE #k<OPAL:LINE>
                        (:X1 70) (:Y1 45) (:X2 95) (:Y2 70))
            (:RIGHT-LINE #k<OPAL:LINE>
                        (:X1 95) (:Y1 70) (:X2 120) (:Y2 30)))
  ...
  :IS-A = #k<OPAL:AGGREGADGET>

NIL
lisp>
```

Figure 7.3: The printout of the CHECK-MARK aggregadget.

```
[lisp>] (ps (gv CHECK-MARK :right-line))

#k<KR-DEBUG:RIGHT-LINE-226>
  :PARENT =  #k<CHECK-MARK>
  :KNOWN-AS =  :RIGHT-LINE
  ...
  :Y2 =  30
  :X2 =  120
  :Y1 =  70
  :X1 =  95
  :IS-A =  #k<OPAL:LINE>

NIL
[lisp>]
```

Figure 7.4: The `:right-line` component of CHECK-MARK.

As shown in figure [agg-ps-ref], page 324, CHECK-MARK has two components: RIGHT-LINE-226 which is a line created according to the definition of `:right-line` in the `:parts` slot of the CHECK-MARK aggregadget, and LEFT-LINE-220 corresponding to the definition of the `:left-line` part. The CHECK-MARK aggregadget also has two slots, `:right-line` and `:left-line`, whose values are the corresponding components.

7.4.3 Destroying Aggregadgets

opal:Destroy *gadget*⟨undefined⟩ [method], page ⟨undefined⟩,

opal:Destroy-Me *gadget*⟨undefined⟩ [method], page ⟨undefined⟩,

The `destroy` method destroys an aggregadget or aggrelist and its instances. To destroy a gadget means to destroy its interactors, components, and item-prototype-object as well as the gadget schema itself. The `destroy-me` method for aggregadgets and aggrelists destroys the prototype but not its instances. *Note:* users of gadgets should call `destroy`; implementors of subclasses should override `destroy-me`.

7.4.4 Constants and Aggregadgets

The ability to define constant slots is an advanced feature of Garnet that is discussed in detail in the KR chapter. However, the aggregadgets use some of the features of constant slots by default.

All aggregadgets created with an initial `:parts` list have constant `:components`. That is, after the aggregadget has been created with all of its parts, the `:components` slot becomes constant automatically, and the components of the aggregadget are not normally modifiable. Also, the `:known-as` slot of each part and the slot in the aggregadget corresponding to the name of each part is constant. By declaring these slots constant, Garnet is able to automatically get rid of the greatest number of formulas possible, thereby freeing up memory for other objects.

For example, given the following instance of an `aggregadget`,

```
(create-instance 'MY-AGG opal:aggregadget
```

```
    (:parts
     '((:obj1 ,opal:rectangle
             (:left 20) (:top 40))
        (:obj2 ,opal:circle
             (:left 50) (:top 10)))))
```

the slots `:components`, `:obj1`, and `:obj2` will be constant in MY-AGG. The result is that you cannot remove components or add new components to this aggregadget without disabling the constant mechanism.

If you really want to add another component to the aggregadget, you could use the macro `with-constants-disabled`, which is described in the KR chapter:

```
    (with-constants-disabled
      (opal:add-component MY-AGG (create-instance NIL opal:roundtangle
                                   (:known-as :obj3)  ; will become a constant slot
                                   (:left 40) (:top 20))))
```

Adding components to a constant aggregadget is discouraged because the aggregadget's dimension formulas that were already thrown away (if they were evaluated) will not be updated with the dimensions of the new components. That is, if OBJ3 in the example above is outside of the original bounding box of MY-AGG (calculated by the formulas in MY-AGG's `:left`, `:top`, `:width`, and `:height` slots), then Opal will fail to display the new component correctly because it only updates the area enclosed by MY-AGG's bounding box.

A better solution than forcibly adding components is to create a non-constant aggregadget to begin with. Since only aggregadgets that are created with a `:parts` slot are constant, you should start with an aggregadget without a `:parts` list, and add your components using `add-component`. Thus, the better way to build the aggregadet above is:

```
    (create-instance 'MY-AGG opal:aggregadget)
    (opal:add-components MY-AGG (create-instance NIL opal:rectangle
                                   (:known-as :obj1)
                                   (:left 20) (:top 40))
                                (create-instance NIL opal:circle
                                   (:known-as :obj2)
                                   (:left 50) (:top 10)))
    ; Then later...
    (opal:add-component MY-AGG (create-instance NIL opal:roundtangle
                                   (:known-as :obj3)
                                   (:left 40) (:top 20)))
```

Note that you will have to supply your own `:known-as` slots in the components if you want the aggregadet to have slots referring to those components.

7.4.5 Implementation of Aggregadgets

An aggregadget is an instance of the prototype `opal:aggregate`, with an initialize method that interprets the `:parts` slot and provides other functions. This initialize method performs the following tasks:

   an instance of every part is created,

all these instances are added (with `add-component`) as the components of the aggregadget,

for each part, a slot is created in the aggregate. The name of this slot is the name of the part, and its value is the instance of the corresponding part.

The slot `:known-as` in the part is set with the part's name.

In some cases (described in detail later), some or all of the structure of the prototype aggregadget is inherited by the new instance.

7.4.6 Dependencies Among Components

Aggregadgets are designed to facilitate the definition of dependencies among their components. When a slot of one component depends on the value of a slot in another component of the same aggregadget, that dependency is expressed using a formula.

The aggregadget is considered the parent of the components, and the components are all siblings within the aggregadget. Thus, the `:parent` slot of each component can be used to travel up the hierarchy, and the slot names of the aggregadget and its components can be used to travel down.

Consider the following modification to the CHECK-MARK schema defined in section [what-an-agg], page 321. In figure [check-mark], page 322, the `:x1` and `:y1` slots of the `:right-line` object are the same as the `:x2` and `:y2` slots of the `:left-line` object so that the two lines meet at a common point. Rather than explicitly repeating these coordinates in the `:right-line` object, dependencies can be defined in the `:right-line` object that cause its origin to always be the terminus of the `:left-line`. Figure [modified-check-mark], page 327, shows the definition of this modified schema.

```
(create-instance 'MODIFIED-CHECK-MARK opal:aggregadget
  (:parts
   '((:left-line ,opal:line
                 (:x1 70)
                 (:y1 45)
                 (:x2 95)
                 (:y2 70))
     (:right-line ,opal:line
                 (:x1 ,(o-formula (gvl :parent :left-line :x2)))
                 (:y1 ,(o-formula (gvl :parent :left-line :y2)))
                 (:x2 120)
                 (:y2 30)))))
```

Figure 7.5: A modified CHECK-MARK schema.

Commas must precede the calls to `o-formula` and the references to the `opal:line` prototype because these items must be evaluated immediately. Without commas, the `o-formula` call, for example, would be interpreted as a quoted list due to the backquoted `:parts` list.

The macro `gvl-sibling` is provided to abbreviate references between the sibling components of an aggregadget:

opal:gvl-sibling *sibling-name* **&rest** *slots*   [Macro]

For example, the `:x1` slot of the `:right-line` object in figure [modified-check-mark], page 327, may be given the equivalent value

```
,(o-formula (opal:gvl-sibling :left-line :x2))
```

7.4.7 Multi-level Aggregadgets

Aggregadgets can be used to define more complicated objects with a multi-level hierarchical
structure. Consider the picture of a check-box shown in figure [check-box-pict-ref], page 329.
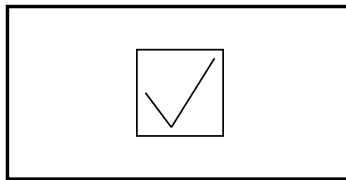
Figure 7.6: A picture of a check-box.

The check-box can be considered a hierarchy of objects: the CHECK-MARK object defined in figure [check-mark], page 322, and a box. This hierarchy is illustrated in figure [check-box-hier-ref], page 330.

```
                    check-box
                     /  \
                    /    \
                  box   check-mark
                         /  \
                        /    \
                left-line    right-line
```
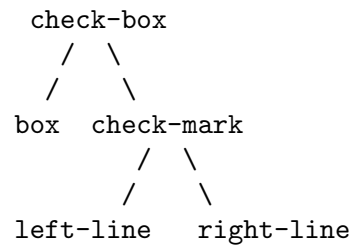
Figure 7.7: The hierarchical structure of a check-box.

The CHECK-BOX hierarchy is implemented through aggregadgets in figure [check-box-def-ref], page 330. Although the CHECK-BOX schema defines the `:box` component explicitly, the details of the `:mark` object have been defined elsewhere in the CHECK-MARK schema (see figure [check-mark], page 322). The aggregadget definition for the CHECK-MARK part could have been written out explicitly, as in the more complicated CHECK-BOX schema of figure ⟨undefined⟩ [custom-check-box1], page ⟨undefined⟩. However, the CHECK-BOX definition presented here uses a modular approach that allows the reuse of the CHECK-MARK schema in other applications.

```
(create-instance 'CHECK-BOX opal:aggregadget
   (:parts
    `((:box ,opal:rectangle
            (:left 75)
            (:top 25)
            (:width 50)
            (:height 50))
      (:mark ,CHECK-MARK))))
```

Figure 7.8: The definition of a check-box.

See section ⟨undefined⟩ [Custom-check-box2], page ⟨undefined⟩, for another example of a modularized multi-level aggregadget, and see section [instances-sec], page 343, for information about inheriting structure from other multi-level aggregadgets.

7.4.8 Nested Part Expressions for Aggregadgets

Recall that parts are specified in a `:parts` slot and that the syntax for a part is

(*name prototype slot value+[*]*)

where *name* is either a keyword or `nil`, *prototype* is a prototype for the part, and *slots* is a list of local slot definitions. If *prototype* is an aggregadget, then *slots* may contain another parts slot; thus, an entire aggregadget tree can be specified by nested `:parts` slots.

For example, figure [x-box-fig], page 331, implements a box containing an X. Notice how the `:mark` part of X-BOX is an aggregadget containing its own parts.
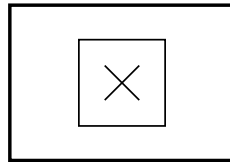
```
;;; compute vertical position in :box according to a proportion
(defun vert-prop (frac)
  (+ (gvl :parent :parent :box :top)
     (round (* (gvl :parent :parent :box :height)
        frac))))


;;; compute horizontal position in :box according to a proportion
(defun horiz-prop (frac)
  (+ (gvl :parent :parent :box :left)
     (round (* (gvl :parent :parent :box :width)
```

7.4.9 Creating a Part with a Function

Instead of defining a prototype as a part, the designer may specify a function which will be called in order to generate the part. This feature can be useful when you plan to create several instances of an aggregadget that are similar, but with different objects as parts. For example, the aggregadgets in figure [single-part-fn], page 333, all have the same prototype.
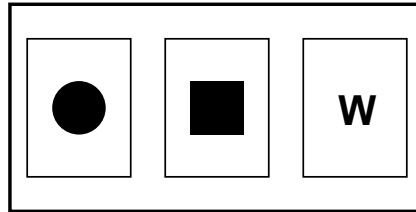
Figure 7.10: Aggregadgets that generate a part through a function.

The syntax for generating a part with a function is to specify a function within the **:parts** list where a prototype for the part would usually go. The function must take one argument, which is the aggregadget whose part is being generated. Slots of the aggregadget may be accessed at any time inside the function.

The purpose of the function is to return an object that will be a component of the aggregadget. You should **not** add the part to the aggregadget yourself in the function. However, you must be careful to always return an object that can be used directly as a component. For example, the `opal:circle` object would not be a suitable object to return, since it is the prototype of many other objects. Instead, you would return an *instance* of `opal:circle`.

Additionally, you must be careful to consider the case where the object to be used has already been used before. That is, if you wanted the function to return a rectangle more than once, the function must be smart enough to return a particular rectangle the first time, and return a different rectangle the second time and every time thereafter. Usually it is sufficient to look at the `:parent` slot of the object to check if it is already part of another aggregadget. The following code, which generates the figure in [single-part-fn], page 333, takes this multiple usage of an object into consideration.

```
(defun Get-Label (agg)
  (let* ((item (gv agg :item))
           ;; Item may be an object or a string
   (new-label (if (schema-p item)
 (if (gv item :parent)
     ;; The item has been used already --
     ;; Use it as a prototype
     (create-instance NIL item)
     ;; Use the item itself
     item)
         (create-instance NIL opal:text
   (:string item)
   (:font (opal:get-standard-font
                                    :sans-serif :bold :very-large))))))
    (s-value new-label :left (o-formula (opal:gv-center-x-is-center-of (gvl :parent)))
    (s-value new-label :top (o-formula (opal:gv-center-y-is-center-of (gvl :parent))))
    new-label))

(create-instance 'AGG-PROTO opal:aggregadget
  (:item "Text")
  (:top 20) (:width 60) (:height 80)
  (:parts
   '((:frame ,opal:rectangle
       (:left ,(o-formula (gvl :parent :left)))
       (:top ,(o-formula (gvl :parent :top)))
       (:width ,(o-formula (gvl :parent :width)))
       (:height ,(o-formula (gvl :parent :height))))
       (:label ,#'Get-Label))))

(create-instance 'CIRCLE-LABEL opal:circle
  (:width 30) (:height 30)
  (:line-style NIL)
  (:filling-style opal:black-fill))
```

```
(create-instance 'SQUARE-LABEL opal:rectangle
  (:width 30) (:height 30)
  (:line-style NIL)
  (:filling-style opal:black-fill))

(create-instance 'AGG1 AGG-PROTO
  (:left 10)
  (:item CIRCLE-LABEL))

(create-instance 'AGG2 AGG-PROTO
  (:left 90)
  (:item SQUARE-LABEL))

(create-instance 'AGG3 AGG-PROTO
  (:left 170)
  (:item "W"))
```

Some of the functionality provided by a part-generating function is overlapped by the customization syntax for aggregadget instances described in section [overriding-slots], page 345. For example, the labels in figure [single-part-fn], page 333, could have been customized from the prototype by supplying prototypes in the local :parts list of each instance. However, for some applications using aggrelists, this feature is indespensable (see section [multi-parts-fn], page 361).

7.4.10 Creating All of the Parts with a Function

As an alternative to supplying a list of component definitions in the :parts slot, the designer may instead specify a function which will generate the parts of the aggregadget during its initialization. This feature is useful when the components of the aggregadget are related in some respect that is easily described by a function procedure, as in figure [multi-line-pict-ref], page 336.
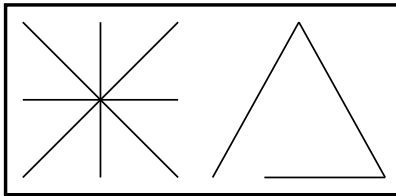
Figure 7.11: The multi-line picture.

This feature of aggregadgets is not usually used since, in most cases, aggrelists supply the same functionality. When all the components of an aggregadget are instances of the same prototype, the designer should consider implementing an itemized aggrelist, discussed in chapter [aggrelists], page 352.

The function may be specified in the `:parts` slot as either a previously defined function or a lambda expression. The function must take one parameter: the aggregadget whose parts are being created. The function must return a list of the created parts (e.g., a list of instances of `opal:line`) and, optionally, a list of the names of the parts. If supplied, the names must be keywords which will become slot names for the aggregadget, providing access to the individual components (see section [agg-dependencies], page 327). (*Note:* The standard lisp function `values` may be used to return two arguments from the generating function.)

Figure [multi-line1-ref], page 337, shows how to create an aggregadget made of multiple lines, with the end-points of the lines given in the special slot `:line-end-points`. The figure creates the object on the left of figure [multi-line-pict-ref], page 336.

```
(create-instance 'MULTI-LINE opal:aggregadget
  (:parts
   '(,#'(lambda (self)
   (let ((lines NIL))
     (dolist (line-ends (gv self :lines-end-points))
       (setf lines (cons (create-instance NIL opal:line
   (:x1 (first line-ends))
   (:y1 (second line-ends))
   (:x2 (third line-ends))
   (:y2 (fourth line-ends)))
 lines)))
     (reverse lines)))))
   (:lines-end-points '((10 10 100 100)
(10 100 100 10)
(55 10 55 100)
(10 55 100 55))))
```

Figure 7.12: An aggregadget with a function to create the parts.

Figure [multi-line2-ref], page 338, shows how to create the same aggregadgets as in figure [multi-line-pict-ref], page 336, but with a separately defined function rather than a lambda expression. In addition, this function returns the list of the names of the parts. Two instances of the aggregadget are created, with only one of these instances having names for the lines.

```
(defun Make-Lines (lines-agg)
  (let ((lines NIL))
    (dolist (line-ends (gv lines-agg :lines-end-points))
      (setf lines (cons (create-instance NIL opal:line
  (:x1 (first line-ends))
  (:y1 (second line-ends))
          (:x2 (third line-ends))
  (:y2 (fourth line-ends)))
lines)))
    (values (reverse lines) (gv lines-agg :lines-names))))

(create-instance 'MY-MULTI-LINE1 opal:aggregadget
  (:parts '(,#'Make-Lines))
  (:lines-end-points '((10 10 100 100)
                       (10 100 100 10)
                       (55 10 55 100)
                       (10 55 100 55)))
  (:lines-names
   '(:down-diagonal :up-diagonal :vertical :horizontal)))

(create-instance 'MY-MULTI-LINE2 opal:aggregadget
  (:parts '(,#'Make-Lines))
  (:lines-end-points '((120 100 170 10)
                       (170 10 220 100)
                       (220 100 150 100))))
```

Figure 7.13: An aggregadget with a function to create named parts.

It should be noted that the use of a function to create parts is *not* inherited. If the `:parts` slot is omitted, then the actual parts (not the function that created the parts) are inherited from the prototype. It is possible to override the `:initialize` method to obtain a different instantiation convention, but probably it is simplest just always to specify the `:parts` slot indicating the function that creates parts.

7.5 Interactors in Aggregadgets

Interactors may be grouped in aggregadgets in precisely the same way that objects are grouped. The slot `:interactors` is analogous to the `:parts` slot, and may contain a list of interactor definitions that will be attached to the aggregadget.

As with the `:parts` slot, `:interactors` must contain a backquoted list of lists with commas preceding everything that should be evaluated immediately⟨undefined⟩ [dash], page ⟨undefined⟩prototypes, function calls, variable references, etc. The name of a function that generates a set of interactors can also be given with the same parameters and functionality as the `:parts` function described in section [run-time], page 335.

If a keyword is supplied as the name for an interactor, then a slot with that name will be automatically created in the aggregadget, and the value of that slot will be the interactor. For example, in figure [framed-text], page 340, a slot called `:text-inter` will be created in

the aggregadget to refer to the text interactor. The system will also add to the aggregadget
a `:behaviors` slot, containing a list of pointers to the interactors. This slot is analogous to
the `:components` slot for graphical objects.

Each interactor will be given a new `:operates-on` slot which is analogous to the `:parent`
slot for component objects. The `:operates-on` slot contains a pointer to the aggregadget
that the interactor belongs to. This slot should be used when referring to the aggregadget
from within interactors.

In order to activate any interactor in Garnet, its `:window` slot must contain a pointer to
the window in which the interactor operates. In most cases, the window for the interactor
will be found in the `:window` slot of the aggregadget, which is internally maintained by
aggregates. Hence, the following slot definition should be included in all interactors defined
in an aggregadget:

```
(:window ,(o-formula (gv-local :self :operates-on :window)))
```

Note: in this formula, `gv-local` is used to follow local links `:operates-on` and `:window`.
Using `gv-local` instead of `gv` or `gvl` when referring to these slots helps avoid accidental
references to these slots in the aggregdagets' prototype. Most values for the `:window` slots
of aggregadget interactors will resemble this formula.

The interactors are independent of the parts, and either feature may be used with or
without the other. When using both parts and interactors, any object may refer to any
other using the methods described in section [agg-dependencies], page 327.

Figure [framed-text], page 340, shows how to create a "framed-text" aggregadget that
allows the input and display of text. This aggregadget is made of two parts, a frame (a
rectangle) and a text object, and one interactor (a text-interactor). Figure [agg-inter-ps-
ref], page 341, is a partial printout of the FRAMED-TEXT aggregadget with its built-in
interactor, illustrating the slots created by the system. A picture of the aggregadget is
shown in figure [framed-text-pix], page 342.

```
(create-instance 'FRAMED-TEXT opal:aggregadget
   (:left 0)        ; Set these slots to determine
   (:top 0)         ; the position of the aggregadget.
   (:parts
    `((:frame ,opal:rectangle
          (:left ,(o-formula (gvl :parent :left)))
          (:top ,(o-formula (gvl :parent :top)))
          (:width ,(o-formula (+ (gvl :parent :text :width) 4)))
          (:height ,(o-formula (+ (gvl :parent :text :height) 4))))
      (:text ,opal:text
          (:left ,(o-formula (+ (gvl :parent :left) 2)))
          (:top ,(o-formula (+ (gvl :parent :top) 2)))
          (:cursor-index NIL)
          (:string ""))))
   (:interactors
     ; Press on the text object (inside the frame) to edit the string
    `((:text-inter ,inter:text-interactor
  (:window ,(o-formula (gv-local :self :operates-on :window)))
          (:feedback-obj NIL)
          (:start-where ,(o-formula
                           (list :in (gvl :operates-on :text))))
          (:abort-event #\control-\g)
          (:stop-event (:leftdown #\RETURN))))))
```

Figure 7.14: Definition of an aggregadget with a built-in interactor.

```
[lisp>] (ps FRAMED-TEXT)
#k<FRAMED-TEXT>
  ...
  :COMPONENTS =  #k<KR-DEBUG:FRAME-205> #k<KR-DEBUG:TEXT-207>
  :FRAME =  #k<KR-DEBUG:FRAME-205>
  :TEXT =  #k<KR-DEBUG:TEXT-207>
  :BEHAVIORS =  #k<KR-DEBUG:TEXT-INTER-214>
  :TEXT-INTER =  #k<KR-DEBUG:TEXT-INTER-214>
  ...
  :IS-A =  #k<OPAL:AGGREGADGET>

NIL
[lisp>] (ps (gv FRAMED-TEXT :text-inter))

#k<KR-DEBUG:TEXT-INTER-214>
  ...
  :OPERATES-ON =  #k<FRAMED-TEXT>
  ...
  :IS-A =  #k<INTERACTORS:TEXT-INTERACTOR>

NIL
[lisp>]
```

Figure 7.15: The printouts of an aggregadget and its attached interactor.

```
┌─────────────────────┐
│  ┌─────────────┐    │
│  │ Framed Text │    │
│  └─────────────┘    │
└─────────────────────┘
```
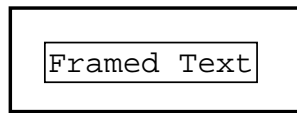
Figure 7.16: A picture of the FRAMED-TEXT aggregadget.

7.6 Instances of Aggregadgets

The preceding chapter discussed the use of the `:parts` slot to define the structure of new aggregadgets. Once an aggregadget is created, the structure will be inherited by instances. The `:parts` slot can be used to extend or override this default structure.

7.6.1 Default Instances of Aggregadgets

By default, when an instance of an aggregadget is created, an instance of each component and interactor is also created. Figure [instance-fig], page 344, illustrates an aggregadget on the left and its instance on the right. Notice that each object within the prototype aggregadget serves as a prototype for each corresponding object in the instance aggregadget. The structure of the instance aggregate matches the structure of the prototype, including "external" references to objects not in either aggregate, as illustrated by the reference from C to D. Since D is external to the aggregate, there is no D', and the reference to D is inherited by C'.
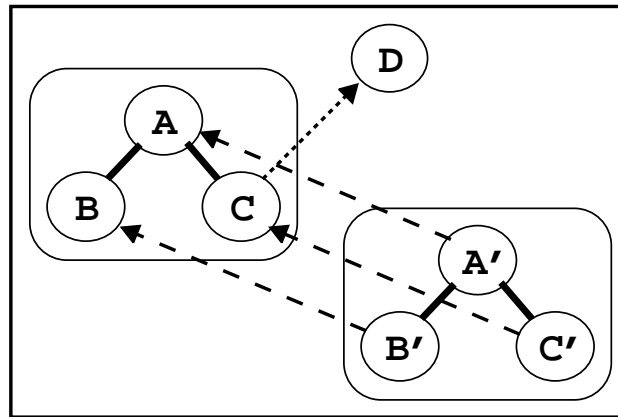
Figure 7.17: A prototype aggregate and one instance. The dashed lines go from instances to their prototypes, solid lines join children to parents, and the dotted line from C to D represents a formula dependence which is inherited by C'.

When creating instances, it is possible to override slots and parts of the prototype aggregadget, provided that these slots were not declared constant in the prototype.

7.6.2 Overriding Slots and Structure

Just as instances of KR objects can override slots with local values, aggregadgets can override slots or even entire parts (objects) with local values. The `:parts` and `:interactors` syntax is used to override details of an aggregadget when constructing an instance.

When creating an instance of an aggregadget that already has components, there are several variations of the `:parts` syntax that can be used to inherit components. As illustrated in these examples, if *any* parts are listed in a `:parts` list, then *all* parts should be listed. This is explained further in section [more-syntax-sec], page 350:

If the entire `:parts` slot is omitted, then the components are instantiated in the default manner described above. For example,

```
(create-instance 'NEW-X-BOX X-BOX (:left 100))
```

will instantiate the `:box` and `:mark` parts of `x-box` by default.

Any element in the list of parts may be a keyword rather than a list. The keyword must name a component of the prototype, and an instance of that component is created. Parts are always added in the order they are listed, regardless of their order in the prototype. For example:

```
(:parts '(:shadow :box :feedback))
```

Any element in the list of parts may be a list of the form (*name* `:omit`), where *name* is the name of a component in the prototype, and `:omit` indicates that an instance of that part is not included in the instance aggregadget. For example:

```
(:parts '((:shadow :omit)
  :box
  :feedback))
```

Any element in the list of parts may be a list of the form (*name* `:modify` *slots*), where *name* is the part name, `:modify` means to use the default prototype, and *slots* is a standard list of slot names and values which override slots inherited from the prototype. Only the changed slots need to be listed; the others are inherited from the prototype. [Note: this is different from the `:parts` slot, where you must list all the parts if you are changing any of them.] If the object is an aggregadget, then one of the slots may be a `:parts` list to further specify components. For example:

```
(:parts '((:shadow :modify (:offset 5))
  :box
  :feedback))
```

Any element of the list of parts may be a list of the form (*name* *prototype* *slots*), as described in section [parts-syntax-sec], page 321. This indicates that the part should be added to the instance aggregadget. If *name* names an existing component in the aggregadget, then the new part will override the part that would otherwise be inherited.

7.6.3 Simulated Multiple Inheritance

In some cases, it is desirable to inherit particular slots from a default prototype object, but to override the actual prototype. For example, one might want to change rectangles in a prototype into circles but still inherit the `:top` and `:left` slots. Alternatively, one might want to replace a number box with a dial but still inherit a `:color` slot from the prototype number box.

The `:parts` syntax has a special variation to accomplish this form of multiple inheritance. If the keyword `:inherit` occurs at the top level in the *slots* list, then the next element of *slots* must be a list of slot names. All the slots not mentioned in the `:inherit` clause are inherited from the new prototype (the circle in the example below). For example:

```
(:parts '((:shadow ,opal:circle
    (:offset 5)
    :inherit (:left :top :width :height :filling-style))
  :box
  :feedback))
```

7.6.4 Instance Examples

Figure [circle-x-box-fig], page 347, illustrates how to override and inherit parts from an aggregadget. The prototype aggregadget is the `x-box` aggregadget shown in figure [x-box-fig], page 331. In the instance named CIRCLE-X-BOX, a circle has been inserted between the box and the "X" mark, and the box has a gray fill.
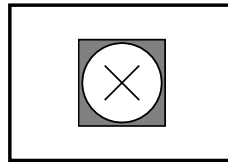
```
(create-instance 'CIRCLE-X-BOX X-BOX
   (:left 150)
   (:top 160)
   (:parts
    `((:box :modify (:filling-style ,opal:gray-fill))
      (:circle ,opal:circle
       (:left ,(o-formula (+ (gvl :parent :left) 2)))
       (:top ,(o-formula (+ (gvl :parent :top) 2)))
       (:width ,(o-formula (- (gvl :parent :width) 4)))
       (:height ,(o-formula (- (gvl :parent :height) 4)))
```

In figure [circle-box-fig], page 349, the CIRCLE-X-BOX aggregadget is further modified by replacing the "X" with a circle.
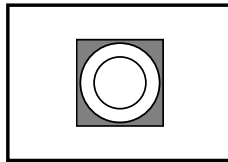
```
(defun circle-box-test ()
  (create-instance 'CIRCLE-BOX CIRCLE-X-BOX
   (:left 150)
   (:top 220)
   (:parts
    '(:box
       :circle
       (:mark :omit)
       (:inner-circle ,opal:circle
       (:left ,(o-formula (+ (gvl :parent :left) 10)))
```

7.6.5 More Syntax: Extending an Aggregadget

Normally, each part of a prototype should be explicitly mentioned in the :parts list. This is perhaps tedious, but it makes the code clear. There is one exception that is provided to make it simple to add things to existing prototypes.

If *none* of the parts of a prototype are mentioned in the parts list, then instances of *all* of the prototype's parts are included in the instance aggregadget. If additional parts are specified, they are added after the default parts, so they will appear graphically on top. *It is an error to mention some but not all of a prototype's parts in a* :parts *list.* (The current implementation only looks to see if the *first* part of the prototype is mentioned in the :parts list in order to decide whether or not to include all of the prototype parts.)

Figure [x-sq-box-fig], page 351, illustrates the extension of the :mark part of the x-box prototype with a rectangle. Since parts :line1 and :line2 are not mentioned, they are included in the :mark part automatically.
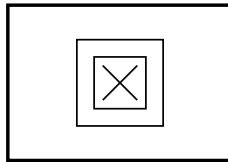
```
(defun x-sq-box-test ()
  (create-instance 'X-SQ-BOX X-BOX
    (:left 210)
    (:top 20)
    (:parts
     '(:box ; inherit the box with no change
       (:mark :modify ; modify the mark
(:parts ; since :line1 and :line2 are not mentioned,
; they are inherited as is
 ((:square ,opal:rectangle ; add a new part to the mark
```

7.7 Aggrelists

Many interfaces require the arrangement of a set of objects in a graphical list, such as menus and parallel lines. Aggrelists are designed to facilitate the arrangement of objects in graphical lists while providing many customizable slots that determine the appearance of the list. The methods `add-component` and `remove-component` can be used to alter the components in the list after the aggrelist has been instantiated. (See section [aggrelist-manipulation-sec], page 374.)

A special style of aggrelists, called "itemized aggrelists", may be used when the items of the list are all instances of the same prototype (e.g., all items in a menu are text strings). These aggrelists use the methods `add-item` and `remove-item` to manipulate the components of the list.

Aggrelists are independent of aggregadgets and may be used separately or inside aggregadgets. Aggrelists may also have aggregadgets as components in order to create objects such as menus or choice lists.

Interactors may be defined for aggrelists using the same methods that implement interactors in aggregadgets (section [agg-interactors], page 338).

7.7.1 How to Use Aggrelists

The definition of the `aggrelist` prototype in Opal is:

```
(create-instance 'opal:aggrelist opal:aggregate
  (:maybe-constant '(:left :top :width :height :direction :h-spacing :v-spacing█
     :indent :h-align :v-align :max-width :max-height
     :fixed-width-p :fixed-height-p :fixed-width-size
     :fixed-height-size :rank-margin :pixel-margin :items :visible))
  (:left 0)
  (:top 0)
  (:width (o-formula ...))
  (:height (o-formula ...))
  (:direction :vertical)        ; Can be :horizontal, :vertical, or NIL
  (:h-spacing 5)        ; Pixels between horizontal elements
  (:v-spacing 5)        ; Pixels between vertical elements
  (:indent 0)        ; How much to indent on wraparound
  (:h-align :left)        ; Can be :left, :center, or :right
  (:v-align :top)        ; Can be :top, :center, or :bottom
  (:max-width  (o-formula (...)))
  (:max-height (o-formula (...)))
  (:fixed-width-p NIL)        ; Whether to use fixed-width-size
  (:fixed-height-p NIL)        ; Whether to use fixed-height-size
  (:fixed-width-size NIL)        ; The width of all components
  (:fixed-height-size NIL)        ; The height of all components
  (:rank-margin NIL)        ; If non-NIL, the number of components in each row/colum
  (:pixel-margin NIL)        ; Same as :rank-margin, but with pixels
  (:head NIL)        ; The first component (read-only slot)
  (:tail NIL)        ; The last component (read-only slot)
```

```
    (:items NIL)                    ; List of the items or a number
    (:item-prototype NIL)          ; Specification of prototype of the items (when itemized
    (:item-prototype-object NIL) ; The actual object, set internally (read-
  only slot)
    ...)
```

Aggrelists are easily customized by providing values for the controlling slots. Any slot listed below may be given a value during the definition of an aggrelist. The slots can also be modified (using the KR function `s-value`) after the aggrelist is displayed to change the appearance of the objects. However, each slot has a default value and the designer may choose to ignore most of the slots.

The list in `:maybe-constant` contains those slots that will be declared constant in an aggrelist whose `:constant` slot contains T. That is, when you create an aggrelist with the slot (`:constant T`), then all of these slots are guaranteed not to change, and all formulas that depend on those slots will be removed and replaced by absolute values. This removal of formulas has the potential to save a large amount of storage space.

The following slots are available for customization of aggrelists:

> `:left` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, The leftmost coordinate of the aggrelist (default is 0).

> `:top` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, The topmost coordinate of the aggrelist (default is 0).

> `:items` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, A number (indicating the number of items in the aggrelist) or a list of values that will be used by the components. If the value is a list, then do not destructively modify the value; instead, set the value with a new list (using `list`) or use `copy-list`.

> `:item-prototype` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, Either a schema or a description of a schema (see section [the-i-p-slot], page 354).

> `:direction` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, Either `:horizontal`, `:vertical` or `nil`. If the value is either `:horizontal` or `:vertical`, the system will install values in the `:left` and `:top` slots of each component, in order to lay out the list properly according to the direction. If the value is `nil`, then the designer must provide formulas for the `:left` and `:top` slots of each component (default is `:vertical`).

> `:v-spacing` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, Vertical spacing between elements (default is 5).

> `:h-spacing` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, Horizontal spacing between elements (default is 5).

> `:fixed-width-p` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, If set to T, all the components will be placed in fields of constant width. These fields will be of the size of the widest component, unless the slot `:fixed-width-size` is non-NIL, in which case it will default to the value stored there (default is `nil`).

> `:fixed-width-size` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, The width of all components, if `:fixed-width-p` is T (default is `nil`).

> `:fixed-height-p` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, If set to T, all the components will be placed in fields of constant height. These fields will be of the

size of the tallest component, unless the slot `:fixed-height-size` is non-NIL, in which case it will default to the value stored there (default is `nil`).

`:fixed-height-size` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, The height of all components, if `:fixed-width-p` is T (default is `nil`).

`:h-align` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, The type of horizontal alignment to use within a field (only applicable if `fixed-width-p` is T). Allowed values are `:left`, `:center`, or `:right` (default is `:left`).

`:v-align` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, The type of vertical alignment to use within a field (only applicable is `fixed-height-p` is T). Allowed values are `:top`, `:center`, or `:bottom` (default is `:top`).

`:rank-margin` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, If non-NIL, then after this many components, a new row will be started for horizontal lists, or a new column for vertical lists (default is `nil`).

`:pixel-margin` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, If non-NIL, then this acts as an absolute position in pixels in the window; if adding the next component would result in extending beyond this value, then a new row or column is started (default is `nil`).

`:indent` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, The amount to indent upon starting a new row/column (in pixels) (default is 0).

7.7.2 Itemized Aggrelists

When all the components of an aggrelist are instances of the same prototype, the aggrelist is referred to as an itemized aggrelist. This type of aggrelist provides for the automatic generation of the components from a specified item prototype. This feature is convenient when creating objects such as menus or button panels, whose components are all similar. (In a non-itemized aggrelist, the components may be of several types, though they still take advantage of the layout mechanisms of aggrelists, as in section [non-itemized-sec], page 364.)

To cause an aggrelist to generate its components from a prototype, the `:item-prototype` and the `:items` slot may be set.

7.7.3 The :item-prototype Slot

The `:item-prototype` slot contains a description of the prototype object that will be used to create the items. This slot is analogous to the `:parts` slot for aggregadgets. Garnet builds an object from the `:item-prototype` description and stores this object in the `:item-prototype-object` slot of the aggrelist. **Do not specify or set the `:item-prototype-object` slot**.

The prototype may be any Garnet object, including aggregadgets, and may be given either as an existing schema name or as a quoted list holding an object definition, as in

```
(:item-prototype '(,opal:rectangle (:width 100)
                                   (:height 50)))
```

The keyword `:modify` may be used to indicate changes to an inherited item prototype, as in

```
(:item-prototype '(:modify (:width 100)
                           (:height 50)))
```

The prototype for the `:item-prototype-object` in this case will be the `:item-prototype-object` of the prototype of the aggrelist being specified. This form would be used to modify the default in some way (see section [modify-item-sec], page 367).

If no local `:item-prototype` slot is specified, the default is to create an instance of the `:item-prototype-object` of the prototype aggrelist. If there is no `:item-prototype-object`, then this is not an itemized aggrelist (see section [non-itemized-sec], page 364).

7.7.4 The :items Slot

The `:items` slot holds either a number or a list. If it is a number n, then n identical instances of `:item-prototype-object` will be created and added to the aggrelist. If it is a list of n elements, n instances of `:item-prototype-object` will be created and added to the aggrelist.

When `:items` is a list of elements, the designer must define a formula in the `:item-prototype` that extracts the desired element from the list for each component. In a menu, for example, the `:items` slot will usually be a list of strings. Components should index their individual strings from the `:items` list according to their `:rank`. The following slot definition, to be included in the `:item-prototype`, would yield this functionality:

```
(:string (o-formula (nth (gvl :rank) (gvl :parent :items))))
```

This formula assigns the nth string in the `:items` list to the nth component of the aggrelist.

The `:items` slot may also hold a nested list so that the components can extract more than one value from it. For example, if the components of a menu are characterized both by a label and a function (to be called when the item is selected), the `:item` slot of the menu will be a list of pairs '((*label function*) ...), and the components will access their strings and associated functions with formulas such as:

```
(:string (o-formula (first (nth (gvl :rank) (gvl :parent :items)))))
(:function (o-formula (second (nth (gvl :rank)
                                   (gvl :parent :items)))))
```

The list in the `:items` list may not be destructively modified. If you need to modify the current value of the slot, you should create a new list (e.g., with `list`) or use `copy-list` on the current value and modify the resulting copied list.

7.7.5 Aggrelist Components

When the value of `:items` changes, the number of components corresponding to the change will be adjusted automatically during the next call to `opal:update`. In most cases, users will never have to do anything special to cause the components to become consistent with the `:items` list.

In some cases, an application might need to refer to the new components (or the new positions of the components) *before* calling `opal:update`. It is possible to explicitly adjust the number of components in the aggrelist after setting the `:items` list by calling:

opal:Notice-Items-Changed *aggrelist*⟨undefined⟩ [method], page ⟨undefined⟩,

where *aggrelist* is the aggrelist whose `:items` slot has changed. This function will additionally execute the layout function on the components, so that they will have up-to-date `:left` and `:top` values.

7.7.6 Constants and Aggrelists

Constant :items and :components

All aggrelists created with a constant `:items` slot have a constant `:components` slot automatically. That is, after the aggrelist has been created with all of its components according to its `:items` list, the `:components` slot becomes constant by default, and the items and components become unmodifiable (with the two exceptions below). In addition, the `:head` and `:tail` slots of the aggrelist, which point to the first and last component, also become constant. By declaring these slots constant, Garnet is able to automatically get rid of the greatest number of formulas possible.

If you really want to add another item to a constant aggrelist, you could wrap a call to `add-item` in `with-constants-disabled`, which disables the protective constant mechanism, and is described fully in the KR chapter. However, just as with aggregadgets (discussed in section [constants-and-aggregadgets], page 325), this is discouraged due to the likelihood that the dimension formulas of the aggrelist will have already been evaluated and thrown away before the new item is added, resulting in an incorrect bounding box for the aggrelist.

A better solution is to create a non-constant aggrelist to begin with. If you plan to change the `:items` slot, then do not include it in the `:constant` list. If you are using T in the constant list, be sure to `:except` the `:items` slot.

Constant :left and :top in Components

The `:left` and `:top` slots of each component are set during the layout of the aggrelist. If all of the slots controlling the layout are constant in the aggrelist, then the `:left` and `:top` slots of the components will be declared constant after they are set. The slots controlling the layout are:

> :left
>
> :top
>
> :items
>
> :direction
>
> :v-spacing
>
> :h-spacing
>
> :indent
>
> :v-align
>
> :h-align
>
> :fixed-width-p
>
> :fixed-height-p
>
> :fixed-width-size
>
> :fixed-height-size
>
> :rank-margin
>
> :pixel-margin

Even if you do not supply customized values for these slots, you will still need to declare them constant for the desired effect. They are all included in the aggrelist's
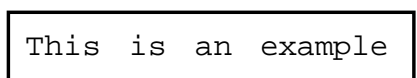
`:maybe-constant` list, so it is easy to declare them all constant with a `:constant` value of T.

Since the aggrelist layout function sets the `:left` and `:top` slots of each component, it is important **not** to declare these slots constant yourself, unless you do so after the aggrelist has already been laid out.

7.7.7 A Simple Aggrelist Example

The following code is a short example of an itemized aggrelist composed of text strings, and the picture of this aggrelist is in figure [aggitem-expl-pict], page 358. Note that the `:left` and `:top` slots of the `:item-prototype` have been left undefined. The aggrelist will fill these slots with the appropriate values automatically.

```
(create-instance 'MY-AGG opal:aggrelist
   (:left 10) (:top 10)
   (:direction :horizontal)
   (:items '("This" "is" "an" "example"))
   (:item-prototype
    '(,opal:text
       (:string ,(formula '(nth (gvl :rank) (gvl :parent :items)))))))
```

```
This  is  an  example
```

Figure 7.21: The picture of an itemized aggrelist.

7.7.8 An Aggrelist with an Interactor

As another example of an itemized aggrelist, consider the schema FRAMED-TEXT-LIST
defined in figure [framed-text-list], page 359. A picture of the FRAMED-TEXT-LIST ag-
grelist appears in figure [framed-text-list-pix], page 360.

```
(create-instance 'FRAMED-TEXT-LIST opal:aggrelist
    (:left 0) (:top 0)
    (:items '("An aggrelist" "using an" "aggregate"
      "as an" "item-prototype"))
    (:item-prototype
     `(,opal:aggregadget
        (:parts
         ((:frame ,opal:rectangle
    (:left ,(o-formula (gvl :parent :left)))
    (:top ,(o-formula (gvl :parent :top)))
    (:width ,(o-formula (+ (gvl :parent :text :width) 4)))
    (:height ,(o-formula (+ (gvl :parent :text :height) 4))))
  (:text ,opal:text
    (:left ,(o-formula (+ (gvl :parent :left) 2)))
    (:top ,(o-formula (+ (gvl :parent :top) 2)))
    (:cursor-index NIL)
    (:string ,(o-formula
        (nth (gvl :parent :rank)
        (gvl :parent :parent :items)))))))
        (:interactors
         ((:text-inter ,inter:text-interactor
    (:window ,(o-formula
        (gv-local :self :operates-on :window)))
    (:feedback-obj NIL)
    (:start-where ,(o-formula
      (list :in (gvl :operates-on :text))))
    (:abort-event #\control-\g)
    (:stop-event (:leftdown #\RETURN))
    (:final-function
     ,#'(lambda (inter text event string x y)
   (let ((elem (gv inter :operates-on)))
     (change-item (gv elem :parent)
  string
  (gv elem :rank))))) )))))
```

Figure 7.22: An aggrelist using an aggregadget as the `:item-prototype`.

```
┌─────────────────────────┐
│  ┌───────────┐          │
│  │An aggrelist│          │
│  ├──────────┤           │
│  │using an  │           │
│  ├──────────┐           │
│  │aggregate │           │
│  ├─────────┤            │
│  │as an    │            │
│  ├─────────────┐        │
│  │item-prototype│        │
│  └─────────────┘        │
└─────────────────────────┘
```
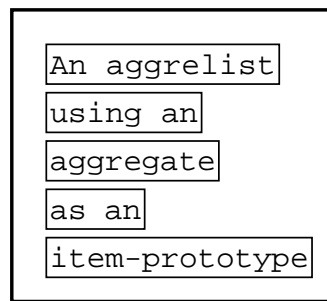
Figure 7.23: A picture of the FRAMED-TEXT-LIST aggrelist.

This aggrelist explicitly defines an aggregadget as the `:item-prototype`. This aggregadget is similar to the FRAMED-TEXT schema defined in figure [framed-text], page 340, but there is an additional `:final-function` slot (see figure [framed-text-list], page 359).

The purpose of the `:final-function` is to keep the strings in the `:items` list consistent with the strings in the components.

Interaction works as follows: Each item is an aggregadget with its own `text-interactor` behavior and `text` component. The cursor text `:string` slot is constrained to the corresponding element in the FRAMED-TEXT-LIST's `:items` slot, but this is a one-way constraint. The text interactor modifies the `:string` slot of the cursor text using `s-value`, which leaves the formula in place, but temporarily changes the slot value. At this point, the `:items` slot and the cursor text `:string` slots are inconsistent, and any change to `:items` would cause all `:string` slot formulas to re-evaluate, possibly losing the string data set by the interactor. To avoid this problem, the `:final-function` of the text interactor directly sets the `:items` slot using `change-item` to be consistent with the formula. This initiates a re-evaluation, but because all values are consistent, no data is lost. Furthermore, if the FRAMED-TEXT-LIST is saved (see section [write-gadget-sec], page 375), the `:items` list will have the current set of strings, and what is written will match what is displayed.

Since the aggregadget defined here is similar to the FRAMED-TEXT schema defined in figure [framed-text], page 340, the `:item-prototype` slot definition could be replaced with

```
(:item-prototype
 '(,FRAMED-TEXT
   (:parts
    (:frame
     (:text :modify
(:string ,(o-formula (nth (gvl :parent :rank)
   (gvl :parent :parent :items)))))))
    (:interactors
     ((:text-inter :modify
   (:final-function
    ,#'(lambda (inter text event string x y)
(let ((elem (gv inter :operates-on)))
  (change-item (gv elem :parent)
        string
        (gv elem :rank)))))))))))
```

provided that the definition for the FRAMED-TEXT schema preceded the FRAMED-TEXT-LIST definition.

See section [Menu-Aggrelist-Example], page 383, for an example of a menu made with an itemized aggrelist.

7.7.9 An Aggrelist with a Part-Generating Function

Section [creating-part-fn], page 332, discussed a feature of aggregadgets that allows you to create parts of an aggregadget by specifying part-generating functions. This feature of aggregadgets can be especially useful when an aggregadget is the `:item-prototype` of an aggrelist. While the same principles hold for aggregadgets whether they are solitary or used in aggrelists, there is a special consideration regarding the `:item-prototype-object` that warrants further discussion.

A typical application of aggrelists that would involve a part-generating function might specify a list of objects in its `:items` list and generate components that have those objects as

parts. Such an application is pictured in figure [esp-cards], page 362. The `:item-prototype` for this aggrelist is an aggregadget with a part-generating function that determines its label. The definition of the aggrelist, along with its part-generating function appears below.
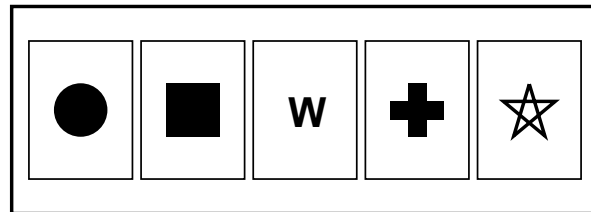
Figure 7.24: An aggrelist that uses a part-generating function in its :item-prototype

```
(defun Get-Label-In-Aggrelist (agg)
```

```
    (let ((alist (gv agg :parent)))
      (if alist  ;; The item-prototype has no parent
 (let* ((item (gv agg :item))
        (new-label (if (schema-p item)
       (if (gv item :parent)
  ;; The item has been used already --
  ;; Use it as a prototype
  (create-instance NIL item)
  ;; Use the item itself
  item)
       (create-instance NIL opal:text
 (:string item)
 (:font (opal:get-standard-font
 :sans-serif :bold :very-large))))))
   (s-value new-label :left
    (o-formula (+ (gvl :parent :left)
 (round (- (gvl :parent :width)
   (gvl :width)) 2))))
   (s-value new-label :top
    (o-formula (+ (gvl :parent :top)
 (round (- (gvl :parent :height)
   (gvl :height)) 2))))
  new-label)
        ;; Give the item-prototype a bogus part
        (create-instance NIL opal:null-object))))

(create-instance 'CIRCLE-LABEL opal:circle
  (:width 30) (:height 30)
  (:line-style NIL)
  (:filling-style opal:black-fill))

(create-instance 'SQUARE-LABEL opal:rectangle
  (:width 30) (:height 30)
  (:line-style NIL)
  (:filling-style opal:black-fill))

(create-instance 'PLUS-LABEL opal:aggregadget
  (:width 30) (:height 30)
  (:parts
   `((:rect1 ,opal:rectangle
       (:left ,(o-formula (+ 10 (gvl :parent :left))))
       (:top ,(o-formula (gvl :parent :top)))
       (:width 10) (:height 30)
       (:line-style NIL) (:filling-style ,opal:black-fill))
     (:rect2 ,opal:rectangle
       (:left ,(o-formula (gvl :parent :left)))
       (:top ,(o-formula (+ 10 (gvl :parent :top))))
       (:width 30) (:height 10)
```

```
                     (:line-style NIL) (:filling-style ,opal:black-fill)))))
         (create-instance 'STAR-LABEL opal:polyline
           (:width 30) (:height 30)
           (:point-list (o-formula
        (let* ((width (gvl :width))    (width/5 (round width 5))
               (height (gvl :height))  (x1 (gvl :left))
               (x2 (+ x1 width/5))     (x3 (+ x1 (round width 2)))
               (x5 (+ x1 width))       (x4 (- x5 width/5))
               (y1 (gvl :top))         (y2 (+ y1 (round height 3)))
               (y3 (+ y1 height)))
          (list x3 y1  x2 y3  x5 y2  x1 y2  x4 y3  x3 y1))))
           (:line-style opal:line-2))

         (create-instance 'ALIST opal:aggrelist
           (:left 10) (:top 20)
           (:items (list CIRCLE-LABEL SQUARE-LABEL "W" PLUS-LABEL STAR-LABEL))
           (:direction :horizontal)
           (:item-prototype
            `(,opal:aggregadget
              (:item ,(o-formula (nth (gvl :rank) (gvl :parent :items))))
              (:width 60) (:height 80)
              (:parts
               ((:frame ,opal:rectangle
                (:left ,(o-formula (gvl :parent :left)))
                (:top ,(o-formula (gvl :parent :top)))
                (:width ,(o-formula (gvl :parent :width)))
                (:height ,(o-formula (gvl :parent :height))))
                (:label ,#'Get-Label-In-Aggrelist)))))))
```

The parts-generating function Get-Label-In-Aggrelist takes into account the aggregadget that will be generated for the `:item-prototype-object` in ALIST. In this example, we are concerned about reserving our label prototypes solely for use in the visible components. We could ignore this case, but then one of our prototypes (like CIRCLE-LABEL) would become a component of the `:item-prototype-object` which never appears in the window. (Additionally, problems could arise if we destroyed the aggrelist along with its `:item-prototype-object` and still expected to use the label as a prototype). Instead, we specifically check if we are generating a part for the `:item-prototype-object` and return a bogus object, saving our real labels for the visible instances.

The gadgets that use aggrelists (like the button panels and menus) all use this feature, so you can have Garnet objects in the `:items` list of a gadget. See the Gadgets chapter for further details.

7.7.10 Non-Itemized Aggrelists

Non-itemized aggrelists may be specified with the `:parts` slot, just as in aggregadgets, except aggrelists will automatically set the `:left` and `:top` slots (among others). Figure [parts-list-fig], page 365, creates an aggrelist with three components, and a picture of this aggrelist is shown in figure [agglist-expl-ref], page 374.

```
(create-instance 'MY-AGG opal:aggrelist
  (:left 10) (:top 10)
  (:parts
   '((:obj1 ,opal:rectangle (:width 60) (:height 30))
     (:obj2 ,opal:oval (:width 60) (:height 30))
     (:obj3 ,opal:roundtangle (:width 60) (:height 30)))))
```

Figure 7.25: Example of an aggrelist with a parts slot.
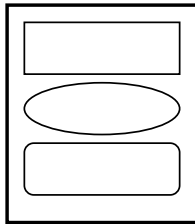
Figure 7.26: The picture of an aggrelist with three components.

Instances of aggrelists are similar to instances of aggregadgets except for the handling of default components and the `:item-prototype-object` slot. Unlike aggregadgets, components that were generated by a `:parts` list are not automatically inherited, so an aggrelist with an empty `:parts` slot will *not* inherit the parts of its prototype. The only way to

inherit these components is to name them in the prototype and to list each name as one of the instance's `:parts`. For example, the following instance of MY-AGG (defined above) will inherit the parts defined in the prototype:

```
(create-instance 'MY-INST MY-AGG
  (:left 100)
  (:parts '(:obj1 :obj2 :obj3)))
```

Note that this syntax is consistent with the rules for customizing the parts of aggregadgets described in section [agg-insts], page 343.

Like aggregadgets, aggrelists created with a `:parts` slot have constant `:components` by default. To cause the `:left` and `:top` slots of the components to become constant after the aggrelist is laid out, all of the layout parameters listed in section [constants-in-aggrelists], page 356, (including the `:items` slot) must be declared constant.

7.8 Instances of Aggrelists

When an instance is made of an itemized aggrelist, components are automatically created as instances of the item prototype object according to the local or inherited `:items` slot.

A consequence of these rules for making instances is that a default instance of a non-itemized aggrelist will typically have no components, while a default instance of an itemized aggrelist will typically have the same component structure as its prototype due to the inherited `:items` slot.

7.8.1 Overriding the Item Prototype Object

For itemized aggrelists, an instance of the item prototype object is made automatically and stored in the `:item-prototype-object` slot of the instance aggrelist. The same syntax used in the `:parts` slot can be used to override slots of the item prototype object. For example, figure [modify-item-fig], page 368, illustrates a variation on the text list in figure [framed-text-list], page 359. Here, the `:frame` component is inherited and modified to be gray and relatively wider than its prototype, a new component, `:white-box` is added, and the `:text` component is inherited and modified to be centered in the new larger surrounding `:frame`. The text interactor is inherited without modification by default.

Note: The `:items` list, if left unspecified, would be shared with FRAMED-TEXT-LIST. It is generally a good idea to specify the `:items` to avoid sharing.
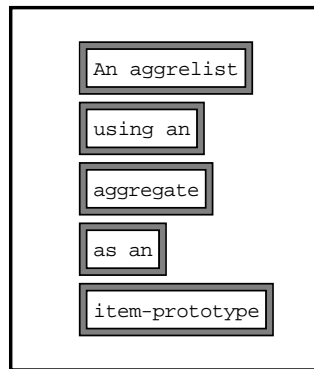
```
(create-instance 'BOXED-TEXT-LIST FRAMED-TEXT-LIST
   (:items '("An aggrelist" "using an" "inherited"
     "but modified" "item-prototype"))
   (:left 120)
   (:top 0)
   (:item-prototype
    '(:modify
       (:parts
        ((:frame :modify
   ; make the frame gray
```

7.9 Manipulating Gadgets Procedurally

A collection of functions is available to alter aggregadget and aggrelist prototypes. When the prototype is altered, the changes propagate down to instances of the prototype. Inheritance of slots is a standard feature of KR, but inheritance of structural changes is unique to aggregadgets and aggrelists and is implemented by the functions and methods described in this chapter.

The philosophy behind structural inheritance is simply stated: *changing a prototype and then making an instance should be equivalent to making an instance and then changing the prototype.* In practice, this equivalence is difficult to achieve completely; exceptions will be noted.

7.9.1 Copying Gadgets

`opal:Copy-Gadget` *gadget*[function], page 90,

This function copies an aggregadget, aggrelist, aggregate, or Opal graphical object. The copy will have the same structure as the original. This is different from (and more expensive than) creating an instance because nothing will be inherited from the original.

When copying an itemized aggrelist, components are not copied, because they inherit from the local items-prototype-object. Instead, the `:items` slot and the item-prototype-object are copied, and new components are generated accordingly.

7.9.2 Aggregadget Manipulation

7.9.3 Add-Component

`opal:Add-Component` *gadget element* [[`:where`] *position* [*locator*]]⟨undefined⟩ [method], page ⟨undefined⟩,

This function behaves just like the `add-component` method for aggregates (see the *Opal Reference Chapter*) except that,

if *gadget* is a prototype, then instances of *element* are also added to instances of *gadget*. This is recursive so that instances of instances, etc., are also affected;

if *element* has slot `:known-as` with value *name*, then the *name* slot of *gadget* is set to be *element*. This creates the standard link from *gadget* to *element* (see Section [known-as-sec], page 326). Ordinarily, the `:known-as` slot of *element* should be set before calling `add-component`.

Note: Names of components and interactors must be unique within their parent. For example, there must not be two components named `:box`.

The *position* and *locator* arguments can be used to adjust the placement of *graphical-object* with respect to the rest of the components of *gadget*.

position can be any of these five values:

> `:front :back :behind :in-front :at` or any of the following aliases:
>
> `:tail :head :before :after :at`

The keyword `:where` is optional; for example,

```
(add-component aggrelist new-component :where :head)
```

```
(add-component aggrelist new-component :head)
```

are valid and equivalent calls to `add-component`. The default value for `:where` is `:tail` (add to the end of the list, which is graphically on top or at the front).

If *position* is either `:before`/`:behind` or `:after`/`:in-front` then the value of *locator* should be a graphical object already in the component list of the aggregate, in which case *graphical-object* is placed with respect to *locator*.

If *position* is `:at`, *graphical-object* is placed at the *locator*th position in the component list, where the zeroth position is the head of the list.

Note: The `add-component` method will always add the component at the most reasonable position if the specified location does not exist. For example, if `add-component` is asked to add a component after another one that does not exist, the new component will be added at the tail.

Instances of *element* are created and added to instances of *gadget* using recursive calls to `add-component`. Since instances of *gadget* may not have the same structure as *gadget*, it is not always obvious where to add a component. In particular, a given *locator* object will never exist in instances, so a new instance *locator* must be inferred from the prototype *locator* as follows:

If the instance gadget has a component that is an instance of the prototype *locator*, then that component is the instance *locator*.

Otherwise, if the instance gadget has a component with the same *name* (`:known-as`) as the prototype *locator*, then that component is the instance *locator*.

Otherwise, a warning is printed, and there is no locator.

Given this procedure for finding an instance *locator*, the insert point is determined as follows:

The default position is `:front`.

If the *position* is specified as `:front` or `:tail`, always insert the component at the `:front`.

If the *position* is specified as `:back` or `:head`, always insert the component at the `:back`.

If the *position* is `:behind` or `:before` *locator*, and an instance *locator* is found, then insert `:behind` the instance *locator*, otherwise insert at the `:front` (the rationale here is to err toward the front, making errors immediately visible).

If the *position* is `:in-front` or `:after` *locator*, and an instance *locator* is found, then insert `:in-front` of the instance *locator*, otherwise insert at the `:front`.

If the *position* is `:at`, then *locator* is an index. Use the same index to insert an *element* instance in each *gadget* instance.

7.9.4 Remove Component

`opal:Remove-Component` *gadget element* [*destroy?*][No value for "Method"]

The `remove-component` method removes the *element* from *gadget*. If *gadget* is connected to a window, then *element* will be erased when the window next has an update message sent to it.

Because aggregadgets allow even the prototype of a component to be overridden in an instance, determining what components to remove is not always straightforward. First, `remove-component` removes all instances of *component* from their parents *if* the parent `is-a-p` the *gadget* argument. (This avoids breaking up aggregates that use instances of components but which are not instances of *gadget*.) Then, `remove-component` removes all parts of instances of *gadget* that have a `:known-as` slot that matches that of the *component*. Components are removed with recursive calls to `remove-component` to affect the entire instance tree.

If *destroy?* is not `nil` (the default is `nil`), then the removed objects are destroyed.

7.9.5 Add-Interactor

Interactors can be added by calling

   `opal:Add-Interactor` *gadget interactor* ⟨undefined⟩ [method], page ⟨undefined⟩,

where *gadget* is an aggregadget or aggrelist. If the interactor has a `:known-as` slot, then this becomes the name of the interactor. The `:operates-on` slot in the interactor is set to the gadget.

An instance of *interactor* is added to each instance of *gadget* using a recursive call to `add-interactor`.

Note: *gadget* should not have an interactor or component with the same name (`:known-as` slot) already. Otherwise, an inconsistent gadget will result.

7.9.6 Remove-Interactor

`opal:Remove-Interactor` *gadget interactor* [*destroy?*]⟨undefined⟩ [method], page ⟨undefined⟩,

is used to remove an interactor. The interactor `:operates-on` slot is destroyed, as is the link from *gadget* to the *interactor* (determined by the value of the *interactor*'s `:known-as` slot). In addition, the *interactor*'s `:active` slot is set to `nil`. The *interactor* is also destroyed if the optional *destroy?* parameter is not `nil`.

Instances of *interactor* that belong to instances of *gadget* (as determined by the `:operates-on` slot) are recursively removed. As with `remove-component`, interactors that have the same name as *interactor* are removed from instances of *gadget*. (This will only have an effect if, in an instance of *gadget*, the default inherited interactor has been overridden or replaced by a different one.)

Note: Since a call to `remove-interactor` will deactivate the interactor, be sure to set the `:active` slot appropriately if the interactor is subsequently added to a gadget.

7.9.7 Take-Default-Component

`opal:Take-Default-Component` *gadget name* [*destroy?*]⟨undefined⟩ [method], page ⟨undefined⟩,

This function removes a local component named by *name*, e.g. `:box`, and replaces it with an instance of the corresponding component in *gadget*'s prototype. The removed component is destroyed if and only if the optional *destroy* argument is not `nil`.

The placement of the new component is inherited as well as the component itself. As with `add-component`, "inherited position" is not well defined when the structure of *gadget* does not match the structure of its prototype. The algorithm for choosing the position is as follows: If the prototype component is the first one, then the instance becomes the first component of *gadget*. Otherwise, a locator (see `add-component`) is found in the prototype such that the locator is "`:in-front`" of the prototype component. If this locator has an instance in *gadget*, the instance is used as a locator in a call to `add-component`, with the *position* parameter being `:in-front`. If the locator does not exist in *gadget*, then the *position* used is `:front`, so at least any error should become visibly apparent.

Changes are propagated to instances of *gadget*.

7.9.8 Itemized Aggrelist Manipulation

7.9.9 Add-Item

`opal:Add-Item` *aggrelist* [*item*] [[`:where`] *position* [*locator*] [`:key` *function-name*]] ⟨undefined⟩ [method], page ⟨undefined⟩,

If supplied, *item* will be added to the `:items` slot of *aggrelist*, and a new instance of `:item-prototype-object` will be added to the components of *aggrelist*. The `add-item` method will perform the necessary bookkeeping to maintain the appearance of the list.

It is an error (actually, a continuable break condition) to add an item to an aggrelist whose `:items` slot is constant. To work around this error, consult section [constants-and-aggregadgets], page 325.

The *position*, *locator* and *function-name* arguments can be used to adjust the placement of *item* with respect to the rest of the items of *aggrelist*.

position can be any of these five values:

> `:front :back :behind :in-front :at` or any of the following aliases:
>
> `:tail :head :before :after :at`

Note: the graphically *front* object is at the *tail* of the components list, etc. If position is either `:before`/`:behind` or `:after`/`:in-front` then the value of *locator* should be an item already in the `:items` slot of the aggrelist, in which case *item* is placed with respect to *locator*.

For example, the following line will add a new item to the aggrelist defined in section [aggitem-expl-ref], page 357:

    (opal:add-item MY-AGG "really" :after "is")

The string "really" will be added to `my-agg` with the resulting aggrelist appearing as "This is really an example".

Furthermore, if the `:items` slot holds a nested list, *:key function-name* can be used to match *locator* only with the result of *function-name* applied to each element of `:items`. For example, if the `:items` slot of `an-aggrelist` is (("foo" 4) ("bar" 2) ("foo" 7)),

    (add-item an-aggrelist '("foobar" 3) :after "foo" :key #'car)

compare "foo" only to the `cars` of the list, and therefore will add the new item as the second element of the list. The line

    (add-item an-aggrelist '("barfoo" 5) :before 7 :key #'cadr)

will add the new item just before the last one.

Note: `add-item` will add the item at the most reasonable position if the specified position does not exist. For example, if `add-item` is asked to add a component after another one that does not exist, the new component will be added at the tail.

7.9.10 Remove-Item

`opal:Remove-Item` *aggrelist* [*item* [`:key` *function-name*]] ⟨undefined⟩ [Method], page ⟨undefined⟩,

The method `remove-item` removes *item* from the `:items` list and the `:components` list of *aggrelist*.

It is an error (actually, a continuable break condition) to add an item to an aggrelist whose `:items` slot is constant. To work around this error, consult section [constants-and-aggregadgets], page 325.

If the `:items` slot holds a nested list, *:key function-name* can be used to specify to try to match *item* only with the result of *function-name* applied to each element of `:items`. For example, if the `:items` slot of `an-aggrelist` is `(("foo" 4) ("bar" 2) ("foo" 7))`,

    (remove-item AN-AGGRELIST "foo" :key #'car)

removes the first item, while

    (remove-item AN-AGGRELIST '("foo" 7))

removes the last one.

7.9.11 Remove-Nth-Item

`opal:Remove-Nth-Item` *aggrelist* n⟨undefined⟩ [method], page ⟨undefined⟩,

To remove an item by position rather than by content, use `remove-nth-item`. The n(th) item is removed from the `:items` slot of *aggrelist*, and the component corresponding to that item will be removed during the next call to `opal:update`.

It is an error to add an item to an aggrelist whose `:items` slot is constant. To work around this error, consult section [constants-and-aggregadgets], page 325.

7.9.12 Change-Item

To change just one item in the `:items` list, call

    opal:Change-Item *aggrelist* *item* n⟨undefined⟩ [method], page ⟨undefined⟩,

where *aggrelist* is the aggrelist to be modified, *item* is a new value for the `:items` list, and n is the index of the item to be changed (the index of the first item is zero).

This function is potentially more efficient than calling `add-item` and `remove-item`, because it ensures that the component corresponding to the changed item will be reused if possible, instead of destroying and reallocating a new component.

7.9.13 Replace-Item-Prototype-Object

`opal:Replace-Item-Prototype-Object` *aggrelist* *item-proto*⟨undefined⟩ [method], page ⟨undefined⟩,

This function is used to replace the `:item-prototype-object` slot of an itemized aggrelist. Any aggrelists which inherit the slot from this one will also be affected. The

components of affected aggrelists are replaced with instances of the new `:item-prototype-object`.

For example, suppose an application uses a number of instances of radio buttons, an aggrelist whose item prototype object determines the appearance of a single button. By calling `replace-item-prototype-object` on the radio buttons prototype, all button throughout the application will change to reflect the new style.

7.9.14 Ordinary Aggrelist Manipulation

7.9.15 Add-Component

The `add-component`, defined in section [add-component-sec], page 369, can also be used to add components to an aggrelist. The system automatically adjusts the appearance of the aggrelist to accommodate the changes in the list of components.

In addition to adding *graphical-object* to *aggrelist*, `add-component` will add some slots to *graphical-object*, or modify existing slots. The slots created or modified by `add-component` are:

> `:left, :top` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, Unless the `:direction` slot of *aggrelist* is `nil`, the system will set these slots with integers that arrange *graphical-object* neatly in the layout of the aggrelist components.
>
> `:rank` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, This slot is set with a number that indicates the position of this component in the list (the head has rank 0). If this component is not visible, then this value has no meaning.
>
> `:prev` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, This contains the previous component in the list, regardless of what is visible.
>
> `:next` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, This contains the next component in the list, regardless of what is visible.

Note: `add-components` (plural) can be used to add several components to an aggrelist.

An alternative implementation of figure [parts-list-fig], page 365, is shown in figure [agglist-expl-ref], page 374. In each component of the aggrelist, the `:left` and `:top` slots have been left undefined. The aggrelist will fill these slots with the appropriate values automatically.

```
(create-instance 'MY-AGG opal:aggrelist (:top 10) (:left 10))
(create-instance 'MY-RECT opal:rectangle
   (:width 100) (:height 30))
(create-instance 'MY-OVAL opal:oval
   (:width 100) (:height 30))
(create-instance 'MY-ROUND opal:roundtangle
   (:width 100) (:height 30))
(add-components MY-AGG MY-RECT MY-OVAL MY-ROUND)
```

Figure 7.28: Example of an aggrelist built using add-component.

7.9.16 Remove-Component

See section [remove-component-sec], page 370, for a description of this method.

Useful hint: It is possible to make components of an aggrelist temporarily disappear by simply setting their `:visible` slot to `nil` — the list will adjust itself so that there is no gap where the item once was. If a gap is desired, then an `opal:null-object` may be inserted into the list — this is an `opal:view-object` that has its `:visible` slot set to T, but has no draw method.

7.9.17 Remove-Nth-Component

`opal:Remove-Nth-Component` *aggrelist* n⟨undefined⟩ [method], page ⟨undefined⟩,

The n(th) component of *aggrelist* is removed by invoking `remove-local-component`. Instances of *aggrelist* are *not* affected.

7.9.18 Local Modification

A number of functions exist to modify gadgets without changing their instances. Their behavior is exactly like the corresponding recursive version described earlier, except that changes are not propagated to instances.

`opal:Add-Local-Component` *gadget element* [[`:where`] *position* [*locator*]]⟨undefined⟩ [method], page ⟨undefined⟩,

`opal:Remove-Local-Component` *gadget element* [*destroy?*][No value for "Method"]

`opal:Add-Local-Interactor` *gadget interactor*⟨undefined⟩ [method], page ⟨undefined⟩,

`opal:Remove-Local-Interactor` *gadget interactor* [*destroy?*]⟨undefined⟩ [method], page ⟨undefined⟩,

`opal:Add-Local-Item` *aggrelist* [*item*] [[`:where`] *position* [*locator*] [`:key` *function-name*]] ⟨undefined⟩ [method], page ⟨undefined⟩,

`opal:Remove-Local-Item` *aggrelist* [*item* [`:key` *function-name*]] ⟨undefined⟩ [Method], page ⟨undefined⟩,

7.10 Reading and Writing Aggregadgets and Aggrelists

An aggregadget or aggrelist may be written to a file. This creates a compilable lisp program that can be reloaded to recreate the object that was saved. To save an aggregadget, use the `opal:write-gadget` function:

7.10.1 Write-Gadget

`opal:Write-Gadget` *gadget file-name* &optional *initialize?*[function], page 90,

where *gadget* is a graphical object, an aggregadget or an aggrelist (or a list of these), and *file-name* is the file name (a string) to be written, or `t` to write to `*standard-output*`. If several calls are made to `write-gadget` to output a sequence of gadgets to the same stream, set the *initialize?* flag to `nil` after the first call. The default value of *initialize?* is T.

If the gadget has any references to gadgets that are not part of the standard set of Opal objects or Interactors, then a warning is printed. *Note: gadget* must not be a symbol or list of symbols:

```
(write-gadget (list BUTTON SLIDER) "misc.lisp")  !; RIGHT!
```

```
(write-gadget '(BUTTON SLIDER) "misc.lisp"); WRONG!
```

Slots that are ordinarily created automatically are not written by `write-gadget`. For example, the `:is-a-inv` slot (maintained by KR) and the `:update-slots-values` slot (maintained by Opal) are not written. The slots to ignore are found in the `:do-not-dump-slots` slot, which is normally inherited. In some cases, it may be desirable to suppress the output of certain slots, e.g. bookkeeping information, and this can be done by setting `:do-not-dump-slots` as follows:

```
(s-value my-proto
         :do-not-dump-slots
         (append list-of-slots (gv my-proto :do-not-dump-slots)))
```

Do not destructively modify `:do-not-dump-slots`! Putting the slot name `:do-not-dump-slots` on the list will prevent the `:do-not-dump-slots` slot from being written. This is probably not a good idea, since if the object is written and reloaded, the local `:do-not-dump-slots` information will be lost.

7.10.2 Avoiding Deeply Nested Parts Slots

One would expect an instance of a standard gadget (see the *Garnet Gadgets Reference chapter*) to have a very concise output representation; however, once the instance is manipulated, various slots are set by interactors. Often, these slots are deeply nested in the gadget structure, and the output has correspondingly deeply nested `:parts` slots. This is a consequence of the fact that Garnet maintains little separation between the gadget definition and local state information.

One solution is to carefully install slot names on the `:do-not-dump-slots` slot to suppress the output of slots for which the default inherited value is acceptable. Another, more drastic, solution is to set the `:do-not-dump-objects` slot in selected objects. This slot may have one of three values:

> NIL ⟨undefined⟩ [shortdash], page ⟨undefined⟩, The default; write out all slots and parts that differ from the prototype.

> `:me` interactors are inherited without modification, so there is no need to write `:parts`, `:interactors`, or `:item-prototype` slots at this level. Other slots, such as `:left` and `:top` should be written.

> `:children` ⟨undefined⟩ [shortdash], page ⟨undefined⟩, Write out `:parts`, `:interactors` and `:item-prototype` slots, but do not allow further nesting. This is equivalent to setting the `:do-not-dump-objects` slot of each component, interactor, and the item-prototype to `:me`.

7.10.3 More Details

The `write-gadget` function makes no attempt to write out objects that are needed as prototypes or that are referenced by formulas. It is the user's responsibility to make sure these objects are loaded before loading a gadget; otherwise, an "unbound symbol" error is likely to occur. If the *gadget* argument is a list, then each aggregadget or aggrelist of the list is written in sequence to the file.

To load a gadget after it has been written, the standard lisp loader (`load`) should be used.

When an aggregadget is written that uses a function to create parts (see section [runtime], page 335), the created parts are written explicitly and in full, as opposed to simply writing out the original `:parts` slot. This guarantees that any modifications to the aggregadget after it was created will be correctly written.

opal:**verbose-write-gadget**⟨undefined⟩ [variable], page ⟨undefined⟩,

If **verbose-write-gadget** is non-NIL, objects will be printed to **error-output** as they are visited by `write-gadget`. Indentation indicates the level of the object in the aggregate hierarchy. *Note:* objects will be printed even if, due to inheritance, nothing needs to be written.

7.10.4 Writing to Streams

The `write-gadget` function can be used as is for simple applications, but it is sometimes desirable to write a header to a file and perhaps embed code written by `write-gadget` into a function definition. This is done by temporarily re-binding **standard-output** as in the following example:

```
(with-open-file (*standard-output* "my-file.lisp"
 :direction :output :if-exists :supersede)
    ;; write header to standard output:
    (format T "... file header info goes here ...")
    ;; write a gadget:
    (write-gadget my-gadget t)
    ;; if there are more gadgets, call with initialize? set to NIL:
    (write-gadget another-gadget T NIL))
```

7.10.5 References to External Objects

Gadgets may contain references to "external objects", that is, objects that are not part of the gadget. When an external object is written, A warning is ordinarily printed to notify the user that the object must be present when the gadget code is loaded.

opal:**standard-names**⟨undefined⟩ [variable], page ⟨undefined⟩,

Many objects, including standard Opal objects, standard Interactors, and objects in the Garnet Gadget library, are considered part of the Garnet environment, so no warning is written for these references. The list **standard-names** tells `write-gadget` what object symbols to assume will be defined when the gadget is loaded. This list can be extended with new new names before calling `write-gadget`.

opal:**defined-names**⟨undefined⟩ [variable], page ⟨undefined⟩,

The global variable **defined-names** is initialized to **standard-names** when you call `write-gadget`. As gadgets are written, their names are pushed onto **defined-names**, so if a list of gadgets is written and the second references the first, no warning will be printed. **defined-names** (not **standard-names**) is what `write-gadgets` actually searches to see if a name is defined.

opal:**required-names**⟨undefined⟩ [variable], page ⟨undefined⟩,

The variable **required-names** is initialized to `nil` when you call `write-gadget`. Whenever a name is written that is not on **defined-names**, it is pushed onto **required-names** and a warning is printed. Inspecting the value of **required-names**

after calling `write-gadget` can give the caller information about what additional gadgets should be saved.

The initialization of `*defined-names*` and `*required-names*` is suppressed when the *initialize?* argument to `write-gadget` is set to `nil`.

7.10.6 References to Graphic Qualities

A reference to an `opal:graphic-quality` object is handled as a special case. Graphic qualities include `opal:filling-style`, `opal:line-style`, `opal:color`, `opal:font`, and `opal:font-from-file`. Although these are objects, they are treated more like record structures throughout the Garnet system. For example, changing a slot in a graphic quality will not automatically cause an update; only replacing a graphic quality with a new one (or faking it with a call to `kr:mark-as-changed`) will cause the update.

Because of the way graphic qualities are used, it is best to think of graphic qualities as values rather than shared objects. Consequently, `write-gadget` writes out graphic qualities by calling `create-instance` to construct an equivalent object rather than by writing an external reference that is likely to be undefined when the file is loaded.

For example, here is a rectangle with a special color, and the output generated by `write-gadget`:

```
(create-instance 'MY-RED RED
   (:red 0.5))

(create-instance 'MY-RECT RECTANGLE
   (:color my-red))

* (write-gadget MY-RECT T)
(create-instance 'MY-RECT RECTANGLE
   (:COLOR (create-instance NIL COLOR
              (:BLUE 0.0)
      (:GREEN 0.0)
      (:RED 0.5))))
```

7.10.7 Saving References From Within Formulas

Writing direct references from within `o-formula`'s to other objects is not possible (in a lisp implementation-independent way) because `o-formula` builds a closure, and bindings within the closure are not externally visible. For example, in

```
(let ((thermometer THERMOMETER-1))
   (o-formula (gv thermometer :temperature)))
```

the variable *thermometer* is bound inside the `let` and is not accessible to any routine that would write the formula. Even though the expression (`gv thermometer :temperature`) is saved in the formula in the current KR implementation, this does not reveal the binding needed to reconstruct the formula.

Fortunately, aggregadgets and aggrelists rarely make direct references to objects. Typically, references to objects take the form of paths in formulas, for example, (`gvl :parent :box :left`). However, there may be occasions when a direct reference is required, for example, when an aggregadget depends upon the value of some separate application object.

There are several ways to avoid problems associated with direct references from formulas:

Use `formula` instead of `o-formula`. The `formula` function interprets its expression, so expressions with embedded references can be constructed at run-time. For example, the thermometer example could be written as:

```
(formula '(gv ',thermometer :temperature))
```

embedding the actual reference directly into the expression. This expression can be written and read back in without problems. However, since formula expressions are interpreted, re-evaluation of the formula will be much slower than the corresponding o-formula.

Put the object reference into a slot, avoiding direct references altogether. For example, to create a dependency on the `:temperature` slot of object THERMOMETER-1, set the `:thermometer` slot of the gadget to THERMOMETER-1, and reference the slot from the formula:

```
(o-formula (gvl :thermometer :temperature))
```

Since the reference to THERMOMETER-1 is now a slot value rather than a hidden binding in a closure, it can be written and read back in without problems. The only performance penalty of this approach will be the extra slot access, which should not add much overhead. There is, however, the added problem of choosing slot names so as not to interfere with other formulas.

Use an @code{e-formula}, described below. This provides the functionality
and speed of @code{o-formula} as well as the ability to save to files
at the expense of a little more work for the programmer and some
extra function definitions.

@subsection The e-formula function
@node The e-formula function
@cindex{e-formula}
@code{e-formula} @emph{expression}@ref{function}

The argument to @code{e-formula} is an expression that, when evaluated,
will return a formula. The expression is retained so that the
original @code{e-formula} expression can be reconstructed when the
formula is written to a file. Returning once again to the thermometer
example, here is how the problem would be solved using @code{e-formula}:
@example
(defun temperature-formula (thermometer)
   (o-formula (gv thermometer :temperature)))

(create-instance 'DISPLAY-1 DISPLAY
   (:value (e-formula '(temperature-formula ',THERMOMETER-1))))
@end example
The first expression defines
an auxiliary (compiled) function @code{temperature-formula}.
The second expression creates an instance of the prototype
@code{display} (not implemented here) whose @code{:value} slot holds

```
the desired formula.
```

```
When the @code{e-formula} expression is evaluated, the argument,
@example
(temperature-formula '#k<THERMOMETER-1>)
@end example
is evaluated. The @code{temperature-formula} function in turn produces
an @code{o-formula} in which the reference to @code{#k<thermometer-1>} is
captured by a compiled closure. Because the @code{o-formula} is based
on a compiled closure, it will evaluate quickly. Note that the
Lisp interpreter is invoked only to @emph{create} the formula, not to evaluate█
it.
```

```
When @code{display-1} is written to a file, @code{write-gadget} will
write
@example
(e-formula '(temperature-formula ',THERMOMETER-1))
@end example
which is the same expression used to create the original formula.
A warning will be issued when the THERMOMETER-1 is written:
@example
Warning: non-standard schema written as THERMOMETER-1
@end example
to warn that a direct reference to THERMOMETER-1 was written
and must be defined when the schema is reloaded.
In order to reload this formula, the function @code{temperature-formula}
must be also be defined.
```

7.11 More Examples

7.11.1 A Customizable Check-Box

Figure [example-1], page 381, shows the definition of a check-box whose position and size can be determined by the programmer when it is used as a prototype object.

The :parts slot defines the :box object as an instance of opal:rectangle with coordinates dependent on the parent aggregadget. Similarly, the :mark object is an opal:aggregadget itself, and its components are dependent on slots in the top-level aggregadget.

Two instances of CHECK-BOX are created — the first one using the default values for the coordinates and the second one using both default and custom coordinates. Both are pictured in figure [example-1-pic], page 382.

```
(create-instance 'CHECK-BOX opal:aggregadget
   (:left 20)
   (:top 20)
   (:width 50)
   (:height 50)
   (:parts
    '((:box ,opal:rectangle
         (:left ,(o-formula (gvl :parent :left)))
         (:top ,(o-formula (gvl :parent :top)))
         (:width ,(o-formula (gvl :parent :width)))
         (:height ,(o-formula (gvl :parent :height))))
      (:mark ,opal:aggregadget
         (:parts
          ((:left-line ,opal:line
              (:x1 ,(o-formula (+ (gvl :parent :parent :left)
                    (floor (gvl :parent :parent :width) 10))))
              (:y1 ,(o-formula (+ (gvl :parent :parent :top)
                    (floor (gvl :parent :parent :height) 2))))
              (:x2 ,(o-formula (+ (gvl :parent :parent :left)
                    (floor (gvl :parent :parent :width) 2))))
              (:y2 ,(o-formula (+ (gvl :parent :parent :top)
                    (floor (* (gvl :parent :parent :height) 9)
   10))))
              (:line-style ,opal:line-2))
           (:right-line ,opal:line
              (:x1 ,(o-formula
    (opal:gvl-sibling :left-line :x2)))
              (:y1 ,(o-formula
    (opal:gvl-sibling :left-line :y2)))
              (:x2 ,(o-formula (+ (gvl :parent :parent :left)
                    (floor (* (gvl :parent :parent :width) 9)
   10))))
              (:y2 ,(o-formula (+ (gvl :parent :parent :top)
                    (floor (gvl :parent :parent :height) 10))))
              (:line-style ,opal:line-2)))))))))

(create-instance 'CB1 CHECK-BOX)

(create-instance 'CB2 CHECK-BOX (:left 90) (:width 100) (:height 60))
```
Figure 7.29: The definition of a customizable check-box.
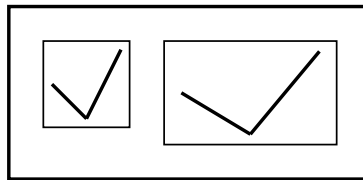
Figure 7.30: Instances of the customizable check-box.

7.11.2 Hierarchical Implementation of a Customizable Check-Box

Figure [example-2], page 383, shows the definition of a customizable check-box as in figure [example-1], page 381. However, this second CHECK-BOX definition exploits the hierar-

chical structure of the check box to modularize the definition of the schema. The modular style allows for the reuse of previously defined code — the `check-mark` schema may now be used for other applications as well.

```
(create-instance 'CHECK-MARK opal:aggregadget
  (:parts
   '((:left-line ,opal:line
       (:x1 ,(o-formula (+ (gvl :parent :parent :left)
(floor (gvl :parent :parent :width) 10))))
       (:y1 ,(o-formula (+ (gvl :parent :parent :top)
(floor (gvl :parent :parent :height) 2))))
       (:x2 ,(o-formula (+ (gvl :parent :parent :left)
(floor (gvl :parent :parent :width) 2))))
       (:y2 ,(o-formula (+ (gvl :parent :parent :top)
(floor (* (gvl :parent :parent :height) 9) 10))))
       (:line-style ,opal:line-2))
     (:right-line ,opal:line
       (:x1 ,(o-formula (opal:gvl-sibling :left-line :x2)))
       (:y1 ,(o-formula (opal:gvl-sibling :left-line :y2)))
       (:x2 ,(o-formula (+ (gvl :parent :parent :left)
(floor (* (gvl :parent :parent :width) 9) 10))))
       (:y2 ,(o-formula (+ (gvl :parent :parent :top)
(floor (gvl :parent :parent :height) 10))))
       (:line-style ,opal:line-2)))))

(create-instance 'CHECK-BOX opal:aggregadget
  (:left 20)
  (:top 20)
  (:width 50)
  (:height 50)
  (:parts
   '((:box ,opal:rectangle
       (:left ,(o-formula (gvl :parent :left)))
       (:top ,(o-formula (gvl :parent :top)))
       (:width ,(o-formula (gvl :parent :width)))
       (:height ,(o-formula (gvl :parent :height))))
     (:mark ,check-mark))))
```

Figure 7.31: A hierarchical implementation of a customizable check-box.

7.11.3 Menu Aggregadget with built-in interactor, using Aggrelists

The figure [menu-aggrelist-ref], page 386, shows how to create a menu aggregadget, by using itemized aggrelist to create the items of the menu. This example also shows how to attach an interactor to such an object. The menu is made of four parts: a frame, a shadow, a feedback and an items-agg, which is an aggrelist containing the items of the menu. Each item is an instance of the prototype `menu-item`. The items are created according to the

labels and notify-functions given in the `:items` slot of the menu. The menu also contains a built-in interactor which, when activated, will call the functions associated to the selected item.

The figure [menu-aggrelist2-ref], page 387, shows how to create an instance of the menu. A picture of these menus (the prototype and its instance) is shown in figure [menu-aggitem-pict], page 385.
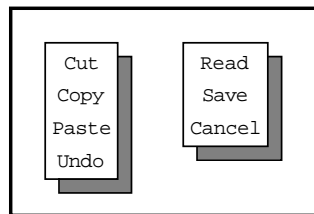
Figure 7.32: The two menus (prototype and instance) made with itemized aggrelist.

```
(defun my-cut () (format T "~%Function CUT called~%"))
(defun my-copy () (format T "~%Function COPY called~%"))
(defun my-paste () (format T "~%Function PASTE called~%"))
(defun my-undo () (format T "~%Function UNDO called~%"))

(create-instance 'MENU-ITEM opal:text
   (:string (o-formula (car (nth (gvl :rank) (gvl :parent :items)))))
   (:action (o-formula (cadr (nth (gvl :rank)
 (gvl :parent :items))))))

(create-instance 'MENU opal:aggregadget
   (:left 20) (:top 20)
   (:items '(("Cut" (my-cut)) ("Copy" (my-copy))
             ("Paste" (my-paste)) ("Undo" (my-undo))))
   (:parts
    `((:shadow ,opal:rectangle
(:filling-style ,opal:gray-fill)
(:left ,(o-formula (+ (gvl :parent :frame :left) 8)))
(:top ,(o-formula (+ (gvl :parent :frame :top) 8)))
(:width ,(o-formula (gvl :parent :frame :width)))
(:height ,(o-formula (gvl :parent :frame :height))))
      (:frame ,opal:rectangle
(:filling-style ,opal:white-fill)
(:left ,(o-formula (gvl :parent :left)))
(:top ,(o-formula (gvl :parent :top)))
(:width ,(o-formula (+ (gvl :parent :items-agg :width) 8)))
(:height ,(o-formula (+ (gvl :parent :items-agg :height) 8))))
      (:feedback ,opal:rectangle
(:left ,(o-formula (- (gvl :obj-over :left) 2)))
(:top ,(o-formula (- (gvl :obj-over :top) 2)))
(:width ,(o-formula (+ (gvl :obj-over :width) 4)))
(:height ,(o-formula (+ (gvl :obj-over :height) 4)))
(:visible ,(o-formula (gvl :obj-over)))
(:draw-function :xor))
      (:items-agg ,opal:aggrelist
(:fixed-width-p T)
(:h-align :center)
(:left ,(o-formula (+ (gvl :parent :left) 4)))
(:top ,(o-formula (+ (gvl :parent :top) 4)))
(:items ,(o-formula (gvl :parent :items)))
(:item-prototype ,menu-item))))
   (:interactors
    `((:press ,inter:menu-interactor
(:window ,(o-formula (gv-local :self :operates-on :window)))
(:start-where ,(o-formula (list :element-of
(gvl :operates-on :items-agg))))
(:feedback-obj ,(o-formula (gvl :operates-on :feedback)))
(:final-function
  ,#'(lambda (interactor final-obj-over)
(eval (gv final-obj-over :action)))))))))
```

Figure 7.33: Definition of a menu with built-in interactor and itemized aggrelist.

```
(defun my-read () (format T "~%Function READ called~%"))
(defun my-save () (format T "~%Function SAVE called~%"))
(defun my-cancel () (format T "~%Function CANCEL called~%"))

(create-instance 'MY-MENU MENU
   (:left 100) (:top 20)
   (:items '(("Read" (my-read)) ("Save" (my-save))
             ("Cancel" (my-cancel)))))
```
Figure 7.34: Creation of an instance of MENU.

7.12 Aggregraphs

The purpose of Aggregraphs is to allow the easy creation and manipulation of graph objects, analogous to the creation and manipulation of lists by Aggrelists. In addition to the standard `aggregraph`, Opal provides the `scalable-aggregraph` which will fit inside dimensions supplied by the programmer, and the `scalable-aggregraph-image` which changes appearance in response to changes in the original graph.

7.13 Using Aggregraphs

In order to generate an aggregraph from a source graph, the source graph must be described by defining its roots (a graph may have more than one root) and a function to generate children from parent nodes. When the aggregraph is initialized, the generating function is first called on the root(s), then on the children of the roots, and so on. For each *source-node* in the original graph, a new *graph-node* is created and added to the aggregraph. Graphical links are also created which connect the graph-nodes appropriately. The layout function (which can be specified by the user) is then called to layout the graph in a pleasing manner. The resulting aggregraph instance can then be displayed and manipulated like any other Garnet object.

Although most programmers will be satisfied with the graphs generated by the default layout function, section [layout-graph], page 397, contains a discussion of how to customize the function used to compute the locations for the nodes in the graph.

See the file `demo-graph.lisp` for a complete interface that uses many features of aggregraphs.

7.13.1 Accessing Aggregraphs

The aggregraph files are **not** automatically loaded when the file `garnet-loader.lisp` is used to load Garnet. There is a separate file called `aggregraphs-loader.lisp` that is used to load all the aggregraphs files. This file is loaded when the line (`load Garnet-Aggregraphs-Loader`) is executed after Garnet has been loaded with `garnet-loader.lisp`.

Aggregraphs reside in the `Opal` package. We recommend that programmers explicitly reference the `Opal` package when creating instances of aggregraphs, as in `opal:aggregraph`. However, the package name may be dropped if the line (`use-package 'opal`) is executed before referring to any object in that package.

7.13.2 Overview

In general, programmers will be able to ignore most of the aggregraph slots described in the following sections, since they are used to customize the layout function of the aggregraph. However, three slots must be set before before any aggregraph can be initialized:

`:children-function` – This slot should contain a function that generates a list of child nodes from a parent node. The function takes the parameters `(lambda (source-node depth))` where *depth* is a number maintained internally by aggregraphs that corresponds to the distance of the current node from the root, and *source-node* is an object in the source graph to be expanded. The function should return a list of the children of *source-node* in the source graph, or `nil` to indicate the node either has no children or should not be expanded (when depth `>` 1, for example).

`:info-function` – The function in this slot should take the parameter `(lambda (source-node))` where *source-node* is an object in the source graph. It should return a string associated with the *source-node* so that a label can be placed on its corresponding graph node in the aggregraph. (If the node-prototype is customized by the programmer, then this function might return some other identifying object instead of a string.) The value returned by the function is stored in the `:info` slot of the graph node.

`:source-roots` – A list of roots in the source graph.

Caveats:

The source nodes must be distinguishable by one of the tests `#'eq`, `#'eql`, or `#'equal`. The default is `#'eql`. (Refer to the `:test-to-distinguish-source-nodes` slot in section [aggregraph-slots], page 393.)

Instances of aggregraphs can be used as prototypes for other aggregraphs without providing values for all the required slots in the prototype.

7.13.3 Aggregraph Nodes

Each type of aggregraph has its own type of node and link prototypes. For the `aggregraph`, the prototypes are `aggregraph-node-prototype` and `aggregraph-link-prototype`, which are defined in the slots `:node-prototype` and `:link-prototype`. To change the look of the nodes or the links in an aggregraph, the programmer will need to define new prototype objects in these slots. Section [aggregraph-with-interactor], page 391, contains an example aggregraph schema that modifies the node prototype.

The node and link prototypes for `scalable-aggregraph` are `scalable-aggregraph-node-prototype` and `scalable-aggregraph-link-prototype`. The prototypes for `scalable-aggregraph-image` are `scalable-aggregraph-image-node-prototype` and `scalable-aggregraph-image-link-prototype`.

The actual nodes and links of the aggregraph are kept in "sub-aggregates" of the aggregraph. The aggregates in the `:nodes` and `:links` slots of the top-level aggregraph have the nodes and links as their components. To access the individual links and nodes, look at the `:components` slot of these aggregates. For example, the instruction

```
(opal:do-components (gv graph :nodes)
                    #'(lambda (node)
                        (format T "~A~%" (gv node :info))))
```

will print out the names of all the nodes in the graph.

As each graph-node is created, a pointer to the corresponding source-node is put in the slot `:source-node` of the graph-node. This allows access to the source node from the graph-node. If desired, a user can supply a function in the slot `:add-back-pointer-to-nodes-function`. This function will be called on each source-node/graph-node pair, and should put a pointer to the graph-node in the source-node data structure. This can be used to establish back pointers in the programmer's data structure.

The function in the slot `:source-to-graph-node` can be useful in finding a particular node in the graph. When this method is given a source-node, it will return the corresponding graph-node if one already exists in the graph.

Useful slots in the node objects include:

> `:left` and `:top` – These slots must be set either directly by the layout function or indirectly through formulas (probably dependent on other slots in the node that are set by the layout function).

> `:width` and `:height` – Dimensions of the node.

> `:links-to-me` and `:links-from-me` – Each slot contains a list of links that point to or from the given node. To get the nodes on the other side of the links, reference the `:from` and `:to` slots of the links, respectively.

> `:source-node` – A pointer to the corresponding node in the source graph (i.e., the source-node of this graph-node). See `:add-back-pointer-to-nodes-function` for back-pointers from the source-node to the graph-node.

> `:layout-info-...` – Several slots that begin with `":layout-info-"` are reserved for bookkeeping by the layout function. Do not set these slots except as part of a customized layout function.

7.13.4 A Simple Example
```
(create-instance 'SCHEMA-GRAPH opal:aggregraph
   (:children-function #'(lambda (source-node depth)
   (if (> depth 1)
       NIL
       (gv source-node :is-a-inv))))
   (:info-function #'(lambda (source-node)
       (string-capitalize
 (kr:name-for-schema source-node))))
   (:source-roots (list opal:view-object)))
```
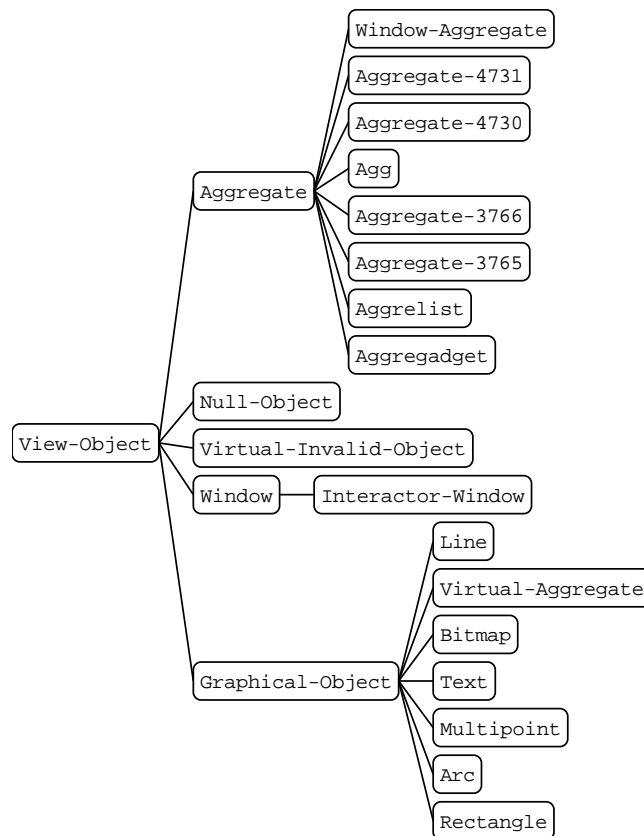
Figure 7.35: Graph generated by SCHEMA-GRAPH

The graph pictured in figure [schema-graph-pix], page 390, is a result of the definition of the SCHEMA-GRAPH object above. The aggregraph was given a description of the Garnet inheritance hierarchy just by defining the root of the graph and a child-generating function.

The generating function in the `:children-function` slot is defined to return the instances of a given schema until the aggregraph reaches a certain depth in the graph. In this case, if the function is given a node that is more than one link away from the root, then the function will return `nil`.

The function in the `:info-function` slot returns the string name of a Garnet schema.

7.13.5 An Example With an Interactor

```
(create-instance 'SCHEMA-GRAPH-2 opal:aggregraph
   (:children-function #'(lambda (source-node depth)
   (when (< depth 1)
     (gv source-node :is-a-inv))))
   (:info-function #'(lambda (source-node)
      (string-capitalize
   (kr:name-for-schema source))))
   (:source-roots (list opal:view-object))
   ; Change the node prototype so that it will go black
   ; when the interactor sets :interim-selected to T
   (:node-prototype
    (create-instance NIL opal:aggregraph-node-prototype
       (:interim-selected NIL)  ; Set by interactor
       (:parts
 `((:box :modify
   (:filling-style ,(o-formula (if (gvl :parent :interim-selected)
   opal:black-fill
   opal:white-fill)))
   (:draw-function :xor) (:fast-redraw-p T))
  :text-al))))
   ; Now define an interactor to work on all nodes of the graph
   (:interactors
    `((:press ,inter:menu-interactor
       (:window ,(o-formula (gv-local :self :operates-on :window)))
       (:start-where ,(o-formula (list :element-of
                                  (gvl :operates-on :nodes))))
       (:final-function
  ,#'(lambda (inter node)
      (let* ((graph (gv node :parent :parent))
     (source-node (gv node :source-node)))
        (format T "~%~% ***** Clicked on ~S *****~%" source-node)
        (kr:ps source-node)))))))))
```
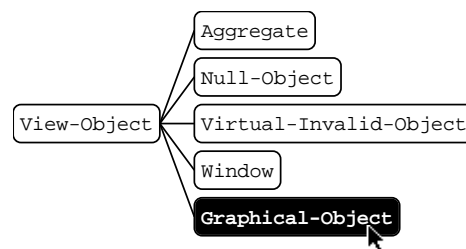
Figure 7.36: Graph generated by SCHEMA-GRAPH-2

The graph of figure [schema-graph-2-pix], page 392, comes from the definition of SCHEMA-GRAPH-2. This aggregraph models the same Garnet hierarchy as in the previous example, but it also modifies the node-prototype for the aggregraph and adds an interactor to operate on the graph.

The `:node-prototype` slot must contain a Garnet object that can be used to display the nodes of the graph. In this case, the customized node-prototype is an instance of the default node-prototype (which is an aggregadget) with some changes in the roundtangle part. The formula for the `:filling-style` will make the node black when the user presses on it with the mouse.

The interactor is defined as in aggregadgets and aggrelists. Note that the `:start-where` slot looks at the components of the `:nodes` aggregate in the top-level aggregraph.

Aggregadget nodes can be moved easily with an `inter:move-grow-interactor`. By setting the `:slots-to-set` slot of the interactor to (`list T T nil nil`), you can change the `:left` and `:top` of the aggregraph nodes as you click and drag on them.

7.14 Aggregraph

Features and operation of an Aggregraph

Creates a graph in which each node determines its own size based on information to be displayed in it. (The information is determined by the function `:info-function`.)

The user must supply a list of source-nodes to be the root of the graph, a children-function which can be used to walk the user's graph, and an info-function to determine what will be displayed in each graph node.

It is an instance of aggregadget, and interactors can be defined as in aggregadgets.

Customizable slots

`:left`, `:top` – The position of the aggregraph. Default is 0,0.

`:source-roots` – List of source nodes to be used as the roots of the graph.

`:children-function` – A function which takes a source node and the depth from the root and returns a list of children. The children are treated as unordered by the default layout-function.

`:info-function` – A function which takes a source node and returns information to be used in the display of the node prototype. The result is put in the `:info` slot of the corresponding graph-node. The default node-prototype expects a string to be returned.

`:add-back-pointer-to-nodes-function` – A function or `nil`. The function, if present, will be called on every source-node graph-node pair. The result of the function is ignored. This allows pointers to be put in the source-nodes for corresponding graph-nodes.

`:node-prototype` – A Garnet object for node prototype, or list of prototypes (in which case a `:node-prototype-selector-function` must be provided– see below). In the instances, the `:info` slot is set with the result of the info-function called on the corresponding source-node. The `:source-node` slot is set to the corresponding source node. And, the `:links-to-me` and `:links-from-me` slots are set to lists of graph links pointing to the node and from the node respectively. The slots whose names begin with `":layout-info"` are reserved for use by the layout functions for internal bookkeeping and so should not be set by the user (unless writing a new layout or associated methods). If any `scalable-aggregraph-image` graphs are made of this graph, the

`:image-nodes` slot is set to a list containing the nodes that correspond to this node. The default prototype expects a string in the `:info` slot, and displays the string with a white-filled roundtangle surrounding it.

`:link-prototype` – Garnet object for link prototype, or list of prototypes (in which case a `:link-prototype-selector-function` must be provided–see below). The `:from` and `:to` slots are set to the graph-nodes that this link connects. The `:image-links` slot is set to a list of corresponding links to this one in associated `scalable-aggregraph-image` graphs. The default prototype is a line between these two graph nodes. (It is connected to the center of the right side of the `:from` node and to the center of the left side of the `:to` node. This assumes a left to right layout of the graph for pleasing display. For other layout strategies, a different prototype may be desired.) For directed graphs, a link prototype with an arrowhead may be desired.

`:node-prototype-selector-function` – A function which takes a source node and the list of prototypes provided in the `:node-prototype` slot and returns one of the prototypes. Will only be used if the value in the `:node-prototype` slot is a list.

`:link-prototype-selector-function` – A function which takes a "from" graph-node, a "to" graph-node and the list of prototypes provided in the `:link-prototype` slot and returns one of the prototypes. Will only be used if the value in the `:link-prototype` slot is a list.

`:h-spacing` – The minimum distance in pixels between nodes horizontally if using default layout-function. The default value is 20.

`:v-spacing` – The minimum distance in pixels between nodes vertically if using default layout-function. The default value is 5.

`:test-to-distinguish-source-nodes` – Must be one of `#'eq`, `#'eql`, or `#'equal`. The default is `#'eql`.

`:interactors` – Specified in the same format as aggregadgets.

`:layout-info-...` – Several slots that begin with `":layout-info-"` are reserved for bookkeeping by the layout function. Do not set these slots except as part of a customized layout function.

Read-only slots

`:nodes` – The aggregate which contains all of the graph-node objects.

`:links` – The aggregate which contains all of the graph link objects.

`:graph-roots` – The list of graph nodes corresponding to the `:source-roots`.

`:image-graphs` – The list of `scalable-aggregraph-image` graphs that are images of this graph.

[Methods] (can be overridden)

`:layout-graph` – a function which is called to determine the locations for all of the nodes in the graph. Takes the graph object as input and sets appropriate slots in each node to position the node (usually `:left` and `:top` slots.) Automatically called when graph is initially created.

:delete-node – Takes the graph object and a graph node and deletes it and all links attached to it. If a node is deleted that is a root of the graph, then it is removed from :graph-roots and the corresponding source-node is removed from :source-roots.

:add-node – The arguments are the graph object, a source-node, a list of parent graph-nodes, and a list of children graph-nodes. It creates a new graph node and places it in the graph positioning it appropriately. Returns nil.

:delete-link – Takes the graph object and a graph link and removes the link from the graph.

:add-link – Takes the graph object and two graph nodes and creates a link from the first node to the second.

:source-to-graph-node – Takes the graph object and a source node and returns the corresponding graph-node.

:find-link – Takes the graph object and two graph-nodes and returns the list of graph link-objects from the first to the second.

:make-root – Takes the graph object and a graph node of the graph and adds the graph node to the root lists of the graph.

:remove-root – Takes the graph object and a graph node and removes the node from the root lists of the graph.

Note that these eight methods depend on each other for intelligent layout. If one is changed it will either have to keep certain bookkeeping information, or other functions will have to be changed as well.

The functions which add and delete nodes and links all attempt to minimally change the graph. The relayout function may dramatically change it.

7.15 Scalable Aggregraph

Features and operation of a Scalable Aggregraph

This object is similar to the normal aggregraph except that it can be scaled by the user. Text will be displayed only if it will fit within the scaled size of the graph nodes with the default prototypes. The scale factor is set by the :scale-factor slot.

The scalable aggregraph will automatically resize if the :scale-factor slot is changed.

It is an instance of aggregadget, and interactors can be defined as in aggregadgets.

[Customizable slots] (same as for aggregraphs except for the following):

:scale-factor – A multiplier of full size which determines the final size of the graph (e.g. 1 causes the graph to be full size, 0.5 causes the graph to be half of full size, etc.) The full size of the graph is determined by the size of the node prototypes and layout of the nodes.

:node-prototype – Must be able to set the :width and :height, otherwise the same as in aggregraph. These slots must have initial values which will be used as their default value (i.e. the width and height of the nodes is :scale-factor * the values in :height and :width slots respectively.

:`link-prototype` – Position and size must depend on the nodes it is attached to (by the :`from` and :`to` slots) with formulas.

:`h-spacing` and :`v-spacing` are the default values. The actual values are :`scale-factor` * these values.

Read-only slots

The same read-only slots are available as with `aggregraph` (see section [aggregraph-slots], page 393).

Methods (can be overridden)

The same methods are available as with `aggregraph` (see section [aggregraph-slots], page 393).

7.16 Scalable Aggregraph Image

Features and operation of Scalable Aggregraph Image

This is designed to show another view of an existing aggregraph. This image is created with the same shape as the original, i.e. the size of nodes and relative positions are in proportion to the original. The proportion is determined by the :`scale-factor` or :`desired-height` and :`desired-width` slots.

The size and shape are determined by Garnet formulas. This has the effect of maintaining the likeness to the original even as the original is manipulated and changed.

The default prototypes (in particular the node prototype), are designed for the image to be used as an overview of a graph which perhaps doesn't fit on the screen. This is why no text is displayed in nodes, for example. This is not the only use of the gadget, especially if the prototypes are changed.

Customizable slots

:`left`, :`top` – The position of the aggregraph. Default is 0,0.

:`desired-width` and :`desired-height` – Desired width and height of the entire graph. The graph will be scaled to fit inside these maximums.

:`source-aggregraph` – The aggregraph to make an image of.

:`scale-factor` – A multiplier of full size which determines the final size of the graph (e.g. 1 causes the graph to be the same size as the source aggregraph, 0.5 causes the graph to be half the size, etc.) Scale-factor overrides the :`desired-width` and :`desired-height` slots if all are specified. The default value is 1.

:`node-prototype` – Garnet object for node prototype, or a list of prototypes (in which case a :`node-prototype-selector-function` must be provided–see below). The :`width`, :`height`, :`left` and :`top` slots must all be settable, and the node size and position must depend on these slots. They will all be overridden with formulas in the created instances. The :`corresponding-node` slot is set to the corresponding node in the source aggregraph. The default node-prototype is a roundtangle proportional to the bounding box of the corresponding node in the source aggregraph (because of the formulas).

:link-prototype – Same as in aggregraph, except the :corresponding-link slot is set to the corresponding link in the source aggregraph and there is no :from or :to slot. The :x1, :y1, :x2 and :y2 slots of the link must all be settable, and the link endpoints must depend on their values. They will all be overridden with formulas in the created instances. The default link-prototype is a line.

:node-prototype-selector-function – A function which takes the appropriate corresponding-node and the list of prototypes provided in the :node-prototype slot and returns one of the prototypes. Will only be used if the value in the :node-prototype slot is a list.

:link-prototype-selector-function – A function which takes the corresponding-link and the list of prototypes provided in the :link-prototype slot and returns one of the prototypes. Will only be used if the value in the :link-prototype slot is a list.

:interactors – Specified in the same format as aggregadgets.

Read-only slots

The same read-only slots are available as with `aggregraph` except :graph-roots (see section [aggregraph-slots], page 393).

Methods (probably shouldn't be overridden)

The methods of a `scalable-aggregraph-image` call the methods of the source aggregraph, and changes are reflected in the image. If the methods of the source graph are called directly, the changes will also be reflected.

When this aggregraph image is created, pointers are created in the source aggregraph and all of its nodes and links to the corresponding image graph, nodes and links. These pointers are added to a list in the slot :image-graphs, :image-nodes and :image-links of the aggregraph, nodes and links. Pointers from the image to the source are in the slots :source-aggregraph, :source-node and :source-link as indicated below. These links are used by the methods (both in this gadget and in the two gadgets described above) to maintain the image.

7.17 Customizing the :layout-graph Function

NOTE: Writing a customized layout function is a formidable task that few users will want to try. This section is provided for programmers whose aggregraph application requires a graph layout that is not suited to the default tree layout function.

The function stored in :layout-graph computes the locations for all of the nodes and links of the graph. It takes the graph as its argument, and the returned value is ignored. All nodes have been created with their height and width, and are connected to the appropriate links and nodes, before the function is called.

The default layout function is `layout-tree` defined in `Opal`. This function can be called repeatedly on the graph, but may drastically change the look of the graph (if a series of adds and deletes were done before the relayout). Features of `layout-tree` are:

It works best for trees and DAGs which are tree-like (i.e. DAGs in which the width becomes larger toward the leaves).

It takes linear time in the number of nodes.

Children are treated as unordered.

Add and delete (both nodes and links) attempt to minimally change the graph.

If a new layout function is written without regard to the bookkeeping slots or the various methods associated with the aggregraph, the other methods will work with the new layout function but will probably not keep the graph looking as nice as possible.

With the default link prototypes it is only necessary to place the nodes, because the links attach to the nodes automatically with Garnet formulas. (Note that the default links are designed for a left to right layout of the graph. If a different layout is desired another prototype may be desired. Of course, formulas can still be used rather than explicitly placing each link.)

In general, the other graph methods may need to maintain or use the same bookkeeping information as the layout function. For example, `add-node` and `delete-node` both affect the `":layout-info-"` slots used by the default layout function. (Specifically they add or delete rectangles respectively from the object stored in the `:layout-info-rect-conflict-object` slot of the graph object. This object keeps track of all rectangles (nodes) placed on the graph and when queried with a new rectangle returns any stored rectangles that it overlaps.) When redefining the layout function, it may be necessary to redefine these functions.

8 Garnet Gadgets

by Andrew Mickish, Brad A. Myers, Rajan Parthasarathy,

10 May 2020

8.1 Abstract

The Garnet Gadget Set contains common user interface objects which can be customized for use in an interface. Because the objects are extremely versatile, they may be employed in a wide range of applications with a minimum of modification. Examples of provided gadgets include menus, buttons, scroll bars, sliders, and gauges.

8.2 Introduction

Many user interfaces that span a wide variety of applications usually have several elements in common. Menus and scroll bars, for example, are used so frequently that an interface designer would waste considerable time and effort recreating those objects each time they were required in an application.

The intent of the Garnet Gadget Set is to supply several frequently used objects that can be easily customized by the designer. By importing these pre-constructed objects into a larger Garnet interface, the designer is able to specify in detail the desired appearance and behavior of the interface, while avoiding the programming that this specification would otherwise entail.

This document is a guide to using the Gadget Set. The objects were constructed using the complete Garnet system, and their descriptions assume that the reader has some knowledge of KR, Opal, Interactors, and Aggregadgets.

8.3 Current Gadgets

Most of the gadgets described in this chapter are pictured in figures [scroll-group], page 400, through [db-group], page 406,

(a)   (b)

(c)
Gauge

(d)

(e)

(f)
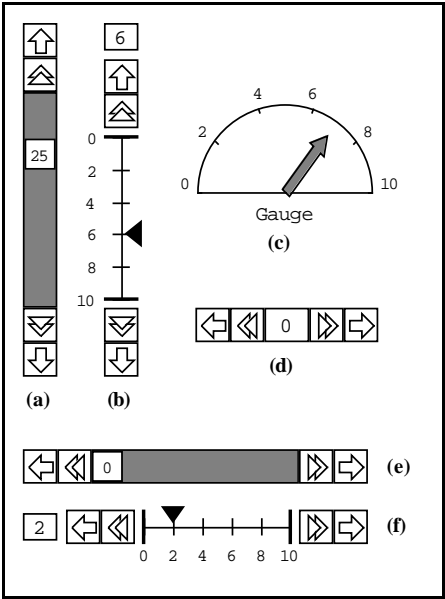
0

Gadgets used to choose a value from a range of values

`v-scroll-bar` - Vertical scroll bar (p. [scroll-bars], page 416)

`v-slider` - Vertical slider (same idea as a scroll bar, but with a tic-marked shaft rather than a rectangular bounding box) (p. [sliders], page 419)

`gauge` - Semi-circular gauge (the needle on the gauge may be moved to select a value) (p. [gauge], page 427)

`trill-device` - Number input box with increment/decrement trill boxes (p. [trill-device], page 423)

`h-scroll-bar` - Horizontal scroll bar (p. [scroll-bars], page 416)

`h-slider` - Horizontal slider (p. [sliders], page 419)

`motif-v-scroll-bar` - Vertical scroll bar (p. [motif-scroll-bars], page 522)

`motif-slider` - Vertical slider (same idea as a scroll bar, but with text beside the indicator showing the current value) (p. ⟨undefined⟩ [motif-slider], page ⟨undefined⟩)

`motif-gauge` - Semi-circular gauge (p. [motif-gauge], page 529)

`motif-trill-device` - Number input with trill boxes (p. [motif-trill-device], page 528)

`motif-h-scroll-bar` - Horizontal scroll bar (p. [motif-scroll-bars], page 522)
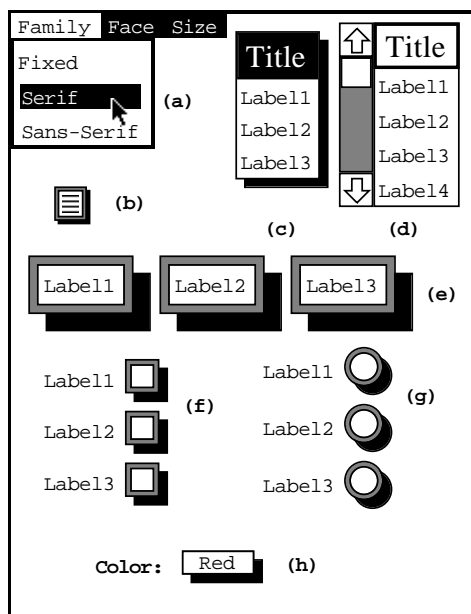
Gadgets used to choose items from a list of possible choices

`menubar` - A pull-down menu (p. [menubar], page 447)

`popup-menu-button` - A button which pops up a menu when pressed. The appearance of the button does not change with the selection. (p. [popup-menu-button], page 438)

`menu` - Vertical menu, single selection (p. [menu], page 441)

`scrolling-menu` - A menu with a scroll bar on one side, which allows a subset of all items in the menu to be viewed. (single or multiple selection) (p. [scrolling-menu], page 444)

`text-buttons` - A panel of rectangular buttons, each with a choice centered inside the button. As an option, the currently selected choice may appear in inverse video. (single selection) (p. [buttons], page 429, and [text-buttons], page 431)

`x-buttons` - A panel of square buttons, each with a choice beside the button. An "X" appears inside each currently selected button. (multiple selection) (p. [buttons], page 429, and [x-buttons], page 432)

`radio-buttons` - A panel of circular buttons, each with a choice beside the button. A black circle appears inside the currently selected button. (single selection) (p. [buttons], page 429, and [radio-buttons], page 434)

`option-button` - A button which pops up a menu when pressed. Selection of a choice from the menu causes that item to appear as the new label of the button. (p. [option-button], page 435)

`motif-menubar` - A pull-down menu. (p. [motif-menubar], page 547)

`motif-menu` - Vertical menu, single selection (p. [motif-menu], page 540)

`motif-scrolling-menu` - A menu with an attached scroll bar. (p. [motif-scrolling-menu], page 544)

`motif-text-buttons` - A panel of rectangular buttons, each with a choice appearing inside the button. (single selection) (p. [motif-buttons], page 533, and [motif-text-buttons], page 534)

`motif-check-buttons` - A panel of square buttons, each with a choice beside the buttons. (multiple selection) (p. [motif-buttons], page 533, and [motif-check-buttons], page 536)

`motif-radio-buttons` - A panel of diamond buttons, each with a choice beside the button. (single selection) (p. [motif-buttons], page 533, and [motif-radio-buttons], page 537)

`motif-option-button` - A button which pops up a menu when pressed. Selection of a choice from the menu causes that item to appear as the new label of the button. (p. [motif-option-button], page 538)

Title: | Labeled-box | **(a)**   Scrolling-input-str …   **(c)**

Title: | Scrolling-labele … | **(b)**   **Label:** | otif-scrolling-l | **(d)**

Gadgets used to handle text input

`labeled-box` - A framed text object that may be edited. As the string gets longer, the frame expands. (p. [labeled-box], page 456)

`scrolling-labeled-box` - A scrolling input string in a box with a label. The frame stays fixed, and the string scrolls. (p. [scrolling-labeled-box], page 459)

`scrolling-input-string` - Input a text string, but using a fixed width area and scroll the string horizontally if necessary. (p. [scrolling-input-string], page 457)

`motif-scrolling-labeled-box` - A labeled box with text inside that may be edited. (p. [motif-scrolling-labeled-box], page 555)

`multifont-gadget` - A text editing gadget that includes word wrap, text selection, and many functions that allow manipulation of the text. This gadget is discussed in the Opal chapter.
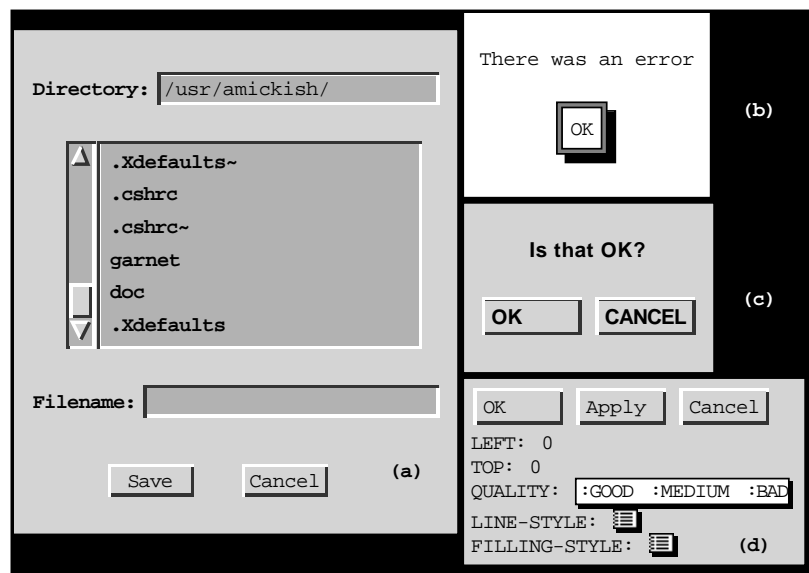
Figure 8.4: Garnet dialog boxes. **(a)** `motif-save-gadget`, **(b)** `error-gadget`,
(c) `motif-query-gadget`, **(d)** `motif-prop-sheet-with-OK`

Dialog boxes for reading and writing to files (the `motif-save-gadget` is pictured in
figure [db-group], page 406)

`save-gadget` - Saves a file in a directory whose contents are displayed in a scrolling menu. (p. [save-gadget], page 492)

`load-gadget` - Loads a file from a directory whose contents are displayed in a scrolling menu. (p. [load-gadget], page 498)

`motif-save-gadget` - Saves a file in a directory whose contents are displayed in a Motif style scrolling menu. (p. [motif-save-gadget], page 559)

`motif-load-gadget` - Loads a file from a directory whose contents are displayed in a Motif style scrolling menu. (p. [motif-load-gadget], page 560)

Dialog boxes for reporting errors to the user and asking for user input (the `error-gadget` and `motif-query-gadget` are pictured in figure [db-group], page 406).

`error-gadget` - Used to display error messages in a window with an "OK" button (p. [error-gadget], page 488)

`query-gadget` - A dialog box like the `error-gadget`, but with multiple buttons and the ability to return values. (p. [query-gadget], page 492)

`motif-error-gadget` - A dialog box used to display error messages with an "OK" button in the Motif style. (p. [motif-error-gadget], page 557)

`motif-query-gadget` - A Motif style dialog box with multiple buttons. (p. [motif-query-gadget], page 558)

Property sheet gadgets (a Motif property sheet is pictured in figure [db-group], page 406)

`prop-sheet` - Displays a set of labels and values and allows the values to be edited. This gadget can be easily displayed in its own window. (p. [propertysheets], page 499)

`prop-sheet-for-obj` - A property sheet designed to display the slots in a Garnet object. (p. [propsheetforobj], page 503)

`prop-sheet-with-OK` - A property sheet with OK-Cancel buttons. (p. [propsheetwithok], page 508)

`prop-sheet-for-obj-with-OK` - A property sheet designed to display the slots in a Garnet object with attached OK-Cancel buttons. (p. [propsheetforobjwithok], page 509)

`motif-prop-sheet-with-OK` - A property sheet with OK-Cancel buttons in the Motif style. (p. [motif-prop-sheets], page 561)

`motif-propt-sheet-for-obj-with-OK` - A Motif style property sheet designed to display the slots in a Garnet object with attached OK-Cancel buttons. (p. [motif-prop-sheet-for-obj-with-ok], page 562)

Scrolling windows

`scrolling-window` - Supports a scrollable window (p. [scrolling-windows], page 471)

`scrolling-window-with-bars` - Scrolling window complete with scroll bars. (p. [scrolling-windows], page 471)

`motif-scrolling-window-with-bars` - Motif style scrolling window (p. [motif-scrolling-window], page 566)

Special gadgets

> `arrow-line` - A line with an arrowhead at one end (p. [arrow-line], page 478)
>
> `double-arrow-line` - A line with arrowheads at both ends (p. [double-arrow-line], page 478)
>
> `browser-gadget` - Used to examine structures and hierarchies (p. [browser-gadget], page 479)
>
> `graphics-selection` - Bounding boxes and interactors to move and change the size of other graphical objects. (p. [graphics-selection], page 460)
>
> `multi-graphics-selection` - Same as `graphics-selection`, but for multiple objects. (p. [multi-gs], page 464)
>
> `polyline-creator` - For creating and editing polylines. (p. [polyline-creator], page 485)
>
> `MouseLine` and `MouseLinePopup` - A gadget that pops up a "help" string, informing the user about the object that the mouse is held over.
>
> `standard-edit` - A module of predefined "cut" and "paste" procedures, and many other common editing functions. (p. [standardeditsec], page 514)

8.4 Customization

is the ability to create a variety of interface styles from a small collection of prototype objects. Each gadget includes many parameters which may be customized by the designer, providing a great deal of flexibility in the behavior of the gadgets. The designer may, however, choose to leave many of the default values unchanged, while modifying only those parameters that integrate the object into the larger user interface.

The location, size, functionality, etc., of a gadget is determined by the values in each of its slots. When instances of gadgets are created, the instances inherit all of the slots and slot values from the prototype object except those slots which are specifically assigned values by the designer. The slot values in the prototype can thus be considered "default" values for the instances, which may be overridden when instances are created.[1] The designer may also add new slots not defined in the gadget prototype for use by special applications in the larger interface. Slot values may be changed after the instances are created by using the KR function `s-value`.

8.5 Using Gadget Objects

The gadget objects reside in the `GARNET-GADGETS` package, which has the nickname "GG". We recommend that programmers explicitly reference the name of the package when creating instances of the gadgets, as in `garnet-gadgets:v-scroll-bar` or `gg:v-scroll-bar`. However, the package name may be dropped if the line (`use-package "GARNET-GADGETS"`) is executed before referring to gadget objects.

---

[1] See the KR chapter for a more detailed discussion of inheritance.

Before creating instances of gadget objects, a set of component modules must be loaded. These modules are loaded in the correct order when the "-loader" files corresponding to the desired gadgets are used (see Chapter [accessing], page 413).

Since each top-level object is exported from the `GARNET-GADGETS` package, creating instances of gadget objects is as easy as instantiating any other Garnet objects. To use a gadget, an instance of the prototype must be defined and added to an interactor window. The following lines will display a vertical scroll bar in a window:

```
(create-instance 'MY-WIN inter:interactor-window
    (:left 0) (:top 0) (:width 300) (:height 500))
(create-instance 'MY-AGG opal:aggregate)
(s-value my-win :aggregate my-agg)
(create-instance 'MY-SCROLL-BAR garnet-gadgets:v-scroll-bar)
(opal:add-component my-agg my-scroll-bar)
(opal:update my-win)
```

interactor window named `my-win` and an aggregate named `my-agg`. The third instruction sets the `:aggregate` slot of `my-win` to `my-agg`, so that all graphical objects attached to `my-agg` will be shown in `my-win`. The next two instructions create an instance of the `v-scroll-bar` object named `my-scroll-bar` and add it as a component of `my-agg`. The last instruction causes `my-win` to become visible with `my-scroll-bar` inside.

In most cases, the use of a gadget will follow the same form as the preceding example. The important difference will be in the instantiation of the gadget object (the fifth instruction above), where slots may be given values that override the default values defined in the gadget prototype. The following example illustrates such a customization of the vertical scroll bar.

Suppose that we would like to create a vertical scroll bar whose values span the interval [0..30], with its upper-left coordinate at (25,50). This vertical scroll bar may be created by:

```
(create-instance 'CUSTOM-BAR garnet-gadgets:v-scroll-bar
    (:left 25)
    (:top 50)
    (:val-1 0)
    (:val-2 30))
```

CUSTOM-BAR which is an instance of `v-scroll-bar`. The vertical scroll bar CUSTOM-BAR has inherited all of the slots that were declared in the `v-scroll-bar` prototype along with their default values, except for the coordinate and range values which have been specified in this schema definition (see section [scroll-bars], page 416, for a list of customizable slots in the scroll bar objects).

8.6 Application Interface

There are several ways that the gadgets can interface with your application. This section describes several ways the you can get the gadgets to "do something" to your application.

8.6.1 The :value slot

In most gadgets, there is a top-level `:value` slot. This slot is updated automatically after a user changes the value or position of some part of the gadget. This is therefore the main slot through which the designer perceives action on the part of the user.

The `:value` slot may be accessed directly (by the KR functions `gv` and `gvl`) order to make other objects in the larger interface dependent on the actions of the user. The slot may also be set directly by the KR function `s-value` to change the current value or selection displayed by the gadget (except in the scrolling menu gadget, where the `:selected-ranks` slot must be set).

An instance of a gadget can be given initial values by setting the `:value` slot after the instance has been created. In most gadgets, this slot may **not** be given a value in the `create-instance` call, since this would override the formula in the slot. Therefore, the general procedure for selecting an initial value in a gadget is to create the instance, access the `:value` slot using `gv` (to initialize the formula in the slot and establish dependencies), and then use `s-value` to set the slot to the desired initial value.

See sections [use-value], page 569, and [sel-buttons], page 570, for examples of the `:value` slot in use.

8.6.2 The :selection-function slot

In most gadgets there is a `:selection-function` slot which holds the name of a function to be called whenever the `:value` slot changes due to action by the user (such as the pressing of a button). The `:selection-function` is not automatically called when the designer's interface sets the `:value` slot directly.

This is probably the most important link between the gadgets and your application. By supplying a gadget with a selection function, then the gadget can execute some application-specific procedure when the user operates it.

In the scroll bars, sliders, trill device, and gauge, this function is called after the user changes the value by moving the indicator or typing in a new value (the function is called repeatedly while the user drags an indicator). In buttons and menus, it is called when the user changes the currently selected item or set of items, and it precedes the function attached locally to the item. In the labeled box, scrolling-input-string and scrolling-labeled-box, it is called after the user has finished editing the text (i.e., after a carriage return). In the `:graphics-selection` gadget, it is called whenever the user selects a new object or deselects the current object.

In the scrolling menu gadget, there are two selection functions, named `:scroll-selection-function` and `:menu-selection-function` which are called independently when the user moves the scroll bar or selects a menu item, respectively.

The function must take two parameters: the top-level gadget itself and the value of the top-level `:value` slot:

```
(lambda (gadget-object value))
```

In x-buttons, the parameter `value` will be a list of strings. The scrolling menu sends the menu item (a Garnet schema) on which the user just clicked as its `value`. Other gadgets will have only a single number or string as their `value`.

An example use of `:selection-function` is in section [use-selection], page 569.

8.6.3 The :items slot

The button and menu gadgets are built up from items supplied by the designer. These items are supplied as a list in the `:items` slot of the gadgets. **Note:** Do not destructively

modify the :items list; instead, create a new list using list or copy the old value with copy-list and modify the copy.

8.6.4 Item functions

There are several ways to specify items:

List of strings - This is the obvious case, such as '("Open" "Close" "Erase").

List of atoms - In Garnet, the values of slots are often specified by atoms – symbols preceded by a colon (e.g., :center). If a formula in the larger interface depends upon the :value slot of the button panel, then the designer may wish the items to be actual atoms rather than strings, so that the value is immediately used without being coerced. Such a list would look like '(:left :center :right). The items will appear to the user as capitalized strings without colons.

List of objects - In addition to string labels, the gadgets can have labels that are objects (like circles and rectangles). Such a list might look like '(,MY-CIRCLE ,MY-SQUARE ,OBJ3). Objects, strings, and atoms can be mixed together in any :items list. Most of the demo functions for the gadgets use at least one object in the example.

List of label/function pairs - This mode is useful when the designer wishes to execute a specific function upon selection of a button. If the :items slot contained the list '(("Cut" My-Cut) ("Paste" My-Paste)), then the function My-Cut would be executed when the button labeled "Cut" becomes selected. The designer must define these functions with two parameters:

```
(lambda (gadget-object item-string))
```

The *gadget-object* is the top-level gadget (such as a text-button-panel) and the *item-string* is the string (or atom) of the item that was just selected.

The item functions are executed along with the selection function whenever the user operates the gadget. These functions are different, however, because the selection function is executed when **any** item is selected, and the item functions are only executed when the item associated with them is selected.

The gadgets always assume that if an element of the :items list is a list, then the first element in the item is a label and the second element is a function. If you intend to use the :items list for storing application-specific data, you should avoid storing data in these reserved positions of the item elements. It is fine to store arbitrary data in the third and subsequent elements of an item list.

Section [use-item-fn], page 570, shows an example implementation of item functions.

8.6.5 Adding and removing items

There are two ways to add and remove items from a button or menu gadget: use add-item and remove-item to change the :items list, or set the :items slot by hand using s-value. Both ways to change items are shown in the example below.

The various methods for changing items are

opal:Add-Item *gadget item* [[:where] *position[locator]* [:key *function-name*]]⟨un■ defined⟩ [method], page ⟨undefined⟩

opal:Remove-Item *gadget* [*item* [:key *function-name*]]⟨undefined⟩ [method],
page ⟨undefined⟩

opal:Remove-Nth-Item *gadget n*⟨undefined⟩ [method], page ⟨undefined⟩

opal:Change-Item *gadget item n*⟨undefined⟩ [method], page ⟨undefined⟩

These methods are described in the Aggregadgets chapter. `Add-item` will add *item* to
the `:items` list of *gadget*, and will place it in the list according to the *position*, *locator*, and
key parameters.

All gadgets that have an `:items` slot support `add-item` and the other methods (except
for the `browser-gadget`, which has other item maintenance functions). The documentation
for the `menubar` and `motif-menu` describes special features supported by those gadgets.

For example, consider adding an item to the X-BUTTONS-OBJ in the `x-button-panel`
demo.

```
; Use opal:add-item in one step
(opal:add-item gg:X-BUTTONS-OBJ "newitem-1")

; Use s-value (directly or indirectly)
(push "newitem-2" (gv gg:X-BUTTONS-OBJ :items))
```

The `push` function uses `s-value` indirectly. `S-value` may also be used explicitly. After
changing the `:items` list with `s-value`, the components of the gadget (like the individual
buttons in a button panel) will be adjusted during the next call to `opal:update`. If infor-
mation about the gadget (like its new dimensions) is required *before* the next update, the
components can be adjusted chapterly with a call to `opal:notice-items-changed` with
the gadget as a parameter. See the Aggregadgets Chapter for more information about
`opal:notice-items-changed`.

Because of internal references to the `:items` slot, destructive modification of the `:items`
list is not allowed. If you change the list in the `:items` slot, you should create a new list
(e.g., with `list`), or use `copy-list` on the original, and destructively modify the copy.

8.7 Constants with the Gadgets

At the top of most gadget definitions, there is a slot called `:maybe-constant` with a list
of slots as its value. These are the slots that will be declared constant in an instance of a
gadget, if the instance was created with its `:constant` slot set to T. By declaring a slot
constant, the user promises that the value of that slot will never change, and all formulas
that depend on it can be thrown away and replaced by absolute values.

Removing formulas that depend on constant slots can free up a large amount of storage
space. Therefore, users who have finished designing part of an interface may want to go
back through their gadget instances and delclare constant as many slots as possible.

In addition to using the special T value in a `:constant` list, you can selectively declare
slots constant by listing them explicitly (e.g., `(:constant '(:left :top))`). You can also
use the `:except` keyword, as in the following schema:

```
(create-instance NIL gg:motif-radio-button-panel
   (:constant '(T :except :active-p))
```

```
(:left 10)(:top 30)
(:items '("Start" "Pause" "Quit")))
```

In this example, the user declares constant all of the slots in the `:maybe-constant` list, with the exception of `:active-p`. This allows the value of the `:active-p` slot to change, and retains all the formulas that depend on it (so that the gadget will update its appearance correctly when the value is toggled).

Constants are discussed in detail in the KR chapter.

8.8 Accessing the Gadgets

8.9 Gadgets Modules

package are modularized so that one schema may be used by several objects. For example, trill boxes with arrows pointing to the left and right are used in the horizontal scroll bar, the horizontal slider, and the trill device. As a result, all of the code for the gadget objects has a consistent style, and the gadgets themselves have a uniform look and feel.

8.10 Loading the Gadgets

Since much of the gadget code is shared by the top-level objects, a set of "parts" modules must be loaded before some of the top-level gadgets. The required modules are loaded in the proper order when the loader files corresponding to the desired gadgets are used. The standard gadgets and their associated loader files are listed in figure [loader-files-figure], page 414. The motif gadgets and loader files appear in figure [motif-loader-files-figure], page 415. It is safe to load the "xxx-loader" files multiple times, they will not re-load the objects the second time.

`arrow-line` - "arrow-line-loader"

`browser-gadget` - "browser-gadget-loader"

`double-arrow-line` - "arrow-line-loader"

`error-gadget` - "error-gadget-loader"

`gauge` - "gauge-loader"

`graphics-selection` - "graphics-loader"

`h-scroll-bar` - "h-scroll-loader"

`h-slider` - "h-slider-loader"

`labeled-box` - "labeled-box-loader"

`load-gadget` - "save-gadget-loader"

`menu` - "menu-loader"

`menubar` - "menubar-loader"

`MouseLine` and `MouseLinePopup` - "mouseline-loader"

`multifont-gadget` - "multifont-loader"

`multi-graphics-selection` - "multi-selection-loader"

`option-button` - "option-button-loader"

`popup-menu-button` - "popup-menu-button-loader"

`prop-sheet` - "prop-sheet-loader"

`prop-sheet-for-obj` - "prop-sheet-loader"

`prop-sheet-for-obj-with-OK` - "prop-sheet-win-loader"

`prop-sheet-with-OK` - "prop-sheet-win-loader"

`query-gadget` - "error-gadget-loader"

`radio-button` - "radio-buttons-loader"

`radio-button-panel` - "radio-buttons-loader"

`save-gadget` - "save-gadget-loader"

`scrolling-input-string` - "scrolling-input-string-loader"

`scrolling-labeled-box` - "scrolling-labeled-box-loader".

`scrolling-menu` - "scrolling-menu-loader"

`scrolling-window` - "scrolling-window-loader"

`scrolling-window-with-bars` - "scrolling-window-loader"

`standard-edit` - "standard-edit-loader"

`text-button` - "text-buttons-loader"

`text-button-panel` - "text-buttons-loader"

`trill-device` - "trill-device-loader"

`v-scroll-bar` - "v-scroll-loader"

`v-slider` - "v-slider-loader"

`x-button` - "x-buttons-loader"

`x-button-panel` - "x-buttons-loader"

Figure 8.5: Loader files for Garnet Gadgets

```
motif-check-button
```
 - "motif-check-buttons-loader"

```
motif-check-button-panel
```
 - "motif-check-buttons-loader"

```
motif-error-gadget
```
 - "motif-error-gadget-loader"

```
motif-gauge
```
 - "motif-gauge-loader"

```
motif-h-scroll-bar
```
 - "motif-h-scroll-loader"

```
motif-load-gadget
```
 - "motif-save-gadget-loader")

```
motif-menu
```
 - "motif-menu-loader"

```
motif-menubar
```
 - "motif-menubar-loader"

```
motif-option-button
```
 - "motif-option-button-loader"

```
motif-prop-sheet-...
```
 - "motif-prop-sheet-win-loader"

```
motif-query-gadget
```
 - "motif-error-gadget-loader"

```
motif-radio-button
```
 - "motif-radio-buttons-loader"

```
motif-radio-button-panel
```
 - "motif-radio-buttons-loader"

```
motif-save-gadget
```
 - "motif-save-gadget-loader"

```
motif-scrolling-labeled-box
```
 - "motif-scrolling-labeled-box-loader"

```
motif-scrolling-menu
```
 - "motif-scrolling-menu-loader"

```
motif-scrolling-window-with-bars
```
 - "motif-scrolling-window-loader"

```
motif-slider
```
 - "motif-slider"

```
motif-text-button
```
 - "motif-text-buttons-loader"

```
motif-text-button-panel
```
 - "motif-text-buttons-loader"

```
motif-trill-device
```
 - "motif-trill-device-loader"

```
motif-v-scroll-bar
```
 - "motif-v-scroll-loader"

Figure 8.6: Loader files for Motif Gadgets

To load the entire Gadget Set, execute (`load Garnet-Gadgets-Loader`) after loading the `Garnet-Loader`. *This is not recommended, since there are so many gadgets, and you will only need a few of them!* To load particular objects, such as the `v-slider` and `menu` gadgets, load the specific loader files:

```
(garnet-load "gadgets:v-slider-loader")
(garnet-load "gadgets:menu-loader")
```

For a discussion of the `garnet-load` function, see the Overview at the beginning of this reference chapter.

8.11 Gadget Files

There are several gadgets files that normally have names that are longer than 31 characters. Since the Mac restricts the length of filenames to 31 characters, some gadget files have their names truncated on the Mac. Mac users may continue to specify the full-length names of these files by using `user::garnet-load`, described in the Overview section of this chapter, which translates the regular names of the gadgets into their truncated 31-character names so they can be loaded. It is recommended that `garnet-load` be used whenever any Garnet

file is loaded, so that typically long and cumbersome pathnames can be abbreviated by a short prefix.

8.12 Gadget Demos

Most gadgets have small demo functions that are loaded along with their schema definitions.[2] For example, after loading the `"v-slider-loader"`, you can do `gg:v-slider-go` to see a demo of the vertical slider.

A complete list of all gadget demos is included in the Demonstration Programs section of this reference chapter. The names of all gadget demos are also mentioned at the top of each section in this Gadget chapter.

8.13 The Standard Gadget Objects

Each of the objects in the Gadget Set is an interface mechanism through which the designer obtains chosen values from the user. The scroll bars, sliders, gauge, and trill device all have a "continuous" flavor, and are used to obtain values between maximum and minimum allowed values. The buttons and menus are more "discrete", and allow the selection of a single choice from several alternatives.

The sections of this chapter describe the gadgets in detail. Each object contains many customizable slots, but the designer may choose to ignore most of them in any given application. If slot values are not specified when instances are created, then the default values will be used.

Each description begins with a list of the customizable slots and default values for the gadget object.

8.14 Scroll Bars

```
(create-instance 'gg:V-Scroll-Bar opal:aggregadget
  (:maybe-constant '(:left :top :height :min-width :val-1 :val-2 :scr-trill-p
                     :page-trill-p :indicator-text-p :page-incr :scr-incr
                     :int-feedback-p :scroll-p :format-string :indicator-font
                     :visible))
  (:left 0)
  (:top 0)
  (:height 250)
  (:min-width 20)
  (:val-1 0)
  (:val-2 100)
  (:scr-incr 1)
  (:page-incr 5)
  (:scr-trill-p T)
  (:page-trill-p T)
  (:indicator-text-p T)
  (:int-feedback-p T)
```

---

[2] Unless the `:garnet-debug` key was removed from from the `*features*` list when the Garnet software was compiled or loaded (see the Hints chapter).

```
      (:scroll-p T)
      (:indicator-font (opal:get-standard-font :fixed :roman :small))
      (:value (o-formula ...))
      (:format-string "~a")
      (:selection-function NIL)    ; (lambda (gadget value))
      )

  (create-instance 'gg:H-Scroll-Bar opal:aggregadget
      (:maybe-constant '(:left :top :width :min-height :val-1 :val-2 :scr-trill-p█
                         :page-trill-p :indicator-text-p :page-incr :scr-incr█
                         :int-feedback-p :scroll-p :format-string :indicator-font :visibl
      (:left 0)
      (:top 0)
      (:width 250)
      (:min-height 20)
      (:val-1 0)
      (:val-2 100)
      (:scr-incr 1)
      (:page-incr 5)
      (:scr-trill-p T)
      (:page-trill-p T)
      (:indicator-text-p T)
      (:int-feedback-p T)
      (:scroll-p T)
      (:indicator-font (create-instance NIL opal:font (:size :small)))
      (:value (o-formula ...))
      (:format-string "~a")
      (:selection-function NIL)    ; (lambda (gadget value))
      )
```
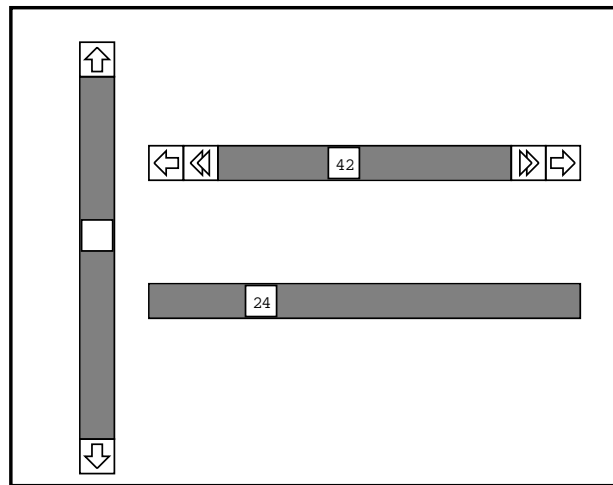
Figure 8.7: Vertical and horizontal scroll bars

The loader file for the `v-scroll-bar` is "v-scroll-loader". The loader file for the `h-scroll-bar` is "h-scroll-loader".

The scroll bar is a common interface object used to specify a desired position somewhere in a range of possible values. The distance of the indicator from the top and bottom of its bounding box is a graphical representation of the currently chosen value, relative to the minimum and maximum allowed values.

The scroll bars in the Gadget Set, `v-scroll-bar` and `h-scroll-bar`, allow the interface designer to specify the minimum and maximum values of a range, while the `:value` slot is a report of the currently chosen value in the range. The interval is determined by the values in `:val-1` and `:val-2`, and either slot may be the minimum or maximum of the range. The value in `:val-1` will correspond to the top of the vertical scroll bar and the left of the horizontal scroll bar. The `:value` slot may be accessed directly by some function in the larger interface, and other formulas in the interface may depend on it. If the `:value` slot is set directly, then the appearance of the scroll bar will be updated accordingly.

The trill boxes at each end of the scroll bar allow the user to increment and decrement `:value` by precise amounts. The intent of the two sets of boxes is to give the user a choice between increment values – either a conventional scroll of `:scr-incr` in the single arrow box or `:page-incr` in the double arrow box. There is no restriction on whether one value must be larger or smaller than the other.

In fact, the designer may choose to leave the trill boxes out completely. The slots `:scr-trill-p` and `:page-trill-p` may be set to `nil` in order to prevent the appearance of the scroll boxes or page boxes, respectively.

The indicator may also be moved directly by mouse movements. Dragging the indicator while the left mouse button is pressed will cause a thick lined box to follow the mouse. The indicator then moves to the position of this feedback box when the mouse button is released. If `:int-feedback-p` is set to `nil`, the thick lined box will not appear, and the indicator itself will follow the mouse. A click of the left mouse button in the background of the scroll bar will cause the indicator to jump to the position of the mouse.

With each change of the indicator position, the `:value` slot is updated automatically to reflect the new position. The current value is reported as a text string inside the indicator unless the slot `:indicator-text-p` is set to `nil`.

Since the scroll bar must be wide enough to accommodate the widest text string in its range of values, the width of the vertical scroll bar (and similarly the height of the horizontal scroll bar) is the maximum of the width of the widest value and the `:min-width`. The `:min-width` will be used if there is no indicator text (i.e., `:indicator-text-p` is `nil`), or if the `:min-width` is greater than the width of the widest value.

The slot `:scroll-p` is used to enable and disable the scrolling feature of the scroll bar. When `:scroll-p` is set to `nil`, the trill boxes of the scroll bar become inactive and the background turns white. This ability to disable scrolling is useful in applications where the range of the scroll bar is not fixed. For example, in the `scrolling-menu` gadget, the scroll bar is disabled there are not enough items to fill the entire menu.

The font in which `:value` is reported in the indicator may be set in the slot `:indicator-font`.

8.15 Sliders

```
(create-instance 'gg:V-Slider opal:aggregadget
```

```
      (:maybe-constant '(:left :top :height :shaft-width :scr-incr :page-incr :val-1 :val
                         :num-marks :scr-trill-p :page-trill-p :tic-marks-p :enumerate-p█
                         :value-feedback-p :scroll-p :value-feedback-font :enum-font█
                         :format-string :enum-format-string :visible))
      (:left 0)
      (:top 0)
      (:height 250)
      (:shaft-width 20)
      (:scr-incr 1)
      (:page-incr 5)
      (:val-1 0)
      (:val-2 100)
      (:num-marks 11)
      (:scr-trill-p T)
      (:page-trill-p T)
      (:tic-marks-p T)
      (:enumerate-p T)
      (:value-feedback-p T)
      (:scroll-p T)
      (:value-feedback-font opal:default-font)
      (:enum-font (create-instance NIL opal:font (:size :small)))
      (:format-string "~a")
      (:enum-format-string "~a")
      (:value (o-formula ...))
      (:selection-function NIL)   ; (lambda (gadget value))
      )

   (create-instance 'gg:H-Slider opal:aggregadget
      (:maybe-constant '(:left :top :width :shaft-height :scr-incr :page-incr :val-1 :val
                         :num-marks :tic-marks-p :enumerate-p :scr-trill-p :page-trill-p█
                         :scroll-p :value-feedback-p :value-feedback-font :enum-font█
                         :format-string :enum-format-string :visible))
      (:left 0)
      (:top 0)
      (:width 300)
      (:shaft-height 20)
      (:scr-incr 1)
      (:page-incr 5)
      (:val-1 0)
      (:val-2 100)
      (:num-marks 11)
      (:tic-marks-p T)
      (:enumerate-p T)
      (:scr-trill-p T)
      (:page-trill-p T)
      (:value-feedback-p T)
      (:scroll-p T)
```

```
(:value-feedback-font opal:default-font)
(:enum-font (create-instance NIL opal:font (:size :small)))
(:format-string "~a")
(:enum-format-string "~a")
(:value (o-formula ...))
(:selection-function NIL)   ; (lambda (gadget value))
)
```
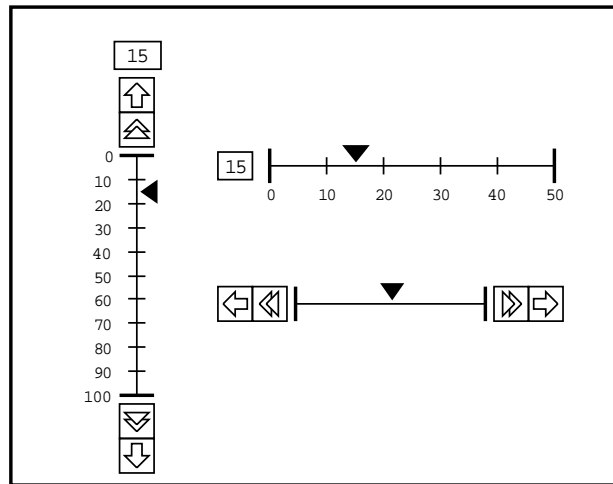
The loader file for the `v-slider` is "v-slider-loader". The loader file for the `h-slider` is "h-slider-loader".

The `v-slider` and `h-slider` gadgets have the same functionality as scroll bars, but they are used when the context requires a different style. The slider is comprised of a shaft with perpendicular tic-marks and an indicator which points to the current chosen value. Optional trill boxes appear at each end of the slider, and the indicator can be moved with the same mouse commands as the scroll bar. The vertical slider has an optional feedback

box above the shaft where the current value is displayed (this box is to the left of the horizontal slider). The value that appears in the feedback box may be edited directly by the user by pressing in the text box with the left mouse button and entering a new number.[3]

The slots `:value`, `:val-1`, `:val-2`, `:scr-incr`, `:page-incr`, `:scr-trill-p`, and `:page-trill-p` all have the same functionality as in scroll bars (see section [scroll-bars], page 416).

The designer may specify the number of tic-marks to appear on the shaft in the slot `:num-marks`. This number includes the tic-marks at each end of the shaft in addition to the internal tic-marks. Tic-marks may be left out by setting the `:tic-marks-p` slot to NIL. If the slot `:enumerate-p` is set to T, then each tic-mark will be identified by its position in the range of allowed values. Also, numbers may appear without tic-marks marks by setting `:enumerate-p` to T and `:tic-marks-p` to `nil`. The slot in which to specify the font for the tic-mark numbers is `:enum-font`.

The slot `:shaft-width` in the vertical slider (analogously, `:shaft-height` in the horizontal slider) is used to specify the width of the trill boxes at the end of the shaft. This determines the dimensions of the (invisible) bounding box for the interactors which manipulate the indicator.

The slot `:scroll-p` is used to enable and disable the scrolling feature of the sliders, just as in the scroll bars. When `:scroll-p` is set to `nil`, the trill boxes of the slider become inactive, and the indicator ceases to move.

The font for the feedback of the current value (which appears at the end of the shaft) may be specified in `:value-feedback-font`. The value feedback may be left out completely by setting `:value-feedback-p` to `nil`.

The `:format-string` and `:enum-format-string` slots allow you to control the formatting of the text strings, in case the standard formatting is not appropriate. This is mainly useful for floating point numbers. The slots should each contain a string that can be passed to the lisp function `format`. The default string is `"~a"`.

8.16 Trill Device

```
(create-instance 'gg:Trill-Device opal:aggregadget
   (:maybe-constant '(:left :top :min-frame-width :min-height :scr-incr :page-incr
                      :val-1 :val-2 :scr-trill-p :page-trill-p :scroll-p
                      :value-feedback-p :format-string :value-feedback-font :visible))
   (:left 0)
   (:top 0)
   (:min-frame-width 20)
   (:min-height 20)
   (:scr-incr 1)
   (:page-incr 5)
   (:val-1 0) (:val-2 100)
   (:scr-trill-p T)
   (:page-trill-p T)
```

---

[3] Backspace and several editing commands are provided through Interactors. See "Text-Interactor" in the Interactors chapter.

```
(:scroll-p T)
(:value-feedback-p T)
(:value-feedback-font opal:default-font)
(:value 20)
(:format-string "~a")
(:selection-function NIL)   ; (lambda (gadget value))
)
```
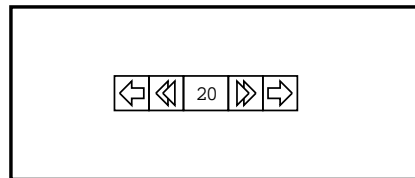
The loader file for the `trill-device` is "trill-device-loader".

The `trill-device` is a compact gadget which allows a value to be incremented and decremented over a range as in the scroll bars and sliders, but with only the numerical value as feedback. All slots function exactly as in horizontal sliders, but without the shaft and tic-mark features. As with sliders, the feedback value may be edited by the user.

A unique feature of the trill box is that either or both `:val-1` or `:val-2` may be `nil`, implying no lower or upper bound on the input value, respectively. If numerical values for both slots are supplied, then clipping of the input value into the specified range occurs as

usual. Otherwise, `:val-1` is assumed to be the minimum value, and clipping will not occur at the `nil` endpoints of the interval.

The width of the trill device may be either static or dynamic. If both `:val-1` and `:val-2` are specified, then the width of the value frame is the maximum of the widest allowed value and the `:min-frame-width`. Otherwise, the value frame will expand with the width of the value, while never falling below `:min-frame-width`.

The height of the trill device is the maximum of the greatest string height of all values in the range and the value of the slot `:min-height`. The `:min-height` will be used if there is no indicator text or if the `:min-height` is greater than the height of the tallest value.

The `:format-string` slot allows you to control the formatting of the text string, in case the standard formatting is not appropriate. This is mainly useful for floating point numbers. This slot takes a string that can be passed to the lisp function `format`. The default string is `"~a"`. For example:

```
(create-instance 'TRILL garnet-gadgets:trill-device
   (:left 35)(:top 70)(:val-1 0.0)(:val-2 1.0)(:scr-incr 0.01)
   (:page-incr 0.1)(:format-string "~4,2F"))
```

8.17 Gauge
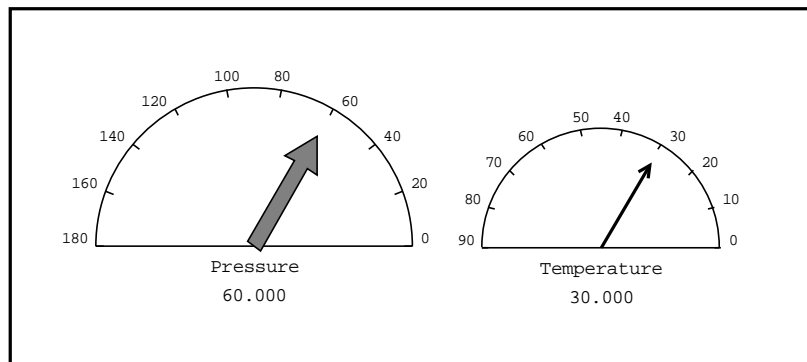

```
(create-instance 'gg:Gauge opal:aggregadget
    (:maybe-constant '(:left :top :width :polygon-needle-p :int-feedback-p▉
        :title :title-font :value-font :enum-font :num-marks
        :tic-marks-p :enumerate-p :value-feedback-p :text-offset
```

```
            :val-1 :val-2 :visible))
      (:left 0)
      (:top 0)
      (:width 230)
      (:val-1 0)
      (:val-2 180)
      (:num-marks 10)
      (:tic-marks-p T)
      (:enumerate-p T)
      (:value-feedback-p T)
      (:polygon-needle-p T)
      (:int-feedback-p T)
      (:text-offset 5)
      (:title "Gauge")
      (:title-font opal:default-font)
      (:value-font opal:default-font)
      (:enum-font (create-instance NIL opal:font (:size :small)))
      (:value (o-formula ...))
      (:format-string "~a")        ; How to print the feedback value
      (:enum-format-string "~a")   ; How to print the tic-mark values
      (:selection-function NIL)    ; (lambda (gadget value))
      )
```

The loader file for the `gauge` is `"gauge-loader"`.

The `gauge` object is a semi-circular meter with tic-marks around the perimeter. As with scroll bars and sliders, this object allows the user to specify a value between minimum and maximum values. A needle points to the currently chosen value, and may either be a bare arrow or a thick, arrow-shaped polygon with a gray filling. The needle may be rotated by dragging it with the left mouse button pressed. Text below the gauge reports the current value to which the needle is pointing.

If the slot `:polygon-needle-p` is T, then the needle will be thick with a gray filling. If `nil`, then the needle will be a bare arrow.

If `:int-feedback-p` is T, then the needle will not follow the mouse directly, but instead a short line will appear and be rotated. When the mouse button is released, the large needle will swing over to rest at the new location. The needle will follow the mouse directly if `:int-feedback-p` is set to `nil`.

The slots `:num-marks`, `:tic-marks-p`, `:enumerate-p`, `:val-1`, `:val-2`, and `:enum-font` are implemented as in the sliders (see section [sliders], page 419). The value in `:val-1` corresponds to the right side of the gauge.

The title of the gauge is specified in `:title`. No title will appear if `:title` is `nil`. The fonts for the title of the gauge and the current chosen value are specified in `:title-font` and `:value-font`, respectively.

If `:value-feedback-p` is T, then numerical text will appear below the gauge indicating the currently chosen value. The value in `:text-offset` determines the distance between the gauge and the title string, and between the title string and the value feedback.

The `:format-string` and `:enum-format-string` slots allow you to control the formatting of the text strings, in case the standard formatting is not appropriate. This is mainly useful for floating point numbers. The slots should each contain a string that can be passed to the lisp function `format`. The default string is `"~a"`.

8.18 Buttons

The button objects in the Garnet Gadgets can be either a single stand-alone button, or a panel of buttons. Each button in the set is related to the others by common interactors and constraints on both the sizes of the buttons and the text beside (or inside) the buttons.

The button objects all have several common features.

When used as a panel, the buttons are implemented with aggrelists, so all slots that can be customized in an aggrelist can be customized in the button panels.[4] These slots are:

> `:direction` — `:vertical` or `:horizontal` (default `:vertical`)
>
> `:v-spacing` — distance between buttons, if vertical orientation (default 5)
>
> `:h-spacing` — same, if horizontal orientation
>
> `:fixed-width-p` — whether all the buttons should have the width of the value in `:fixed-width-size`, or the width of each button should be determined by the width of the string associated with that button (default T)
>
> `:fixed-height-p` — same, but with heights
>
> `:fixed-width-size` — width of all components (default is the width of the widest button, as determined by the widest string)
>
> `:fixed-height-size` — same, but with heights
>
> `:h-align` — How to align buttons, if vertical orientation. Allowed values are `:left`, `:center`, or `:right`. (default `:right` for radio-buttons and x-buttons, `:center` for text-buttons)
>
> `:rank-margin` — after this many buttons, a new row (or column) will be started (default `nil`)
>
> `:pixel-margin` — absolute position in pixels after which a new row (or column) will be started (default `nil`)
>
> `:indent` — amount to indent the new row (or column) in pixels (default 0)

In the button and menu objects, the `:value` slot contains to the string or atom of the currently selected item (in the `x-button-panel` this value is a list of selected items). The currently selected object is named in the `:value-obj` slot. In order to set an item to be selected, either the `:value` slot of the button panel must be set with the desired string or atom from the `:items` list, or the `:value-obj` slot must be set with the desired button object (see section [sel-buttons], page 570, for examples of selecting buttons).

---

[4] See the Aggregadgets chapter for greater detail.

The `:width` of the buttons is determined by the width of the longest item, and therefore cannot be specified by the designer. However, the `:width` is computed internally and may be accessed after the object is instantiated. (The :height is computed similarly.)

The shadow below each button has the effect of simulating a floating three-dimensional button. When the left mouse button is clicked on one of the gadget buttons, the button frame moves onto the shadow and appears to be depressed. The slot `:shadow-offset` specifies the amount of shadow that appears under the button when it is not pressed. A value of zero implies that no shadow will appear (i.e., no floating effect).

There is a gray border in the frame of each of the buttons, the width of which may be specified in the slot `:gray-width`.

The strings or atoms associated with each button are specified in the `:items` slot. See section [items-slot], page 410, for a discussion of specifying items and item functions.

The font in which the button labels appear may be specified in the `:font` slot.

Most of the buttons and button panels have a `:toggle-p` slot. When the value of this slot is T, then the button will become deselected if it is clicked a second time. Otherwise, after the button is selected the first time, it is always selected (though its `:selection-function` and associated item functions will continue to be executed each time it is pressed.
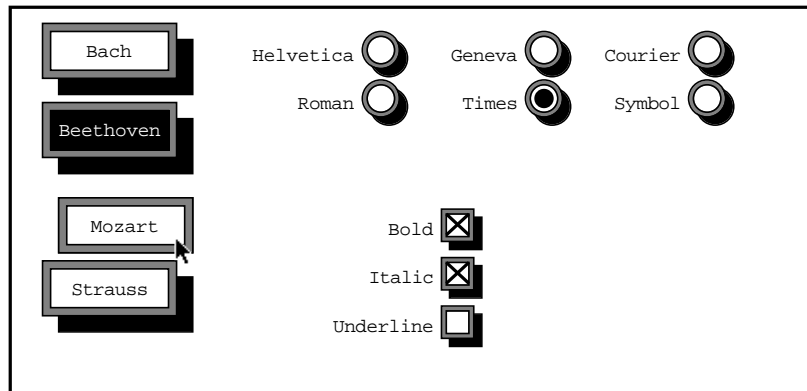
Figure 8.8: Text buttons, radio buttons, and x-buttons

8.18.1 Text Buttons

```
(create-instance 'gg:Text-Button opal:aggregadget
   (:maybe-constant '(:left :top :shadow-offset :text-offset :gray-width
```

```
                :string :toggle-p :font :final-feedback-p :visible))
        (:left 0)
        (:top 0)
        (:shadow-offset 10)
        (:text-offset 5)
        (:gray-width 5)
        (:string "Text Button")
        (:toggle-p T)
        (:font opal:default-font)
        (:final-feedback-p T)
        (:value (o-formula (if (gvl :selected) (gvl :string))))
        (:selected (o-formula (gvl :value)))   ;  This slot is set by the interactor
        (:selection-function NIL)   ; (lambda (gadget value))
        )

    (create-instance 'gg:Text-Button-Panel opal:aggregadget
        (:maybe-constant '(:left :top :direction :v-spacing :h-spacing :h-align
           :fixed-width-p :fixed-width-size :fixed-height-p
           :fixed-height-size :indent :rank-margin :pixel-margin
           :shadow-offset :text-offset :gray-width :final-feedback-p
           :toggle-p :font :items :visible))
        (:left 0)
        (:top 0)
        (:shadow-offset 10)
        (:text-offset 5)
        (:gray-width 5)
        (:final-feedback-p T)
        (:toggle-p NIL)
        (:font opal:default-font)
        (:items '("Text 1" "Text 2" "Text 3" "Text 4"))
        (:value-obj NIL)
        (:value (o-formula (gvl :value-obj :string)))
        (:selection-function NIL)   ; (lambda (gadget value))
        <All customizable slots of an aggrelist>)
```

The loader file for the `text-button` and `text-button-panel` is "text-buttons-loader".

The `text-button-panel` object is a set of rectangular buttons, with the string or atom associated with each button centered inside. When a button is pressed, the text of the button will appear in inverse video if `:final-feedback-p` is T. The `text-button` is just a single button.

The distance from the beginning of the longest label to the inside edge of the button frame is specified in `:text-offset`. The value in `:text-offset` will affect the height and width of every button when specified.

8.18.2 X Buttons

```
    (create-instance 'gg:X-Button opal:aggregadget
```

```
      (:maybe-constant '(:left :top :button-width :button-height
         :shadow-offset :text-offset :gray-width
         :text-on-left-p :toggle-p :string :font :visible))
      (:left 0)
      (:top 0)
      (:button-width 20)
      (:button-height 20)
      (:shadow-offset 5)
      (:text-offset 5)
      (:gray-width 3)
      (:text-on-left-p T)
      (:string "X Button")
      (:toggle-p T)
      (:font opal:default-font)
      (:value (o-formula (if (gvl :selected) (gvl :string))))
      (:selected (o-formula (gvl :value))) ; Set by interactor
      (:selection-function NIL)              ; (lambda (gadget value))
      )

  (create-instance 'gg:X-Button-Panel opal:aggregadget
      (:maybe-constant '(:left :top :direction :v-spacing :h-spacing :h-align█
         :fixed-width-p :fixed-width-size :fixed-height-p :fixed-height-size█
                          :indent :rank-margin :pixel-margin :button-width :button-height█
                          :shadow-offset :text-offset :gray-width :text-on-left-p█
         :font :items :visible))
      (:left 0)
      (:top 0)
      (:button-width 20)
      (:button-height 20)
      (:shadow-offset 5)
      (:text-offset 5)
      (:gray-width 3)
      (:text-on-left-p T)
      (:font opal:default-font)
      (:items '("X-label 1" "X-label 2" "X-label 3"))
      (:value-obj NIL)
      (:value (o-formula (mapcar #'(lambda (object)
    (gv object :string))
         (gvl :value-obj))))
      (:selection-function NIL)  ; (lambda (gadget value))
      <All customizable slots of an aggrelist>)
```

The loader file for the x-button and x-button-panel is "x-buttons-loader".

The x-button-panel object is also a set of rectangular buttons, but the item associated
with each button appears either to the left or to the right of the button. Any number
of buttons may be selected at one time, and clicking on a selected button de-selects it.

Currently selected buttons are graphically indicated by the presence of a large "X" in the button frames. The `x-button` is just a single button.

Since the `x-button-panel` allows selection of several items at once, the `:value` slot is a list of strings (or atoms), rather than a single string. Similarly, `:value-obj` is a list of objects.

The slot `:text-on-left-p` specifies whether the text will appear on the right or left of the x-buttons. A `nil` value indicates the text should appear on the right. When text appears on the right, the designer will probably want to set `:h-align` to `:left` in order to left-justify the text against the buttons.

The distance from the labels to the buttons is specified in `:text-offset`.

The slots `:button-width` and `:button-height` specify the width and height of the x-buttons. The "X" will stretch to accommodate these dimensions.

8.18.3 Radio Buttons

```
(create-instance 'gg:Radio-Button opal:aggregadget
   (:maybe-constant '(:left :top :button-diameter :shadow-offset :text-offset█
                      :gray-width :string :text-on-left-p :toggle-p :font :visible))█
   (:left 0) (:top 0)
   (:button-diameter 23)
   (:shadow-offset 5) (:text-offset 5) (:gray-width 3)
   (:string "Radio button")
   (:toggle-p T)
   (:text-on-left-p T)
   (:font opal:default-font)
   (:value (o-formula (if (gvl :selected) (gvl :string))))
   (:selected (o-formula (gvl :value))) ; Set by interactor
   (:selection-function NIL)             ; (lambda (gadget value))
   )

(create-instance 'gg:Radio-Button-Panel opal:aggregadget
   (:maybe-constant '(:left :top :direction :v-spacing :h-spacing :h-align█
                      :fixed-width-p :fixed-width-size :fixed-height-p :fixed-height-s
                      :indent :rank-margin :pixel-margin :button-diameter :shadow-offs
                      :text-offset :gray-width :text-on-left-p :toggle-p :font█
                      :items :visible))
   (:left 0)
   (:top 0)
   (:button-diameter 23)
   (:shadow-offset 5)
   (:text-offset 5)
   (:gray-width 3)
   (:text-on-left-p T)
   (:toggle-p T)
   (:font opal:default-font)
   (:items '("Radio-text 1" "Radio-text 2" "Radio-text 3" "Radio-text 4"))█
   (:value-obj NIL)
```

```
(:value (o-formula (gvl :value-obj :string)))
(:selection-function NIL)   ; (lambda (gadget value))
<All customizable slots of an aggrelist>)
```

The loader file for the `radio-button` and `radio-button-panel` is "radio-buttons-loader".

The `radio-button-panel` is a set of circular buttons with items appearing to either the left or right of the buttons (implementation of `:text-on-left-p` and `:text-offset` is identical to x-buttons). Only one button may be selected at a time, with an inverse circle indicating the currently selected button. A `radio-button` is a single button.

8.19 Option Button

```
(create-instance 'gg:Option-Button opal:aggregadget
  (:maybe-constant '(:left :top :text-offset :label :button-offset :button-shadow-offs
                     :items :initial-item :button-font :label-font :button-fixed-width
                     :v-spacing :keep-menu-in-screen-p :menu-h-align))
  (:left 40) (:top 40)
  (:text-offset 4)
  (:label "Option button:")
  (:button-offset 10)
  (:button-shadow-offset 5)
  (:items '("Item 1" "Item 2" "Item 3" "Item 4"))
  (:initial-item (o-formula (first (gvl :items))))
  (:button-font opal:default-font)
  (:label-font (opal:get-standard-font NIL :bold NIL))
  (:value (o-formula (gvl :option-text-button :string)))
  (:button-fixed-width-p T)
  (:v-spacing 0)
  (:menu-h-align :left)
  (:keep-menu-in-screen-p T)
  (:selection-function NIL)   ; (lambda (gadget value))
  ...)
```

`Color:` `Blue`

The loader file for the `option-button` is "option-button-loader".

When the left mouse button is clicked and held on an option button, a menu will pop up, from which items can be selected by moving the pointer to the desired item and releasing the mouse button. Figure [option-button-tag], page 436, shows an option button in its normal state (on the left) and when the button is pressed.

The `:items` slot is a list of strings or Garnet objects, which will appear in the menu. The `:initial-item` slot contains the initial item that will appear in the button. This slot MUST be non-NIL, and should contain either an element of the `:items` list, or a formula to calculate the same.

The `:text-offset` slot specifies how far from the frame the text should begin. The slot `:button-offset` specifies how far from the label the button should begin. The `:button-shadow-offset` contains the size of the button's shadow.

The `:label` slot contains a string that appears before the option button. If no label is desired, this slot can be set to the empty string, `""`.

The `:button-font` and `:label-font` slots specify the fonts to use in the button and the label. The font of the items in the menu is the same as the font in the `:button-font` slot.

The `:value` slot contains the currently selected item, which is the same as the value in the `:string` slot of the button.

The `:button-fixed-width-p` slot specifies whether to keep the button's width constant or not. If it is set to T, the button's width will be the width of the longest string in the `:items` slot. If it is set to NIL, the width of the button will be the width of the currently selected item.

The value in `:v-spacing` specifies the amount of space between each menu item.

The `:menu-h-align` slot should be either `:left`, `:center`, or `:right`, and specifies the justification of the menu items.

If the `:keep-menu-in-screen-p` slot is T, then the menu will never pop out of the screen, i.e., the top of the menu will never be less than the screen's top, and the bottom of the menu will never be greater than the screen's bottom. If this slot is set to `nil`, the menu may pop out of the top or out of the bottom of the screen. NOTE: If the number of items in the menu makes it so that both the top of the menu and the bottom of the menu are out of the screen, this slot will be disregarded.

8.20 Popup-Menu-Button

```
Selected: "Blue"
```

Red
Blue
Green
Yellow
Aquamarine
Cyan
Fluorescent
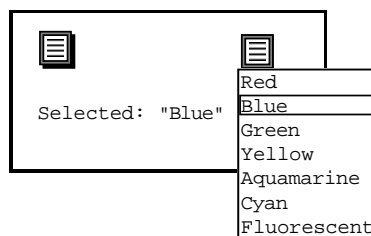
```
(create-instance 'gg:Popup-Menu-Button gg:text-button
  (:left 0)
  (:top 0)
  (:string gg:lines-bitmap)
```

```
(: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)    ; (lambda (gadget value))
(: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 [items-slot], page 410, for more information.

The `:v-spacing, :h-align,` and `:item-font` control the display of the menu items. See the Gadgets chapter 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.

8.21 Menu

```
(create-instance 'gg:Menu opal:aggregadget
   (:maybe-constant '(:left :top :v-spacing :h-align :shadow-offset
       :text-offset :title :title-font :items :item-font
       :item-to-string-function :visible))
   (:left 0)
   (:top 0)
   (:v-spacing 0)
   (:h-align :left)
   (:shadow-offset 5)
   (:text-offset 4)
   (:min-menu-width 0)
   (:title NIL)
   (:title-font (create-instance NIL opal:font
                   (:family :serif)
   (:size :large)
   (:face :roman)))
   (:items '("Item 1" "Item 2" "Item 3" "Item 4"))
   (: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)))
   ""))) 
```

```
(:selection-function NIL)    ; (lambda (gadget value))
(:value-obj NIL)
(:value (o-formula (gvl :value-obj :string))))
```
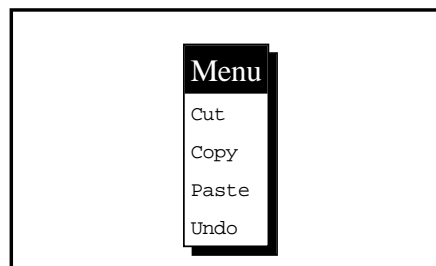

The loader file for the `menu` is "menu-loader".

The `menu` object is a set of text items surrounded by a rectangular frame, with an optional title above the items in inverse video. When an item is selected, a box is momentarily drawn around the item and associated item functions and global functions are executed.

The `:items` slot may be a list of strings, atoms, string/function pairs or atom/function pairs, as with buttons (see section [buttons], page 429). If this slot is `s-value`'d with a new list of items, the components of the gadget will be adjusted automatically during the next call to `opal:update`.

The amount of shadow that appears below the menu frame (the menu frame is stationary) is specified in `:shadow-offset`. A value of zero implies that no shadow will appear.

The slot `:h-align` determines how the menu items are justified in the frame. Allowed values are `:left`, `:center` and `:right`.

The slot `:text-offset` is the margin spacing – the distance from the frame to the longest string.

The slot `:item-font` determines the font in which the items will appear.

A title for the menu may be specified as a string in the `:title` slot. If `:title` is `nil`, then no title will appear. The font in which the title should appear is specified in `:title-font`.

any objects, including strings, atoms, schemas, string/function pairs, etc. The default scrolling menu assumes that `:items` contains a list as described in section [buttons], page 429, but this can be easily changed by the designer. A function defined in `:item-to-string-function` takes an item (or the first element of an item pair) and returns a string corresponding to that item for display in the menu. The default function for this slot is

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

This function takes an item and returns it if it is a string, or coerces it into a string if it was an atom. See section [sm-ex], page 571, for an example where the `:items` list is composed of Garnet schemas.

8.22 Scrolling Menu
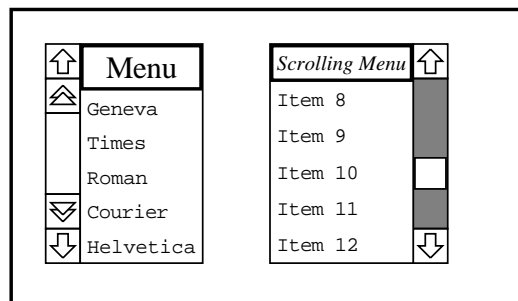

```
(create-instance 'gg:Scrolling-Menu opal:aggregadget
   (:maybe-constant '(:left :top :scroll-on-left-p :min-scroll-bar-width :scr-trill-p
                      :page-trill-p :indicator-text-p :scr-incr :page-incr
```

```
                                  :int-scroll-feedback-p :indicator-font :min-frame-width :v-spaci
                                  :h-align :multiple-p :items :item-to-string-function :item-font▮
                                  :num-visible :int-menu-feedback-p :final-feedback-p :text-offset|
                                  :title :title-font :visible))
        (:left 0) (:top 0)

        ;; Scroll bar slots
        (:scroll-on-left-p T)
        (:min-scroll-bar-width 20)
        (:scr-trill-p T)
        (:page-trill-p T)
        (:indicator-text-p NIL)
        (:scr-incr 1)
        (:page-incr 5)
        (:int-scroll-feedback-p NIL)
        (:indicator-font (create-instance NIL opal:font (:size :small)))
        (:scroll-selection-function NIL)

        ;; Menu slots
        (:min-frame-width 0)
        (:v-spacing 6)
        (:h-align :left)
        (:multiple-p T)
        (:toggle-p T)
        (:items '("Item 1" "Item 2" "Item 3" ... "Item 20"))
        (:item-to-string-function
           #'(lambda (item)
               (if item
                   (if (stringp item)
                       item
                       (string-capitalize (string-trim ":" item)))
                   "")))
        (:item-font opal:default-font)
        (:num-visible 5)
        (:int-menu-feedback-p T)
        (:final-feedback-p T)
        (:text-offset 4)
        (:title NIL)
        (:title-font (create-instance NIL opal:font
        (:family :serif)
        (:size :large)
        (:face :roman)))
        (:menu-selection-function NIL)
        (:selected-ranks NIL)
        (:value (o-formula ...)))
```

The loader file for the `scrolling-menu` gadget is "scrolling-menu-loader".

The `scrolling-menu` object is a combination of a vertical scroll bar and a menu which allows the user to only see a subset of the available choices in a menu at one time. The set of visible choices is changed by moving the scroll bar, which causes the choices to scroll up or down in the menu.

8.22.1 Scroll Bar Control

T, then the scroll bar will appear on the left side of the menu. Otherwise, the scroll bar will be on the right.

The slot `:min-scroll-bar-width` determines the minimum width of the scroll bar. The scroll bar will be wider than this width only if the indicator text is too wide to fit into this width.

The interim feedback of the scroll bar is controlled by the slot `:int-scroll-feedback-p`. If this slot is set to T, then a thick-lined box will follow the mouse when the user drags the indicator. Otherwise, the indicator will follow the mouse directly.

A function may be specified in `:scroll-selection-function` to be executed whenever the user changes the scroll bar, either by clicking on the trill boxes or by dragging the indicator. The function takes the same parameters as the usual selection function described in section [sel-fn], page 410.

The slots `:scr-trill-p`, `:page-trill-p`, `:scr-incr`, `:page-incr`, `:indicator-text-p`, and `:indicator-font` are all used for the scroll bar in the scrolling menu in the same way as the vertical scroll bar described in section [scroll-bars], page 416.

8.22.2 Menu Control

frame is determined by `:min-frame-width`. The scrolling menu will appear wider than this value only if the title or the longest item string will not fit in a menu of this width.

The `:v-spacing` slot determines the distance between each item in the menu.

The justification of the items in the menu is determined by the slot `:h-align` which may be either `:left`, `:center`, or `:right`.

If the value of `:multiple-p` is T, then the user may make multiple selections in the menu by clicking on items while holding down the shift key. If this slot is `nil`, then only single selections are permitted.

The `:toggle-p` slot specifies whether to toggle the current selection when it is clicked on again. If `:toggle-p` is `nil`, then a selected item can be clicked upon for any number of times and it will stay selected. If the `:toggle-p` slot is set to T (the default), clicking on an already selected item will cause the item to become unselected. NOTE: Clicking on a selected item while doing multiple selections will always toggle the selection, regardless of the value of the `:toggle-p` slot.

The `:item-to-string-function` slot is identical in operation to the one described for the `gg:menu` in section [menu], page 441. 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.

The slot `:num-visible` determines how many items should be visible in the menu at one time.

A box will appear around the item being selected while the mouse button is held down if the slot `:int-menu-feedback-p` is T.

Selected items will appear in inverse video if the slot `:final-feedback-p` is set to T.

The slot `:text-offset` determines the distance from each string to the menu frame.

A title will appear above the menu if a title string is specified in `:title`. If `:title` is `nil`, then no title will appear. The font of the title is in `:title-font`.

The font of the items is in `:item-font`.

The `:selected-ranks` slot is used by the designer to select items in the menu. The slot contains a list of indices which correspond to the ranks of the selected items in the `:items` list. The ranks are zero-based. For example, if the `:selected-ranks` slot contained `'(0 3)`, then the first and fourth items (not necessarily visible) in the scrolling menu will be selected.

A function defined in `:menu-selection-function` will be executed whenever the user selects an item from the menu. This function takes two parameters,

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

where *gadget* is the programmer's instance of the **scrolling-menu** and *scrolling-menu-item* is the object just selected by the user. The item associated with the user's selection can be obtained through the `:item` slot of the *scrolling-menu-item*:

```
(gv scrolling-menu-item :item) --> The item just selected
```

8.23 Menubar

```
(create-instance 'gg:Menubar opal:aggrelist
   (:left 0)(:top 0)
   (:items NIL)
   (:title-font (create-instance NIL opal:font (:size :large)))
   (:item-font (create-instance NIL opal:font (:face :italic)))
   (:selection-function NIL)   ; (lambda (gadget value))
   )
```
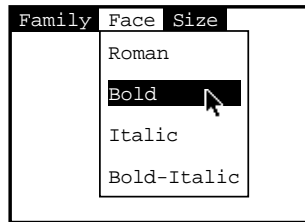
Figure 8.10: Picture of a pull-down menu (an instance of `menubar`)

The `menubar` gadget is a set of pull down menus that is similar to the Macintosh design. When the user clicks on an inverse bar item, a submenu pops up and the user can then choose one of the displayed items.

NOTE: There is no `:value` slot in this gadget. The designer should define functions in the `:selection-function` or `:items` slots to propagate the user's selections to the rest of the interface (see below).

The complete `menubar` gadget is a collection of three objects. In addition to the top-level `menubar` object, there are `bar-item` and `submenu-item` objects. The `menubar` is an aggrelist of `bar-item` objects, which are the inverse-video text objects that appear horizontally at the top of the window. The `submenu-item` objects are vertically arranged in an aggrelist within each `bar-item`.

The programmer may approach the `menubar` from two perspectives: the traditional Garnet way which involves setting the `:items` slot and allowing the gadget to maintain its own components, or from a bottom-up approach which involves creating the sub-objects and chapterly adding (and removing) them from the `menubar` instance.

Programmers who choose the Garnet approach can ignore most of the functions described below, since interaction with the `menubar` will almost exclusively involve setting the `:items` slot. The other approach requires creating instances of `bar-item` and `submenu-item` gadgets and adding them as components to a `menubar` using the support functions.

8.23.1 Item Selection Functions

There are three levels of functions in the `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 three types of functions take the parameters `(lambda (gadget menu-item submenu-item))`

When a function is supplied in the `:selection-function` slot of the `menubar`, it will be executed whenever any item is selected from any of the submenus. If a function is attached to a submenu (e.g., it is the value for *m1func* in the `:items` syntax of section [garnet-menubar-programming], page 449), then it is executed when any item is chosen from that submenu. If a function is attached to a submenu-item (e.g., *mX,Yfunc*), then it is executed only when that submenu-item is selected.

The order for calling these functions is: first, the submenu function is called, then the submenu-item function is called, and finally the `:selection-function` is called.

8.23.2 Programming the Menubar in the Traditional Garnet Way

The `:items` slot of the `menubar` is a complicated list with the following format:

```
(:items '(("m1" m1func (("m1,1" [m1,1func])...("m1,N" [m1,Nfunc])))
          ("m2" m2func (("m2,1" [m2,1func])...("m2,N" [m2,Nfunc])))
          ...))
```

where "*mN*" is a string or atom that is the title of a menu (atoms appear as capitalized strings in the submenu titles), "*mX,Y*" is a string or atom in menu X, row Y, *mNfunc* is executed when any item in menu N is selected, and *mX,Yfunc* is executed when item "*mX,Y*" is selected. See section [item-selection-functions], page 449, for the parameters of these functions. NOTE: the syntax above requires that the submenu-items be described with lists, even when no submenu-item functions are supplied (i.e., the list `("m1,1")` is required instead of just the string `"m1,1"`).

In order to maintain the syntax of the sublists, the submenu functions (*m1func* and *m2func* above) must always be supplied. Thus, `nil` should be placed in this position if no

function is to be executed for the submenu. The submenu-item functions (*m1,1func* etc. above) are optional and may be omitted.

The `:title-font` is the font for the `bar-item` objects which appear in inverse video, and the `:item-font` is the font for the `submenu-item` objects arranged vertically in the pop-up menus.

8.23.3 An example

```
(create-instance 'WIN inter:interactor-window
   (:left 750)(:top 80)(:width 200)(:height 200)
   (:title "Menubar Example"))
(s-value WIN :aggregate (create-instance 'TOP-AGG opal:aggregate))
(opal:update win)

(defun Fixed-Fn (gadget menu-item submenu-item)
  (format t "Calling Fixed-Fn with ~S ~S ~S.~%" gadget menu-item submenu-item))█

(defun Face-Fn (gadget menu-item submenu-item)
  (format t "Calling Face-Fn with ~S ~S ~S.~%" gadget menu-item submenu-item))█

(create-instance 'DEMO-MENUBAR garnet-gadgets:menubar
   (:items
    '(("family" NIL
       (("fixed" Fixed-Fn)("serif")("sans-serif")))
      ("face" Face-Fn
       (("roman")("bold")("italic")("bold-italic")))
      ("size" NIL
       (("small")("medium")("large")("very-large"))))))

(opal:add-component TOP-AGG DEMO-MENUBAR)
(opal:update win)
```

Figure 8.11: The code to generate the picture in Figure [menubar-pix-1], page 448

The code in Figure [menubar-code-1], page 450, creates the `menubar` picture shown in Figure [menubar-pix-1], page 448. It illustrates the Garnet method for handling the `menubar` gadget.

8.23.4 Adding items to the menubar

There are two types of items that can be added to a `menubar`: an entire submenu can be added to the top-level `menubar`, or single submenu-item can be added to a submenu.

The `add-item` method for the `menubar` can be used to add submenus – opal:Add-Item *menubar submenu [[:where] position [locator] [:key index-function]]*⟨undefined⟩ **[method], page** ⟨undefined⟩ Using the standard Garnet method, the *submenu* parameter should be a sublist of a top-level items list,
(e.g., '("underline" NIL (("none") ("single") ("double"))). The remaining optional

parameters follow the standard `add-item` definition described in the Aggregadgets chapter, and refer to the placement of the new submenu among the existing submenus. *Locator* should be some element of the current *:items* list, or may be the title of a submenu when the *index-function* is `#'car` (see examples below).

For example, each of the following lines will add a new submenu to DEMO-MENUBAR in Figure [menubar-code-1], page 450:

```
(opal:add-item DEMO-MENUBAR '("font-name" NIL (("courier") ("times") ("geneva"))))
(opal:add-item DEMO-MENUBAR
               '("other-fonts" NIL (("helvetica") ("chicago")))
        :after '("family" NIL (("fixed" Fixed-Fn)("serif")("sans-serif"))))
(opal:add-item DEMO-MENUBAR
               '("symbols" NIL (("mathematical") ("greek")))
        :before "face" :key #'car)
```

Individual submenu-items can be added to a `menubar` with the following function:

```
add-submenu-item menubar submenu-title submenu-item [[:where] position [lo-
cator] [:key index-function]])
```

This function adds the new *submenu-item* to the menubar, and places it in the submenu named by *submenu-title*. The new *submenu-item* description should be a list containing a string (or atom) and an optional function (e.g., `'("italic")` or `'("italic" italic-fn)`).

For example, the following lines will add new submenu-items to the DEMO-MENUBAR in Figure [menubar-code-1], page 450:

```
(garnet-gadgets:add-submenu-item DEMO-MENUBAR "face" '("outline"))
(garnet-gadgets:add-submenu-item DEMO-MENUBAR "size" '("very small")
                                        :before "small" :key #'car)
```

As shown in the second example, the *position* and *locator* parameters should correspond to existing submenu items.

8.23.5 Removing items from the menubar

Just as submenus and submenu-items can be added to the `menubar`, these two types of items can be removed.

`opal:Remove-Item` *menubar submenu*⟨undefined⟩ `[method]`, `page` ⟨undefined⟩ This function removes the *submenu* from *menubar*. For traditional Garnet programming, the *submenu* should be a sublist of the top-level `:items` list, or just the title of a submenu (a string or atom).

For example, the following lines will remove an item from the DEMO-MENUBAR in Figure [menubar-code-1], page 450:

```
(opal:remove-item DEMO-MENUBAR
                  '("face" Face-Fn (("roman")("bold")("italic")("bold-italic"))))
(opal:remove-item DEMO-MENUBAR "size")
```

The following function is used to remove submenu-items from a `menubar`:

`gg:Remove-Submenu-Item` *menubar submenu-title submenu-item* `[function]`, `page 90` *Submenu-item* may either be the list description of the submenu-item (i.e., `("italic")`) or just the string (or atom) of the submenu-item (i.e., `"italic"`).

For   example,   `(garnet-gadgets:remove-submenu-item DEMO-MENUBAR "size"`
`"small")`

8.23.6 Programming the Menubar with Components

In the bottom-up approach to programming the `menubar`, the user must create components
of the `menubar` (i.e., instances of `bar-item` and `submenu-item` gadgets) and attach them
piece-by-piece. This design is loosely based on the interface to the Macintosh menubar
in Macintosh Common Lisp. The functions for creating the components are described in
section [creating-menubar-components], page 453. Section [adding-menubar-components],
page 453, explains how to attach these components to the `menubar`.

8.23.7 An example

```
(create-instance 'WIN inter:interactor-window
   (:left 750)(:top 80)(:width 200)(:height 200)
   (:title "Menubar Example"))
(s-value WIN :aggregate (create-instance 'TOP-AGG opal:aggregate))
(opal:update win)

; Create the menubar and the bar-item
(setf demo-menubar (garnet-gadgets:make-menubar))
(setf mac-bar (garnet-gadgets:make-bar-item :title "Mac Fonts"))

; Create submenu-items
(setf sub-1 (garnet-gadgets:make-submenu-item :desc '("Gothic")))
(setf sub-2 (garnet-gadgets:make-submenu-item :desc '("Venice")))
(setf sub-3 (garnet-gadgets:make-submenu-item :desc '("Old English")))

; Add the submenu-items to the bar-item
(opal:add-item mac-bar sub-1)
(opal:add-item mac-bar sub-2 :before sub-1)
(opal:add-item mac-bar sub-3 :after "Venice" :key #'car)

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

; Add the bar-item to the menubar and update the window
(opal:add-item demo-menubar mac-bar)
(opal:update win)
```

Figure 8.12: The creation of a menubar and its components

The code in Figure [menubar-code-2], page 452, creates a `menubar` and several component
pieces, and then attaches the components to the `menubar`. This illustrates the bottom-up
approach to programming the `menubar`.

Notice that the `menubar` instance must be added to the top-level aggregate before any bar-items are attached. This ensures that the `menubar` will be initialized with the proper main window before new submenu windows are added.

8.23.8 Creating components of the menubar

The functions in this section are used to create the three types of components that comprise a pull-down menu – the `menubar` (the top-level part), the `bar-item` (which contains a submenu), and the `submenu-item`. Once the parts of the pull-down menu are created, they are attached using the functions of section [adding-menubar-components], page 453. Please see section [adding-menubar-components], page 453, for examples of the creation functions and attachment functions together.

`gg:make-menubar`   [Function]
>   Returns an instance of `menubar`.

`gg:make-bar-item` **&key** *desc font title*   [Function]
>   this function returns an instance of `bar-item`. If any of the keys are supplied, then the corresponding slots of the `bar-item` instance are set with those values. The *desc* parameter is the description of a submenu (e.g., `'("underline" NIL (("none")("single")("double")))`). The *font* is the font of the submenu-items within the submenu, and *title* is a string or atom that will be the heading of the submenu. If the title was already specified in the *desc* parameter, then no *title* parameter should be supplied.

`gg:make-submenu-item` **&key** *desc enabled*   [Function]
>   this function returns an instance of `submenu-item`. If any of the keys are supplied, then the corresponding slots of the `submenu-item` instance are set with those values. The *desc* parameter is the description of a submenu-item (e.g., `'("italic")` or `'("italic" italic-fn)`). The default for *enabled* is T.

8.23.9 Adding components to the menubar

Just as with the traditional Garnet approach, the two types of components that can be added to the `menubar` gadget are instances of the `bar-item` gadget and instances of the `submenu-item` gadget. The `add-item` method can be used to add new bar-items to a menubar, or to add new submenu-items to existing bar-items. Also, the following `Set-...` functions can be used to install a collection of components all at once.

`gg:set-menubar` *menubar new-bar-items*   [Function]
>   Removes all current bar-items from *menubar* and installs the *new-bar-items*. The *new-bar-items* should be a list of `bar-item` instances.

`gg:Set-Submenu` *bar-item submenu-items*   [Function]
>   Sets *bar-item* to have *submenu-items* in its submenu. *Submenu-items* is a list of `submenu-item` instances.

menubar *bar-item* [[*:where*] *position* [*locator*] [*:key*   [Method on `opal:add-item`]
      index-function]]
bar-item *submenu-item* [[*:where*] *position* [*locator*]   [Method on `opal:add-item`]
      [*:key index-function*]]

The *menubar*, *bar-item*, and *submenu-item* parameters above should be supplied with objects created by the functions in section [creating-menubar-components], page 453. The optional parameters follow the standard `add-item` definition described in the Aggregadgets chapter, and refer to the placement of the new `bar-item` among the existing bar-items. *Locator* may be either an existing `menubar` component, or some element of the `:items` list (like a submenu-title) used together with the *index-function* (see below).

After creating three `bar-item` instances, the example below shows how the bar-items can be attached as components to a `menubar`.

```
(setf bar1 (garnet-gadgets:make-bar-item
             :desc '("font-name" NIL (("courier") ("times") ("geneva")))))█
(setf bar2 (garnet-gadgets:make-bar-item
             :desc '("other-fonts" NIL (("helvetica") ("chicago")))))█
(setf bar3 (garnet-gadgets:make-bar-item
             :desc '("symbols" NIL (("mathematical") ("greek")))))
(opal:add-item DEMO-MENUBAR bar1)
(opal:add-item DEMO-MENUBAR bar2 :after "family" :key #'car)
(opal:add-item DEMO-MENUBAR bar3 :after bar2)
```

The following instructions show how submenu-items can be attached to oa `bar-item`. A `bar-item` object is first created, and then several submenu-items are attached to it using `add-item`:

```
(setf mac-bar (garnet-gadgets:make-bar-item :title "Mac Fonts"))
(setf sub-1 (garnet-gadgets:make-submenu-item :desc '("Gothic")))
(setf sub-2 (garnet-gadgets:make-submenu-item :desc '("Venice")))
(setf sub-3 (garnet-gadgets:make-submenu-item :desc '("Old English")))█
(opal:add-item mac-bar sub-1)
(opal:add-item mac-bar sub-2 :before sub-1)
(opal:add-item mac-bar sub-3 :after "Venice" :key #'car)
```

8.23.10 Removing components from the menubar

The `bar-item` and `submenu-item` components can be removed from the `menubar` with the `remove-item` method:

menubar *bar-item*   [Method on `opal:remove-item`]
bar-item *submenu-item*   [Method on `opal:remove-item`]

For example, if we have already created a `bar-item` called BAR-1 and added it to DEMO-MENUBAR, then the following line will remove that item: `(opal:remove-item DEMO-MENUBAR bar1)`

The `remove-item` method can also be used to remove submenu-items from bar-items. In order to remove a submenu item from the `bar-item` instance MAC-BAR, the following line can be used (provided SUB-1 is an existing `submenu-item` that was added to MAC-BAR): `(opal:remove-item mac-bar sub-1)`

8.23.11 Finding Components of the Menubar

gg:`Menubar-Components` *menubar*   [Function]
gg:`submenu-components` *bar-item*   [Function]
>   returns a list of `submenu-item` instances that are installed in *bar-item*'s submenu.

gg:`get-bar-component` *menubar item*   [Function]
>   returns the first `bar-item` object in *menubar* that corresponds to *item*. The *item* parameter may be a string or an atom, or one of the sublists of the *menubar*'s `:items` list.

gg:`get-submenu-component` *bar-item item*   [Function]
>   returns the first `submenu-item` object in *bar-item* that corresponds to *item*. The *item* parameter may be a string or an atom, or a string/function pair that describes a `submenu-item`.

gg:`find-submenu-component` *menubar submenu-title submenu-item*   [Function]
>   similar to `get-submenu-component`, except that `find-submenu-component` finds the appropriate `bar-item` instance in the given *menubar*. Returns the `submenu-item` object that corresponds to *submenu-item*. The parameter *submenu-title* should be the string or atom that is the title of some submenu. *Submenu-item* should be a string or atom, or a string/function pair that describes a `submenu-item` already installed in *submenu-title*.

8.23.12 Enabling and Disabling Components

gg:`menubar-disable-component` *menubar-component*   [Function]
>   disables *menubar-component*'s interactors and makes its label grayed-out. The user will not be able to select *menubar-component* while it is disabled. *Menubar-component* is an instance of `bar-item` or `submenu-item`.

gg:`menubar-enable-component` *menubar-component*   [Function]
>   enables *menubar-component*'s interactors and returns its label to solid text. *Menubar-component* is an instance of `bar-item` or `submenu-item`.

gg:`menubar-enabled-p` *menubar-component*   [Function]
>   Returns T if the *menubar-component* is enabled. *Menubar-component* is an instance of `bar-item` or `submenu-item`.

8.23.13 Other Menubar Functions

gg:`menubar-get-title` *menubar-component*   [Function]
>   returns the string or atom associated with *menubar-component*. The *menubar-component* must be an instance of a `bar-item` or `submenu-item` gadget.

gg:`menubar-set-title` *menubar-component string*   [Function]
>   changes the title of *menubar-component* to *string* and, if *menubar-component* is installed in a `menubar`, sets the `:items` slot of the `menubar` appropriately. *Menubar-component* can be either an instance of `bar-item` or `submenu-item`. *String* is a string or an atom, suitable for putting in the `:items` slot. Returns *string*.

gg:`menubar-installed-p` *menubar-component*   [Function]
  Returns `nil` if *menubar-component* is not attached to a `menubar`; otherwise returns
  the object it is installed in (either a `menubar` or a `bar-item`.

8.24 Labeled Box

```
┌─────────────────────────┐
│                         │
│        Label: ┌──────┐  │
│               │Field │  │
│               └──────┘  │
│                         │
└─────────────────────────┘
```

```
(create-instance 'gg:Labeled-Box opal:aggregadget
```

```
        (:maybe-constant '(:left :top :label-offset :field-offset :min-frame-width█
                           :label-string :field-font :label-font :visible))
        (:left 0)
        (:top 0)
        (:min-frame-width 10)
        (:label-offset 5)
        (:field-offset 6)
        (:label-string "Label:")
        (:value "Field")
        (:field-font opal:default-font)
        (:label-font (create-instance NIL opal:font (:face :bold)))
        (:selection-function NIL)   ; (lambda (gadget value))
        )
```

The loader file for the `labeled-box` is "labeled-box-loader".

The `labeled-box` gadget is comprised of a dynamic box with text both inside and to the left of the box. The text to the left of the box is a permanent label and may not be changed by the user. The text inside the box may be edited, however, and the width of the box will grow with the width of the string. As always, the current string inside the box may be accessed by the top level `:value` slot.
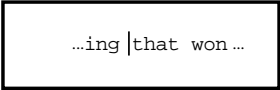
The width of the text frame will not fall below `:min-frame-width`.

The label to appear beside the box is in `:label-string`. The distance from the label to the left side of the box is specified in `:label-offset`, and the font of the label is in `:label-font`.

The distance from the box to the inner text is in `:field-offset`, and the font of the inner text is in `:field-font`.

8.25 Scrolling-Input-String

```
    (create-instance 'gg:Scrolling-Input-String opal:aggregadget
       (:maybe-constant '(:left :top :width :font :line-style :visible))
       (:left 0)
       (:top 0)
       (:width 100) ;  The width of the string area in pixels.
       (:value "Type here") ; The string that will originally appear in the
    ;  box and that will be changed.
       (:selection-function NIL) ; Function to be executed after editing text█
       (:font opal:default-font) ; **Must be fixed width**
       (:line-style opal:default-line-style)) ; line style can be used to set the color of
```

...ing |that won ...

The loader file for the `scrolling-input-string` gadget is "scrolling-input-string-loader".

This allows the user to enter a one-line edited string of arbitrary length, but only requires a fixed area on the screen since if the string gets too long, it is automatically scrolled left and right as needed. Three little dots (an ellipsis) are displayed on either side of the string if any text is not visible on that side.

The user interface is as follows: To start editing, click with the left mouse button on the string. To stop, hit `return`. To abort, hit `^g`. If the string gets to be too large to fit into

the specified width, then the string inside is scrolled left and right so the cursor is always visible. The cursor can be moved and text deleted with the usual editing commands (see the Interactors chapter, page 170).

The top level :value slot is set with the final value of the string appearing inside the box. This slot may be set directly to change the initial value, and formulas may depend on it. A function may be specified in the :selection-function slot to be executed after the field text has changed (i.e., after the carriage return). Room is left on both sides of the string for a "..." symbol which shows whether the string has been scrolled or not. Therefore, the string will not appear exactly at the :left or extend the full :width (since room is left for the ...'s even when they are not needed).

8.26 Scrolling-Labeled-Box

```
(create-instance 'gg:Scrolling-Labeled-Box opal:aggregadget
   (:maybe-constant '(:left :top :width :label-offset :field-offset
      :label-string :field-font :label-font :visible))
   (:left 0) (:top 0)
   (:width 130) ; The width of the entire area in pixels.  This must be big enough
         ; for the label and at least a few characters of the string!
   (:value "Field") ; The string that will originally appear in the
    ;   box and that will be changed.
   (:selection-function NIL) ; Function to be executed after editing text
   (:field-font opal:default-font) ; **Must be fixed width**

   (:label-string "Label:") ; The string that will appear beside the box
   (:label-offset 5) ; The distance between the label and the box
   (:field-offset 2) ; The distance between the field text and the box
   (:label-font (create-instance NIL opal:default-font (:face :bold))))
     ; The font of the string beside the box, can be variable-width
```
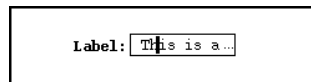
The loader file for the `scrolling-labeled-box` gadget is "scrolling-labeled-box-loader".

This is a combination of the `scrolling-input` gadget and the `labeled-box` gadget. It has a label and a box around the text. It operates just like the `scrolling-input-string`.

8.27 Graphics-Selection

```
(create-instance 'gg:Graphics-Selection opal:aggregadget
```

```
(:start-where NIL)
(:start-event :leftdown)
(:running-where T)
(:modify-function NIL)
(:check-line T)
(:movegrow-boxes-p T)
(:movegrow-lines-p T)
(:value NIL)
(:active-p T)
(:selection-function NIL)   ; (lambda (gadget value))
)
```
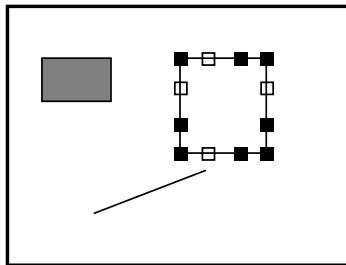
The loader file for `graphics-selection` is "graphics-loader".

The `graphics-selection` object is used to move and change the size of other graphical objects. (The `multi-graphics-selection` can select and move multiple objects – see section [multi-gs], page 464.) When the user clicks on a graphical object (from a set of objects chosen by the designer), the object becomes selected and small selection squares appear around the perimeter of the object. The user can then drag the white squares to move the object or drag the black boxes to change the size of the object. Pressing in the background (i.e., on no object) causes the currently selected object to become unselected. Clicking on an object also causes the previously selected object to become unselected since only one object may be selected at a time. While moving and growing, if the mouse goes outside of `:running-where` or if the `^g` key is pressed, the operation aborts.

The `graphics-selection` gadget should be added as a component to some aggregate or aggregadget of the larger interface, just like any other gadget. The objects in the interface that will be affected by the `graphics-selection` gadget are determined by the slots `:start-where` and `:running-where`.

The `graphics-selection` gadget sets the `:box` slot of the object being moved or grown. This is consistent with the behavior of the `move-grow-interactor`, discussed in the Interactors chapter. Therefore, you should create your objects with `:left`, `:top`, `:width`, and `:height` formulas that reference the `:box` slot.

The `:start-where` slot must be given a value to pass to an interactor to determine which items may be selected. The value must be one of the valid `...-or-none` forms for the interactors `:start-where` slot (see the Interactors chapter for a list of allowable values).

The `:start-event` slot specifies the event that will cause an object to be selected. The default is `:leftdown`, so if the left mouse button is clicked over an object in the `:start-where`, that object will become selected.

The `:running-where` slot is the area in which the objects can move and grow (see the Interactors chapter for allowable values).

If the `:check-line` slot is non-NIL, then the `graphics-selection` gadget will check the `:line-p` slot in the selected object, and if it is non-NIL then the interactor will select and change the object as a line. Instances of `opal:line` and `gg:arrow-line` already have their `:line-p` slots set to T. For other objects that should be selected as lines, the designer must set the `:line-p` slots explicitly (e.g., a composite object like an `arrow-line` is not really a line, though it should be treated like one).

If `:movegrow-lines-p` is `nil`, then the `graphics-selection` object will not allow a user to drag the selection squares of a line, and a beep will be issued if the user clicks on a selection square of a line.

If `:movegrow-boxes-p` is `nil`, then the `graphics-selection object` will not allow a user to drag the selection squares of a non-line, and a beep will be issued if the user clicks on a selection square of a non-line.

The `graphics-selection` gadget will be active when the value of its `:active-p` slot is T. To turn off the gadget, set this slot to `nil`.

The `:selection-function` slot specifies a function to be executed upon the selection of any object by the user. This function must take the parameters:

```
(lambda (gadget-object new-selection))
```

The `new-selection` parameter may be `nil` if no objects are selected (i.e., the user clicks in the background).

The designer can supply a `:modify-function` that will be called after an object is modified. It takes these parameters:

```
(lambda (gadget-object selected-object new-points))
```

The `new-points` will be a list of 4 numbers, either `left,top,width,height` or `x1,y1,x2,y2`.

8.28 Multi-Graphics-Selection

```
(create-instance 'gg:Multi-Graphics-Selection opal:aggregadget
  ;; programmer-settable slots
  (:active-p T)        ;; whether objects can be selected with the gadget
  (:start-where NIL) ;; supply a valid start-where here
  (:running-where T) ;; if supplied, then this is the area in which the
            ;; objects can move and grow
  (:selection-function NIL) ;; this is called when the selection changes
  (:modify-function NIL) ;; called when an object is changed size or position
  (:check-line T)          ;; whether to check for :line-p in objects
  (:check-polygon T)       ;; whether to check for :polygon-p in objects
  (:check-group T)         ;; whether to check for :group-p in objects
  (:check-grow-p NIL)      ;; whether to check for :grow-p in objects
  (:check-move-p NIL)      ;; whether to check for :move-p in objects
  (:move-multiple-p T)     ;; whether to move multiple objects as a group
  (:grow-multiple-p T)     ;; whether to grow multiple objects as a group
  (:input-filter NIL)      ;; used by the move-grow-interactor for grid-
ding, etc.
  (:want-undo NIL)         ;; whether to save information to allow undo
  (:multiple-select-p T) ;; if T, then multiple objects can be selected.

  (:other-multi-graphics-selection NIL]  ;; Used when several multi-selection gad-
gets in
                                      ;; different windows are work-
ing in conjunction.

  (:allow-component-select NIL)        ;; if T, then pressing with control will select
                                      ;; the component under the selected object.
  (:down-to-component-function gg::Point-To-Comp)  ;; a function that gets the
                                              ;; appropriate com-
ponent out
                                              ;; of the object un-
der the mouse.

  ;; slots the programmer can access
  (:current-selected-object NIL) ;; set with new selection or object to be moved
  ;; or grown before other slots are set.
  (:value NIL))  ;; current object or objects selected **DO NOT SET**
```
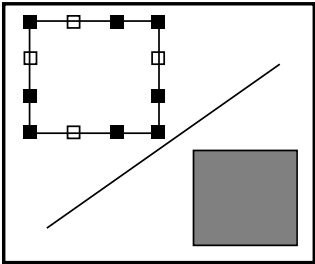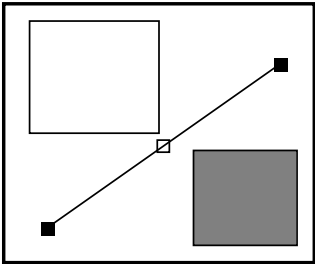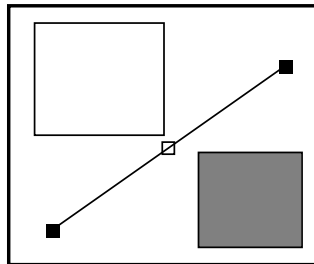
The loader file for `multi-graphics-selection` is "multi-selection-loader".

The `multi-selection` gadget is somewhat like the `graphics-selection`. The major difference is that multiple objects can be selected and manipulated by the user, and that the programmer must use a function to set the `:value` slot. Another difference is the way that the gadget checks whether move and grow is allowed.

This gadget exhibits the following features:

Given a list of graphical objects, the `multi-graphics-selection` aggregadget will cause selection squares to appear on the bounding box of selected objects.

One or more objects may be selected at a time, even when the objects are in different windows.

A built-in interactor displays the selection squares around an object at the time of a specified event (such as clicking a mouse button on the object).

Each selection square allows the user to move or grow the object by dragging the selection square.

The user can move and grow several objects simultaneously.

All of the objects inside a region (drawn by dragging the mouse) can be selected.

8.28.1 Programming Interface

Create an instance of the `gg:multi-graphics-selection` gadget and supply the `:start-where` slot with a valid list that can be passed to an interactor. This `:start-where` must return the items to be selected. It should be an `...-or-none` form, such as `:element-of-or-none`. An example of the parameter to `:start-where` is: `(list :element-of-or-none MYAGG)`

The `:value` slot of the `multi-graphics-selection` object supplies the object(s) the user selects. If `:multiple-select-p` is `nil` (the default), then it is a single object or `nil`. If `:multiple-select-p` is T, then will always be a list or `nil` (even if only one object is selected). Also, a `:selection-function` can be supplied and will be called each time the selection changes. It takes the parameters

`(lambda (gadget new-selection)` where *new-selection* is the new value of `:value`.

When your interface contains selectable objects in several windows, you can put a multi-selection gadget in each window and coordinate them all. Each gadget's `:other-multi-graphics-selection` slot should contain a list of ALL the multi-selection gadgets. Then, each gadget's `:value` will reflect selections in all windows. A known bug is that the selection order is NOT preserved across multiple windows (you can't tell which object was selected first or last). Also, you cannot drag objects from one window to another.

The user can change the size and/or position of the objects by pressing on the selection handles (see below). If the `:check-line` slot is non-NIL, then the `:line-p` slot in the object returned by `:start-where` will be gvd, and if it is non-NIL then the interactor will change the object as a line. Note that instances of `opal:line` and `gg:arrow-line` have their `:line-p` slot set to T by default. For other objects, the programmer must set the `:line-p` slots explicitly. There is analogous interaction between the `:check-group` and `:check-polygon` slots of the gadget and the `:group-p` and `:polygon-p` slots of the selected objects.

The programmer can supply a `:modify-function` that will be called after an object is modified. It takes these parameters: `(gadget selected-object new-points)` The *new-points* will be a list of 4 numbers, either `left,top,width,height` or `x1,y1,x2,y2`.

Programmer-settable slots:

In summary, public slots of the `multi-graphics-selection` gadget are:

> `:active-p` - If T, then the gadget will operate. If NIL, then none of the interactors will work. Setting to NIL does **not** clear the selection, however.

:start-where - Supply a valid start-where here.

:running-where - If supplied, then this is the area in which the objects can move and grow.

:selection-function - This is called when the selection changes.

:modify-function - This is called when an object is changed size or position.

:check-line - If T, the objects are checked for their :line-p slot and if that is non-NIL, then move or grown as a line.

:check-polygon - If T, the objects are checked for their :polygon-p slot and if that is non-NIL, then they are moved or grown as a polygon (by changing their :point-list slot).

:check-group - If T, the objects are checked for their :group-p slot and if that is non-NIL, then the individual components of the group are modified.

:check-grow-p - If T, then checks in each object to see if :grow-p is T, and if so, then will grow, else won't. If nil, then everything can grow. Default nil.

:check-move-p - If T, then checks in each object to see if :move-p is T, and if so, then will move, else won't. If nil, then everything can move. Default nil.

:move-multiple-p - If T, then if multiple items are selected and you press on a move box, then moves all of the objects. If nil, then just moves the object you point to. Default=T.

:grow-multiple-p - If T, then when multiple items are selected, grow boxes appear at the corners of the whole selection, and pressing there will grow all the objects. If nil, then those handles don't appear.

:input-filter - This is used by the move-grow-interactor for gridding. Consult the Interactors chapter for a list of allowed values.

:want-undo - If T, then saves information (conses) so that you can call undo-last-move-grow.

:allow-component-select - Whether to allow selection of components (see below). Default=NIL.

:down-to-component-function - A function that determines which component of an object has just been selected (see below). Default=NIL.

:multiple-select-p - If T, then multiple objects can be selected. Default=NIL.

Slots that can be accessed:

:value - set with list of the current selections, in reverse order the user selected the objects (first selected object is last in the list). **Do not set this slot.** Instead, use the function Set-Selection (see below).

:current-selected-object - set with new selection before other slots are set.

Selecting components of the currently selected object:

You can enable the selecting of the components of the selected objects by setting :allow-component-select to T. For example, if the :start-where lists a set of objects, this feature can allow the selection of the *parts* of those objects. When component selection is enabled, then by pressing the control-left mouse button over a selected object,

that object will be deselected, and its component will be selected instead. Similarly, if the `control-middle` mouse button or the `control-shift-left` mouse button is hit over a selected object, then that object is de-selected, and the object underneath is added to the selection set. The slot `:down-to-component-function` should contain a function to get the appropriate component out of the object under the mouse. This function might call a method in the selected object. Parameters are (`lambda obj x y`). It should return the object to be selected, or NIL. The default function calls `opal:point-to-component` directly.

Slots of the objects that can be selected are:

> `:line-p` - this should be T if the object should be moved as a line, and `nil` if as a rectangle

> `:group-p` - this should be T if the object is some instance of opal:aggregate and all its components should be moved as a group

> `:polygon-p` - this should be T if the object is a polyline and it should be moved by changing its `:point-list` slot

> `:points` - if `:line-p` is T, then the `:points` slot of the object is changed as the object is moved or grown.

> `:box` - if `:line-p` is `nil`, then the `:box` slot of the object is changed as the object is moved or grown.

> `:grow-p` - if this object can be changed size

> `:move-p` - if this object can be moved

Useful Functions:

gg:`set-selection` *gadget new-selection*   [Function]
> *Gadget* should be a `multi-graphics-selection` gadget, and *new-selection* is a list of objects that should be selected, or a single object to be selected, or `nil` to turn off all selections. The list passed in is not damaged.

gg:`undo-last-move-grow` *multi-graphics-selection-gadget*   [Function]
> When `:want-undo` is non-NIL (default is `nil`), then calling this function will undo the last move or grow and the selection will return to whatever it was when the objects were moved or grown. If you call `undo-last-move-grow` again, it undoes the undo (one-level undo). It is your responsibility to make sure that no objects were deleted or whatever between the grow and the call to undo.

> *Garnet does not yet have a general mechanism for Undo, so you should use this feature with care. It is currently your responsibility to keep track of what the last operation was and undo it.*

8.28.2 End User Operation

The user can press on any object with the left button, and it will become selected. Pressing on the background, causes no object to be selected (the current object becomes de-selected). Selecting an object with the left button causes the previous object to be de-selected. If the application allows multiple selection, then clicking with shift-left or middle on an object toggles it in the selection set.

Once an object is selected, it can be grown by pressing with the left button on one of the black boxes or moved by pressing on a white box. While moving and growing, if the mouse goes outside of :running-where or if `^g` is pressed, the operation aborts.

The gadget also allows the user to change the size of several objects at once. When multiple objects are selected, outline handles appear around each object, and the whole set can be moved by pressing on any of these handles. Additionally, when `:grow-multiple-p` is non-NIL, black handles appear at the four corners of the collection of objects, and these can be used to scale the entire group.

The gadget also allows objects to be selected in a region. If you press down and drag before releasing, then only the objects fully inside the dragged rectangle will become selected. If you do this with the left button, then they will be selected. If you do this with shift-left or the middle button, then all objects inside the rectangle will be toggled in the selection set (added if not there, removed if there).

8.29 Scrolling-Windows

There are two scrolling-window gadgets which have the standard Garnet look and feel, and two other scrolling-window gadgets that have the Motif look and feel (see section [motif-scrolling-window], page 566). These windows are based on the design from Roderick J. Williams at the University of Leeds for the Garnet contest. The `scrolling-window` gadget allows you to do your own scrolling. The `scrolling-window-with-bars` gadget comes with a horizontal and vertical scroll bar, which you can have on either side (and can turn off explicitly). Each scroll bar will go blank if the entire area to be scrolled in is visible in the window.
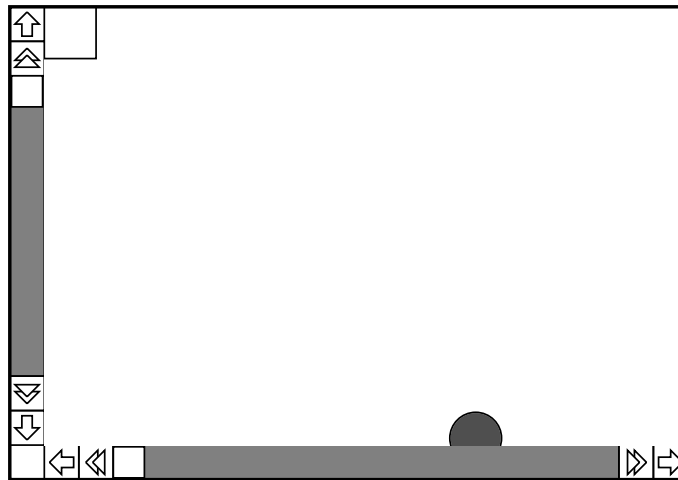
```
(create-instance 'gg:Scrolling-Window opal:aggregadget
   (:maybe-constant '(:title :parent-window))
   (:left 0)  ; left, top, width and height of window
   (:top 0)
   (:position-by-hand NIL) ; if T, then left,top ignored and user asked for win-
dow position
   (:width 150) ; width and height of inside of outer window
   (:height 150)
```

```
    (:border-width 2) ; of window
    (:parent-window NIL) ; window this scrolling-window is inside of, or NIL if top lev
    (:double-buffered-p NIL)
    (:omit-title-bar-p NIL)
    (:title "Scrolling-Window")
    (:icon-title (o-formula (gvl :title))) ; Default is the same as the title█
    (:total-width 200)    ; total size of the scrollable area inside
    (:total-height 200)
    (:X-Offset 0)  ; offset in of the scrollable area
    (:Y-Offset 0)
    (:visible T)  ; whether the entire window is visible (mapped)

    ;;  Read-Only slots
    (:Inner-Window NIL)  ; these are created by the update method
    (:inner-aggregate NIL) ; add your objects to this aggregate (but have to up-█
date first)
    (:outer-window NIL) ; call Opal:Update on this window (or on gadget itself)█

(create-instance 'gg:Scrolling-Window-With-Bars opal:aggregadget
    (:maybe-constant '(:left :top :width :height :border-width :title
                        :total-width :total-height :h-scroll-bar-p :v-scroll-bar-p█
       :h-scroll-on-top-p :v-scroll-on-left-p :min-scroll-bar-width
                        :scr-trill-p :page-trill-p :indicator-text-p :h-scr-incr█
                        :h-page-incr :v-scr-incr :v-page-incr :int-feedback-p█
                        :indicator-font :parent-window :icon-title :visible))█
    ;; Window slots
    (:left 0)  ; left, top, width and height of outermost window
    (:top 0)
    (:position-by-hand NIL) ; if T, then left,top ignored and user asked for win-█
dow position
    (:width 150) ; width and height of inside of outer window
    (:height 150)
    (:border-width 2) ; of outer window
    (:parent-window NIL) ; window this scrolling-window is inside of, or NIL if top lev
    (:double-buffered-p NIL)
    (:omit-title-bar-p NIL)
    (:title "Scrolling-Window")
    (:icon-title (o-formula (gvl :title))) ; Default is the same as the title█
    (:total-width 200)    ; total size of the scrollable area inside
    (:total-height 200)
    (:X-Offset 0)  ; x offset in of the scrollable area. CANNOT BE A FORMULA█
    (:Y-Offset 0)  ; CANNOT BE A FORMULA
    (:visible T)  ; whether the window and bars are visible (mapped)

    (:h-scroll-bar-p T)  ; Is there a horizontal scroll bar?
    (:v-scroll-bar-p T)  ; Is there a vertical scroll bar?
```

```
      ;; Scroll Bar slots
      (:h-scroll-on-top-p NIL)  ; whether horiz scroll bar is on top or bottom
      (:v-scroll-on-left-p T)   ; whether vert scroll bar is on left or right
      (:min-scroll-bar-width 20) ; these control both scroll bars
      (:scr-trill-p T)  ; single-line increment arrow buttons visible?
      (:page-trill-p T) ; page jump arrow buttons visible?
      (:h-scr-incr 10)  ; in pixels
      (:h-page-incr (o-formula (- (gvl :width) 10))) ; default jumps one page mi-
  nus 10 pixels
      (:v-scr-incr 10)  ; in pixels
      (:v-page-incr (o-formula (- (gvl :height) 10))) ; default jumps one page mi-
  nus 10 pixels
      (:int-feedback-p T) ; use NIL to have contents move continuously
      (:indicator-text-p NIL) ; Whether the pixel position is shown in the bars
      (:indicator-font (create-instance NIL opal:font (:size :small)))

      ;; Read-Only slots
      (:Inner-Window NIL)  ; these are created by the update method
      (:inner-aggregate NIL) ; add your objects to this aggregate (but have to up-
  date first)
      (:outer-window NIL) ; call Opal:Update on this window (or on gadget itself)
      (:clip-window NIL)
```

The loader file for both scrolling-window gadgets is "scrolling-window-loader".

Caveats:

If the scrolling-window has a `:parent-window`, update the parent window before instantiating the scrolling-window.

Update the scrolling-window gadget before referring to its inner/outer windows and aggregates.

The instance of the scrolling-window should **not** be added to an aggregate.

These gadgets supply a scrollable region using the X window manager's ability to have subwindows bigger than the parent window. Garnet moves the subwindow around inside the parent window and X handles the clipping. All the objects in the window are instantiated (and therefore take up memory), but they will not be drawn if outside. You must specify the total area to be scrolled in using the `:total-width` and `:total-height` fields. (Therefore, the scrolling window gadgets do not support semi-infinite planes–you must pick a size for the user to scroll around in.) Often, you can compute the size based on the contents to be displayed in the window. There can be a formula in the `:total-*` fields, but it should have an initial value. *Note: It is illegal to have windows with a zero or negative width and height, so the* `:total-width` *and* `:total-height` *should always be greater than zero.*

The width and height specified for the window is the inside of the outer window, not counting the scroll bars. For `scrolling-windows`, this will usually be the same as the size of the visible region. For `Scrolling-Window-With-Bars`, the visible portion is smaller by the size of the scroll bars, which is usually the value of the `:min-scroll-bar-width` slot (unless you turn on indicator text).

Each of these gadgets is special in that they add themselves to the windows that they create. Since windows are not like other Gadgets, you need to follow special rules with scrolling windows.

First, *do not add scrolling-window or scrolling-window-with-bars gadgets to any aggregates or include them in aggregadgets.* If you want a scrolling window to be inside another window, you must use the `:parent-window` slot instead.

Second, *you must call* `opal:update` *on a scrolling window gadget immediately after creating it, and before adding anything to the windows.* The update method causes the windows to be created. If you want to create a prototype of a scrolling window (and specify special values for some of the fields), you can skip the update call, but then you cannot add any contents to the window.

The aggregate to add the contents to is provided in the slot `:inner-aggregate` of the gadget after the update call. To make the scrolling-window a subwindow of another window, specify the `:parent-window` of the scrolling-window. If you want to put a sub-window inside a scrolling-window, use the window in the `:inner-window` slot of the scrolling window as the `:parent` of the newly created window.

As an example:

```
(create-instance 'MYSCROLL garnet-gadgets:scrolling-window-with-bars
    (:left 650)(:top 10)(:width 300)(:height 400)
    ;;note that the next two formulas must have initial values
    (:total-width (o-formula (gvl :inner-aggregate :width) 200))
    (:total-height (o-formula (gvl :inner-aggregate :height) 200)))
(opal:update MYSCROLL) ; Must update scrolling windows before using them.
(opal:add-components (gv MYSCROLL :inner-aggregate)
    all the objects to be added to the scrolling window
    )
;;; create a scrolling window inside the other scrolling window, just for fun
(create-instance 'SUB-SCROLLING-WINDOW garnet-gadgets:scrolling-window-with-bars
    (:left 15)(:top 15)(:width 150)(:height 150)
    (:parent-window (gv MYSCROLL :inner-window)))
```

With `Scrolling-Windows`, but *not* `Scrolling-windows-with-Bars`, you can explicitly set the `:X-offset` and `:Y-Offset` fields using `s-value` to adjust the position of the contents. For `Scrolling-windows-with-Bars`, you must use the following procedures to have your application program scroll the window. This is necessary to get the scroll bars to be updated correctly to show the window position. These procedures also work with `Scrolling-Windows`.

Useful functions:

gg:scroll-win-inc *scroll-win-gadget xinc yinc*   [Function]
This function scrolls a window by adding the specified values, which can be negative. Note that *xinc* and *yinc* are usually zero or negative numbers, since they are the offset top-left corner of the inner window from the top-left of the clipping window, so to see down in the document, the inner window must be moved up.

gg:scroll-win-to *scroll-win-gadget x y*   [Function]
This function scrolls a window by putting the specified coordinate at the upper left corner of the clip window.

`gg:show-box` *scroll-win left top right bottom*   [Function]

This function causes the scrolling-window *scroll-win* to scroll so that the region specified by *left*, *top*, *right* and *bottom* is visible. If the box is already visible, it will not cause the window to scroll. This can be used to cause the cursor in a text window, for example, or a "current item" to be visible. It is also used by the `focus-multifont` interactor.

If the box is larger than the visible region of the scrolling-window, the bottom and/or the rightmost parts of the box may remain hidden.

8.30 Arrow-line and Double-Arrow-Line
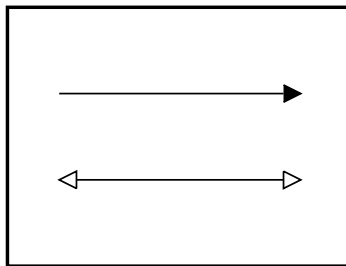
The **arrow-line** and **double-arrow-line** objects are comprised of a line and one or more arrowheads, effectively forming a single- or double-headed arrow. These objects are provided since the standard **opal:arrowhead** does not have an attached line.

8.30.1 Arrow-Line

```
(create-instance 'gg:Arrow-Line opal:aggregadget
  (:maybe-constant '(:x1 :y1 :x2 :y2 :line-style :open-p :filling-style :visible))█
  (:X1 0) (:Y1 0)
  (:X2 20) (:Y2 20)
  (:line-style opal:default-line-style)
  (:filling-style NIL)
  (:open-p T))
```

The loader file for the `arrow-line` is "arrow-line-loader".

The origin (or tail) of the `arrow-line` is the point (`:x1,y1`), and the tip is at (`:x2,y2`). The values for these slots may be formulas that depend on the value of slots in other Garnet objects. For example, if `:x2` and `:y2` depended on the `:left` and `:top` coordinates of some rectangle, then the arrow would point to the top, left corner of the rectangle regardless of the movement of the rectangle.[5]

The appearance of the arrowheads themselves may also be customized. The `:line-style` slot contains a value indicating the thickness of all lines in the `arrow-line` object. Opal exports a set of pre-defined line styles, which must be preceded by the Opal package name, as in `opal:line-0`. Available line style classes are: `no-line`, `thin-line`, `line-0`, `line-1`, `line-2`, `line-4`, `line-8`, `dotted-line` and `dashed-line`. Other line style classes may also be defined (see the Opal Chapter).

The slot `:filling-style` determines the shade of gray that will appear inside the arrowheads. Pre-defined filling styles are exported from Opal, and must again be preceded by the Opal package name. Available filling styles are `no-fill`, `black-fill`, `white-fill`, `gray-fill`, `light-gray-fill`, `dark-gray-fill`, and `diamond-fill`. The Opal function `halftone` may also be used to generate a filling style, as in (`:filling-style` (`opal:halftone 50`)), which is half-way between black and white fill.

The slot `:open-p` determines whether a line is drawn across the base of the arrowhead.

Additional features of the arrowhead may be customized by accessing the slot `:arrowhead` of the `arrow-line`. For example, the following instruction would set the `:diameter` of an `arrow-line` arrowhead to 20:

```
(s-value (gv MY-ARROW-LINE :arrowhead) :diameter 20)
```

The same customization may also be implemented when the instance is created:

```
(create-instance 'MY-ARROW-LINE garnet-gadgets:arrow-line
  (:parts `(:line (:arrowhead :modify
                              (:diameter 20)))))
```

8.30.2 Double-Arrow-Line

```
(create-instance 'gg:Double-Arrow-Line opal:aggregadget
  (:maybe-constant '(:x1 :y1 :x2 :y2 :line-style :open-p :filling-style
      :arrowhead-p :visible))
  (:X1 0) (:Y1 0)
  (:X2 40) (:Y2 40)
```

---

[5] See the KR chapter for a detailed discussion of constraints and formulas.

```
(:line-style opal:default-line-style)
(:filling-style NIL)
(:open-p T)
(:arrowhead-p :both))
```

The loader file for the `double-arrow-line` is "arrow-line-loader".

The endpoints of the `double-arrow-line` are at points (`:x1,:y1`) and (`:x2,:y2`). The slots `:line-style`, `:filling-style`, and `:open-p` are used exactly as in the `arrow-line`, with both arrowheads taking identical properties.

The additional slot `:arrowhead-p` designates which end(s) of the line will have arrowheads. Allowed values are:

> `0` or `NIL` - No arrowheads
>
> `1` - Arrowhead at coordinate (`:x1,:y1`)
>
> `2` - Arrowhead at coordinate (`:x2,:y2`)
>
> `3` or `:both` - Arrowheads at both ends

The arrowheads may be further customized as in the `arrow-line` object. The arrowheads are available in the slots `:arrowhead1` and `:arrowhead2`.

8.31 Browser Gadget

```
(create-instance 'gg:Browser-Gadget opal:aggregadget
  (:maybe-constant '(:left :top :num-menus :num-rows :menu-items-generating-function
                     :menu-function :item-to-string-function :additional-selection-p
     :item-font :title-font :visible))

  ;; Browser parameters
  (:left 0)
  (:top 0)
  (:num-menus 3)
  (:num-rows 5)
  (:menu-items-generating-function NIL)
  (:item-to-string-function #'(lambda (item) item))  ;; assume item is a string

  ;; Additional-selection parameters
  (:additional-selection-p T)
  (:additional-selection (o-formula ... ))
  (:additional-selection-function NIL)
  (:additional-selection-coordinate NIL)

  ;; Scrolling-Menu parameters
  (:item-font opal:default-font)
  (:title-font (create-instance NIL opal:font (:face :italic)))
  (:selection-function NIL)   ; (lambda (gadget value))
  )
```
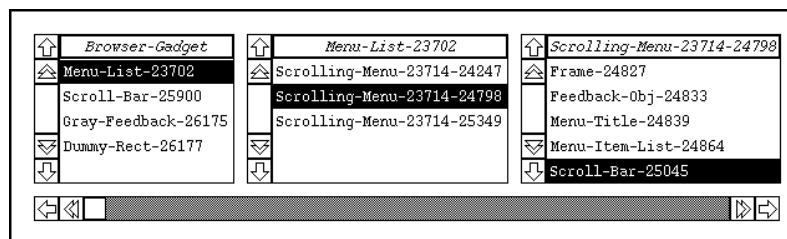
The loader file for the `browser-gadget` is "browser-gadget-loader".

The `browser-gadget` is a sophisticated interface device which may be used to examine hierarchical structures, such as Garnet object networks and directory trees. The gadget is composed of a set of scrolling menus, where the selections in each scrolling menu correspond to the children of the item appearing in the title. Clicking on one of the menu selections causes that selection to appear as the title of the next scrolling menu, with all of its children appearing as choices in the new menu. Additionally, clicking the middle mouse button over a menu selection causes a gray feedback box to appear, indicating a secondary selection.

"demo-file-browser", are included in the Garnet `demos` sub-directory as examples of how the `browser-gadget` is used in an interface. With the schema browser, the user may examine the inheritance and aggregate hierarchies of Garnet, while the file browser can be used to examine the file hierarchy of Unix directories.

8.31.1 User Interface

initially appear in a window with an item already displayed in the first menu. (Alternatively, the designer may provide a mechanism such as a `labeled-box` gadget through which the user initializes a fresh browser with the first item.) The selections in the first menu are derived from the item in the title through a specified function. When the user clicks the left mouse button on one of the menu choices, that selection will appear in the title of the next menu, with all of that item's "children" appearing as choices. If the item that the user selects does not generate any children, then a new menu is not generated.

The user may also click on a menu selection with the middle mouse button, causing the selection to be bordered by a gray rectangle. This selection is called the "additional selection", and there is only one for all of the menus in the `browser-gadget`.

The choices that are visible in each menu are controlled by the scroll bars appearing on the sides of the menus. If there are more menu selections derived from the title item than can be shown in a menu, then the background of the scroll bar will be gray and a white indicator box will appear. Clicking the left mouse button on the trill boxes at the top and bottom of the scroll bars will "scroll" more selections into the menu. Clicking on the single arrow trill boxes causes the visible selections to scroll one at a time, and clicking on the double arrow trill boxes will cause an entire "page" of new selections to appear (one page is equal to the number of items visible in the menu). The user may also drag the indicator of a scrolling menu scroll bar to adjust the visible selections.

Analogously, the horizontal scroll bar appearing below the menus may be adjusted to change which menus are displayed. When there are more menus to show than are allowed at one time, then the trill boxes can be clicked to scroll either one menu at a time or a whole "screen" of menus. Dragging the indicator in this scroll bar will cause a black rectangle to follow the mouse, rather than the indicator box itself. When the user releases the black rectangle, the indicator will jump to the position where it was released.

8.31.2 Programming Interface

8.31.3 Overview

It is important to note that the programming interface to the `browser-gadget` is different than in other Garnet gadgets. Due to the complexity of the gadget, this section is provided as a guide to the essential elements of the `browser-gadget` so that the designer can create and use an instance immediately. Subsequent sections describe in greater detail the slots and functions mentioned in this section.

When creating an instance of the `browser-gadget`, there is one slot that **must** be set. The slot `:menu-items-generating-function` must be provided with a function that generates children from the items that are to be shown in the titles of the menus. This function takes an item and returns a list of items that correspond to menu selections. These items can be of any type, but if they are not strings, then the slot `:item-to-string-function`

must **also** be set with a function to derive strings from the items (its default value is the identity function). These functions are discussed further in section [gen-fns], page 483.

The `:items` slot adheres to the convention that if an element of this list is a list, then the second element is an item-function. The `:item-to-string-function` (described below) is applied to the first element of the item list to get a label for a menu selection. If data is to be stored in the elements of the `:items` list, it should be included as the third or greater elements in the item lists (see section [items-slot], page 410).

To install an item in a `browser-gadget` instance, the function `set-first-item` should be called with the parameters of the name of the browser instance and the new item. A subsequent update of the window containing the instance will show the item appearing in the first menu with all of its children. Other functions used to manipulate the `browser-gadget` are discussed in section [manipulating], page 484.

8.31.4 An example

Suppose that we want to define an instance of the `browser-gadget` to look at the inheritance hierarchy of Garnet schemas. First, create an instance called BROWSER-1 with the appropriate generating functions (these particular lambda-expressions are analyzed in [gen-fns], page 483).

```
(create-instance 'BROWSER-1 garnet-gadgets:browser-gadget
   (:menu-items-generating-function #'(lambda (item)
(gv item :is-a-inv)))
   (:item-to-string-function #'(lambda (item)
 (if item
     (string-capitalize (kr:name-for-schema item))
      "")))) 
```

The BROWSER-1 schema can be added to a Garnet window in the usual way:

```
(create-instance 'WIN inter:interactor-window
   (:width 600) (:height 200)
   (:aggregate (create-instance 'AGG opal:aggregate)))
(opal:add-component AGG BROWSER-1)
(opal:update win)
```

Now, we can initialize the BROWSER-1 object with a Garnet schema, such as the `opal:rectangle` schema:

```
(garnet-gadgets:set-first-item BROWSER-1 opal:rectangle)
(opal:update win)
```

All instances of `opal:rectangle` that currently exist will be shown in the first menu. Clicking on one of the selections in this menu will cause that selection to appear in the title of the second menu, with all of its instances as selections.

Since `opal:rectangle` is an instance of the `opal:graphical-object` schema, we can use the `push-first-item` (described in section [manipulating], page 484) to show all of the objects that are instances of `opal:graphical-object`. If we call

```
(garnet-gadgets:push-first-item BROWSER-1 opal:graphical-object)
```

then the "Rectangle" title will be moved into the second menu along with all of its selections, and the "Graphical-Object" item will be displayed in the first menu with all

of its instances. The "Rectangle" selection under the "Graphical-Object" title will be highlighted, since it was matched with the title of the second menu.

8.31.5 Generating Functions for Items and Strings

`:menu-items-generating-function` contains a function which generates menu selections from each item in the scrolling menu titles. The function takes an *item* as a parameter, and returns a list of menu items which correspond to the selections in the scrolling menus. For example, if a `browser-gadget` instance is to be initialized with a Garnet schema, and the menus should display all of the instances of each item, then the `:menu-items-generating-function` appearing in the example of section [browser1-example], page 482, is appropriate. It should be noted that this function does not need to return a list of strings, but that eventually strings will be generated from the items that it returns (via the function in `:item-to-string-function`).

   `:item-to-string-function` is used to generate strings from arbitrary items obtained from the `:menu-items-generating-function`. If the generated items are strings themselves, then the `:item-to-string-function` may retain its default value. The strings returned by the `:item-to-string-function` will be displayed as the titles and selections of the scrolling menus. In the example of section [browser1-example], page 482, the `:menu-items-generating-function` returns a list of Garnet schemas. So the supplied `:item-to-string-function` takes a schema as a parameter and returns the string name of the schema. Notice that when there are fewer items than there are menus, this function will generate empty strings for the titles of the blank menus.

8.31.6 Other Browser-Gadget Slots

The number of menus to be displayed horizontally in the `browser-gadget` is determined by the slot `:num-menus`. Since the set of menus in the gadget is implemented with an aggrelist, the menu objects will be adjusted automatically to correspond with the new value during the next call to `opal:update`. Analogously, the slot `:num-rows` determines the number of vertical selections to appear at one time in each scrolling menu.

   The slots `:title-font` and `:item-font` control the fonts for the titles of the menus and the menu selections, respectively.

   The function specified in `:selection-function` is executed when the user selects an item from one of the scrolling menus. The parameters of this function are

```
(lambda (browser-instance item))
```

where the *item* is an object generated by the function specified in `:menu-items-generating-function`. This function is executed after some internal bookkeeping is performed to update the `browser-gadget`.

8.31.7 The Additional Selection

button over one of the scrolling menu selections, the outline of a gray rectangle will appear over the selection. The item chosen in this manner is called an "additional selection".

   Whether this feature is active is determined by the value of the slot `:additional-selection-p`.

   The item identified by the additional selection may be accessed through the slot `:additional-selection`. The value in this slot will correspond to some item returned by

the function specified in :menu-items-generating-function. **Note:** this slot cannot be set directly to move the gray feedback box. Instead, the :additional-selection-coordinate slot must be set.

Since items may frequently be scrolled off to the side of the browser, it may not be possible to name explicitly the item which the gray feedback object should appear over. However, the "coordinate" of the additional selection can always be named in the slot :additional-selection-coordinate. This slot is set when the user selects the additional selection, and it may be set directly by the programmer. The :additional-selection-coordinate slot contains a list of two values – the first is the rank of the menu which the selection appears in, and the second is the rank of the selection within the menu. Both ranks are zero-based, and are relative to the full lengths of the two item lists, not just the items currently visible.

The function specified in the slot :additional-selection-function will be executed when the user chooses the additional selection. The parameters are

```
(lambda (browser-instance item))
```

where *item* was just selected by the user. If the user presses over the previous additional selection, it will become deselected, and the :additional-selection-function will be called with nil as the *item* parameter.

8.31.8 Manipulating the browser-gadget

been created, an item can be installed in the instance as starting object by calling set-first-item with the parameters

gg:**Set-First-Item** *browser-instance new-item*   [Function]
>   The effect of calling this function is to install the *new-item* in the :items slot of the instance, and to initialize the bookkeeping slots of the instance.
>
>   to add an item to the front of a browser-gadget instance. It takes the parameters

gg:**push-first-item** *browser-instance new-item*   [Function]
>   and adds the *new-item* to the front of the *browser-instance*'s :items list and adjusts the bookkeeping slots of the instance appropriately. A selection in the first menu is highlighted only if a match is found with the title of the second menu (which causes the browser to appear as though the second menu was actually generated from clicking on the selection in the first menu).
>
>   install a new first item in an instance when the desired item already appears as a selection in one of the scrolling menus. The function is given the parameters

gg:**promote-item** *browser-instance coordinate*   [Function]
>   where *coordinate* is a list of two numbers corresponding to the location of the desired item in the *browser-instance*. The syntax of the coordinate list is defined in section [additional], page 483. If the item whose coordinate is passed is highlighted, then all of the menus to the right of the selection are retained; otherwise, the item becomes the only item in the instance.

8.32 Polyline-Creator
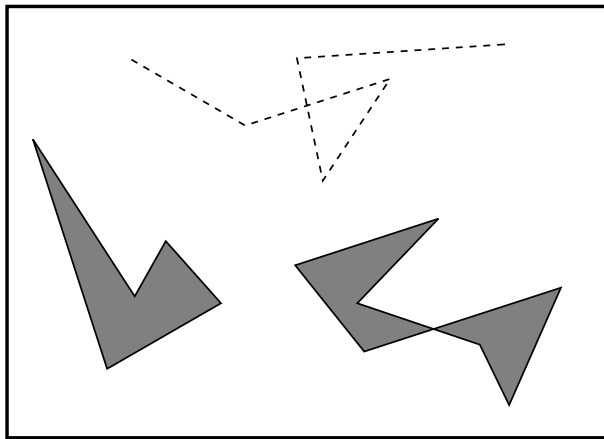

```
(create-instance 'gg:Polyline-Creator opal:aggregadget

     (:selection-function NIL) ; called when have full poly-line
     (:start-event :leftdown)  ; the event to start the whole process on
```

```
     (:start-where NIL)            ; where the mouse should be when the start-█
  event happens
     (:running-where T)
     (:close-enough-value 3)   ; how close a point should be to the first point to stop
     (:input-filter NIL)

     ; Editing parameters
     (: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█
     (:polyline-being-edited NIL)        ; read-only slot

     ; Return value
     (:value NIL) ; set with final point list
```

The loader file for the `polyline-creator` gadget is "polyline-creator-loader". Examples of creating and editing polylines are in the GarnetDraw demo and the small (`gg:polyline-creator-demo-go`) which is loaded by default with the `polyline-creator`.

This gadget allows the user to enter new polylines (lists of points), while providing feedback. It also supports polyline editing, meaning that you can add, remove, and move points of a polyline with the mouse.

8.32.1 Creating New Polylines

The user interface for creating polylines is as follows: The user presses a button (specified in the `:start-event` slot) to start the interaction. Each subsequent button press causes a new segment to be added to the line. Feedback is provided to the user. The Polyline stops when:

> the new point is close enough (within `:close-enough-value` pixels) to the first point of the polyline (in which case the polyline is closed).
>
> a button pressed is different from the start event (in which case the polyline is open).
>
> the application calls the function `Stop-Polyline-Creator` (see below).

The gadget can also be aborted if the user types ^g or the application calls `abort-polyline-creator`.

The function in the `:selection-function` is called to create the new polyline. This function should not destructively modify the point-list, but should instead *copy* the point-list if it will be changed. This functions is called with the parameters

```
     (lambda (gadget new-point-list)
```

where *new-point-list* is of the form: (x1 y1 x2 y2 x3 y3 ...).

The `:input-filter` slot is used just as in the `move-grow-interactor` and the `two-point-interactor`, described in the Interactors chapter.

The `:value` slot is also set by the gadget with the final point-list. Applications are not allowed to set this directly (there can be no default value for this gadget).

8.32.2 Editing Existing Polylines

`gg:Toggle-Polyline-Handles` *polyline-creator-gadget polyline* `[function]`,
`page 90`

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` - Default = `:leftdown`. The event to start moving a
> point.
>
> `:mover-stop-event` - Default = `:leftup`. The event to stop moving a point.
>
> `:adder-start-event` - Default = `:leftdown`. The event to add a point.
>
> `:deleter-start-event` - Default = `:middledown`. The event to delete a point.
>
> `:threshold` - Default = 3. How close you have to click next to a line to add a
> point.

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.

8.32.3 Some Useful Functions

`gg:stop-polyline-creator` *gadget*   [Function]
    This causes the gadget to create the current object. It ignores the current mouse
    position. This is useful if some other gadget (such as a palette changing the drawing
    mode) wants to stop the gadget. You can call this even if the gadget is not operating.

`gg:abort-polyline-creator` *gadget*   [Function]
    This aborts the gadget without creating the polyline.

8.33 Error-Gadget

```
Error: Invalid input from user.
      Press OK to continue.

            [OK]
```
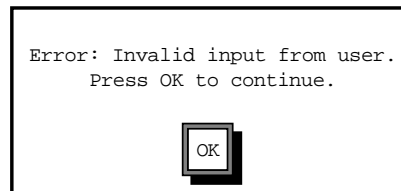
```
(create-instance 'gg:Error-Gadget opal:aggregadget
   (:parent-window NIL)
   (:font opal:default-font)
   (:justification :center)
```

```
        (:modal-p T)
        (:beep-p T)
        (:button-name "OK")
        (:window NIL)                    ; Automatically initialized
        (:selection-function NIL)   ; (lambda (gadget value))
        ...)
```

The loader file for the `error-gadget` is "error-gadget-loader".

The `error-gadget` is a dialog box used to tell the user that an error has occurred. When activated, the user sees a window appear with a multi-line text message and an "OK" button centered in the window. If specified by the designer, all activities in the rest of the interface will be suspended until the user clicks on the "OK" button to cause the error window to disappear.

There is also a `motif-error-gadget`, which is described in section [motif-error-gadget], page 557.

Some utility functions in section [top-careful-eval], page 490, allow you to easily raise an `error-gadget` in the context of checking user input for errors.

Caveats:

Update the parent window before instantiating the error-gadget.

The instance of the error-gadget should **not** be added to an aggregate.

8.33.1 Programming Interface

In order to associate an error window with an application, an instance of the `error-gadget` should be created with the `:parent-window` slot set to the window of the application. The error window is activated by calling one of the functions

gg:display-error *error-gadget* **&optional** *message*   [Function]
gg:display-error-and-wait *error-gadget* **&optional** *message*   [Function]

where the parameter *error-gadget* is the instance created by the user and *message* is a string to be displayed in the window. If *message* is not supplied, then the value in the `:string` slot of the gadget is used. The message may have multiple lines, indicated by carriage returns within the text string. While the `display-error` routine returns immediately when the dialog box appears, `display-error-and-wait` does not return until the user hits the OK button. The return value of both functions is always T.

When the error-gadget is associated with a parent window, the error window will appear centered inside of this window. If `:parent-window` is `nil`, then the error window will appear at coordinates (200,200), relative to the upper left corner of the screen.

The font of the message is specified in the `:font` slot. The `:justification` slot is used to specify whether to align the text against the left or right margin of the window or whether each line should be centered in the window (allowed values are `:left`, `:right`, and `:center`).

If the value of the `:modal-p` slot is T, then all interactors in the rest of the interface will be suspended, and the user will not be able to continue working until the "OK" button has been pressed. If `:modal-p` is `nil`, then the interface will continue to function with the error window visible.

If the `:beep-p` slot is T, then Garnet will sound a beep when the gadget becomes visible. To turn off the beep, set `:beep-p` to `nil`.

The `:button-name` slot determines the label of the button. Since the `display-error` routines do *not* take this as a parameter, it must be set in the gadget itself.

After the instance of the `error-gadget` has been created, the window which will contain the text and the button may be accessed through the `:window` slot of the instance. Note: When the `error-gadget` instance has a parent-window, the `:left` and `:top` coordinates of this window will be relative to the parent-window. Otherwise, they are relative to the full screen.

8.33.2 Error-Checking and Careful Evaluation

There are several 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.

8.33.3 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:

```
lisp> (gg:careful-eval '(+ 4 5))    ;; evaluates successfully
9
lisp> (gg:careful-eval '(+ 4 y))    ;; signals an error
NIL
#<EXCL::SIMPLE-ERROR.0>
lisp> (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
lisp>
```

8.33.4 Careful-Read-From-String

> gg:Careful-Read-From-String *string* &optional *error-gadget error-message* [function],■
> page 90

`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.

8.33.5 Careful-String-Eval

> gg:Careful-String-Eval *string* &optional *error-gadget error-message* [function],■
> page 90

`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*.

8.33.6 Careful-Eval-Formula-Lambda

> gg:Careful-Eval-Formula-Lambda *expr error-gadget error-message* [function],■
> page 90
>
>   *the-obj the-slot the-formula warn-p*

`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:
```
lisp> (create-instance 'R opal:rectangle
         (:my-left 67))
```

```
Object R
#k<R>
lisp> (gg:careful-eval-formula-lambda '(gvl :my-left) NIL NIL
                                            R :left :ignore NIL)
67
lisp>
```

8.34 Query-Gadget

```
(create-instance 'gg:Query-Gadget gg:error-gadget
   (:button-names '("OK" "CANCEL"))
   (:string "Is that OK?")
   (:parent-window NIL)
   (:font opal:default-font)
   (:justification :center)
   (:modal-p T)
   (:beep-p T)
   (:window NIL)                 ; Automatically initialized
   (:selection-function NIL)    ; (lambda (gadget value))
   ...)
```

The loader file for the `query-gadget` is "error-gadget-loader" (the `query-gadget` is in the same file as the `error-gadget`).

The `query-gadget` is similar to the `error-gadget`, but it allows more buttons in the window, so it is useful for a general purpose dialog box. The button names are supplied in the `:button-names` slot of the `query-gadget` or as a parameter to the display functions. The use of the `query-gadget` is the same as the `error-gadget` (and the same caveats apply). There is also a `motif-query-gadget`, which is described in section [motif-query-gadget], page 558.

To display a query-gadget, you first create an instance of `query-gadget`, and then call one of:

> display-query *query-gadget* &optional *message label-list*

> display-query-and-wait *query-gadget* &optional *message label-list*

The *message* is the string to display, and the optional *label-list* allows you to change the buttons. It should be a list of strings, atoms or keywords. If *message* is not supplied, then the value of the `:string` slot of the gadget is used. This function displays the query-gadget on the screen and then returns immediately. The `selection-function` of the query gadget (if any) is called with the item from the label-list the user selected. While the `display-query` routine returns immediately when the dialog box appears, `display-query-and-wait` does not return until the user hits one of the buttons. The return value `display-query-and-wait` is the label of the selected button.

8.35 [Save Gadget]

```
(create-instance 'gg:Save-Gadget opal:aggregadget
  (:maybe-constant '(:parent-window :window-title :window-left :window-top
```

```
          :message-string :num-visible :initial-directory :button-panel-items
                       :button-panel-h-spacing :min-gadget-width :modal-p
                       :check-filenames-p :query-message :query-buttons
                       :dir-input-field-font :dir-input-label-font :message-font█
                       :file-menu-font :file-input-field-font :file-input-label-font█
                       :button-panel-font))
(:parent-window NIL)
(:window-title "save window")
(:min-gadget-width 240)
(:initial-directory "./")
(:message-string "fetching directory...")
(:query-message "save over existing file")
(:button-panel-items '("save" "cancel"))
(:button-panel-h-spacing 25)
(:num-visible 6)
(:check-filenames-p t)
(:modal-p NIL)
(:selection-function NIL)   ; (lambda (gadget value))

(:dir-input-field-font (opal:get-standard-font NIL NIL :small))
(:dir-input-label-font (opal:get-standard-font NIL :bold NIL))
(:file-input-field-font (opal:get-standard-font NIL NIL :small))
(:file-input-label-font (opal:get-standard-font NIL :bold NIL))
(:message-font (opal:get-standard-font :fixed :italic :small))
(:button-panel-font opal:default-font)
(:file-menu-font (opal:get-standard-font NIL :bold NIL))
...)
```
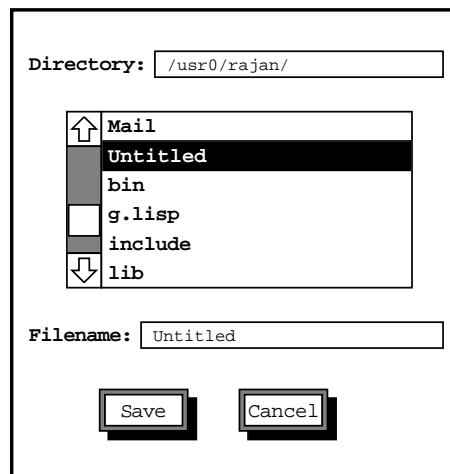
Figure 8.14: A save-gadget showing the contents of directory **/usr0/rajan/**

The loader file for the `save-gadget` is "save-gadget-loader" (which also loads the `load-gadget`). Figure [save-gadget-tag], page 494, shows a picture of the save gadget.

The `save-gadget` is a dialog box used to save a file, while displaying the contents of the destination directory in a scrolling menu. The gadget has an accompanying query-gadget dialog box (not shown) that can ask the user if the file really should be saved before the `save-gadget` appears. This is an extra level of convenience for the application designer.

There is also a `motif-save-gadget`, as well as a `load-gadget` and `motif-load-gadget`.

Caveats:

Update the parent window before instantiating the save-gadget.

The instance of the save-gadget should **not** be added to an aggregate.

8.35.1 Programming Interface

When a save gadget is created, it does not appear automatically. Like the query and error gadgets, it has its own display function. The save window is activated by calling one of these functions:

> `gg:Display-Save-Gadget` *save-gadget* &optional *initial-filename*[No value for ''function'']

> `gg:Display-Save-Gadget-And-Wait` *save-gadget* &optional *initial-filename*[No value for ''function'']

While the `display-save-gadget` routine returns immediately when the dialog box appears, `display-save-gadget-and-wait` does not return until the user hits either the "Save" or "Cancel" button. If an *initial-filename* is provided, it will appear in the "Filename:" box when the gadget is displayed.

NOTE: To change the directory, set the `:initial-directory` slot of the gadget to be the new directory. Then, when you call one of the display methods, the directory will be updated.

To hide a save window, use

`gg:hide-save-gadget` *save-gadget*   [Function]
> The following function is described in section [save-file-if-wanted-fn], page 498.

`gg:save-file-if-wanted` *save-gadget* **&optional** *filename*   [Function]
> (*query-string* **"save file first"**)

> When a save-gadget is first displayed, the "Directory" box will contain the present directory (unless otherwise specified, as explained in the next section); the scrolling-menu will have the contents of that directory; and the "Filename" box will be blank.

> Whenever the directory name is changed by the user, the scrolling menu will also change to list the contents of the new directory. If an invalid directory name is specified, there will be a beep and the invalid name will be replaced by the previous name. Whenever a directory is being fetched, a brief message (by default, "Fetching directory...") will appear, and will go away when the scrolling menu has been updated. When a file name is typed into the "Directory" box, the file name will be moved down to the "Filename" box, and the menu will be updated.

> If a file in the scrolling menu is selected, then the "Filename" box will contain the name of that file. If a directory is selected, the "Directory" box will be set to the selected directory, and the scrolling menu will once again update itself.

If an invalid file name is typed into the "Filename" box, there will be a beep and the "Filename" box will be reset. An invalid file name is one that has a directory name in it ("/usr/garnet/foo", for example).

The following slots may be changed to customize the `save-gadget`:

> `:window-title` - contains the title of the save window, which is by default "Save Window". Window managers usually do not display titles for subwindows (i.e., if a window is specified in `:parent-window`).

> `:parent-window` - if this slot contains a window, then the `save-gadget` will appear as a subwindow of that window. By default, the gadget will automatically be centered inside the parent window. If this is not desired, the `:window-left` and `:window-top` slots can be changed to position the gadget.

> `:window-left` and `:window-top` - specify the coordinates of the dialog box. Default values are 0 for both slots unless there is a parent window.

> `:initial-directory` - the directory to display when the `save-gadget` appears. The default is "./", which is the current directory as determined by the lisp process.

> `:message-string` - the message to display to the user while the save gadget fetches the contents of a new directory. Default is "Fetching directory...".

> `:num-visible` - how many files to display in the scrolling menu. Default is 6.

> `:button-panel-items` - a list of names for the buttons. The default is '("Save" "Cancel"). NOTE: It is important that, when you rename the buttons and use the `default-save-function`, you rename them in the "Save" "Cancel" order. That is, the label that should cause the gadget to save must appear first in the `:items` list, and the label that cancels the gadget's action must appear second. For example, if you rename the `:button-panel-items` slot as '("Go" "Return"), it will produce the correct results. However, if you use '("Return" "Go") instead, the wrong functions will get called.

> `:button-panel-h-spacing` - the distance between the buttons (default 25).

> `:min-gadget-width` specifies the width of the "Directory" and "Filename" boxes. The scrolling menu is centered between them.

> `:modal-p` - when T, then interaction in other Garnet windows will be suspended untill either the "Save" or the "Cancel" button is hit.

> `:check-filenames-p` - whether to check to see whether the file already exists before saving. If the file exists, then a query gadget will pop up and ask for confirmation.

> `:query-message` - the string that will be used in the query gadget that pops up when you try to overwrite a file. If `:check-filenames-p` slot is `nil`, this slot is ignored.

> `:selection-function` - as usual, the function called when the "Save" button is hit.
>
> `:dir-input-field-font` and `:dir-input-label-font` - the fonts for the field and label of the "Directory" box.
>
> `:file-input-field-font` and `:file-input-label-font` - the fonts for the field and label of the "Filename" box.
>
> `:message-font` - the font to use for the message that appears when the directory is being fetched.
>
> `:file-menu-font` - the font of the items inside the scrolling menu
>
> `:button-panel-font` - the font for the buttons

8.35.2 Adding more gadgets to the save gadget

It is possible to add more gadgets, such as extra buttons, etc. to the save gadget. To do this, you simply add more components to the `:parts` list of the save gadget (which is an aggregadget). However, you MUST include the following 5 components in the parts list: `:dir-input`, `:file-menu`, `:file-input`, `:message`, and `:OK-cancel-buttons`.

An example of adding more gadgets to a save gadget follows:

```
(create-instance 'SG gg:save-gadget
  (:parts
   '(:dir-input :file-menu :file-input :message :OK-cancel-buttons
     (:extra-button ,gg:text-button
      (:left 10) (:top 220)
      (:text-offset 2) (:shadow-offset 5) (:gray-width 3)
      (:string "Test")))))
```

This will, in addition to creating the standard save gadget parts, create an additional button. This button can be accessed by using (`gv SG :extra-button`). Naturally, you can have selection functions, etc. to whatever gadgets you add. However, it is extremely important to include the `:dir-input`, `:file-menu`, `:file-input`, `:message` and `:OK-cancel-buttons` in the `:parts` list.

NOTE: The save/cancel buttons automatically position themselves 25 pixels below the last gadget in the `:parts` list, since most people desire the buttons at the bottom of the gadget. If this is not desired, you can modify the `:top` slot of the `:OK-cancel-buttons`.

8.35.3 Hacking the Save Gadget

The slots described above should be enough to customize most applications. However, when that is not the case, it is possible to hack the save gadget.

For example, the save/cancel buttons are centered with respect to the "Filename" box. If this is not desirable, the `:OK-cancel-buttons` slot can be modified to the desired left and top coordinates.

Suppose the left of the save/cancel buttons should be at 10. The save gadget then would look like:

```
(create-instance 'sg gg:save-gadget
  (:parts
   '(:dir-input
```

```
        :message
        :file-menu
        :file-input
        (:OK-cancel-buttons :modify (:left 10)))))
```

8.35.4 The Save-File-If-Wanted function

If you are using a menubar with a "File" menu, you might want to use the `save-file-if-wanted` function. You would call this function before such operations as quit, close, and read if the contents of the window had not yet been saved. The format for this function is:

gg:`save-file-if-wanted` *save-gadget* **&optional** *filename* [Function]
        (*query-string* **"save file first**)

> This function will pop up a query gadget that asks "Save file first?", or whatever you specify as the *query-string*. If "Yes" is selected, then it will call the standard `display-save-gadget-and-wait` function on the given filename, and the return value of this function will be the same as the return value for the *save-gadget*'s `:selection-function`. If "Cancel" is selected, it will return :CANCEL. If "No" is selected, it will return :NO.

> For an example of when and where this function can be used, look at the source code for Garnetdraw, under the section labeled "MENU FUNCTIONS AND MENUBAR". The Open, New and Quit functions all call this function.

> Often, it is necessary to know if the "Cancel" button was hit or not. For this purpose, the functions `save-file-if-wanted` and the `display-save-gadget-and-wait` return `:cancel` if the "Cancel" button was hit. For example, the quit function in Garnetdraw looks like this:

```
    (defun quit-fun (gadget menu-item submenu-item)
      (unless (eq :cancel (gg:Save-File-If-Wanted *save-db* *document-name*))
        (do-stop)))
```

> If the user clicks on "Cancel" either in the "Save file first?" query box, or in the save-gadget itself, `save-file-if-wanted` will return `:cancel`.

8.36 [Load Gadget]

```
    (create-instance 'gg:Load-Gadget opal:aggregadget
      (:maybe-constant '(:parent-window :window-title :window-left :window-top
                         :message-string :num-visible :initial-directory :button-panel-item
                         :button-panel-h-spacing :min-gadget-width :modal-p
                         :check-filenames-p :dir-input-field-font :dir-input-label-font
                         :message-font :file-menu-font :file-input-field-font
                         :file-input-label-font :button-panel-font))
      (:parent-window NIL)
      (:window-title "load window")
      (:min-gadget-width 240)
      (:initial-directory "./")
      (:message-string "fetching directory...")
      (:button-panel-items '("load" "cancel"))
```

```
      (:button-panel-h-spacing 25)
      (:num-visible 6)
      (:check-filenames-p t)
      (:modal-p nil)
      (:selection-function NIL)   ; (lambda (gadget value))

      (:dir-input-field-font (opal:get-standard-font nil nil :small))
      (:dir-input-label-font (opal:get-standard-font nil :bold nil))
      (:file-input-field-font (opal:get-standard-font nil nil :small))
      (:file-input-label-font (opal:get-standard-font nil :bold nil))
      (:message-font (opal:get-standard-font :fixed :italic :small))
      (:button-panel-font opal:default-font)
      (:file-menu-font (opal:get-standard-font nil :bold nil))
      ...)
```

The `load-gadget` is loaded along with the `save-gadget` by the file "save-gadget-loader".

The `load-gadget` is very similar to the `save-gadget`. Both look alike, except for their window titles. The same caveats apply to both the save and load gadgets (see section [save-gadget], page 492).

The `load-gadget` has its own functions for displaying and hiding the gadget, which are analogous to those used by the `save-gadget`:

gg:display-load-gadget *load-gadget* **&optional** *initial-filename*   [Function]
gg:display-load-gadget-and-wait *load-gadget* **&optional**   [Function]
       initial-filename
gg:hide-load-gadget *load-gadget*   [Function]
    When a load gadget is created and `display-load-gadget` is called, the window that pops up contains the same initial contents as in the save gadget. The "Directory" box, the scrolling-menu, and the message, all work identically in both the gadgets.

    The "Filename" box resembles the save gadget in that it beeps when an invalid file name is typed in (unless the `:check-filenames-p` slot is NIL), and is reset to the empty string, "". However, an invalid file name is defined as a file name that does not exist, or a directory.

    As in the save gadget, when you rename the buttons and use the default load function, it is important to put the name corresponding to the "Load" button as the first element of the `:button-panel-items` list.

8.37 Property Sheets

The `prop-sheet` gadget takes a list of values to display, and `prop-sheet-for-obj` takes a KR object to display. The `prop-sheet-with-OK` and `prop-sheet-for-obj-with-OK` gadgets combine a property sheet with OK, Apply and Cancel buttons and functions to display these in windows (using the Garnet look and feel). Similarly, the `motif-prop-sheet-with-OK` and `motif-prop-sheet-for-obj-with-OK` combine a property sheet with buttons, but use the Motif look and feel (see section [motif-prop-sheets], page 561).

8.37.1 User Interface

Press on the value of a slot with the left button to begin typing. Press with the left button again (anywhere) or hit `return` or `^j` to stop editing (if multi-line strings are allowed, then `return` goes to the next line, so you need to use `^j` or left button to stop editing). Pressing with any other button inside the string moves the cursor. Regular editing operations are supported (see the text-interactor in the Interactors chapter). If you hit `tab`, the cursor will move to the next field. If label selection is enabled, then labels can be selected by pressing with any mouse button. If value selection is enabled, then values must be selected with the *right* button while they are not being edited. Selected labels or values are displayed in bold.

8.37.2 Prop-Sheet

```
(create-instance 'gg:Prop-Sheet opal:aggregadget
   (:maybe-constant '(:left :top :items :default-filter :v-spacing
      :multi-line-p :select-label-p :visible
      :label-selected-func :label-select-event
      :select-value-p :value-selected-func :single-select-p))
 ; Customizable slots
  (:left 0) (:top 0)
  (:items NIL) ; put the values to be displayed here
  (:default-filter 'default-filter)
  (: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-selected-func NIL)
  (:label-select-event :any-mousedown)
  (: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
```

```
COLOR: Red
HEIGHT: 34
STATUS: Nervous
DIRECTION: up down diagonal
RANGE (1..20): 1
```

Figure 8.15: Example of a property sheet with an embedded gadget.

The loader for the `gg:prop-sheet` gadget is "prop-sheet-loader".

Customizable slots:

:left, :top - Position of the gadget. Default: 0,0

:items - The control list of the items to be displayed in the gadget. The format for the list is a list of lists, as follows: ((label1 stringval1 [filter1 [realval1 [comment]]]) (label2 ...))

> The labels can be atoms or strings, and are shown at the left.

> The stringval is the initial (default) value displayed. For an example of the use of the various forms of stringval, see section [propexample], page 512. It can be:

>> a string,

>> a formula object which computes a string. Note that all references in the formula must be absolute (since otherwise they would be relative to the property sheet).

>> an instance of a gadget (e.g., a radio-button-panel), in which case that instance is used instead of an editable text field. Note that the instance itself is used, so it will be destroyed if the prop-sheet is destroyed. The gadget instance should supply its value in a slot called :value (as the standard garnet gadgets do). NOTE: If a gadget, no filter functions are called (use the :selection-function of the gadget), the realval is ignored, and the :changed-values slot is not valid. Useful gadgets are described in section [propusefulgadgets], page 511.

> If the filter is non-NIL, it is a function called after the user types the value (see below).

> The realval, if supplied, is the actual value the stringval represents (e.g. if the real values are not strings). If stringval is a list of strings, then realval should be a list of the same length.

> If supplied, the comment is displayed after the label. It can be any string, and will be displayed after the slot label. Typical uses would be to give legal values (e.g.: "(1..20)").

:default-filter - If there is no filter on an individual item, then the global default-filter function is called when the user finishes editing. See below. The default filter does nothing.

:v-spacing - Vertical space between the items. Default = 1

:pixel-margin - Multiple-valued items are represented as an aggrelist, so this determines the maximum pixel value of an item, before wrapping to the next line. Note that this does *not* affect single valued items. Default: nil

:rank-margin - Same as :pixel-margin, but is a count of the number of values. Default: nil

:multi-line-p - Whether the user can enter multi-line strings, which means that return does not exit a field, but makes a new line. Default: nil.

:select-label-p - Whether pressing on the label (with any mouse button) causes the item to be selected. Default: nil.

`:label-select-event` - If you want to make the labels selectable, you can specify which mouse event to use in the slot `:label-select-event`.

`:label-selected-func` - Called with *(gadget label-obj label)* when a label is selected.

`:select-value-p` - Whether pressing on the value (with the right button) causes the value to be selected. NOTE: Values which are specified as gadgets cannot be selected. Default: `nil`.

`:value-selected-func` - Called when a value is selected with *(gadget value-obj value label)* where label is the label of that field.

`:single-select-p` - Whether a single label or value can be selected (T) or multiple fields can be selected (NIL). This is only relevant if one or both of `:select-label-p` or `:select-value-p` is non-NIL. Default: `nil`.

Read-only (output) slots:

`:label-selected` - Will be set with a list of the selected label objects. Call `Get-Val-For-PropSheet-Value` to get label name from the label object.

`:value-selected` - Will be set with a list of the selected value objects. Call `Get-Val-For-PropSheet-value` on an obj to get the value and label from the value object.

`:value` - List of all the slots and their (filtered) values. For example: ((label1 value1) (label2 value2) ...).

`:changed-values` - List of the slots that have changed, as: ((label1 value1) (label2 value2)) This slot is not kept valid if a gadget is used as an item.

Filter functions:

The filter functions allow the program to convert the string values to the appropriate form. The displayed string and the "real" value are stored separately, so they can be different. Filter functions are defined as: `(lambda (prop-sheet-gadget label value-obj new-str old-str))`

The *index* is used for multi-valued slots, and otherwise is zero. The *value-obj* is the actual object used to display the string, and will be needed only by hackers. The filter function can return the value to use (modified *new-str*, not necessarily a string) or it can return three values: `(new-val in-valid-p new-str)` where *new-val* is a value (not necessarily a string) to use, *in-valid-p* is T if the new-str value is invalid (bad), in which case the *new-str* is still used, but it is shown in italic. If *new-str* is returned, then it is displayed instead of what the user typed (for example if the filter function expands or corrects the typed value).

An example of a custom filter function is shown in section [propexample], page 512.

8.37.3 Prop-Sheet-For-Obj

When you want to display a property sheet for a Garnet object, you can use `prop-sheet-for-obj`. The prop-sheet can directly access the `:parameters` list of a Garnet object, which is a list of the slots normally customizable for the object. You can also display and modify slots of *multiple* objects simultaneously. Gilt makes heavy use of many features in this 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 in-
tersection 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
```

```
LEFT:  150
TOP:  10
WIDTH:  50
HEIGHT:  40
LINE-STYLE:  OPAL:LINE-2
FILLING-STYLE:  OPAL:GRAY-FILL
```

Figure 8.16: Example of a property sheet for an object (the object is shown at the upper left).

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

Customizable slots:

:left, :top - Position of the gadget. Default: 0,0

:obj - The KR object or list of objects to be displayed. If this slot contains 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 which has that slot in its :parameters list.

:error-gadget - An error-gadget may be placed in this slot. Type checking is performed before setting a slot, and any errors are reported in this error gadget. 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.

:slots - The list of slots of the object to view. Default value is nil, which means the prop-sheet should get the list of slots from the :parameters slot of the object being edited (see :union?). When relying on :parameters, 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 (see also :type-gadgets). 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.

Alternatively, any element in the list can be a slot name or a sublist: (*slot* "*comment*" *display*):

> If the *comment* is non-nil, it is displayed after the label.

> If the *display* parameter is supplied, it can either be:

>> A list of legal values for the slot, e.g. '(:direction (:horizontal :vertical))

>> A function of the form (lambda (new-val)) which returns T if the value is bad. This function might pop up an error dialog box after testing but before returning. The slot keeps its illegal value, but it is shown in italics.

>> A gadget, in which case the :value slot of the gadget is set with the old value, and the :value slot is queried to get the final value. If gadgets are used, then :set-immediately-p for the property sheet should be nil. A useful gadget is Pop-Up-From-Icon.

:union? - This affects which slots are shown for objects when their :parameters lists are being used. If there are multiple objects, then a value of T for this slot will display the slots that are in *any* of the objects. If the value of this slot is nil, then only those slots that appear in *all* of the :parameters lists (the intersection of the lists) will be displayed. The default is T, to show the union of all :parameters lists.

:eval-p - If nil, then the values set into the slots will be all strings. If T, then evaluates what the user types (using Read-From-String) and sets the result into the slot. Usually, you use T when displaying the graphical fields of graphical objects. Default=T. NOTE: Evaluating a slot may cause the interface to crash if the values are not valid.

:set-immediately-p - If T, then as soon as the user types CR, the object's slot is set. If nil, some external action must set the object's slots (e.g., when using prop-sheet-for-obj-with-OK, the object's slots are not set until the OK button is hit). Default=T.

:type-gadgets - This slot is 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.

The slots :v-spacing, :pixel-margin, :rank-margin, :multi-line-p, :select-label-p, :label-select-event, :label-selected-func, :select-value-p, :value-selected-func, and :single-select-p are the same as for the prop-sheet gadget.

Read-only (output) slots (same as Prop-Sheet)

:label-selected

:value-selected

:value

:changed-values

8.37.4 Useful Functions

gg:reusepropsheet *prop-sheet-gadget new-items*   [Function]
ReUsePropSheet allows you to re-use an old prop-sheet or a prop-sheet-with-OK gadget with a new set of values, which is much more efficient than destroying and creating a new prop-sheet. NOTE: it is NOT sufficient to simply s-value the :items slot. If you plan to reuse property sheets, do not declare the :items slot constant.

gg:reusepropsheetobj *prop-sheet-for-obj* **&optional** *obj slots*   [Function]
ReUsePropSheetObj allows a prop-sheet-for-obj or prop-sheet-for-obj-with-OK gadget to be re-used. If the new *obj* and *slots* are *not* supplied, then they should be set into the object before this function is called. NOTE: it is NOT sufficient to simply s-value the :obj and :slots slot.

gg:`Get-Val-For-Propsheet-Value` (*label-or-value-obj*)   [Function]

The `Get-Val-For-PropSheet-Value` function returns the label when a label is passed in, or for a value-obj, returns multiple values: `value label`, where *label* is the label (name, not object) of that field.

If you want to change the value of a property sheet item without regenerating a new property sheet, you can use the new function `Set-Val-For-PropSheet-Value`. This takes the form:

gg:`set-val-for-propsheet-value` *label-or-value-obj new-value*   [Function]

The *label-or-value-obj* parameter is the object used by the property-sheet to represent the field.

8.37.5 Prop-Sheet-With-OK

The next set of gadgets combine property sheets with OK, Apply and Cancel buttons. There are two pairs: one for Garnet look-and-feel gadgets, and one for Motif look-and-feel gadgets (see section [motif-prop-sheets], page 561, for the Motif version).

```
(create-instance 'gg:Prop-Sheet-With-OK opal:aggregadget
    (:maybe-constant '(:left :top :items :default-filter :ok-function
        :apply-function :Cancel-Function :v-spacing
        :multi-line-p :select-label-p  :visible
        :label-selected-func :label-select-event
        :select-value-p :value-selected-func :single-select-p))
  ; Customizable slots
    (:left 0) (:top 0)
    (:items NIL)
    (:default-filter 'default-filter)
    (:OK-Function NIL)
    (:Apply-Function NIL)
    (:Cancel-Function NIL)
    (: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 entries
    (:label-select-event :any-mousedown)
    (:label-selected-func NIL)
    (:select-value-p NIL)
    (:value-selected-func NIL)
    (:single-select-p NIL)

  ; Read-only slots
    (:label-selected ...)
    (:value-selected ...)
    (:value ...)
    (:changed-values ...))
```

The `prop-sheet-with-OK` gadget is just the `prop-sheet` gadget with Garnet text buttons for OK, Apply, and Cancel.

The loader for `prop-sheet-with-OK` is "prop-sheet-win-loader".

Customizable slots

> `:OK-Function` - Function called when the OK button is hit. Defined as:
>
> `(lambda (Prop-Sheet-With-OK-gadget))` Typically, this would do something with the values gotten from `(gv Prop-Sheet-With-OK-gadget :values)` or `(gv Prop-Sheet-With-OK-gadget :changed-values)`. If you use the `Pop-Up-Win-For-Prop` functions, then the window will be removed before the `OK-function` is called, so you do not have to worry about the window.
>
> `:Apply-Function` - Function called when the Apply button is hit. Defined as:
>
> `(lambda (Prop-Sheet-With-OK-gadget)` Typically, this would do something with the values gotten from `(gv Prop-Sheet-With-OK-gadget :values)` or `(gv Prop-Sheet-With-OK-gadget :changed-values)`.
>
> `:Cancel-Function` - Function called when Cancel button is hit. Defined as: `(lambda (Prop-Sheet-With-OK-gadget))` Programmers typically would not use this. If you use the `Pop-Up-Win-For-Prop` functions, then the window will be removed before the `Cancel-function` is called, so you do not have to worry about the window.

The rest of the slots are the same as for `prop-sheet`.

8.37.6 Prop-Sheet-For-Obj-With-OK

```
(create-instance 'gg:Prop-Sheet-For-Obj-With-OK prop-sheet-with-OK
   (:maybe-constant '(:left :top :obj :slots :eval-p :ok-function
      :apply-function :Cancel-Function :v-spacing
      :multi-line-p :select-label-p :visible
      :label-selected-func :label-select-event
      :select-value-p :value-selected-func :single-select-p))
 ; Customizable slots
  (:OK-Function NIL)
  (:Apply-Function NIL)
  (:Cancel-Function NIL)
  (:left 0) (:top 0)
  (:obj NIL)   ; a single obj or a list of objects
  (:slots NIL) ; list of slots to show. If NIL, get from :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 `prop-sheet-for-obj-with-OK` gadget is just the `prop-sheet-for-obj` gadget with Garnet text buttons for OK, Apply, and Cancel.

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

Given a list of slots for a KR object, displays the values and allows them to be edited. The labels and values can optionally be selectable. Sets the object's slot only when OK or Apply is hit. (So `:set-immediately-p` is always `nil`).

Customizable slots

> `:OK-Function` - Function called when the OK button is hit. Defined as: (lambda (Prop-Sheet-For-Obj-With-OK-gadget)) Since this gadget will set the slots of the object automatically when OK is hit (before this function is called) and the window visibility is handled automatically, programmers rarely need to supply a function here.

> `:Apply-Function` - Function called when the Apply button is hit. Defined as: (lambda (Prop-Sheet-For-Obj-With-OK-gadget)) Since this gadget will set the slots of the object automatically when Apply is hit (before this function is called), programmers rarely need to supply a function here.

> `:Cancel-Function` - Function called when Cancel button is hit. Defined as: (lambda (prop-sheet-for-obj-with-ok-gadget)) Since the window visibility is handled automatically, programmers rarely need to supply a function here.

8.37.7 Useful Functions

`gg:pop-up-win-for-prop` *prop-gadget-with-ok left top title*   [Function]
        &optional *modal-p*
> Given an existing gadget of any of the "OK" types, this function pops up a window which will show the property sheet, and will go away when the user hits either "OK" or "Cancel". The window is allocated by this function to be the correct size. When the *modal-p* parameter is T, then interaction in all other Garnet windows will be suspended until the user clicks either the "OK" or "Cancel" button in this window. This function can be called many times on the **same** gadget, which is much more

efficient than allocating a new gadget and window each time. To change the items or object before redisplaying, use one of the functions below.

gg:pop-up-win-change-items *prop-gadget-with-ok new-items left*   [Function]
       top title **&optional** *modal-p*

Given an existing gadget, `Pop-Up-Win-Change-Items` sets the items field of the gadget to the specified value, and then pops up a window displaying that property sheet. (This function calls `ReUsePropSheetObj` automatically). (Note: if you want to pop up a `Prop-Sheet-With-OK` or `Motif-Prop-Sheet-With-OK` gadget without changing the *items* field, you can simply pass it to `Pop-Up-Win-For-Prop`.

gg:pop-up-win-change-obj *prop-obj-gadget-with-ok obj slots left*   [Function]
       top title **&optional** *modal-p*

Given an existing gadget, `Pop-Up-Win-Change-Obj` sets the *obj* and *slot* fields of the gadget to the specified values, and then pops up a window displaying that property sheet. (This function calls `ReUsePropSheetObj` automatically). (Note: if you want to pop up a `Prop-Sheet-For-Obj-With-OK` or `Motif-Prop-Sheet-For-Obj-With-OK` gadget without changing the *obj* and *slot* fields, you can simply pass it to `Pop-Up-Win-For-Prop`.

8.37.8 Useful Gadgets

This section describes two gadgets that are useful in property sheet fields as the values. Both of these gadgets are shown in Figure [motifpropfix], page 564.

8.37.9 Horiz-Choice-List

The `horiz-choice-list` displays the choices and allows the user to pick one with the left mouse button. The choices can be strings or atoms.

```
(create-instance 'gg:Horiz-Choice-List opal:aggregadget
   (:maybe-constant '(:left :top :items))
   ; Customizable slots
   (:left 0) ; left and top are set automatically when used in a prop-sheet
   (:top 0)
   (:items '("one" "two" "three")) ; the items to choose from
   ; Input and output slot
   (:value NIL) ; what the user selected
   )
```

The loader for `Horiz-Choice-List` is "prop-values-loader", although it is automatically loaded when you load a property sheet.

The `Horiz-Choice-List` is automatically used when you list a set of legal values for the display parameter for a `prop-sheet-for-obj`.

8.37.10 Pop-Up-From-Icon

The `Pop-Up-From-Icon` displays a small icon, and if the user hits on it, then a function is called which can pop-up a dialog box or menu to make the choice.

```
(create-instance 'gg:Pop-Up-From-Icon opal:aggregadget
  (:maybe-constant '(:left :top :icon-image :pop-up-function)))
```

```
; Customizable slots
(:left 0) ; left and top are set automatically when used in a prop-sheet█
(:top 0)
(:icon-image pop-up-icon) ; you can replace with your own picture
(:pop-up-function NIL))   ;put a function here to pop-up the menu or whatever█
```

The loader for `Pop-Up-From-Icon` is "prop-values-loader", although it is automatically loaded when you load a property sheet.

The `pop-up-function` is called when the user presses with the left button and then releases over the icon. It is called as follows: `(lambda(pop-up-from-icon-gadget))` It should stuff its results into the `:value` field of that gadget. See the chapter on Gilt for some functions that are useful for popping up dialog boxes and menus.

8.37.11 Property Sheet Examples

First, an example filter function, which checks if value is a number, and if it is between 1 and 20.

```
(defun string-to-num-filter (prop-gadget label index value-obj new-str old-str)█
  (declare (ignore prop-gadget label index value-obj))
  (let* ((sym (read-from-string new-str))
 (number (when (integerp sym) sym)))
    (if (and number (>= number 1) (<= number 20))
; then OK, return the converted number
(values number NIL new-str)
; else bad, return original string and T to say invalid
(progn
  (inter:beep) ; first, beep
  (values new-str T new-str)))))
```

Now, we will use that filter function in a property sheet. This code creates the property sheet shown in Figure [plainproppix], page 501, in section [propsheetsec], page 500. It contains three regular lines, a slot using a gadget, and then a slot with a filter function and a comment.

```
(create-instance 'PROP1 garnet-gadgets:prop-sheet
   (:items '((:color "Red")
             (:height "34")
     (:status "Nervous")
     (:direction ,(create-instance NIL garnet-gadgets:horiz-choice-list
     (:items '("up" "down" "diagonal"))))
     (:range "1" ,#'string-to-num-filter 1 "(1..20)"))))
```

Finally, a Motif look and feel property sheet for an object with OK, Apply and Cancel buttons in it. The `my-rectangle1` object is only changed when OK or Apply is hit. The resulting window is shown in Figure [motifpropfix], page 564.

```
(create-instance 'MY-OBJ-PROP gg:motif-prop-sheet-for-obj-with-OK
  (:left 0)
  (:top 0)
  (:obj MY-RECTANGLE1)
  (:slots '(:left ; first four slots are normal
```

```
      :top
      :width
      :height
      (:quality (:good :medium :bad)) ;list of options
      ; next two slots use pop-up icon gadgets
      (:line-style ,(create-instance NIL gg:pop-up-from-icon
      (:pop-up-function #'Line-Style-Pop-Up)))
      (:filling-style ,(create-instance NIL gg:pop-up-from-icon
         (:pop-up-function #'Fill-style-pop-up))))))))
```

8.38 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.

8.38.1 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 contain-
   ing 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, Allegro, and LispWorks.**

8.38.2 MouseLinePopup gadget

```
(create-instance 'gg:MouseLinePopup opal:aggregadget
  (:start-event :SHIFT-CONTROL-META-LEFTDOWN)
  (:windows (o-formula (gvl :window))) ; default is the window contain-
ing 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.

8.39 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 changing a `standard-edit` routine will allow it to be useful to your application, let us know.)

The `standard-edit` routines can be loaded using (`garnet-load "gg:standard-edit-loader"`).

8.39.1 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 **selected** 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-line`s.
>
> `:polygon-p` - if non-NIL, then the object is controlled by a `:point-list` list of multiple values. True by default for `opal:polyline`s.
>
> `: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:aggregadget`s. 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`.

8.39.2 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.

8.39.3 Standard Editing Routines

> gg:Standard-Initialize-Gadget *gadget   selection-gadget   agg-of-items* [function], ▮
> page 90
>
>   &key *clipboard   undo-delete?*

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], page 90

Useful for all those functions that are **N**ot **I**mplemented **Y**et. It prints "Sorry, Not Implemented Yet" in the Lisp listener window and beeps.

gg:Standard-Delete *gadget* &rest *args* [function], page 90

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

gg:Standard-Delete-All *gadget* &rest *args* [function], page 90

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

gg:Standard-Undo-Last-Delete *gadget* &rest *args* [function], page 90

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], page 90

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

gg:Standard-To-Bottom *gadget* &rest *args* [function], page 90

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

gg:Standard-Refresh *gadget* &rest *args* [function], page 90

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

gg:Standard-Select-All *gadget* &rest *args* [function], page 90

Causes all of the objects to be selected.

gg:Standard-Cut *gadget* &rest *args* [function], page 90

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], page 90

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], page 90

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], page 90

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], page 90`

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], page 90`

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], page 90`

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:aggregadget`s. 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.

8.39.4 Utility Procedures

> `gg:Sort-Objs-Display-Order` *objs* *draw-agg* `[function], page 90`

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], page 90`

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], page 90`

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

8.40 The Motif Gadget Objects

The Motif gadgets in the Gadget Set were designed to simulate the appearance and behavior of the OSF/Motif widgets. They are analogous to the standard gadgets of Chapter [Standard-Gadgets], page 416, and many of the customizable slots are the same for both sets of gadgets.

As in the previous chapter, the descriptions of the Motif gadgets begin with a list of customizable slots and their default values (any of which may be ignored). The `motif-gadget-prototype` object which occurs in the definition of each Motif gadget is just an instance of an `opal:aggregadget` with several color, filling-style, and line-style slot definitions used by all Motif gadgets.

The Motif gadgets have been implemented to appear on either color or black-and-white screens without changes to the instances. The `:foreground-color` slot is used to compute filling-styles internally on a color screen, and it is ignored on a black-and-white screen. Figure [color-and-bw-motif], page 519, shows how a few of the Motif gadgets look on each type of screen.
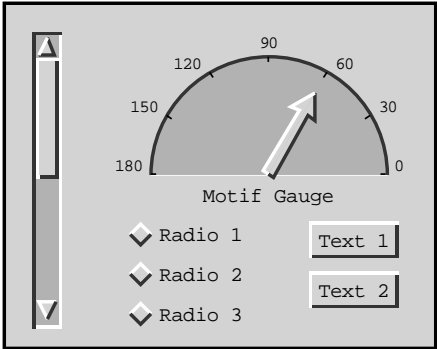
8.41 Useful Motif Objects

In order to facilitate the construction of interfaces containing Motif gadgets, Garnet exports some miscellaneous objects that are commonly found in Motif. The objects described in this section are defined in the "motif-parts" file (automatically loaded with all Garnet Motif-style "-loader" files).

8.41.1 Motif Colors and Filling Styles

In each Motif gadget, there is a slot for the color of the gadget. The `:foreground-color` is the color that should be shown in the foreground of the gadget (i.e., the part of the gadget that does not appear recessed). The background, shadow, and highlight colors for the gadget are computed internally based on the `:foreground-color` given.

The default `:foreground-color` for the gadgets is `opal:motif-gray`, but the user may provide any instance of `opal:color` in the slot. Additionally, Opal provides the following colors for use with the Motif gadgets. The associated filling styles may be of use in other objects designed by the programmer.

> opal:motif-gray
>
> opal:motif-blue
>
> opal:motif-green
>
> opal:motif-orange
>
> opal:motif-light-gray
>
> opal:motif-light-blue
>
> opal:motif-light-green
>
> opal:motif-light-orange
>
> opal:motif-gray-fill
>
> opal:motif-blue-fill
>
> opal:motif-green-fill
>
> opal:motif-orange-fill
>
> opal:motif-light-gray-fill
>
> opal:motif-light-blue-fill
>
> opal:motif-light-green-fill
>
> opal:motif-light-orange-fill

When the Motif gadgets are used on a black-and-white monitor, the gadgets ignore the `:foreground-color` slot and internally compute reasonable filling-styles that are black, white, or Opal halftones.

8.41.2 Motif-Background

```
(create-instance 'gg:Motif-Background opal:rectangle
   (:foreground-color opal:motif-gray))
```

In order to simulate the Motif three-dimensional effect in an interface, there should be a gray background in a window containing Motif-style gadgets. Garnet provides two ways

to achieve this effect. You could add an instance of the `motif-background` object to the window, which is a rectangle whose dimensions conform to the size of the window in which it appears.

Alternately, you could supply the `:background-color` of your window with an appropriate Opal `color` object (like `opal:motif-gray`). This is generally more efficient, since it is faster to redraw a window with its background color than to redraw a rectangle that occupies the entire window.

NOTE: If you choose to use the `motif-background` object, it is essential that the instance be added to the top-level aggregate before any other Garnet object. This will ensure that the background is drawn **behind** all other objects in the window.

Of course, the `:foreground-color` of the `motif-background` instance or the `:background-color` of the window should be the same as the colors of all the Motif gadgets in the window.

8.41.3 Motif-Tab-Inter

```
(create-instance 'gg:Motif-Tab-Inter inter:button-interactor
   (:window NIL)
   (:objects NIL)
   (:rank 0)
   (:continuous NIL)
   (:start-where T)
   (:start-event '(#\tab :control-tab))
   (:waiting-priority gg:motif-tab-priority-level)
   (:running-priority gg:motif-tab-priority-level)
   (:stop-action #'(lambda (interactor obj-over) ...))
   (:final-function NIL))
```

Each Motif gadget has the ability to be operated by the keyboard as well as the mouse. In traditional Motif interfaces, the keyboard selection box is moved within each gadget with the arrow keys, and it is moved among gadgets with the tab key (i.e., one gadget's keyboard selection is activated while the previous gadget's keyboard selection is deactivated). The keyboard interface can be chapterly activated by setting a Motif gadget's `:keyboard-selection-p` to T, but the bookkeeping becomes formidable when there are a large number of Motif gadgets on the screen and their keyboard status is changing. Thus, Garnet provides the `motif-tab-inter` which handles the bookkeeping among multiple Motif gadgets.

To use the `motif-tab-inter`, create an instance with a list of the Motif gadgets on which to operate in the `:object` slot and the window of the objects in the `:window` slot. Usually, these are the only two slots that will need to be set.

Repeatedly hitting the tab key (or simultaneously hitting `control` and `tab` will cause the keyboard selection to cycle through the list of objects. Specifically, hitting the tab key causes the `:rank` of the `motif-tab-inter` to be incremented, and the interactor checks the `:active-p` slot of the next object in the `:object` list. If the result is T, then that object's `:keyboard-selection-p` slot is set to T. Otherwise, the `:rank` is incremented again and the next object is checked.

The :active-p slots of the "continuous" Motif gadgets – the scroll bars, slider, and gauge – all default to T, while the :active-p slots of the Motif buttons and menu depend on the items in the :inactive-items list.

The :running-priority and :waiting-priority of the motif-tab-inter are both set to be motif-tab-priority-level, which is a higher priority than the default interactor priority levels (but lower than the error-gadget's error-priority-level). This allows the motif-tab-inter to be used at the same time as the inter:text-interactor (as in the motif-scrolling-labeled-box).

The function in the :final-function slot is executed whenever the current selection changes. It takes the parameters (lambda (inter new-object))

Examples of the motif-tab-inter in use may be found in demo-motif and in all three Motif button demos.

8.42 Motif Scroll Bars

```
(create-instance 'gg:Motif-V-Scroll-Bar gg:motif-gadget-prototype
   (:maybe-constant '(:left :top :width :height :val-1 :val-2 :scr-incr
      :page-incr :scr-trill-p :percent-visible :scroll-p
      :foreground-color :visible))
   (:left 0)
   (:top 0)
   (:width 20)
   (:height 200)
   (:val-1 0)
   (:val-2 100)
   (:scr-incr 1)
   (:page-incr 5)
   (:scr-trill-p T)
   (:percent-visible .5)
   (:scroll-p T)
   (:keyboard-selection-p NIL)
   (:foreground-color opal:motif-gray)
   (:value (o-formula ...))
   (:active-p T)
   (:selection-function NIL)   ; (lambda (gadget value))
   )

(create-instance 'gg:Motif-H-Scroll-Bar gg:motif-gadget-prototype
   (:maybe-constant '(:left :top :width :height :val-1 :val-2 :scr-incr
      :page-incr :scr-trill-p :percent-visible :scroll-p
      :foreground-color :visible))
   (:left 0)
   (:top 0)
   (:width 200)
   (:height 20)
   (:val-1 0)
   (:val-2 100)
```

```
(:scr-incr 1)
(:page-incr 5)
(:scr-trill-p T)
(:percent-visible .5)
(:scroll-p T)
(:keyboard-selection-p NIL)
(:foreground-color opal:motif-gray)
(:value (o-formula ...))
(:active-p T)
(:selection-function NIL)   ; (lambda (gadget value))
)
```
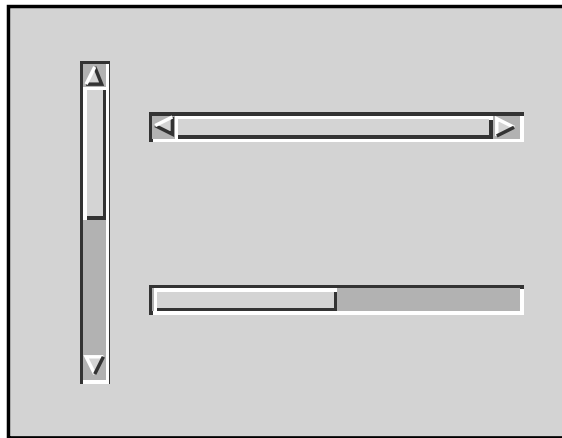
The loader file for the `motif-v-scroll-bar` is "motif-v-scroll-loader". The loader file for the `motif-h-scroll-bar` is "motif-h-scroll-loader".

The Motif scroll bars allow the specification of the minimum and maximum values of a range, while the `:value` slot is a report of the currently chosen value in the range. The interval is determined by the values in `:val-1` and `:val-2`, and either slot may be the minimum or maximum of the range. The value in `:val-1` will correspond to the top of the vertical scroll bar and to the left of the horizontal scroll bar. The `:value` slot may be accessed directly by some function in the larger interface, and other formulas in the interface

may depend on it. If the `:value` slot is set directly, then the appearance of the scroll bar will be updated accordingly.

The trill boxes at each end of the scroll bar allow the user to increment and decrement `:value` by the amount specified in `:scr-incr`. The designer may choose to leave the trill boxes out by setting `:scr-trill-p` to `nil`.

The indicator may also be moved directly by mouse movements. Dragging the indicator while the left mouse button is pressed will change the `:value` accordingly. A click of the left mouse button in the background trough of the scroll bar will cause the `:value` to increase or decrease by `:page-incr`, depending on the location of the indicator relative to the mouse click.

When `:keyboard-selection-p` is T, then a black-selection box is drawn around the scroll bar and the indicator can be moved with the arrow keys (uparrow and downarrow for the `motif-v-scroll-bar`, leftarrow and rightarrow for the `motif-h-scroll-bar`).

The `:percent-visible` slot contains a value between 0 and 1, and is used to specify the length of the indicator relative to the length of the trough. If `:percent-visible` is .5, then the length of the indicator will be half the distance between the two trill boxes. This feature might be useful in a scrolling menu where the length of the indicator should correspond to one "page" of items in the menu (e.g., for three pages of items, set `:percent-visible` to .33).

The slots `:scroll-p` and `:active-p` are used to enable and disable the scrolling feature of the scroll bar. When either is set to `nil`, the trill boxes of the scroll bar become inactive and the indicator cannot be moved. The difference is that when `:active-p` is set to `nil`, then the keyboard selection cannot be enabled.

8.43 Motif Slider

```
(create-instance 'gg:Motif-Slider gg:motif-v-scroll-bar
   (:maybe-constant '(:left :top :height :trough-width :val-1 :val-2
      :scr-incr :page-incr :scr-trill-p :text-offset
      :scroll-p :indicator-text-p :indicator-font
      :foreground-color :visible))
   (:left 0)
   (:top 0)
   (:height 200)
   (:trough-width 16)
   (:val-1 0)
   (:val-2 100)
   (:scr-incr 1)
   (:page-incr 5)
   (:scr-trill-p NIL)
   (:text-offset 5)
   (:scroll-p T)
   (:indicator-text-p T)
   (:keyboard-selection-p NIL)
   (:indicator-font opal:default-font)
   (:foreground-color opal:motif-gray)
```

```
(:value (o-formula ...))
(:active-p T)
(:selection-function NIL)   ; (lambda (gadget value))
(:parts (...)))
```
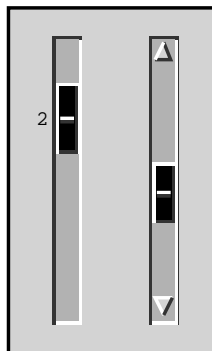

"motif-slider-loader".

The `motif-slider` is similar to the `motif-v-scroll-bar`, except that it has a fixed-size indicator with accompanying text feedback. The mouse can be used to drag the indicator, and the arrow keys can be used when keyboard-selection is activated.

The slots `:value`, `:val-1`, `:val-2`, `:scr-incr`, `:page-incr`, `:scr-trill-p`, `:scroll-p`, `:active-p` and `:keyboard-selection-p` all have the same functionality as in the `motif-v-scroll-bar`.

The `:trough-width` slot determines the width of the scroll-bar part of the slider. The actual `:width` of the gadget is not user-settable because of the changing value feedback width.

The current `:value` of the slider is displayed beside the trough if `:indicator-text-p` is T. The font of the indicator text is in `:indicator-font`. The distance from the indicator text to the trough is in `:text-offset`.

8.44 Motif-Trill-Device
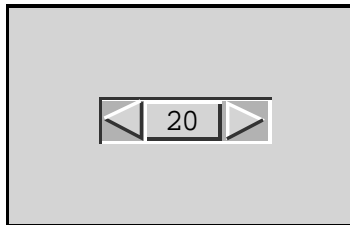

```
(create-instance 'gg:Motif-Trill-Device gg::motif-gadget-prototype
  (:left 0) (:top 0)
  (:width 150) (:height 40)
  (:val-1 0) (:val-2 100)
  (:value 20)
```

```
(:foreground-color opal:motif-gray)
(:format-string "~a")
(:value-feedback-font opal:default-font)
(:value-feedback-p T)
(:scroll-incr 1)
(:selection-function NIL)    ; (lambda (gadget value))
)
```

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 section
[trill-device], page 423, for more information.

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

8.45 Motif Gauge

```
(create-instance 'gg:Motif-Gauge gg:motif-gadget-prototype
   (:maybe-constant '(:left :top :width :title :foreground-color :title-font█
      :value-font :enum-font :num-marks :tic-marks-p
      :enumerate-p :value-feedback-p :text-offset :val-1 :val-2
      :scr-incr :format-string :enum-format-string :visible))
   (:left 0)
   (:top 0)
   (:width 230)
   (:title "Motif Gauge")
   (:foreground-color opal:motif-gray)
   (:title-font opal:default-font)
   (:value-font opal:default-font)
   (:enum-font (create-instance NIL opal:font (:size :small)))
   (:num-marks 10)                ; Includes endpoints
   (:tic-marks-p T)
   (:enumerate-p T)
   (:value-feedback-p T)
   (:text-offset 5)
```

```
(:val-1 0)
(:val-2 180)
(:scr-incr 5)
(:format-string "~a")        ; How to print the feedback value
(:enum-format-string "~a")   ; How to print the tic-mark values
(:keyboard-selection-p NIL)
(:value (o-formula ...))
(:selection-function NIL)    ; (lambda (gadget value))
)
```
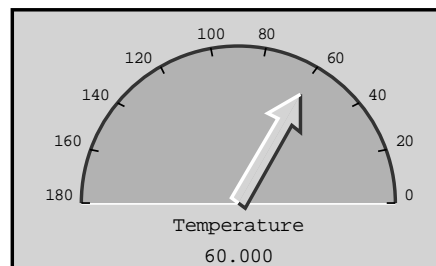
The `motif-gauge` is a semi-circular meter with tic-marks around the perimeter. As with scroll bars and sliders, this object allows the user to specify a value between minimum and maximum values. An arrow-shaped polygon points to the currently chosen value, and may be rotated either by dragging it with the mouse or by the arrow keys when keyboard selection is activated. Text below the gauge reports the current value to which the needle is pointing.

The slots `:num-marks`, `:tic-marks-p`, `:enumerate-p`, `:val-1`, `:val-2`, and `:enum-font` are implemented as in the standard Garnet sliders (see section [sliders], page 419). The value in `:val-1` corresponds to the right side of the gauge.

The title of the gauge is specified in `:title`. No title will appear if `:title` is `nil`. The fonts for the title of the gauge and the current chosen value are specified in `:title-font` and `:value-font`, respectively.

If `:value-feedback-p` is T, then numerical text will appear below the gauge indicating the currently chosen value. The value in `:text-offset` determines the distance between the gauge and the title string, and between the title string and the value feedback.

The `:format-string` and `:enum-format-string` slots allow you to control the formatting of the text strings, in case the standard formatting is not appropriate. This is mainly useful for floating point numbers. The slots should each contain a string that can be passed to the lisp function `format`. The default string is `"~a"`.

Setting `:keyboard-selection-p` to T activates the keyboard interface to the `motif-gauge`. The left and right arrow keys can then be used to change the value of the gauge. The increment by which the value of the gauge changes during each press of an arrow key is in `:scr-incr`.

8.46 Motif Buttons
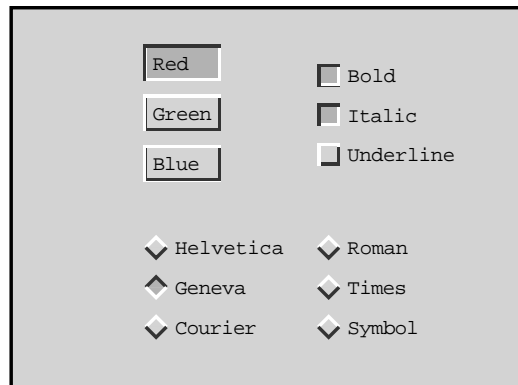
Figure 8.18: Motif Text Buttons, Check Buttons, and Radio Buttons

As with the standard Garnet buttons, the Motif buttons can be either a single, stand-alone button or a panel of buttons. Use of the Motif gadgets is identical to the use of standard Garnet buttons in the following respects (see Section [buttons], page 429).

All slots that can be customized in an aggrelist can be customized in the Motif button panels.

The `:value` slot contains the string or atom of the currently selected item (in the `motif-check-button-panel` this value is a list of selected items). In button panels, the currently selected component of the panel's aggrelist is named in the `:value-obj` slot.

The `:width` and `:height` of button panels are determined internally, and may not be set directly. Instead, refer to the slots `:fixed-width-size` and `:fixed-height-size`. The `:width` and `:height` slots may be accessed after the object is instantiated.

The `:items` slot can be either a list of strings, a list of atoms, or a list of string/function or atom/function pairs (see section [items-slot], page 410).

The font in which the button labels appear may be specified in the `:font` slot.

Most of the buttons and button panels have a `:toggle-p` slot that controls whether buttons can become deselected. If the value of this slot is T, then clicking on a selected button deselects it. Otherwise, the button always stays selected, though the `:selection-function` and the item functions will continue to be executed each time the button is pressed.

The following slots provide additional functionality for the Motif buttons:

In single Motif buttons, if the `:active-p` slot is `nil`, then the string of the button appears in "grayed-out" text and the button is not user selectable.

Analogously, the `:inactive-items` slot of the Motif button panels contains a list of strings or atoms corresponding to the members of the `:items` list. The text of each item listed in `:inactive-items` will appear "grayed-out" and those buttons will not be user selectable. If `:active-p` is set to `nil`, then all items will appear "grayed-out".

When the slot `:keyboard-selection-p` is T, the keyboard interface to the button gadgets is activated. The arrow keys will move the selection box among the buttons in a button panel, and the space-bar will select the boxed button. The component of the button panel aggrelist currently surrounded by the selection box is named in `:keyboard-selection-obj`, and its string is in `:keyboard-selection`. Thus, the slot `:keyboard-selection` may be set with a string (or an atom, depending on the `:items` list) to put the selection box around a button. Since this slot contains a formula, the programmer may not supply an initial value at create-instance time. Instead, as with the `:value` slot, the user must first gv the `:keyboard-selection` slot and then s-value it to the desired initial value.

NOTE: When keyboard selection is activated, the space-bar is used to select buttons, while the return key is used to select items in the `motif-menu`.

8.46.1 Motif Text Buttons

```
(create-instance 'gg:Motif-Text-Button gg:motif-gadget-prototype
   (:maybe-constant '(:left :top :text-offset :active-p :string :toggle-p :font
                      :final-feedback-p :foreground-color :visible))
```

```
            (:left 0)
            (:top 0)
            (:text-offset 5)
            (:active-p T)
            (:string "Motif Text Button")
            (:font opal:default-font)
            (:final-feedback-p NIL)
            (:toggle-p T)
            (:keyboard-selection-p NIL)
            (:foreground-color opal:motif-gray)
            (:value (o-formula (if (gvl :selected) (gvl :string))))
            (:selected (o-formula (gvl :value)))  ;Set by interactor
            (:selection-function NIL)   ; (lambda (gadget value))
            )

   (create-instance 'gg:Motif-Text-Button-Panel motif-gadget-prototype
       (:maybe-constant '(:left :top :text-offset :final-feedback-p :toggle-p :items :font
                          :foreground-color :direction :v-spacing :h-spacing :v-align
                          :h-align :indent :fixed-width-p :fixed-width-size :fixed-height-
                          :fixed-height-size :rank-margin :pixel-margin :visible))
       (:left 0)
       (:top 0)
       (:text-offset 5)
       (:final-feedback-p NIL)
       (:items '("Text 1" "Text 2" "Text 3" "Text 4"))
       (:inactive-items NIL)
       (:toggle-p NIL)
       (:keyboard-selection-p NIL)
       (:keyboard-selection (o-formula ...))
       (:keyboard-selection-obj (o-formula ...))
       (:font opal:default-font)
       (:foreground-color opal:motif-gray)
       (:value-obj NIL)
       (:value (o-formula ...))
       (:active-p (o-formula ...))
       (:selection-function NIL)   ; (lambda (gadget value))
       <All customizable slots of an aggrelist>)
```

The loader file for the `motif-text-button` and `motif-text-button-panel` is "motif-text-buttons-loader".

The `motif-text-button-panel` is a set of rectangular buttons, with the string or atom associated with each button aligned inside. The button will stay depressed after the mouse is released only if `:final-feedback-p` is T.

The distance from the beginning of the longest label to the inside edge of the button frame is specified in `:text-offset`.

8.46.2 Motif Check Buttons

```
(create-instance 'gg:Motif-Check-Button gg:motif-gadget-prototype
   (:maybe-constant '(:left :top :button-width :text-offset :text-on-left-p█
      :active-p :toggle-p :string :font :foreground-color :visible))
   (:left 0)
   (:top 0)
   (:button-width 12)
   (:text-offset 5)
   (:text-on-left-p NIL)
   (:active-p T)
   (:string "Motif Check Button")
   (:font opal:default-font)
   (:toggle-p T)
   (:keyboard-selection-p NIL)
   (:foreground-color opal:motif-gray)
   (:value (o-formula (if (gvl :selected) (gvl :string))))
   (:selected (o-formula (gvl :value))) ;Set by interactor
   (:selection-function NIL)   ; (lambda (gadget value))
   )

(create-instance 'gg:Motif-Check-Button-Panel motif-gadget-prototype
   (:maybe-constant '(:left :top :button-width :text-offset :text-on-left-p :items█
                      :font :foreground-color :direction :v-spacing :h-spacing█
                      :v-align :h-align :indent :fixed-width-p :fixed-width-size█
                      :fixed-height-p :fixed-height-size :rank-margin :pixel-margin█
                      :visible))
   (:left 0)
   (:top 0)
   (:button-width 12)
   (:text-offset 5)
   (:text-on-left-p NIL)
   (:items '("Check 1" "Check 2" "Check 3"))
   (:inactive-items NIL)
   (:keyboard-selection-p NIL)
   (:keyboard-selection (o-formula ...))
   (:keyboard-selection-obj (o-formula ...))
   (:font opal:default-font)
   (:foreground-color opal:motif-gray)
   (:value-obj NIL)
   (:value (o-formula ...))
   (:active-p (o-formula ..))
   (:selection-function NIL)   ; (lambda (gadget value))
   <All customizable slots of an aggrelist>)
```

The loader file for the `motif-check-button` and the `motif-check-button-panel` is "motif-check-buttons-loader".

The `motif-check-button-panel` is analogous to the `x-button-panel` from the standard Garnet Gadget Set. Any number of buttons may be selected at one time, and clicking on a selected button de-selects it.

Since the `motif-check-button-panel` allows selection of several items at once, the `:value` slot is a list of strings (or atoms), rather than a single string. Similarly, `:value-obj` contains a list of button objects.

The slot `:text-on-left-p` specifies whether the text will appear on the right or left of the buttons. A `nil` value indicates that the text should appear on the right. When text appears on the right, the designer will probably want to set `:h-align` to `:left` in order to left-justify the text against the buttons.

The distance from the labels to the buttons is specified in `:text-offset`.

The slot `:button-width` specifies the height and width of each button square.

8.46.3 Motif Radio Buttons

```
(create-instance 'gg:Motif-Radio-Button gg:motif-gadget-prototype
   (:maybe-constant '(:left :top :button-width :text-offset :text-on-left-p
      :toggle-p :active-p :string :font :foreground-color :visible))
   (:left 0)
   (:top 0)
   (:button-width 12)
   (:text-offset 5)
   (:text-on-left-p NIL)
   (:active-p T)
   (:string "Motif Radio Button")
   (:font opal:default-font)
   (:toggle-p T)
   (:keyboard-selection-p NIL)
   (:foreground-color opal:motif-gray)
   (:value (o-formula (if (gvl :selected) (gvl :string))))
   (:selected (o-formula (gvl :value)))  ; Set by interactor
   (:selection-function NIL)              ; (lambda (gadget value))
   )

(create-instance 'gg:Motif-Radio-Button-Panel motif-gadget-prototype
   (:maybe-constant '(:left :top :button-width :text-offset :text-on-left-p :toggle-p
                      :items :font :foreground-color :direction :v-spacing :h-spacing
                      :v-align :h-align :indent :fixed-width-p :fixed-width-size
      :fixed-height-p :fixed-height-size :rank-margin :pixel-margin
                      :visible))
   (:left 0)
   (:top 0)
   (:button-width 12)
   (:text-offset 5)
   (:text-on-left-p NIL)
   (:items '("Radio 1" "Radio 2" "Radio 3"))
   (:inactive-items NIL)
```

```
(:toggle-p NIL)
(:keyboard-selection-p NIL)
(:keyboard-selection (o-formula ...))
(:keyboard-selection-obj (o-formula ...))
(:font opal:default-font)
(:foreground-color opal:motif-gray)
(:value-obj NIL)
(:value (o-formula ...))
(:active-p (o-formula ...))
(:selection-function NIL)   ; (lambda (gadget value))
<All customizable slots of an aggrelist>)
```

The loader file for the `motif-radio-button` and `motif-radio-button-panel` is "motif-radio-buttons-loader".

The `motif-radio-button-panel` is a set of diamond buttons with items appearing to either the left or the right of the buttons (implementation of `:button-width`, `:text-on-left-p` and `:text-offset` is identical to the motif check buttons). Only one button may be selected at a time.

8.47 [Motif Option Button]

```
(create-instance 'gg:Motif-Option-Button opal:aggregadget
  (:maybe-constant '(:left :top :text-offset :label :button-offset :items :initial-item
                     :button-font :label-font :button-fixed-width-p :v-spacing█
                     :keep-menu-in-screen-p :menu-h-align :foreground-color))█
  (:left 40) (:top 40)
  (:text-offset 6)
  (:label "Option button:")
  (:button-offset 2)
  (:items '("Item 1" "Item 2" "Item 3" "Item 4"))
  (:initial-item (o-formula (first (gvl :items))))
  (:button-font opal:default-font)
  (:label-font (opal:get-standard-font NIL :bold NIL))
  (:foreground-color opal:motif-gray)
  (:value (o-formula (gvl :option-text-button :string)))
  (:button-fixed-width-p T)
  (:v-spacing 8)
  (:keep-menu-in-screen-p T)
  (:menu-h-align :left)
  (:selection-function NIL)   ; (lambda (gadget value))
  ...)
```

Color: Blue

This is a Motif version of the `option-button` gadget. When the left mouse button is clicked on the option button, a menu will pop up, from which items can be selected by moving the mouse over the desired item and releasing the button. The selected item will appear as the new label of the button. Figure [motif-option-button-tag], page 539, shows a Motif option button in its normal state and after the button has been pressed.

This button works exactly like the standard `option-button` described in section [option-button], page 435. The customizations are also alike, except that the `motif-option-button` does not have a `:button-shadow-offset` slot and adds a `:background-color` slot. The loader file for the motif option button is named "motif-option-button-loader".

8.48 Motif Menu

```
(create-instance 'gg:Motif-Menu gg:motif-gadget-prototype
   (:maybe-constant '(:left :top :min-frame-width :text-offset :v-spacing :h-align
                      :items :accelerators :bar-above-these-items :item-font
                      :accel-font :item-to-string-function :final-feedback-p
                      :foreground-color :visible))
   (:left 0)
   (:top 0)
   (:min-frame-width 0)
   (:text-offset 6)
   (:v-spacing 8)
   (:h-align :left)
   (:items '("Menu 1" "Menu 2" "Menu 3" "Menu 4" "Menu 5"))
   (:inactive-items NIL)
   (:accelerators NIL)
   (:bar-above-these-items NIL)
   (:item-to-string-function
    #'(lambda (item)
(if item
    (if (stringp item)
item
(string-capitalize (string-trim ":" item)))
    "")))
   (:final-feedback-p T)
   (:keyboard-selection-p NIL)
   (:keyboard-selection (o-formula ...))
   (:keyboard-selection-obj (o-formula ...))
   (:item-font opal:default-font)
   (:accel-font opal:default-font)
   (:foreground-color opal:motif-gray)
   (:value-obj NIL)
   (:value (o-formula ...))
   (:selection-function NIL)   ; (lambda (gadget value))
   )
```
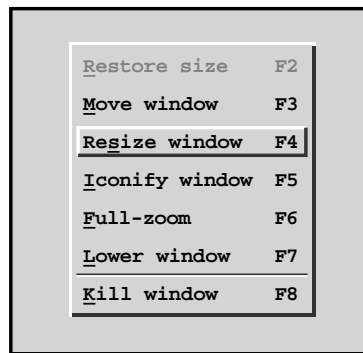
The loader file for the `motif-menu` is "motif-menu-loader".

8.48.1 Programming Interface

The `motif-menu` is analogous to the `menu` from the standard Gadget Set, with the addition of an `:accelerators` slot which facilitates the selection of a menu item by the user. Only one item may be selected at a time.

The `:accelerators` slot is a list of triples which correspond to the items in the `:items` list. Consider the following slot definitions in an instance of the `motif-menu`:

```
(:items '("Remove-window" "Move-window" ...))
(:accelerators '((#\R "Alt+F2" :META-F2) (#\M "Alt+F3" :META-F3) ...))
```

Since the #\M character appears in the second accelerator pair, the "M" in the "Move-window" item will be underlined in the menu. The string "Alt+F3" will appear to the right of the "Move-window" item in the menu. Interactors are defined in the `motif-menu` that allow the user to press the "M" key whenever keyboard selection is activated to select "Move-window". And, after properly initializing an instance of the `motif-menu-accelerator-inter` (described below), simultaneously pressing the "Alt" and "F3" keys will also select "Move window".

Since this menu supports only single selection, the `:value` slot contains the currently selected item (from the `:items` list) and the `:value-obj` slot contains the currently selected object in the menu's aggrelist.

The `:items` and `:item-to-string-function` slots are implemented as in the `:scrolling-menu` from the standard Gadget Set (see Section [scrolling-menu], page 444). Each item (the actual item, not its string conversion) specified in the `:inactive-items` list will appear "grayed-out" and will not be selectable.

A separator bar will appear above each item listed in the slot `:bar-above-these-items`.

The minimum width of the menu frame is determined by `:min-frame-width`. The menu will appear wider than this value only if the longest item string (and its corresponding accelerator, if any) will not fit in a menu of this width.

The `:v-spacing` slot determines the distance between each item in the menu, and `:text-offset` determines the distance from the menu frame to the items (and the distance between the longest item and its corresponding accelerator, if any).

The justification of the items in the menu is determined by the slot `:h-align` and may be either `:left`, `:center`, or `:right`.

A feedback box will appear around the currently selected item if `:final-feedback-p` is T.

When the slot `:keyboard-selection-p` is T, the keyboard interface to the `motif-menu` is activated. The arrow keys will move the selection box among the items in the menu, and the return key will select the boxed item. The component of the menu's aggrelist currently surrounded by the selection box is named in `:keyboard-selection-obj`, and its string is in `:keyboard-selection`. Thus, the slot `:keyboard-selection` may be set with a string (or an atom, depending on the `:items` list) to put the selection box around an item. Since this slot contains a formula, the programmer may not supply an initial value at create-instance time. Instead, as with the `:value` slot, the user must first gv the `:keyboard-selection` slot and then s-value it to the desired initial value. **NOTE:** The return key is used to select items in the `motif-menu`, while the space-bar is used to select Motif buttons.

The fonts in which to display the items and the accelerator strings are in `:item-font` and `:accel-font`, respectively.

8.48.2 The Motif-Menu Accelerator Interactor

```
(create-instance 'gg:Motif-Menu-Accelerator-Inter inter:button-interactor
   (:window NIL)
   (:menus NIL)
```

```
    (:continuous NIL)
    (:start-where T)
    (:start-event (o-formula (multiple-value-call #'append
        (values-list (gvl :accel-chars)))))
    (:accel-chars (o-formula (mapcar #'(lambda (menu)
  (gv menu :global-accel-chars))
      (gvl :menus))))
    (:waiting-priority gg:motif-tab-priority-level)
    (:running-priority gg:motif-tab-priority-level)
    (:stop-action #'(lambda (interactor obj-over) ...))
    (:final-function NIL))
```

The `motif-menu-accelerator-inter` interactor is used with a set of `motif-menu` instances to implement the global character selection feature (:META-F2, etc. above). When an instance is supplied with a list of menus in the `:menus` slot and the window of the menus in the `:window` slot, then when the user strikes any of the accelerator keys defined in the menus, the corresponding menu item will be selected and its functions will be executed. Only one item may be assigned to each global accelerator character. An example of the `motif-menu-accelerator-inter` may be found in demo-motif and in the `motif-menu` demo.

8.48.3 Adding Items to the Motif-Menu

The `add-item` method for the `motif-menu` is similar to the standard method, except that the programmer may supply an accelerator to be added to the menu which corresponds to the item being added.

```
opal:add-item motif-menu item [:accelerator accel] [[:where] position [lo-
cator] [:key function-name]]
```

The value for *accel* should be an accelerator triplet that can be inserted into the `:accelerators` list of the *motif-menu*, such as '(#\R "Alt+F2" :META-F2). Note that the accelerator parameter must come **before** the "where" keys.

The usual `remove-item` method is used for the `motif-menu`, with the additional feature that the accelerator corresponding to the old item is automatically removed from the `:accelerators` list (if there is one).

8.49 [Motif Scrolling Menu]
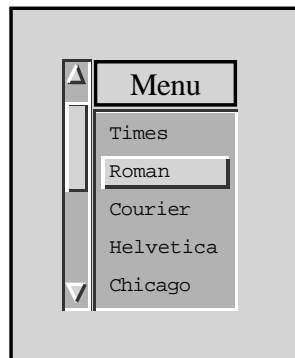

```
(create-instance 'gg:Motif-Scrolling-Menu motif-gadget-prototype
   (:maybe-constant '(:left :top :scroll-on-left-p
      :scr-incr :page-incr :min-frame-width :v-spacing :h-align
      :multiple-p :items :item-to-string-function
```

```
                    :item-font :num-visible :int-menu-feedback-p
                    :final-feedback-p :text-offset :title :title-font
                    :visible))
            (:left 0) (:top 0)
            (:active-p T)

            ;; Scroll bar slots
            (:scroll-on-left-p T)
            (:scr-incr 1)
            (:page-incr (o-formula (gvl :num-visible)))
            (:scroll-selection-function NIL)

            ;; Menu slots
            (:toggle-p T)
            (:min-frame-width 0)
            (:v-spacing 6)
            (:h-align :left)
            (:multiple-p T)
            (:items '("Item 1" "Item 2" "Item 3" "Item 4" "Item 5" "Item 6" "Item 7"█
              "Item 8" "Item 9" "Item 10" "Item 11" "Item 12" "Item 13"
              "Item 14" "Item 15" "Item 16" "Item 17" "Item 18" "Item 19"
              "Item 20"))
            (:item-to-string-function
             #'(lambda (item)
        (if item
            (if (stringp item)
        item
        (string-capitalize (string-trim ":" item)))
            "")))
            (:item-font opal:default-font)
            (:num-visible 5)
            (:int-menu-feedback-p T)
            (:final-feedback-p T)
            (:text-offset 6)
            (:title NIL)
            (:title-font (opal:get-standard-font :serif :roman :large))
            (:menu-selection-function NIL)
            (:selected-ranks NIL)
            (:foreground-color opal:motif-gray)
            (:value (o-formula ...)))
```

The loader file for the `motif-scrolling-menu` is named "motif-scrolling-menu-loader".

The `motif-scrolling-menu` is very much like the standard `scrolling-menu`, but there are a few differences. Since the scrolling window has a motif-v-scroll-bar as a part of it, the slots `:min-scroll-bar-width`, `page-trill-p`, `:indicator-text-p`, and `:int-scroll-feedback-p` are not applicable.

Also, the `motif-scrolling-menu` has a slot `:foreground-color`, which is absent in the standard `scrolling-menu`.

8.50 Motif-Menubar
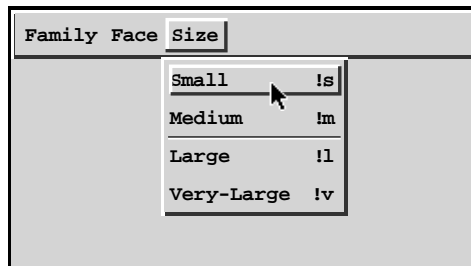
Figure 8.20: An instance of the [motif-menubar] gadget

```
(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 [menubar], page 447. 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 [motif-menubar-pix], page 547.

8.50.1 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 *submenus* 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 *item* 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.

8.50.2 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.

8.50.3 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 [motif-menubar-pix], page 547, there is a bar above third item in the last submenu.

8.50.4 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 [mmbar-components], page 552, 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 [mmbar-sel-fns], page 548, 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.

8.50.5 An Example

The following example creates the motif-menubar pictured in Figure [motif-menubar-pix], page 547. 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)
```

8.50.6 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*] ⟨undefined⟩ [method],█
page ⟨undefined⟩

                                                                [[: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* ⟨undefined⟩ [method],█
page ⟨undefined⟩

                              [: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)
```

8.50.7 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>` ⟨undefined⟩ `[method]`, page ⟨undefined⟩

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>` ⟨undefined⟩ `[method]`, page ⟨undefined⟩

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

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

8.50.8 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.

8.50.9 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.

8.50.10 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], page 90`

Will return an instance of `motif-menubar`.

`gg:Make-Motif-Bar-Item &key <desc font title> [function], page 90`

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.

8.50.11 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> ⟨undefined⟩ [method],`
page ⟨undefined⟩

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> ⟨undefined⟩ [method],`
page ⟨undefined⟩

Sets the <motif-bar-item> to have the new submenu-items. For more information on these two functions, see section [menubar], page 447.

```
    opal:Add-Item <motif-menubar motif-bar-item> [:accelerators <accels>] ⟨un▮
    defined⟩ [method], page ⟨undefined⟩
              <                                                > [[:where] <position>
```

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 [mmbar-accelerators], page 549.

> opal:Add-Item <motif-bar-item motif-menu-item> ⟨undefined⟩ [method], page ⟨un▮ defined⟩
>
>   [: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)
```

8.50.12 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)
```

8.50.13 Methods Shared with the Regular Menubar

The following methods have the same effect on the motif-menubar as they have on the standard menubar. Please see section [menubar], page 447, for more information.

> gg:Menubar-Components <motif-menubar> ⟨undefined⟩ [method], page ⟨undefined⟩▮
> gg:Submenu-Components <motif-bar-item> ⟨undefined⟩ [method], page ⟨undefined⟩▮
> gg:Get-Bar-Component <motif-menubar> <item> ⟨undefined⟩ [method], page ⟨un▮ defined⟩
> gg:Get-Submenu-Component <motif-bar-item> <item> ⟨undefined⟩ [method], page ⟨un▮ defined⟩
> gg:Find-Submenu-Component <motif-menubar> <submenu-title> <submenu-item> ⟨un▮ defined⟩ [method], page ⟨undefined⟩

gg:Menubar-Disable-Component <motif-menubar-component> ⟨undefined⟩ [method],█
page ⟨undefined⟩
gg:Menubar-Enable-Component <motif-menubar-component> ⟨undefined⟩ [method],█
page ⟨undefined⟩
gg:Menubar-Enabled-P <motif-menubar-component> ⟨undefined⟩ [method], page ⟨un█
defined⟩
gg:Menubar-Get-Title <motif-menubar-component> ⟨undefined⟩ [method], page ⟨un█
defined⟩
gg:Menubar-Set-Title <motif-menubar-component> ⟨undefined⟩ [method], page ⟨un█
defined⟩
gg:Menubar-Installed-P <motif-menubar-component> ⟨undefined⟩ [method], page ⟨un█
defined⟩

8.51 Motif-Scrolling-Labeled-Box

```
(create-instance 'gg:Motif-Scrolling-Labeled-Box motif-gadget-prototype
   (:maybe-constant '(:left :top :width :field-offset :label-offset :label-string█
                      :field-font :label-font :foreground-color :active-p :visible))█
   (:left 0)
   (:top 0)
   (:width 135)
   (:field-offset 4)
   (:label-offset 5)
   (:label-string "Label:")
   (:value "Field")
   (:field-font opal:default-font)    ;;**Must be fixed width**
   (:label-font (create-instance NIL opal:font (:face :bold)))
   (:foreground-color opal:motif-gray)
   (:keyboard-selection-p NIL)
   (:active-p T)
   (:selection-function NIL)   ; (lambda (gadget value))
   )
```

Label: `Field`
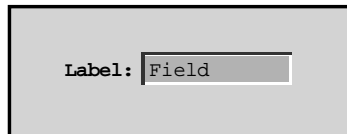
The loader file for the `motif-scrolling-labeled-box` is "motif-scrolling-labeled-box-loader".

This gadget is a Motif version of the `scrolling-labeled-box`. The `:width` of the box is fixed, and the `:value` string will scroll inside the box if it is too long to be displayed.

When the `:active-p` slot is set to `nil`, both the label and the field will appear "grayed-out" and the field will not be editable.

8.52 Motif-Error-Gadget

Error: Invalid input from user.
Press OK to continue.

OK
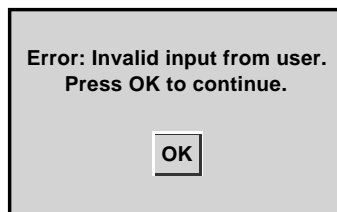
```
(create-instance 'gg:Motif-Error-Gadget opal:aggregadget
   (:string "Error")
   (:parent-window NIL)
   (:font (opal:get-standard-font :sans-serif :bold :medium))
```

```
(:justification :center)
(:modal-p T)
(:beep-p T)
(:window NIL)          ;; Automatically initialized
(:foreground-color opal:motif-orange)
(:selection-function NIL)   ; (lambda (gadget value))
)
```

The loader file for the `motif-error-gadget` is "motif-error-gadget-loader".

The `motif-error-gadget` is a dialog box that works exactly the same way as the `error-gadget` described in section [error-gadget], page 488. The same caveats apply, and the functions `display-error` and `display-error-and-wait` may be used to display the dialog box.

There is an additional slot provided in the `motif-error-gadget` which determines the color of the dialog box. The `:foreground-color` slot may contain any instance of `opal:color`.

8.53 Motif-Query-Gadget

```
(create-instance 'gg:Motif-Query-Gadget gg:motif-error-gadget
  (:string "Is that OK?")
  (:button-names '("OK" "CANCEL"))
  (:parent-window NIL)
  (:font (opal:get-standard-font :sans-serif :bold :medium))
  (:justification :center)
  (:modal-p T)
  (:beep-p T)
  (:window NIL)          ;; Automatically initialized
  (:foreground-color opal:motif-orange)
  (:selection-function NIL)   ; (lambda (gadget value))
  )
```

The loader file for the `motif-query-gadget` is "motif-error-gadget-loader" (it is defined in the same file as the `motif-error-gadget`).

The `motif-query-gadget` works exactly the same way as the `query-gadget` described in section [query-gadget], page 492. It has more buttons than the `motif-error-gadget`, so it can be used as a general-purpose dialog box. The functions `display-query` and `display-query-and-wait` may be used to display the dialog box.

The additional `:foreground-color` slot may contain any instance of `opal:color`, and determines the color of the dialog box.

8.54 [Motif Save Gadget]
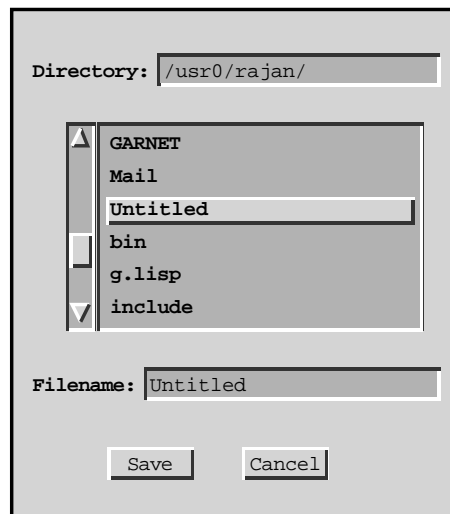
Figure 8.21: An instance of the Motif save gadget. "/usr0/rajan" is the current directory.

```
(create-instance 'motif-save-gadget opal:aggregadget
  (:maybe-constant '(:left :top :parent-window :window-title :window-left :window-top
```

```
                    :message-string :num-visible :initial-directory :button-panel-items
                    :button-panel-h-spacing :min-gadget-width :check-filenames-p
                                :modal-p :query-message :query-buttons :foreground-color█
                                :dir-input-field-font :dir-input-label-font :file-input-field-fon
                                :file-input-label-font :file-menu-font :button-panel-font█
                                :message-font))
          (:window-title "save window")
          (:initial-directory "./")
          (:message-string "fetching directory...")
          (:num-visible 6)
          (:button-panel-items '("save" "cancel"))
          (:button-panel-h-spacing 25)
          (:min-gadget-width 240)
          (:modal-p NIL)
          (:check-filenames-p t)
          (:query-message "save over existing file")
          (:foreground-color opal:motif-light-blue)
          (:selection-function NIL)    ; (lambda (gadget value))
          (:dir-input-field-font (opal:get-standard-font NIL NIL :small))
          (:dir-input-label-font (opal:get-standard-font NIL :bold NIL))
          (:file-input-field-font (opal:get-standard-font NIL NIL :small))
          (:file-input-label-font (opal:get-standard-font NIL :bold NIL))
          (:message-font (opal:get-standard-font :fixed :italic :small))
          (:file-menu-font (opal:get-standard-font NIL :bold NIL))
          (:button-panel-font opal:default-font)
          ...)
```

The motif-save-gadget works exactly like the save-gadget, described in section [save-gadget], page 492. The only difference is that the motif-save-gadget has a slot called :foreground-color which allows the user to set the color of the gadget. This slot can be set to any opal:color object.

The loader file for the motif-save-gadget is named "motif-save-gadget-loader". Figure [motif-save-gadget-tag], page 559, shows an instance of the Motif save gadget.

8.55 [Motif Load Gadget]

```
    (create-instance 'gg:Motif-Load-Gadget opal:aggregadget
      (:maybe-constant '(:left :top :parent-window :window-title :window-left█
          :window-top :dir-input-field-font :dir-input-label-font
          :message-font :message-string :num-visible :file-menu-font
          :initial-directory :file-input-field-font
          :file-input-label-font :button-panel-items :button-panel-font
          :button-panel-h-spacing :min-gadget-width :modal-p
          :check-filenames-p :foreground-color)))
      (:parent-window NIL)
      (:window-title "load window")
      (:message-string "fetching directory...")
      (:num-visible 6)
```

```
(:initial-directory "./")
(:button-panel-items '("load" "cancel"))
(:button-panel-h-spacing 25)
(:min-gadget-width 240)
(:modal-p NIL)
(:check-filenames-p t)
(:foreground-color opal:motif-light-blue)
(:selection-function NIL)   ; (lambda (gadget value))
(:dir-input-field-font (opal:get-standard-font NIL NIL :small))
(:dir-input-label-font (opal:get-standard-font NIL :bold NIL))
(:file-input-field-font (opal:get-standard-font NIL NIL :small))
(:file-input-label-font (opal:get-standard-font NIL :bold NIL))
(:message-font (opal:get-standard-font :fixed :italic :small))
(:file-menu-font (opal:get-standard-font NIL :bold NIL))
(:button-panel-font opal:default-font)
...)
```

The `motif-load-gadget` is loaded along with the `motif-save-gadget` by the file "motif-save-gadget-loader".

The `motif-load-gadget` works the same way as the standard `load-gadget`. The only difference is that the motif gadget has an additional `:foreground-color` slot, which can be set to any `opal:color` object.

8.56 Motif Property Sheets

The following property sheets are similar to the standard property sheets, except that they use the Motif look and feel. For a complete discussion of the use of property sheets, see section [propertysheets], page 499.

8.56.1 Motif-Prop-Sheet-With-OK

```
(create-instance 'gg:Motif-Prop-Sheet-With-OK opal:aggregadget
    (:maybe-constant '(:left :top :items :default-filter :ok-function :apply-function
                       :cancel-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 :visibl

    ; Customizable slots
    (:foreground-color opal:motif-gray) ; the color for the background
    (:left 0) (:top 0)
    (:items NIL)
    (:default-filter 'default-filter)
    (:OK-Function NIL)
    (:Apply-Function NIL)
    (:Cancel-Function NIL)
    (:v-spacing 1)
    (:pixel-margin NIL)
    (:rank-margin NIL)
```

```
        (:multi-line-p NIL)
        (:select-label-p NIL)
        (:label-select-event :any-mousedown)
        (:label-selected-func NIL)
        (:select-value-p NIL)
        (:value-selected-func NIL)
        (:single-select-p NIL)

      ; Read-only slots
        (:label-selected ...
        (:value-selected ...
        (:value ...
        (:changed-values ...
```

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

This is the same as `Prop-Sheet-With-OK` (described in section [propsheetwithok], page 508, except that it uses the Motif look-and-feel, and you can set the foreground color.

8.56.2 Motif-Prop-Sheet-For-Obj-With-OK

```
    (create-instance 'Motif-Prop-Sheet-For-Obj-With-OK Motif-Prop-Sheet-With-OK█
        (:maybe-constant '(:left :top :obj :slots :eval-p :ok-function :apply-function█
                           :cancel-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 :visibl

      ; Customizable slots
        (:foreground-color opal:motif-gray)
        (:OK-Function NIL)
        (:Apply-Function NIL)
        (:Cancel-Function NIL)
        (:left 0) (:top 0)
        (: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 in-█
    tersection 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
```

Figure 8.22: An example of `motif-prop-sheet-for-obj-with-OK` containing some gadgets. The code to create this is shown in section [propexample], page 512.

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

The implementation and operation of `motif-prop-sheet-for-obj-with-ok` is identical to the `prop-sheet-for-obj-with-ok` gadget (described in section [propsheetforobjwithok], page 509, with the exception that the `:foreground-color` slot may be set to any `opal:color` object.

8.57 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 'gg: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 in-
tersection 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".

8.58 Motif Scrolling Window

```
    (create-instance 'gg:Motif-Scrolling-Window-With-Bars opal:aggregadget
       (:maybe-constant '(:left :top :width :height :border-width :title :total-width▉
                          :total-height :foreground-color :h-scroll-bar-p :v-scroll-bar-p▉
          :h-scroll-on-top-p :v-scroll-on-left-p :h-scr-incr :h-page-incr
                          :v-scr-incr :v-page-incr :icon-title :parent-window :visible))▉

       ;; Window slots
       (:left 0)  ; left, top, width and height of outermost window
       (:top 0)
       (:position-by-hand NIL) ; if T, then left,top ignored and user asked for win-▉
    dow position
       (:width 150) ; width and height of inside of outer window
       (:height 150)
       (:border-width 2) ; of outer window
       (:parent-window NIL) ; window this scrolling-window is inside of, or NIL if top lev
       (:double-buffered-p NIL)
       (:omit-title-bar-p NIL)
       (:title "Motif-Scrolling-Window")
       (:icon-title (o-formula (gvl :title))) ; Default is the same as the title▉
       (:total-width 200)   ; total size of the scrollable area inside
       (:total-height 200)
       (:X-Offset 0)  ; x offset in of the scrollable area. CANNOT BE A FORMULA▉
       (:Y-Offset 0)  ; CANNOT BE A FORMULA
       (:visible T)  ; whether the window and bars are visible (mapped)
       (:foreground-color opal:motif-gray)

       (:h-scroll-bar-p T)  ; Is there a horizontal scroll bar?
       (:v-scroll-bar-p T)  ; Is there a vertical scroll bar?

       ;; Scroll Bar slots
       (:h-scroll-on-top-p NIL)  ; whether horiz scroll bar is on top or bottom▉
       (:v-scroll-on-left-p T)   ; whether vert scroll bar is on left or right▉
       (:h-scr-incr 10)  ; in pixels
       (:h-page-incr (o-formula (- (gvl :width) 10))) ; default jumps one page mi-▉
    nus 10 pixels
       (:v-scr-incr 10)  ; in pixels
```

```
      (:v-page-incr (o-formula (- (gvl :height) 10))) ; default jumps one page mi-
  nus 10 pixels

      ;; Read-Only slots
      (:Inner-Window NIL)   ; these are created by the update method
      (:inner-aggregate NIL) ; add your objects to this aggregate (but have to up-
  date first)
      (:outer-window NIL) ; call Opal:Update on this window (or on gadget itself)
      (:clip-window NIL)
      ...)
```
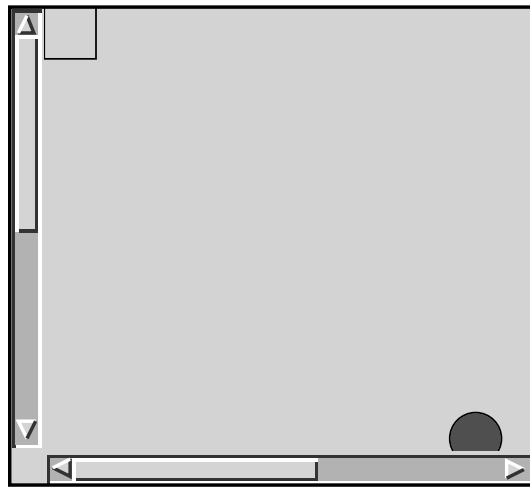
The loader file for the `motif-scrolling-window-with-bars` gadget is "motif-scrolling-window-loader".

The use of `motif-scrolling-window-with-bars` is identical to the `scrolling-window-with-bars` gadget described in section [scrolling-windows], page 471, with the exception that the parameters to the scroll bars are slightly different and the `:foreground-color` can be set.

Caveats:

> If the motif-scrolling-window has a `:parent-window`, update the parent window before instantiating the motif-scrolling-window.
>
> Update the scrolling-window gadget before referring to its inner/outer windows and aggregates.
>
> The instance of the motif-scrolling-window should **not** be added to an aggregate.

The `motif-scrolling-window-with-bars` gadget is not a window itself; it is an aggregadget that points to its own windows. These windows are accessed through the `:outer-window`, `:inner-window`, and `:clip-window` slots of the gadget, as in (g-value MY-SCROLL-WIN :outer-window). So you cannot call `opal:make-ps-file` with the scrolling-window gadget as an argument. You have to send one of the windows that it points to:

```
> (opal:make-ps-file (g-value SCROLL-WIN :outer-window)
                     "fig.PS" :LANDSCAPE-P T :BORDERS-P :MOTIF)
T
>
```

8.59 Using the Gadgets: Examples

8.60 Using the :value Slot

In order to use the value returned by a gadget, we have to access the top level `:value` slot. As an example, suppose we want to make an aggregadget out of a vertical slider and a circle, and that we want the diameter of the circle to be dependent on the current value of the slider. We may create such a unit by putting a formula in the `:diameter` slot of the circle that depends on the value returned from the slider. Such an aggregadget is implemented below. The formula in the `:diameter` slot of the circle uses the KR function `gvl` to access the `:value` slot of the vertical slider.

```
(create-instance 'BALLOON opal:aggregadget
   (:parts
    `((:slider ,gg:v-slider
               (:left 10)
               (:top 20))
      (:circle ,opal:circle
               (:diameter ,(o-formula (gvl :parent :slider :value)))
               (:left 100) (:top 50)
               (:width ,(o-formula (gvl :diameter)))
               (:height ,(o-formula (gvl :diameter)))))))
```

8.61 Using the :selection-function Slot

In order to execute a function whenever any new value or item is selected (i.e., when the `:value` slot changes), that function must be specified in the slot `:selection-function`. Suppose we want a set of buttons which give us a choice of several ancient cities. We would also like to define a function which displays a message to the screen when a new city is selected. This panel can be created with the definitions below.

```
(create-instance 'MY-BUTTONS gg:text-button-panel
```

```
      (:selection-function #'Report-City-Selected)
      (:items '("Athens" "Babylon" "Rome" "Carthage")))

  (defun Report-City-Selected (gadgets-object value)
    (format t "Selected city:  ~S~%~%" value)
    (format t "Pressed button object ~S~%"
              (gv gadgets-object :value-obj)))
```

8.62 Using Functions in the :items Slot

In order to execute a specific function when a specific menu item (or button) is selected, the desired function must be paired with its associated string or atom in the :items list. A menu which executes functions assigned to item strings appears below. Only one function (My-Cut) has been defined, but the definition of the others is analogous.

```
  (create-instance 'MY-MENU gg:menu
     (:left 20)
     (:top 20)
     (:title "Menu")
     (:items '(("Cut" my-cut) ("Copy" my-copy) ("Paste" my-paste))))

  (defun my-cut (gadgets-object item-string)
    (format t "Function CUT called~%~%"))
```

8.63 Selecting Buttons

In order to directly select a button in a button panel (rather than allowing the user to select the button with the mouse), either the :value or :value-obj slots may be set. However, neither of these slots may be given values at the time that the button panel is created (i.e., *do not supply values in the* create-instance *call for these slots*), since this would permanently override the formulas in the slots.

The :value slot may be set with any of the items (or the first element in any of the item pairs) in the :items list of the button panel. The example below shows how buttons on a text-button-panel and an x-button-panel could be chapterly selected. In both cases, the selected items (i.e., those appearing in the :value slot) will appear selected when the button panels are displayed in a window.

```
  (create-instance 'MY-TEXT-BUTTONS gg:text-button-panel
     (:items '(:left :center :right)))
  (gv MY-TEXT-BUTTONS :value)   ;; initialize the formula in the :value slot
  (s-value MY-TEXT-BUTTONS :value :center)

  (create-instance 'MY-X-BUTTONS gg:x-button-panel
     (:items '("Bold" "Underline" "Italic")))
  (gv MY-X-BUTTONS :value)   ;; initialize the formula in the :value slot
  ;; Value must be a list because x-buttons have multiple selection
  (s-value MY-X-BUTTONS :value '("Bold" "Underline"))
```

Buttons may also be selected by setting the :value-obj slot to be the actual button object or list of button objects which should be selected. This method requires the designer

to look at the internal slots of the button gadgets. The example below shows how the same
results may be obtained using this method as were obtained in the above example.

```
(create-instance 'NEW-TEXT-BUTTONS gg:text-buttons-panel
   (:items '(:left :center :right)))
(s-value NEW-TEXT-BUTTONS
         :value-obj
         ;; The second button corresponds to the item ":center"
         (second (gv NEW-TEXT-BUTTONS :text-button-list :components)))
```

The `:value` slot of a single button will either contain the `:string` of the button or `nil`.
Single buttons will appear selected when the `:value` slot contains any non-NIL value.

8.64 The :item-to-string-function Slot

The `:items` slot of the scrolling menu may be a list of any objects at all, including the
standard items described in section [buttons], page 429. The mechanism which allows strings
to be generated from arbitrary objects is the user-defined `:item-to-string-function`. The
default scrolling menu will handle a list of standard items, but for a list of other objects a
suitable function must be supplied.

As discussed in section [items-slot], page 410, the elements of the `:items` list can be
either single atoms or lists. When an element of the `:items` list is a list, then the `:item-to-
string-function` is applied only to the first element in the list, rather than the list itself. In
other words, the `:item-to-string-function` takes the `car` of the item list as its parameter,
rather than the entire item list.

Suppose the list in the `:items` slot of the scrolling menu is

```
(list v-scroll-bar v-slider trill-device)
```

which is a list of Garnet Gadget schemas. A function must be provided which returns
a string identifying an item when given a schema as input. The following slot/value pair,
inserted into the definition of the new schema, will accomplish this task:

```
(:item-to-string-function #'(lambda (item)
                               (if item
                                   (name-for-schema item)  ;; imported from KR
                                   "")))
```

9 Debugging Tools for Garnet Reference chapter

by Roger B. Dannenberg, Andrew Mickish, Dario Giuse

10 May 2020

9.1 Abstract

Debugging a constraint-based graphical system can be difficult because critical interdependencies can be hard to visualize or even discover. The debugging tools for Garnet provide many convenient ways to inspect objects and constraints in Garnet-based systems.

9.2 Introduction

This chapter is intended for users of the Garnet system and assumes that the reader is familiar with Garnet. Other reference chapters cover the object and constraint system KR *KRTR2*, the graphics system Opal *OpalCHAPTER*, Interactors *InterCHAPTER* for handling keyboard and mouse input, Aggregadgets *AggregadgetsCHAPTER* for making instances of aggregates of Opal objects.

9.3 Notation in this Chapter

In the examples that follow, user type-in follows the asterisk (*), which is the prompt character in CMU Common Lisp on the RT. Function results are printed following the characters "-->". This is not what CMU Common Lisp prints, but is added to avoid confusion, since most debugging functions print text in addition to returning values:

```
* (some-function an-arg or-two)
some-function prints out this information,
    which may take several lines
--> function-result-printed-here
```

9.4 Loading and Using Debugging Tools

Normally, debugging tools will be loaded automatically when you load the file `garnet-loader.lisp`. Presently, the debugging tools are located in the files `debug-fns.lisp` and `objsize.lisp`. A few additional functions are defined in the packages they support.

Most of the debugging tools are in the `GARNET-DEBUG` package, and you should ordinarily type

```
(use-package "GARNET-DEBUG")
```

to avoid typing the package name when using these tools. Functions and symbols mentioned in this document that are *not* in the `GARNET-DEBUG` package will be shown with their full package name.

9.5 Inspecting Objects

9.5.1 Inspector

The `Inspector` is a powerful tool that can be of significan help in debugging. It pops up a window showing an object, and also shows 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 [inspectorfig], page 574.

```
┌──────────────────────────────────────────────────────────────────────────┐
│ ▭                          Showing #k<TEXT-6508>                      ▫  □ │
├──────────────────────────────────────────────────────────────────────────┤
│ ┌───────────┐┌───────────┐┌─────────┐┌────────────┐┌────┐┌────────┐       │
│ │Show Object││Show in New││Re-Fetch ││Dependencies││Done││Done All│       │
│ └───────────┘└───────────┘└─────────┘└────────────┘└────┘└────────┘       │
│ ┌─────┐┌──────┐┌─────┐┌──────┐┌────────────┐┌───────────────┐┌───────┐     │
│ │Flash││Search││Break││Notify││Clear Breaks││Inherited Slots││Objects│     │
│ └─────┘└──────┘└─────┘└──────┘└────────────┘└───────────────┘└───────┘     │
│ #k<TEXT-6508>                                                              │
│                                                                            │
│  :IS-A = (#k<TEXT-6509>)                                                   │
│  :LEFT - #k<F2975-4250-6074> (V) = 114                                     │
│  :TOP - #k<F2976-4252-6075> (V) = 7                                        │
│  :WIDTH - #k<F32-3310-3311-5501> (V) = 27                                  │
│  :HEIGHT - #k<F31-3333-3334-5511> (V) = 14                                 │
│  :VISIBLE - #k<F16-352-357-3277-3278-5448> (V) = T                         │
│  :LINE-STYLE - #k<F2979-4254-6076> (V) = #k<OPAL:DEFAULT-LINE-STYLE>       │
│  :PARENT (C) = #k<MOTIF-BAR-ITEM-3024-5430>                                │
│  :FONT - #k<F2977-3327-5508> (V) = #k<FONT-192>                            │
│  :KNOWN-AS (C) = :TEXT                                                     │
│  :STRING - #k<F2978-3313-5502> (V) = "Size"                               │
│  :UPDATE-INFO = #<Update-Info dirty-p NIL invalid-p NIL>                   │
│  :WINDOW = #k<DEMO-MULTIFONT::WIN>                                         │
└──────────────────────────────────────────────────────────────────────────┘
```
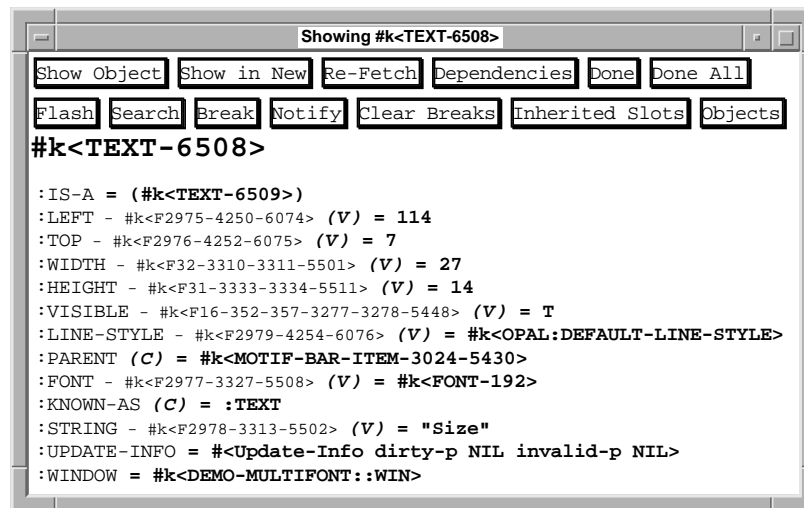
Figure 9.1: The Inspector showing a text object.

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.

An example of using the Inspector is included in the Tutorial at the beginning of this Reference chapter.

9.5.2 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 [inspectorfig], page 574. 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], page 90
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], page 90
```

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

9.5.3 Schema View

The schema view shown in Figure [inspectorfig], page 574, 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.

9.5.4 Object View

Hitting the `"Objects"` button brings up the view in Figure [inspectorobjects], page 577. This view shows the name of the object being inspected at the top, then the `is-a` hierarchy. In Figure [inspectorobjects], page 577, `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.

```
┌─────────────────────────────────────────────────────────────────┐
│ ▢                      Showing #k<TEXT-6508>                ▫ ▢   │
├─────────────────────────────────────────────────────────────────┤
│ ┌───────────┐┌───────────┐┌─────────┐┌────────────┐┌────┐┌─────┐ │
│ │Show Object││Show in New││Re-Fetch ││Dependencies││Done││Done All│
│ └───────────┘└───────────┘└─────────┘└────────────┘└────┘└─────┘ │
│ ┌─────┐┌──────┐┌─────┐┌──────┐┌────────────┐┌──────────────┐┌───────┐
│ │Flash││Search││Break││Notify││Clear Breaks││Inherited Slots││Objects│
│ └─────┘└──────┘└─────┘└──────┘└────────────┘└──────────────┘└───────┘
│ #k<TEXT-6508>                                                    │
│                                                                 │
│                                                                 │
│ For #k<TEXT-6508> :                                             │
│                                                                 │
│ IS-A hierarchy:                                                 │
│     #k<TEXT-6509>                                               │
│     #k<TEXT-7180>                                               │
│     #k<OPAL:TEXT>                                               │
│     #k<OPAL:GRAPHICAL-OBJECT>                                   │
│     #k<OPAL:VIEW-OBJECT>                                        │
│                                                                 │
│ Aggregate hierarchy (:parent slot):                            │
│     #k<MOTIF-BAR-ITEM-3024-5430>                               │
│     #k<MENUBAR-ITEMS-7179>                                     │
│     #k<DEMO-MULTIFONT::PULL-DOWN>                              │
│     #k<DEMO-MULTIFONT::TOP>                                    │
│     #k<DEMO-MULTIFONT::WIN>                                    │
│                                                                 │
│ Objects Shown in This Window:                                  │
│     #k<TEXT-6508>                                              │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```
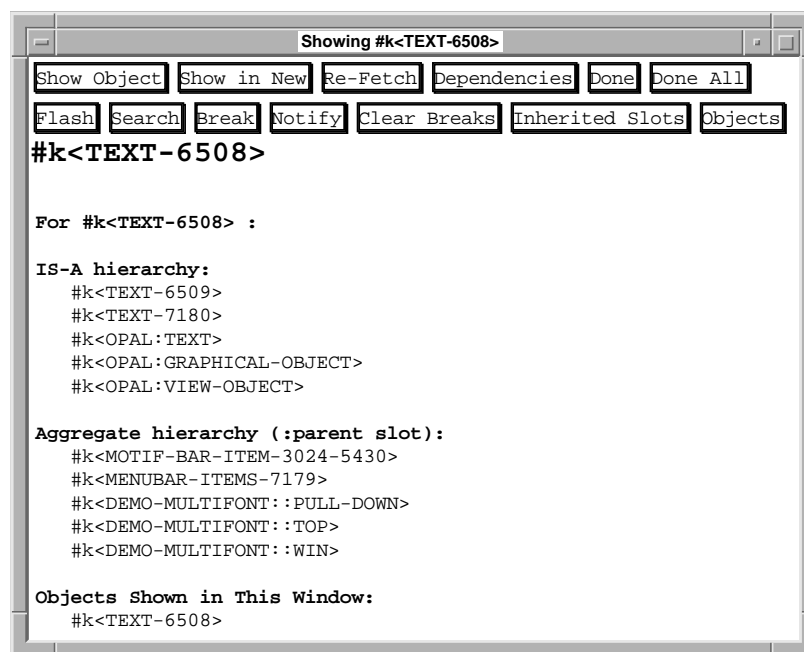
Figure 9.2: The Inspector showing the objects related to the text object of Figure [inspectorfig], page 574.

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.

9.5.5 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 [inspectordepfig], page 579. This slows the slots used in calculating the value in the formula.

```
┌─────────────────────────────────────────────────────────────────────────┐
│ ▭                        Showing #k<TEXT-6508>                       ▫ ▢ │
├─────────────────────────────────────────────────────────────────────────┤
│ [Show Object] [Show in New] [Re-Fetch] [Dependencies] [Done] [Done All] [Flash] [Search] │
│ [Break] [Notify] [Clear Breaks] [Inherited Slots] [Objects]             │
│ #k<TEXT-6508>                                                           │
│                                                                         │
│ Slot :VISIBLE of #k<TEXT-6508> (formula = #k<F16-352-357-3277-3278-5448>) = T: │
│ Expression = (LET ((OPAL::PARENT (GVL :PARENT)))   (OR (NULL OPAL::PARENT) (GV OPAL::PARENT :VIS │
│   (OR (NULL OPAL::PARENT) (GV OPAL::PARENT :VISIBLE)))) │
│ Dependencies:                                                          │
│    :VISIBLE of #k<MOTIF-BAR-ITEM-3024-5430> = T                         │
│       :VISIBLE of #k<MENUBAR-ITEMS-7179> = T                            │
│          :VISIBLE of #k<DEMO-MULTIFONT::PULL-DOWN> = T                  │
│             ...                                                         │
│       :PARENT of #k<MOTIF-BAR-ITEM-3024-5430> = #k<MENUBAR-ITEMS-7179>  │
└─────────────────────────────────────────────────────────────────────────┘
```
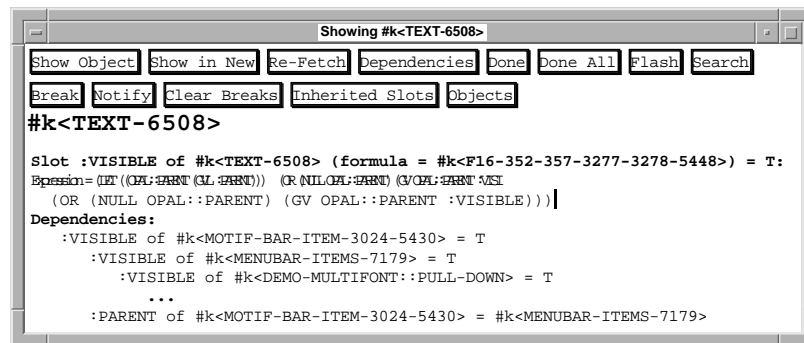
Figure 9.3: The Inspector showing the dependencies of the `:visible` slot of the object shown in Figure [inspectorfig], page 574.

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 any slot name and hit the `"Dependencies"` button.

9.5.6 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 [dependencysec], page 578).

`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 [inspectobjects], page 576).

9.6 PS – Print Schema

The same information that is shown in the Inspector for an object's slots and values can be printed with the simpler `kr:ps`. This function does not create a new window to show the information, but instead prints right into the lisp listener.

```
kr:PS object &key types-p  all-p (control T) (inherit NIL) [function], page 90
                (indent 0) (stream *standard-output*)
```

(All the nuances of this function are described in the KR chapter.)

9.7 Look, What, and Kids

For quick inspection of objects, the `look`, `what`, and `kids` functions may be used:

```
gd:Look object &optional (detail 2)[function], page 90
gd:What object[function], page 90
gd:Kids object[function], page 90
```

The `look` function prints out varying amounts of information about an object, depending upon the optional argument *detail*:

(look obj 0) prints a one-line description of obj. This is equivalent to calling (what obj).

(look obj 1) prints a one-line description of obj and also shows the immediate components of obj if it is an aggregate. This form is equivalent to calling (kids obj).

(look obj 2) recursively prints all components of obj. This is the default, equivalent to typing (look obj). Use it to look at the structure of an aggregate.

(look obj 3) prints slots of obj, using ps, and then prints the tree of components.

(look obj 4) prints slots of obj and its immediate components. Any (trees of) sub-components are also printed.

(look obj 5) prints what is essentially complete information about a tree of objects, including all slots of all components.

For example,

```
* (what mywindow)
#k<MYWINDOW> is-a #k<INTERACTOR-WINDOW> (WINDOW)
--> NIL
* (look mywindow)
#k<MYWINDOW> is-a #k<INTERACTOR-WINDOW> (WINDOW)
   #k<MYAGG> is-a #k<AGGREGATE> (VIEW-OBJECT)
      #k<MYRECT> is-a #k<MOVING-RECTANGLE> (RECTANGLE)
      #k<MYTEXT> is-a #k<CURSOR-MULTI-TEXT> (MULTI-TEXT)
--> NIL
```

9.8 Is-A-Tree

Look prints the parent of the object and then the "standard parent" The "standard parent" is the first named object encountered traveling up the `:is-a` tree. If `look` does not print enough information about an object, the `is-a-tree` function might be useful:

> `gd:Is-A-Tree[function]`, page 90

This function traces up `:is-a` links and prints the resulting tree:

```
* (is-a-tree mytext)
#k<MYTEXT> is-a
   #k<CURSOR-MULTI-TEXT> is-a
      #k<MULTI-TEXT> is-a
         #k<TEXT> is-a
            #k<GRAPHICAL-OBJECT> is-a
               #k<VIEW-OBJECT>
--> NIL
```

9.9 Finding Graphical Objects

It is often necessary to locate a graphical object or figure out why a graphical object is not visible.

> `gd:Where object[function]`, page 90

prints out the *object*'s `:left`, `:top`, `:width`, `:height`, and `:window` in a one-line format.

```
* (where mywindow)
#k<MYWINDOW> :TOP 43 :LEFT 160 :WIDTH 355 :HEIGHT 277
--> NIL
* (where myagg)
#k<MYAGG> :TOP 20 :LEFT 80 :WIDTH 219 :HEIGHT 150 :WINDOW #k<MYWINDOW>
--> NIL
```

If you are not sure which screen image corresponds with a particular Opal object, use the following function:

> `gd:Flash object[function]`, page 90

The `flash` function will invert the bounding box of *object* making the object flash on and off. `flash` has two interesting features:

You can `flash` aggregates, which are otherwise invisible.

If the object is not visible, `flash` will try hard to tell you why not. Possible reasons include:

The object does not have a window,

The window does not have an aggregate,

The object is missing a critical slot (e.g. `:left`),

The object is outside of its window,

The object's `:visible` slot is `nil`,

The aggregate containing the object is not visible, or

The object is outside of its aggregate (a problem with the aggregate).

Flash does not test to see if the object is obscured by another window. If flash does not complain and you do not see any blinking, use where to find the object's window. Then use where (or flash) applied to the window to locate the window on your screen. Bring the window to the front and try again.

The invert function is similar to flash, but it leaves the object inverted. The uninvert function will undo the effect of invert:

        gd:Invert *object*[function], page 90
        gd:Uninvert *object*[function], page 90

invert uses a single Opal rectangle to invert an area of the screen. If the rectangle is in use, it is first removed, so at most one region will be inverted at any given time. Unlike, flash, invert depends upon Opal, so if Opal encounters problems with redisplay, invert will not work (see fix-up-window in Section [fix-up-windows], page 587).

The previous functions are only useful if you know the name of a graphical object. To obtain the name of an object that is visible on the screen, use:

        gd:Ident[function], page 90

Ident waits for the next input event and reports the object under the mouse at the time of the event. In addition to printing the leaf object under the mouse, ident runs up the :parent links and prints the chain of aggregates up to the window. Some interesting features to note are:

   ident will report a window if you do not select an object.

   ident returns a list[1] in the form (*object window x y code*) so you can then use the selection in another expression, e.g. (kr:ps (car (ident))). *Object* will be nil if none was selected.

   ident also prints the input event and mouse location. For instance, use ident if you want to know the Lisp name for the character transmitted when you type the key labeled "Home" on your keyboard or to tell you the window coordinates of the mouse.

Another way to locate

        gd:Windows[function], page 90

which prints a list of Opal windows and their locations. The list of windows is returned. Only mapped windows are listed, so windows will only report a window that has been opal:update'd.

        * (windows)
        #k<MYWINDOW> :TOP 43 :LEFT 160 :WIDTH 355 :HEIGHT 277
        #k<DEMO-GROW::VP> :TOP 23 :LEFT 528 :WIDTH 500 :HEIGHT 300
        --> (#k<DEMO-GROW::VP> #k<MYWINDOW>)

9.10 Inspecting Constraints

Formulasnot always help when formulas and objects are inherited and/or created at run time. To make dependencies explicit, the explain-slot

        gd:Explain-Slot *object slot*[function], page 90

---

[1] A list is returned rather than a multiple value because multiple values print out on multiple lines in CMU Common Lisp, taking too much screen space when ident is used interactively.

explain-slot will track down all dependencies of *object*'s *slot* and prints them. Indirect dependencies that occur when a formula depends upon the value of another formula are also printed. The complete set of dependencies is a directed graph, but the printout is tree-structured, representing a depth-first traversal of the graph. The search is cut off whenever a previously visited node is encountered. This can represent either a cycle or two formulas with a common dependency.

In the following example, the :top of mytext depends upon the :top of myrect which in turn depends upon its own :box slot:

```
* (explain-slot mytext :top)
#k<MYTEXT>'s :TOP is #k<F2449> (20 . T),
which depends upon:
   #k<MYRECT>'s :TOP is #k<F2439> (20 . T),
   which depends upon:
      #k<MYRECT>'s :BOX is (80 20 100 150)
--> NIL
```

When explain-slot is too verbose, a non-recursive version can be used:

gd:Explain-Short *object* *slot*[function], page 90

For example:

```
* (explain-short mytext :top)
#k<MYTEXT>'s :TOP is #k<F2449> (20 . T),
which depends upon:
   #k<MYRECT>'s :TOP is #k<|1803-2439|> (20 . T),
      ...
--> NIL
```

Warning: explain-slot and explain-short may produce incorrect results in the following ways:

Both explain-slot and explain-short rely on dependency pointers maintained for internal use by KR. In the present version, KR sometimes leaves dependencies around that are no longer current. This is not a bug because, at worst, extra dependencies only cause formulas to be reevaluated unnecessarily. However, this may cause explain-slot or explain-short to print extra dependencies.

Formulas may access slots but not use the values. This will create the appearance of a dependency when none actually exists.

Formulas that that try to follow a null link,(gv :self :feedback-obj :top) where :feedback-obj is nil, may be marked as invalid and have their dependency lists cleared. explain-slot and explain-short will detect this case and warn you if it happens.

9.11 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.

9.11.1 Suggest-Constants

`gd:Record-From-Now`[function], page 90

`gd:Suggest-Constants` *object* `&key` *max* (*recompute-p* T) (*level* 1)[function],█
page 90

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, declar-
ing 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 ac-
curate 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
> slots which would, by themselves, eliminate some formula. If *level* is made
> higher, slots will be printed that may not eliminate formulas by themselves,
> but will at least eliminate some dependencies from the formulas that remain.

> For example, consider a formula that depends on slots A and B. Declaring
> constant either A or B alone would not eliminate the formula, so with *level*
> set to 1, slots A and B would not be suggested by `suggest-constants`. Set-
> ting *level* to 2, however, will print both A and B, since the combination of
> the two slots would indeed eliminate a formula. Higher values of *level* make
> `suggest-constants` print out formulas that are less and less likely to eliminate
> formulas.

9.12 Explain-Formulas and Find-Formulas

> gd:Explain-Formulas *aggregate* &optional (*limit* 50) *eliminate-useless-p*[function],∎
> page 90

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],∎
> page 90

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.

9.13 Count-Formulas and Why-Not-Constant

> gd:Count-Formulas *object*[No value for ''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*[No value for ''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.

9.14 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 [inspectorsec], page 573) 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], page 90

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
(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], page 90

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], page 90

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], page 90

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.

9.15 Opal Update Failures

Opal assumes that graphical objects have valid display parameters such as `:top` or `:width`. If a parameter is computed by formula and there is a bug, the problem will often cause an error within Opal's `update` function.

A "quarantine slot" named `:in-progress` 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. The quarantine slot is discussed in detail in the Opal Chapter.

There are several ways to proceed after an update failure. The first and easiest action is to run `opal:update` with the optional parameter `t`:

    (opal:update window t)

This forces `opal:update` to do a complete update of `window` as opposed to an incremental update. This may fix your problem by bringing all slots up-to-date and expunging previous display parameters.

Another possibility is, after entering the debugger, call

    gd:Explain-NIL[function], page 90

This function will check to see if a formula tried to follow a null link (a typical cause of Opal object slots becoming `nil`). If so, the object and slot associated with the formula will be printed followed by objects and slots on which the formula depends[2]. One of the slots depended upon will be the null link that caused the formula to fail.

Warning: `explain-nil` will always attempt to describe the last formula that failed due to a null link *since the last time* `explain-nil` *was evaluated*. This may or may not be relevant to the bug you are searching for. The last error is cleared every time `explain-nil` is evaluated to reduce confusion over old errors. If there has been no failure, `explain-nil` will print

    No errors in formula evaluation detected

A third possibility is to run

`gd:fix-up-window` *window*   [Function]
   on the window in question. (You may want to use `windows` to find the window object.) `fix-up-window` will do type checking without attempting a redisplay. If an error is detected, `fix-up-window` will allow you to interactively remove objects with problems from the window.

   After fixing the problem that caused `update` to crash, you should be able to do a successful total update on the window (discussed above). A successful total update will clear the quarantine slot, and will allow interactions to take place in the window normally.

9.16 Inspecting Interactors

9.17 Tracing

A common problem is to create some graphical objects and an interactor but to discover that nothing happens when you try to interact with the program. If you know what interactor is not functioning, then you can trace its behavior using the function

    inter:Trace-Inter *interactor*[function], page 90

---

[2] `Explain-nil` does not use the same technique for finding dependencies as `explain-slot`, which uses forward pointers from the formula's `:depends-on` slot. Since `:depends-on` is currently cleared when a null link is encountered, `explain-nil` uses back pointers from the objects back to the formula. These are in the `:depended-upon` slot of objects. To locate the back pointers, `explain-nil` searches for all components of all Opal windows. Only objects in windows are searched, so dependencies on non-graphical objects will be missed.

This function enables some debugging printouts in the interactors package that should help you determine what is wrong. A set of things to trace is maintained internally, so you can call `inter:trace-inter` several times to trace several things. In addition to interactors, the parameter can be one of:

   `t` — trace everything.

   `NIL` — untrace everything, same as calling `inter:untrace-inter`.

   `:window` — trace things about interactor windows such as `create` and `destroy` operations.

   `:priority-level` — trace changes to priority levels.

   `:mouse` — trace `set-interested-in-moved` and `ungrab-mouse`.

   `:event` — show all events that come in.

   `:next` — start tracing when the next interactor runs, and trace that interactor.

   `:short` — report only the name of the interactor that runs, so that the output is much less verbose. This is very useful if you suspect that more than one interactor is accidentally running at a time.

Tracing any interactor will turn on `:event` tracing by default. Call (`inter:untrace-inter :event`) (see below) to stop `:event` tracing.

Just typing

   `(inter:trace-inter)`

will print out the interactors currently being traced.

   `inter:Untrace-Inter` *interactor*[function], page 90

can be used to selectively stop tracing a single interactor or other category. You can also pass `t` or `nil`, or no argument to `untrace` to stop all tracing:

   `(inter:untrace-inter)`

9.18 Describing Interactors

If you are not debugging a particular interactor, there are a few ways to proceed other than wading through a complete interactor trace. First, you can find out what interactors are active by calling:

   `gd:Look-Inter` &optional *interactor-or-object detail*[function], page 90

The parameter *interactor-or-object* can be:

   NIL to list all active interactors (see below),

   an interactor to describe,

   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

With no arguments (or `nil` as an argument), `look-inter` will print all active interactors (those with their `:active` and `:window` slots set to something) sorted by priority level

   `* (look-inter)`

```
Interactors that are :ACTIVE and have a :WINDOW are:
Level #k<RUNNING-PRIORITY-LEVEL>:
Level #k<HIGH-PRIORITY-LEVEL>: #k<DEMO-GROW::INTER2>
Level #k<NORMAL-PRIORITY-LEVEL>: #k<MYTYPER> #k<MYMOVER> #k<DEMO-GROW::INTER3>■
 #k<DEMO-GROW::INTER4> #k<DEMO-GROW::INTER1>
--> NIL
```

If *detail* is 1, `look-inter` will show the `:start-event` and `:start-where` of each active interactor:

```
* (look-inter 1)
Interactors that are :ACTIVE and have a :WINDOW are:
Level #k<RUNNING-PRIORITY-LEVEL>:
Level #k<HIGH-PRIORITY-LEVEL>: #k<DEMO-GROW::INTER2>
Level #k<NORMAL-PRIORITY-LEVEL>: #k<MYTYPER> #k<MYMOVER> #k<DEMO-GROW::INTER3>■
 #k<DEMO-GROW::INTER4> #k<DEMO-GROW::INTER1>
#k<DEMO-GROW::INTER2> (MOVE-GROW-INTERACTOR)
    starts when :LEFTDOWN (:ELEMENT-OF #k<AGGREGATE-164>)
#k<MYTYPER> (TEXT-INTERACTOR)
    starts when :RIGHTDOWN (:IN #k<MYTEXT>)
#k<MYMOVER> (MOVE-GROW-INTERACTOR)
    starts when :LEFTDOWN (:IN #k<MYRECT>)
#k<DEMO-GROW::INTER3> (MOVE-GROW-INTERACTOR)
    starts when :MIDDLEDOWN (:ELEMENT-OF #k<AGGREGATE-136>)
#k<DEMO-GROW::INTER4> (MOVE-GROW-INTERACTOR)
    starts when :RIGHTDOWN (:ELEMENT-OF #k<AGGREGATE-136>)
#k<DEMO-GROW::INTER1> (BUTTON-INTERACTOR)
    starts when :LEFTDOWN (:ELEMENT-OF-OR-NONE #k<AGGREGATE-136>)
--> NIL
```

To get information about a single interactor, pass the interactor as a parameter:

```
* (look-inter mymover)
#k<MYMOVER>'s :ACTIVE is T, :WINDOW is #k<MYWINDOW>
#k<MYMOVER> is on the #k<NORMAL-PRIORITY-LEVEL> level
#k<MYMOVER> (MOVE-GROW-INTERACTOR)
    starts when :LEFTDOWN (:IN #k<MYRECT>)
--> NIL
```

In some cases you need to know what interactor will affect a given object (perhaps located using the `ident` function). This is not possible in general since the object(s) an interactor changes may be referenced by arbitrary application code. However, if you use interactors in fairly generic ways, you can call `look-inter` with a graphical object as argument to search for relevant interactors:

```
* (look-inter myrect)
#k<MYMOVER>'s :start-where is (:IN #k<MYRECT>)
--> NIL
* (look-inter mytext)
#k<MYTYPER>'s :start-where is (:IN #k<MYTEXT>)
--> NIL
```

The search algorithm used by look-inter is fairly simple: the current value of
`:start-where` is interpreted to see if it could refer to the argument. Then the
`:feedback-obj` and `:obj-to-change` slots are examined for an exact match with the
argument. If formulas are encountered, only the current value is considered, so there are a
number of ways in which `look-inter` can fail to find an interactor.

9.19 Sizes of Objects

Several functions are provided to help make size measurements of Opal objects and aggre-
gates.

> `gd:ObjBytes` *object*`[function], page 90`

will measure the size of a single Opal object or interactor in bytes.

> `gd:AggBytes` *aggregate* `&optional` *verbose*`[function], page 90`

will measure the size of an Opal aggregate and all of its components in bytes. The first
argument may also be a list of aggregates, a window, or a list of windows. For example, to
compute the total size of all graphical objects, you can type this:

> `(aggbytes (windows))`

The output will include various statistics on size according to object type. Sizes are
printed in bytes, and the returned value will be the total size in bytes. The size information
does not include any interactors because interactors can exist independent of the aggregate
hierarchy. The optional *verbose* flag defaults to `t`; setting it to `nil` will reduce the detail of
the printed information.

> `gd:InterBytes` `&interactor` *window* *verbose*`[function], page 90`

will report size information on the interactors whose `:window` slot *currently* contains
the specified window. If the *window* parameter is omitted, `t` or `nil`, then the size of all
interactors is computed. (Use `objsize` for a single interactor.) Note that an interactor may
operate in more than one window and that interactors can follow objects from window to
window. As with `aggbytes`, the *verbose* flag defaults to `t`; setting it to `nil` will reduce the
detail of the printed information.

> `gd:*Avoid-Shared-Values*`⟨undefined⟩ `[variable], page` ⟨undefined⟩

Normally, `aggbytes` does not consider the fact that list structures may be shared, so
shared storage is counted multiple times. To avoid this (at the expense of using a large
hash table), set `avoid-shared-values` to `t`.

> `gd:*Avoid-Equal-Values*`⟨undefined⟩ `[variable], page` ⟨undefined⟩

To measure the potential for sharing, set this variable to `t`. This will do hashing using
`#'equal` so that equal values will be counted as shared instead of `#'eq`, which measures
actual sharing.

> `gd:*Count-Symbols*`⟨undefined⟩ `[variable], page` ⟨undefined⟩

Ordinarily, storage for object names is not counted as part of the storage for objects.
By setting this variable for true, the sizes reported by `objbytes` and `aggbytes` will include
this additional symbol storage overhead.

Note: Size information for an object includes the size of any attached formulas. At
present, only objects and cons cells are counted. Storage for structures (other than KR
schema), strings, and arrays is *not* counted.

⟨undefined⟩ [References], page ⟨undefined⟩,

10 Demonstration Programs for Garnet

by Brad A. Myers, Andrew Mickish

10 May 2020

10.1 Abstract

This file contains an overview of the demonstration programs distributed with the Garnet toolkit. These programs serve as examples of what Garnet can do, and also of how to write Garnet programs.

10.2 Introduction

Probably the best way to learn about how to code using the Garnet Toolkit is to look at example programs. Therefore, we have provided a number of them with the Toolkit release. In addition, you can load and run the demos to see what kinds of things Garnet can do.

The "best" example program is `demo-editor`, which is included in this technical report. The other example programs serve mainly to show how particular special features of Garnet can be used.

Unfortunately, many of the demonstration programs were implemented before important parts of the Garnet Toolkit were implemented. For example, many of the demos do not use Aggregadgets and Aggrelists. These particular demos are *not* good examples of how we would code today. Hopefully, we will soon re-code all of these old demos using the newest features, but for the time being, you will probably only want to look at the code of the newer demos.

This document provides a guide to the demo programs, what they are supposed to show, and whether they are written with the latest style or not.

10.3 Loading and Compiling Demos

If for some reason the demos were not compiled during the standard installation procedure discussed in the Overview chapter, you can compile just the demos by executing (garnet-load "demos-src:demos-compiler"). This will generate new binaries for the demos, which will need to be copied from the `src/demos/` directory into your `bin/demos` directory.

Normally, the demonstration programs are *not* loaded by the standard Garnet loader. The best way to view the demos is to load the `garnet-loader` as usual and then load the Demos Controller:

```
(garnet-load "demos:demos-controller")
(demos-controller:do-go)
```

This will load the controller itself, but not any of the demos. It will display a window with a set of check buttons in it. Just click with the mouse on a button, and the corresponding demo will be loaded and started. Clicking on the check box again will stop the demo. Clicking again will restart it (but not re-load it). An instruction window will appear at the bottom of the screen with the instructions for the last demo started.

The `demos-controller` application features the `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. For more information about this gadget, see the appropriate section of the Gadgets Chapter.

Using the `demos-controller` causes each demo file to be loaded as it is needed. If you wanted to load *all* of the demos at once (whether you eventually planned to use the `demos-controller` or not), you could set `user::load-demos-p` to be T before loading `garnet-loader`, or execute `load Garnet-Demos-Loader`.

All of the demos described here are in the sub-directory `demos`.

10.4 Running Demo Programs

To see a particular demo program, it is not necessary to use the Demos Controller described in section [loadingandcompilingdemos], page 593. Instead, the file can be loaded and executed by itself.

Almost all of the demonstration programs operate the same way. Once a file `demo-xxx` is loaded, it creates a package called `demo-xxx`. In this package are two procedures – `do-go` to start the demo and `do-stop` to stop it. Therefore, to begin a demo of *xxx*, you would type: `(demo-xxx:do-go)`. The `do-stop` procedure destroys the window that the demo is running in. You can load and start as many demos as you like at the same time. Each will run in its own separate window.

The `do-go` procedure will print instructions in the Lisp window about how to operate the demonstration program.

Demos for the individual gadgets are all in the `garnet-gadgets` package and have unique names. Section [gadgetdemos], page 598, describes how to see these demos.

10.5 Double-Buffered Windows

All the demos can take advantage of the Opal feature for double-buffered windows. The `do-go` routine for each demo has an optional `:double-buffered-p` argument that defaults to `nil`. For instance, to run `demo-3d` on a double-buffered window, say: `(demo-3d:do-go :double-buffered-p T)` and to run it normally, say: `(demo-3d:do-go)`

10.6 Best Examples

10.6.1 GarnetDraw

There a useful utility called `GarnetDraw` which is a relatively simple drawing program written using Garnet. Since the file format for storing the created objects is simply a Lisp file which creates aggregadgets, you might be able to use GarnetDraw to prototype application objects (but Lapidary is probably better for this).

GarnetDraw uses many features of Garnet including gridding, PostScript printing, selection of all objects in a region, moving and growing of multiple objects, menubars, and the `save-gadget` and `load-gadget` dialog boxes. The editing functions like Cut, Copy, and Paste are implemented using the `Standard-Edit` module from `garnet-gadgets`, and objects can be cut and pasted between `GarnetDraw` and `Gilt` (since they share the same clipboard). Accelerators are defined for the menubar commands, like `META-x` for Cut and `META-v` for Paste.

GarnetDraw works like most Garnet programs: select in the palette with any button, draw in the main window with the right button, and select objects with the left button. Select multiple objects with shift-left or the middle mouse button. Change the size of objects by pressing on black handles and move them by pressing on white handles. The line style and color and filling color can be changed for the selected object and for further drawing by clicking on the icons at the bottom of the palette. You can also edit the shape of polylines: create a polyline, select it, and choose "Reshape" from the "Edit" menu.

10.6.2 Demo-Editor

Probably the best example program is the sample graphics editor in the file `demo-editor.lisp`. It demonstrates many of the basic components when building a Garnet application. This demo automatically loads and uses the `text-button-panel`, `graphics-selection`, and `arrow-line` gadgets.

10.6.3 Demo-Arith

`Demo-arith` is a simple visual programming interface for constructing arithmetic expressions. It uses constraints to solve the expressions. There are buttons for producing Post-Script output from the picture. Also, you can create new objects using gestures by dragging with the middle mouse button (rather than selecting them from the palette). The instructions are printed when the program is started.

10.6.4 Demo-Grow

`Demo-grow` shows how to use the `graphics-selection` gadget. It uses the same techniques as in `demo-editor` (section [demoeditor], page 595).

10.6.5 Multifont and Multi-Line Text Input

`Demo-text` shows how multi-line, multi-font text input can be handled. It does not use Aggregadgets or any gadgets, but none are necessary.

10.6.6 Demo-Multifont

To see how to effectively use the multifont text object, along with its interactors, examine the `demo-multifont` demo. Most of the code is actually a good demonstration of how to use the `menubar` and `motif-scrolling-window-with-bars` gadgets, but the `multifont-text` objects and interactors are in there. Features demonstrated include word wrap and how to changing the fonts with the special multifont accelerators.

The `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.

10.6.7 Creating New Objects

`Demo-twop` shows how new lines and new rectangles can be input. It uses the same techniques as in `demo-editor` (section [demoeditor], page 595).

10.6.8 Angles

There are two programs that demonstrate how to use the angle interactor. `Demo-angle` contains circular gauges (but see the `gauge` gadget, section [gadgetdemos], page 598), as well as a demonstration of how to use the "angle-increment" parameter to the angle `:running-action` procedure.

Demo-clock shows a clock face with hands that can be rotated with the mouse.

10.6.9 Aggregraphs

The `demo-graph` file is an example of many features of Aggregraphs.

10.6.10 Scroll Bars

Although sliders and scroll bars are provided in the Garnet Gadget set (the `gadgets` subdirectory), the file `demo-scrollbar` contains some alternative scroll bar objects. The Macintosh scroll bar in this demo was written in the old Garnet style, but there are new versions of scroll bars in the OpenLook, Next, and Motif style.

To see the demo of all four scroll bars, use the functions `demo-scrollbar:do-go` and `demo-scrollbar:do-stop` as usual. There are also functions that display the scroll bars individually called `mac-go`, `open-go`, `next-go`, and `motif-go`.

10.6.11 Menus

Demo-menu shows a number of different kinds of menus that can be created using Garnet. All of them were implemented using Aggregadgets and Aggrelists.

10.6.12 Animation

Demo-animator uses background animation processes to move several objects in a window. One of the objects is a walking figure which moves across the screen by rapidly redrawing a pixmap.

Demo-fade shows a simple animation for the Garnet acronym.

Demo-logo performs the same animation as `demo-fade`, but it also includes the Garnet logo.

10.6.13 Garnet-Calculator

The `garnet-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 demo with (`garnet-load "demos:garnet-calculator"`). To run it, execute (`garnet-calculator:do-go`).

garnet-calculator:Start-Calc &key *double-buffered-p* [function], page 90

garnet-calculator:Stop-Calc *app-object* &optional (*destroy-app-object?* T) [function], page 90

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`.

10.6.14 Browsers

The files `demo-schema-browser` and `demo-file-browser` show two uses of the `browser-gadget`.

10.6.15 Demo-Virtual-Agg

To show off an example of virtual-aggregates, load Demo-Virtual-Agg and say:

      (demo-virtual-agg:do-go :num-dots 1000)

`Demo-virtual-agg:do-go` takes a single optional keyed parameter `:num-dots` which tells how many circles should appear in a window. The default is 1000.

The first 1000 circles are read in from circles.data in the `user::Garnet-DataFile-PathName` directory (because that's faster) and the rest are chosen randomly. A '.' is printed out for every ten circles.

You will also see a little star in the upper left on the screen, in front of the `virtual-aggregate`, and a big gray rectangle underneath the `virtual-aggregate`. These are just to show that the update algorithm is working reasonably well.

> Clicking with the left button creates a new circle (of random radius and color) where you clicked.
>
> Clicking with the right button "destroys" the top-most circle underneath where you clicked, or beeps if there was nothing under there.
>
> Clicking on the little star and dragging moves the little star.
>
> Clicking shift-middle causes the circle underneath the cursor to change to a different random color. (This shows off `change-item`.)
>
> Clicking shift-right causes the entire `virtual-aggregate` to disappear or reappear.

10.6.16 Demo-Pixmap

This new demo shows a two-dimensional `virtual-aggregate` in action. Here, the `virtual-aggregate` is a 50 X 50 array of 5 X 5 rectangles. Each rectangle can be colored from the color palette, and the pattern of colored rectangles is reflected in a pixmap.

You can load a pixmap into the demo (e.g., from the directory `Garnet-Pixmap-PathName`), edit the pixmap with the color palette and virtual-aggregate, and then save the pixmap to a new file. You can also generate PostScript files from this demo, though you have to have a Level 2 printer (that defines the PostScript function `colorimage`) to print a color pixmap image.

10.6.17 Demo-Gesture

`Demo-gesture` is an example of how the new gesture-interactor can be used in an interface. In this demo, you can create perfect circles and rectangles by drawing rough approximations with the mouse, which are interpreted by the gesture recognizer. Gestures may also be used to copy and delete the shapes you have created.

10.6.18 Demo-Unidraw

`Demo-Unidraw` is a gesture-based text editor, which allows you to enter characters with freehand drawing using the mouse. The gestures that this demo understands are comprised

of a shorthand alphabet devised by David Goldberg at Xerox Parc. The gesture patterns are shown in the middle of the demo window, and the canvas for drawing gestures is at the bottom. As the demo recognizes the gestures you draw, it selects the corresponding gesture and puts the new character in the text window.

10.6.19 Gadget Demos

There are separate demo programs of some of the gadgets in the files `demo-gadgets` and `demo-motif`. Each of these packages export the usual `do-go` and `do-stop` procedures, and can be found in the `demos` directory.

Other good examples are the Garnet Gadgets, stored in the `gadgets` sub-directory. These were *all* written using the latest Garnet features. At the end of almost all gadget files is a small demo program showing how to use that gadget. Since all the gadgets are in the same package (`garnet-gadgets`), the gadget demos all have different names. They are:

`Arrow-line-go, Arrow-line-stop` - to demonstrate arrow-lines

`Error-gadget-go, Error-gadget-stop` - to demonstrate both the error gadget and the query gadget

`Gauge-go, Gauge-stop` - to demonstrate circular gauges

`H-scroll-go, H-scroll-stop` - to demonstrate standard horizontal scroll bars

`H-slider-go, H-slider-stop` - to demonstrate standard horizontal sliders

`Labeled-box-go, Labeled-box-stop` - to demonstrate labeled text-type-in objects

`Menu-go, Menu-stop` - to demonstrate a standard menu

`Menubar-go, Menubar-stop` - to demonstrate pull-down menus

`Motif-Check-Buttons-go, Motif-Check-Buttons-stop` - to demonstrate Motif style check buttons

`Motif-Error-Gadget-go, Motif-Error-Gadget-stop` - to demonstrate both the motif error gadget and the motif query gadget

`Motif-Gauge-go, Motif-Gauge-stop` - to demonstrate the Motif style gauge

`Motif-H-Scroll-go, Motif-H-Scroll-stop` - to demonstrate Motif style horizontal scroll bars

`Motif-Menu-go, Motif-Menu-stop` - to demonstrate the Motif style menus

`Motif-Menubar-go, Motif-Menubar-stop` - to demonstrate the Motif style menubar, with accelerators

`Motif-Option-Button-go, Motif-Option-Button-stop` - to demonstrate the Motif style version of this popup menu gadget, whose button changes labels according to the menu selection

`Motif-Radio-Buttons-go, Motif-Radio-Buttons-stop` - to demonstrate Motif style radio buttons

`Motif-Scrolling-Labeled-Box-go, Motif-Scrolling-Labeled-Box-stop` - to demonstrate the Motif style text-type-in field

`Motif-Scrolling-Window-With-Bars-go, Motif-Scrolling-Window-With-Bars-stop` - to demonstrate the Motif style scrolling window gadget

`Motif-Slider-go, Motif-Slider-stop` - to demonstrate the vertical Motif slider

`Motif-Text-Buttons-go, Motif-Text-Buttons-stop` - to demonstrate Motif style text buttons

`Motif-Trill-go, Motif-Trill-stop` - to demonstrate the Motif style trill device

`Motif-V-Scroll-go, Motif-V-Scroll-stop` - to demonstrate the Motif vertical scroll bar

`Mouseline-go, Mouseline-stop` - to demonstrate the mouseline and "balloon help" string

`Multifont-Gadget-go, Multifont-Gadget-stop` - to demonstrate the gadget which is a conglomeration of a multifont-text, a focus-multifont-textinter, and a selection-interactor

`Option-Button-go, Option-Button-stop` - to demonstrate this kind of popup menu gadget, whose button label changes according to the menu selection

`Popup-Menu-Button-go, Popup-Menu-Button-stop` - to demonstrate this kind of popup menu gadget, whose button label is fixed and may be a bitmap or other object

`Prop-Sheet-For-Obj-go, Prop-Sheet-For-Obj-stop` - to demonstrate how prop-sheets can be used to change slot values of Garnet objects

`Radio-Buttons-go, Radio-Buttons-stop` - to demonstrate radio buttons

`Scrolling-Input-String-go, Scrolling-Input-String-stop` - to demonstrate the scrolling input string gadget

`Scrolling-Labeled-Box-go, Scrolling-Labeled-Box-stop` - to demonstrate the standard scrolling labeled box

`Scrolling-Menu-go, Scrolling-Menu-stop` - to demonstrate the scrolling menu gadget

`Scrolling-Window-go, Scrolling-Window-stop` - to demonstrate the standard scrolling window

`Scrolling-Window-With-Bars-go, Scrolling-Window-With-Bars-stop` - to demonstrate the scrolling window with attached vertical and horizontal scroll bars

`Text-Buttons-go, Text-Buttons-stop` - to demonstrate buttons with labels inside

`Trill-go, Trill-stop` - to demonstrate the trill-device gadget

`V-scroll-go, V-scroll-stop` - to demonstrate standard vertical scroll bars

`V-slider-go, V-slider-stop` - to demonstrate standard vertical sliders

`X-Buttons-go, X-Buttons-stop` - to demonstrate X buttons

Each of these has its own loader file, named something like *xxx*-`loader` for gadget *xxx*. See the Gadgets chapter for a table of loader file names.

10.6.20 Real-Time Constraints and Performance

The program `demo-manyobjs` was written as a test of how fast the system can evaluate constraints. The `do-go` procedure takes an optional parameter of how many boxes to create. Each box is composed of four Opal objects.

10.7 Old Demos

10.7.1 Moving and Growing Objects

The best example of moving and growing objects is `demo-grow` (section [demogrow], page 595).

In addition, `demo-moveline` shows how the `move-grow-interactor` can be used to move either end of a line.

10.7.2 Menus

`Demo-3d` shows some menus and buttons where the item itself moves when the user presses over it, in order to simulate a floating button.

10.8 Demos of Advanced Features

10.8.1 Using Multiple Windows

`Demo-multiwin` shows how an interactor can be used to move objects from one window to another. For more information, see the Interactors chapter.

10.8.2 Modes

`Demo-mode` shows how you can use the `:active` slot of an interactor to implement different modes. For more information, see the Interactors chapter.

10.8.3 Using Start-Interactor

`Demo-sequence` shows how to use the `inter:start-interactor` function to have one interactor start another interactor without waiting for the second one's start event. Another example of the use of `inter:start-interactor` is in `demo-editor` (section [demoeditor], page 595) to start editing the text label after drawing a box. For more information on `start-interactor`, see the Interactors chapter.

11 A Sample Garnet Program

by Brad A. Myers

10 May 2020

11.1 Abstract

This file contains a sample program written using the Garnet Toolkit. The program is a simple graphical editor that allows the user to create boxes and arrows.

11.2 Introduction

The program in this file is implemented using the Garnet Toolkit, and is presented as an example of how to write programs using the toolkit. The program implements a graphical editor that allows the user to create boxes with textual labels which can be connected by lines. The lines have arrowheads, and go from the center of one box to the center of another. The boxes can be moved or changed size, and the arrows stay attached correctly. The boxes or lines can also be deleted, and the labels can be edited.

The sample program is in the file `demo-editor.lisp`, and a source and binary (compiled) version should be available in the `demos` sub-directory of the Garnet system files.

This graphical editor shows the use of:

Constraints: to keep the arrows centered, to keep the name labels at the tops of boxes, etc.

Opal objects: roundtangles, cursor-text, windows.

Interactors: to choose which drawing mode (`menu-interactor`), to edit the text strings (two `text-interactor`s), and to create new objects (`two-point-interactor`).

Toolkit widgets: `Text-button-panel` (a form of menu), `graphics-selection` (to show which object is selected and allow it to be moved), and `arrow-line`s. These widgets have built in Opal objects and Interactors.

Aggregadgets: to group the roundtangle and label string.

Creating instances from prototypes (creating the new boxes and arrows).

This code is about 365 lines long, including comments, and took me two hours to code and one hour to debug. I did not use any higher-level Garnet tools to create it (it was all coded directly in Lisp).

11.3 Loading the Editor

After loading `Garnet-loader`, either of the following commands will load the editor:

```
(garnet-load "demos:demo-editor")        ; To load the compiled version
```

or

```
(garnet-load "demos-src:demo-editor")    ; To load the interpreted ver-
                                         sion, which

                                         ; may make experimenting/debugging easier
```

11.4 User Interface

A snap shot of the editor in use is shown in Figure [SampleFig], page 602.
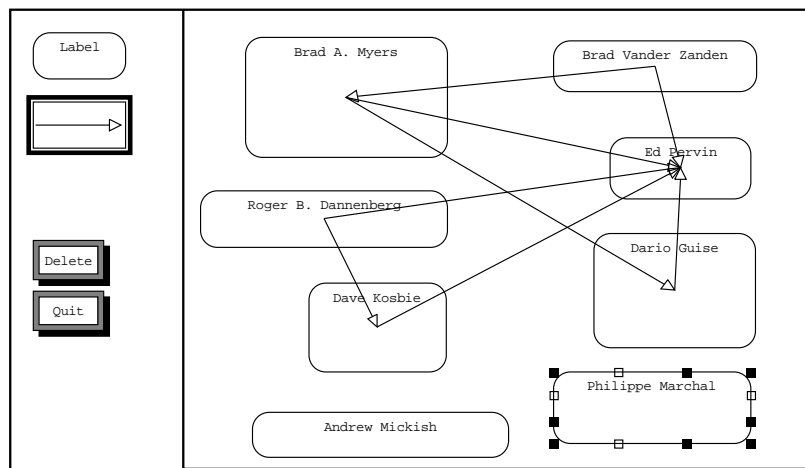
Figure 11.1: A Sample Garnet Application. The code for this application is listed at the end of this technical report.

The user interface is as follows. The menu at the top determines the current mode. When the roundtangle is outlined, the user can draw new boxes, and when the arrow is outlined, the user can draw new arrows. Press with either the left or right mouse buttons over one to change modes.

To create a new roundtangle, press with the *right* button in the workspace window (on the right) and hold down. Drag to the desired size and then release. Next, type in the new name. Various editing characters are supported during typing, including:

> `^h, delete, backspace`: delete previous character.
>
> `^d`: delete next character.
>
> `^u`: delete entire string.
>
> `^b, left-arrow`: go back one character.
>
> `^f, right-arrow`: go forward one character.
>
> `^a, home`: go to the beginning of the string.
>
> `^e, end`: go to the end of the string.
>
> `^y, insert`: insert the contents of the X cut buffer into the string at the current point.

When finished typing, press RETURN or any mouse button to stop.

To create a new arrow, when in arrow mode, press with the *right* button over a roundtangle and release over a different roundtangle. An arrow will be drawn starting at the center of the first roundtangle and going to the center of the other one.

Press with the *left* button on a roundtangle or an arrow to select it. Press on the background to cause there to be no selection. Press and release on the "Delete" button to delete the selected object.

If a roundtangle is selected, you can move it by pressing on a small white square with the left button and dragging to the new position and releasing. You can change its size by pressing with the left button on a black square. While the dotted outline box is displayed, you can abort the operation by moving outside the window and releasing, or by hitting ^G (control-G).

To change the string of a label, press on the label with the left button and begin typing. When finished typing, press RETURN or any mouse button to stop, or press ^G to abort the editing and return to the original string.

11.5 Overview of How the Code Works

The next section contains the actual code for the demo editor. This section presents some the parts of the design and serves as a guide to the code.

The standard "Garnet style" is to `USE-PACKAGE` the `KR` package, and directly reference all the other Garnet packages, so this is how the code is written. Functions such as `create-instance, s-value, o-formula, formula, gv,` and `gvl` are defined in `KR`.

The first part of the code creates *prototypes* of the basic items that the user will create: arrow lines and labeled boxes. When the editor is running, the code will create an *instance* of one of these prototypes to get a new set of objects to be displayed. The arrow line object is composed of one `arrow-line` from the Garnet-Gadget set, with some special *constraints* on

its end-points. The arrow-line is parameterized by the two objects it is connected to. These two objects are kept in slots of the arrow-line: `:from-obj` and `:to-obj`. The constraints on the end-points of the arrow-line are expressed as *formulas* that cause the arrow to go from the center of the object stored in the `:from-obj` slot of the arrowline, to the center of the object in the `:to-obj` slot.

The labeled box is more complicated. It is composed of two parts: a rounded-rectangle ("roundtangle") and a label. An *AggreGadget* is used to compose these together. The boundaries of the roundtangle are defined by the values in the `:box` slot, since the standard `Move-Grow-Interactor` modifies objects by setting this slot. The label string is constrained to be centered at the top of the roundtangle. The actual string used is stored both at the top level AggreGadget, and in the text object, so formulas are set up to keep these two `:string` slots having the same value.

The next function (`create-mode-menu`) creates the top menu that contains a label object and an arrow object. A feedback rectangle is also created to show what the current mode is. This feedback rectangle has formulas that keep it over whatever mode object is selected. An interactor is then created to allow the user to choose the mode.

The main command menu is created using `create-menu`, which simply creates an instance of a `garnet-gadget:text-button-panel` in the correct place. The functions to be executed are `delete-object` and `do-quit`, and these are defined next. The only trick here is that if a labeled box is deleted, the lines to it are also deleted. For quit, destroying a window automatically destroys all of its contents.

Creating a new object is fairly straightforward. The interactor is queried to find out whether to create a line or a box, and the appropriate kind of object is then created. Lines can only start or end in boxes, so the appropriate boxes are found. To appear in a window, the newly created objects must be added to an *aggregate* which is attached to the window, Here, the aggregate found by looking in the `:objs-aggregate` slot of the interactor.

Another important feature of the `Create-New-Obj` procedure is that if the object being created is a box, then it starts an interactor to allow the user to type the text label.

The top-level, exported procedure, `do-go`, starts everything up by creating a window, a sub-window to be the work area, and top-level aggregates for both windows. Another aggregate will hold the user-created objects. The `selection` object will show which object is selected, and also allow that object to be moved or grown (if it is not a line). Two text editing interactors are then created. One is used when a new object is created to have the user type in the initial name. This one is started explicitly using `Start-interactor` in `Create-New-Object`. The other interactor is used when the user presses on the text label of an object to edit the name.

Finally, the interactor to create new objects is defined. This one is a little complex, because it needs to decide whether to use a line or rectangle feedback based on the current mode.

The last step is to add the top level aggregates to the windows and call `update` to get the objects to appear. If you are running Allegro, Lucid, Lispworks, CMUCL, or MCL Common Lisp, they will begin operating by themselves, but under other Common Lisps, the `Main-Event-Loop` call is needed to get the interactors to run. The `Exit-Main-Event-Loop` function in `Do-Quit` causes `Main-Event-Loop` to exit.

11.6 The Code

```
;;; [-*- Mode: LISP; Syntax: Common-Lisp; Package: DEMO-EDITOR; Base: 10 -*-]█
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; [         The Garnet User Interface Development Environment.      ]
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; [This code was written as part of the Garnet project at          ]
;;; [Carnegie Mellon University, and has been placed in the public   ]
;;; [domain.  If you are using this code or any part of Garnet,      ]
;;; [please contact garnet@cs.cmu.edu to be put on the mailing list. ]
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;This file is a sample of a graphics editor created with Garnet.  It is█
;;;designed to be a model for other code development, and therefore uses
;;;all the most up-to-date Garnet features.
;;;
;;;** Call (demo-editor:Do-Go) to start and (demo-editor:Do-Stop) to stop **█
;;;
;;;Designed and implemented by Brad A. Myers


(in-package :DEMO-EDITOR)

;;;  Load text-buttons-loader, graphics-loader, and arrow-line-loader unless█
;;;  already loaded
;;;
(dolist (pair '((:text-buttons "text-buttons-loader")
(:graphics-selection "graphics-loader")
(:arrow-line "arrow-line-loader")))
  (when (not (get :garnet-modules (car pair)))
    (user::garnet-load (concatenate 'string "gadgets:" (cadr pair)))))


;; Eliminate compile warnings for named objects
;;
(declaim (special MYARROWLINE MYLABELEDBOX))
```

```
;;;------------------------------------------------------------------------■
;;;First create the prototypes for the box and lines
;;;------------------------------------------------------------------------■


(create-instance 'myarrowline garnet-gadgets:arrow-line
  (:from-obj NIL) ;set this with the object this arrow is from
  (:to-obj NIL)   ;set this with the object this arrow is from
  (:x1 (o-formula (opal:gv-center-x (gvl :from-obj))))
  (:y1 (o-formula (opal:gv-center-y (gvl :from-obj))))
  (:x2 (o-formula (opal:gv-center-x (gvl :to-obj))))
  (:y2 (o-formula (opal:gv-center-y (gvl :to-obj))))
  (:open-p NIL)
  (:visible (o-formula (and (gvl :from-obj)(gvl :to-obj))))
  (:line-p T) ;so that the selection object will know what kind this is
  )

(create-instance 'mylabeledbox opal:aggregadget
  (:box (list 20 20 40 20)) ;this will be set by the
    ;interactors with the size of this box.

  (:lines-at-this-box NIL)  ;Keep track of lines pointing
    ;to me, in case I am deleted.

  ;;Set up a circular constraint between this string slot and the
  ;;string slot in the label.  If either is changed, the other is
  ;;automatically updated.  For circular constraints, it is
  ;;important to have an initial value, here it is the empty string.
  (:string (o-formula (gvl :label :string) ""))

  (:line-p NIL) ;so that the selection object will know what kind this is■
  (:parts
   '((:frame ,opal:roundtangle
     (:radius 15)
     (:left ,(o-formula (first (gvl :parent :box))))
     (:top ,(o-formula (second (gvl :parent :box))))
     (:width ,(o-formula (third (gvl :parent :box))))
     (:height ,(o-formula (fourth (gvl :parent :box)))))
     (:label ,opal:text
     (:string ,(o-formula (gvl :parent :string) ""))
     (:cursor-index NIL)
     ;;center me horizontally with respect to the frame
     (:left ,(o-formula (- (opal:gv-center-x (gvl :parent :frame))
   (floor (gvl :width) 2))))
     (:top ,(o-formula (+ (gvl :parent :frame :top) 5)))))))
```

```
;;;------------------------------------------------------------------------■
;;;Create main menu object
;;;------------------------------------------------------------------------■

;;Create an arrow and a box menu object, and put them in a menu, with an
;;interactor and feedback object to show which is selected.
;;Agg is the top level aggregate to put the menu in, and window is the window.■
;;The :line-p slot of the agg is set with a formula to tell whether in line mode■
;;or not.
(defun create-mode-menu (agg window)
  (let (feedback boxitem arrowitem)
    (setf boxitem (create-instance NIL mylabeledbox
    (:box (list 20 20 80 40))
    (:string "Label")))
    ;;the arrow will be inside a box.
    (setf arrowitem
  (create-instance NIL opal:aggregadget
    (:parts
     '((:frame ,opal:rectangle
       (:left 20)(:top 80)(:width 80)(:height 40))
       (:line ,garnet-gadgets:arrow-line
       (:open-p NIL)
       (:x1 ,(o-formula (+ (gvl :parent :frame :left) 2)))
       (:y1 ,(o-formula (opal:gv-center-y (gvl :parent :frame))))
       (:x2 ,(o-formula (+ (gvl :parent :frame :left) 76)))
       (:y2 ,(o-formula (gvl :y1)))))))))

    ;;The interactor (defined below) will set the :selected slot of the aggregate.■
    ;;Use this to determine where the feedback should be.
    ;;We need to use formula rather than o-formula here so we can have a direct■
    ;;reference to agg (use formula whenever you need to reference an ob-■
ject that
    ;;is not stored in a slot of the current object).  Notice the use of
    ;;back-quote and comma to get a reference to the agg object.
    (setf feedback (create-instance NIL opal:rectangle
     (:line-style opal:line-4)
     (:filling-style NIL)
     (:left (o-formula (- (gvl :parent :selected :left) 6)))
     (:top (o-formula (- (gvl :parent :selected :top) 6)))
     (:width (o-formula (+ (gvl :parent :selected :width) 12)))
     (:height (o-formula (+ (gvl :parent :selected :height) 12)))
     (:visible (o-formula (gvl :parent :selected)))
     (:draw-function :xor)
     (:fast-redraw-p T)))
    (opal:add-components agg boxitem arrowitem feedback)
```

```
;;use the :menuobjs slot to hold the items that can be selected
(s-value agg :menuobjs (list boxitem arrowitem))

;;default mode is the rectangle
(s-value agg :selected boxitem)

;;The :line-p slot of the agg is set with a formula to tell whether in line mode o
(s-value agg :line-p (o-formula (eq (gvl :selected) (second (gvl :menuobjs)))))█

;;now create an interactor to choose which mode
(create-instance NIL inter:menu-interactor
  (:window window)
  (:start-event '(:leftdown :rightdown)) ;either one
  (:start-where `(:list-element-of ,agg :menuobjs)))))
```

```
;;This creates the menu of commands.  For now, it only has "delete" and "quit" in it.█
;;The menu is stored into the aggregate agg.  Returns the menu created.
(defun create-menu (agg)
  (let ((menu (create-instance NIL Garnet-gadgets:Text-Button-Panel
(:constant T)
(:items '(("Delete" Delete-Object) ("Quit" Do-Quit)))
(:left 20)
(:top 200)
(:font Opal:Default-font)
(:shadow-offset 5)
(:final-feedback-p NIL))))
    (opal:add-components agg menu)
    menu))


;;;**********************************************************************
;;;Procedures to do the work
;;;**********************************************************************

;;;Delete-Line is called from delete object to delete lines
(defun Delete-Line(line-obj)
  (let ((from-obj (g-value line-obj :from-obj))
(to-obj (g-value line-obj :to-obj)))
    ;;remove this line from the boxes' lists
    (s-value from-obj :lines-at-this-box
     (delete line-obj (g-value from-obj :lines-at-this-box)))
    (s-value to-obj :lines-at-this-box
     (delete line-obj (g-value to-obj :lines-at-this-box)))
    (opal:destroy line-obj)))

;;;Delete-object is called from the main menu routine
(defun Delete-Object (toolkit-obj menu-item)
  (declare (ignore menu-item))
  (let ((selected-obj (g-value toolkit-obj :selection-obj :value)))
    (if selected-obj
      (progn
;;first turn off selection
(s-value (g-value toolkit-obj :selection-obj) :value NIL)
;;now delete object
(if (g-value selected-obj :line-p)
    ;;then deleting a line
    (Delete-Line selected-obj)
    ;;else deleting a box
    (progn
      ;;first delete all lines to this box
      (dolist (line-at-box (g-value selected-obj :lines-at-this-box))
(delete-line line-at-box))
```

```
      ;;now delete the box
      (opal:destroy selected-obj))))
;;else nothing selected
(inter:beep))))

(defun Do-Quit (toolkit-obj menu-item)
  (declare (ignore menu-item))
  (opal:destroy (g-value toolkit-obj :window))
  ;;for demo-controller
  (unless (and (fboundp 'User::Garnet-Note-Quitted)
       (User::Garnet-Note-Quitted "DEMO-EDITOR")))
)
```

```
;;;Create a new object.  Get the type of object to create from the interactor.█
;;;This procedure is called as the final-function of the two-point interactor.█
(defun Create-New-Obj (inter point-list)
  (let ((agg (g-value inter :objs-aggregate))
(line-p (g-value inter :line-p))) ;create a line or rectangle

    (if line-p
;;then create a line, first have to find the objects where the line is drawn█
(let ((from-box (opal:point-to-component agg (first point-list)
 (second point-list) :type mylabeledbox))
      (to-box (opal:point-to-component agg (third point-list)
       (fourth point-list) :type mylabeledbox))
      new-line)
  ;;If one end of the arrow is not inside a box, or is from and to the same box, then
  (if (or (null from-box)(null to-box) (eq from-box to-box))
      (inter:beep)
      ;; else draw the arrow.
      (progn
(setf new-line (create-instance NIL myarrowline
(:from-obj from-box)
(:to-obj to-box)))
;;keep track in case boxes are deleted so can delete this line.
(push new-line (g-value from-box :lines-at-this-box))
(push new-line (g-value to-box :lines-at-this-box))

(opal:add-component agg new-line))))
;;else, create a new box
(let ((textinter (g-value inter :textinter))
      (new-box (create-instance NIL mylabeledbox
 (:box (copy-list point-list))))) ;have to make a copy of list since
                  ;the interactor re-uses the same list
  (opal:add-component agg new-box)
  ;;now start the interactor to allow the user to type the label.
  ;;Obj-to-change is the label object of the new box.
  (s-value textinter :obj-to-change (g-value new-box :label))
  (inter:start-interactor textinter)))))

;;;
```

```
;;;**********************************************************************
;;;Main procedures
;;;**********************************************************************

(defparameter current-window NIL) ;this global variable is only used for the de-
bugging function below: do-stop

(defun Do-Go (&key dont-enter-main-event-loop double-buffered-p)
  (let (top-win work-win top-agg work-agg selection objs-agg menu edit-text)
    ;;;create top-level window
    (setf top-win (create-instance NIL inter:interactor-window
    (:left 20) (:top 45)
    (:double-buffered-p double-buffered-p)
    (:width 700) (:height 400)(:title "GARNET Sample Editor")
    (:icon-title "Graphics Editor")))
    (setf current-window top-win)

    ;;;create window for the work area
    (setf work-win (create-instance NIL inter:interactor-window
     (:left 150)
     (:top -2) ;no extra border at the top
     (:width (o-formula (- (gvl :parent :width) 150)))
     (:height (o-formula (gvl :parent :height)))
     (:double-buffered-p double-buffered-p)
     (:border-width 2)
     (:parent top-win)))

    ;;;create the top level aggregate in the windows
    (setq top-agg (create-instance NIL opal:aggregate
     (:left 0)(:top 0)
     (:width (o-formula (gvl :window :width)))
     (:height (o-formula (gvl :window :height)))))

    (setq work-agg (create-instance NIL opal:aggregate
     (:left 0)(:top 0)
     (:width (o-formula (gvl :window :width)))
     (:height (o-formula (gvl :window :height)))))
    ;;;create an aggregate to hold the user-created objects
    (setq objs-agg (create-instance NIL opal:aggregate
     (:left 0)(:top 0)
     (:width (o-formula (gvl :window :width)))
     (:height (o-formula (gvl :window :height)))))
    (opal:add-component work-agg objs-agg)

    ;;;create menus
    (create-mode-menu top-agg top-win)
```

```
(setf menu (create-menu top-agg))

;;;create a graphics selection object
(setq selection (create-instance NIL Garnet-Gadgets:graphics-selection
  (:start-where (list :element-of-or-none objs-agg))
  (:movegrow-lines-p NIL) ;can't move lines
  ;;move objects while cursor in the work window
  (:running-where (list :in work-win))))
(opal:add-component work-agg selection)

;;store the selection object in a new slot of the menu so that the delete
;;function can find which object is selected.
(s-value menu :selection-obj selection)

;;;Create an interactor to edit the text of the labels when they are first
;;;created.  This interactor will never start by itself, but is started
;;;explicitly using Inter:Start-Interactor in the Create-New-Object function.
(setf edit-text (create-instance NIL Inter:Text-Interactor
  (:obj-to-change NIL) ;this is set when the interactor is started
  (:start-event NIL) ;won't start by itself
  (:start-where NIL) ;won't start by itself
  (:stop-event '(#\return :any-mousedown)) ;either stops it
  (:window work-win)))
;;cont., next page...
```

```
    ;;The next interactor edits the text when the user presses on a string.█
    (create-instance NIL Inter:Text-Interactor
      (:stop-event '(#\return :any-mousedown)) ;either stops it
      (:start-where (list :leaf-element-of objs-agg :type opal:text))
      ;;high priority so that if this one runs, the object
      ;;underneath will not become selected.
      (:waiting-priority inter:high-priority-level)
      (:window work-win))

    ;;;create an interactor to create the new objects
    (create-instance NIL Inter:Two-Point-Interactor
      (:start-event :rightdown)
      (:start-where T)
      (:running-where (list :in work-win))
      (:window work-win)
      (:abort-event '(:control-g :control-\g))
      (:line-p (o-formula (gvl :window :parent :aggregate :line-p)))
      ;The next 2 slots are used by the Create-New-Obj procedure,
      ;not by this interactor itself.
      (:objs-aggregate objs-agg)
      (:textinter edit-text)
      (:selection selection)

      (:feedback-obj
       ;;use the feedback objects in the graphics-selection object
       ;;pick which feedback depending on whether drawing line or box
       (o-formula
(if (gvl :line-p)
    (gvl :selection :line-movegrow-feedback)
  (gvl :selection :rect-movegrow-feedback))))
      (:final-function #'Create-New-Obj))

    ;;;Now, add the aggregates to the window and update
    (s-value top-win :aggregate top-agg)
    (s-value work-win :aggregate work-agg)
    (opal:update top-win)  ;;will also update work-win

  ;;** Do-Go **
  (Format T "~%Demo-Editor:
 Press with left button on top menu to change modes (box or line).
 Press with left button on bottom menu to execute a command.
 Press with right button in work window to create a new object
      of the current mode.
 Boxes can be created anywhere, but lines must start and stop inside boxes.█
 After creating a box, you should type the new label.
 Press with left button on text string to start editing that string.
```

```
        While editing a string, type RETURN or press a mouse button to stop.█
   Press with left button in work window to select an object.
   Press with left button on white selection square to move an object.
   Press with left button on black selection square to change object size.█
   While creating, moving, or growing a box, move outside window and release or█
        hit ^G or ^g to abort.
   ~%")

   (unless dont-enter-main-event-loop #-cmu (inter:main-event-loop))

     ;;return top window
     top-win))

;;** This is mainly for debugging, since usually the quit button in the menu will be u
(defun Do-Stop ()
  (opal:destroy current-window))
```

12 Gilt Reference: A Simple Interface Builder for Garnet

by Brad A. Myers

10 May 2020

12.1 Abstract

Gilt is a simple interface layout tool that helps the user design dialog boxes. It allows the user to place pre-defined Garnet gadgets in a window and then save them to a file. There are two versions: one for Garnet look-and-feel gadgets and one for Motif look-and-feel gadgets.

12.2 Introduction

This document is the reference chapter for the *Gilt* tool, which is part of the Garnet User Interface Development System *GarnetIEEE*. Gilt stands for the **G**arnet **I**nterface **L**ayout **T**ool, and is a simple interface builder for constructing dialog boxes. A dialog box is a collection of *gadgets*, such as menus, scroll bars, sliders, etc. Gilt supplies a window containing many of the built-in Garnet gadgets (see Figure ⟨undefined⟩ [gadgetwindow], page ⟨undefined⟩), from which the user can select the desired gadgets and place them in the work window. Gilt does *not* allow constraints to be placed on objects or for new gadgets or application-specific objects to be created.

There are two sets of gadgets in Gilt. Each allows you to create dialog boxes with a consistent look-and-feel. The standard Garnet gadgets are shown in Figure ⟨undefined⟩ [gadgetwindow], page ⟨undefined⟩, and the Motif style gadgets are in Figure [motifgadgetwindow], page 619). Both versions operate the same way. You can toggle between the standard and Motif gadget palettes by selecting `"Load Other Gadgets"` from the main Gilt menubar.
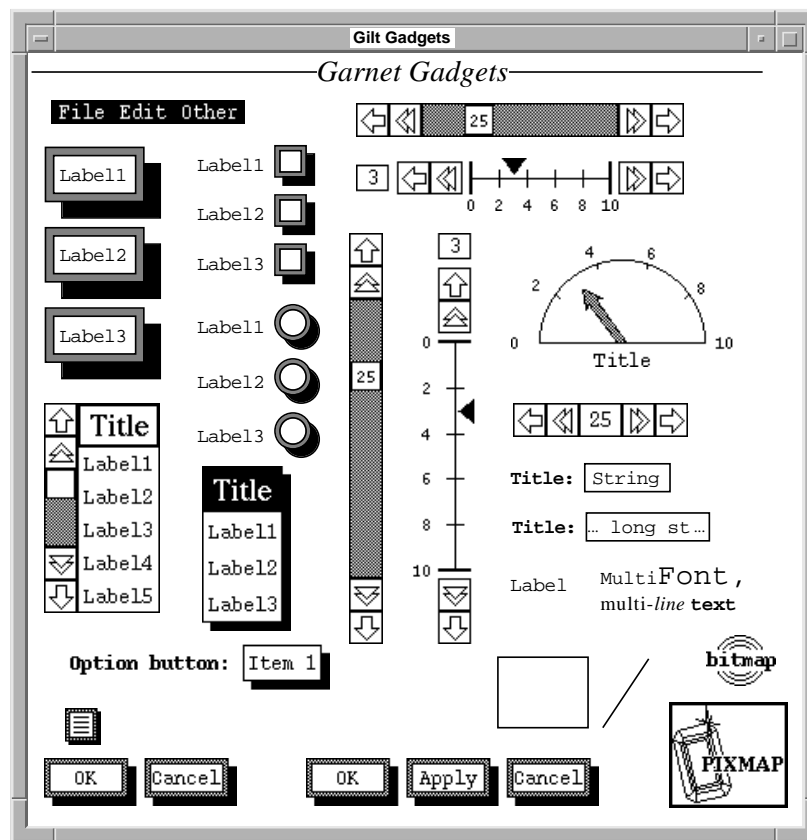
Figure 12.1: The Gilt gadget window for the Garnet look and feel. All of the gadgets that can be put into the window are shown. The check boxes are selected.
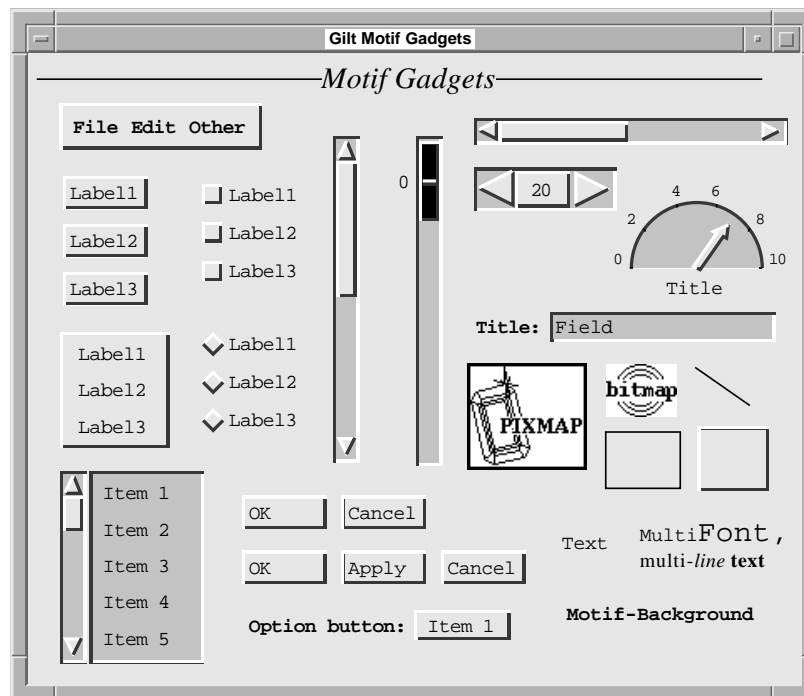
Figure 12.2: The Gilt gadget window for the Motif look and feel.

There is a more powerful interactive design tool in Garnet called Lapidary *garnetLapidary*. Lapidary allows new gadgets to be constructed from scratch, and allows application-specific graphics to be created without programming. However, Lapidary does not support the placement of the existing Garnet gadgets.

12.3 Loading Gilt

Gilt is *not* automatically loaded when you load Garnet. After Garnet is loaded, to load Gilt do:

```
(load Garnet-Gilt-Loader)
```

There is only one version of Gilt, but you can specify what set of gadgets should appear in the palette when the windows appear. This is determined by a required parameter to `do-go`. To start Gilt, do:

```
(gilt:do-go :motif)
(or)
(gilt:do-go :garnet)
```

Gilt can be stopped by selecting `"Quit"` from the menubar, or by executing `(gilt:do-stop)`.

12.4 User Interface

Gilt displays three windows: The gadgets window, the main command window, and the work window. The main command window is shown in Figure ⟨undefined⟩ [commandwindow], page ⟨undefined⟩. Figure ⟨undefined⟩ [workwindow], page ⟨undefined⟩, shows an example session where the work window contains gadgets with the Garnet look-and-feel. The two types of gadget palette windows are shown in Figures ⟨undefined⟩ [gadgetwindow], page ⟨undefined⟩, and [motifgadgetwindow], page 619.

For the Garnet look and feel, examples are in Figures [GadgetWindow], page 618, [CommandWindow], page 621, and [WorkWindow], page 622. Figure [motifgadgetwindow], page 619, shows the Gadget window for the Motif look and feel.
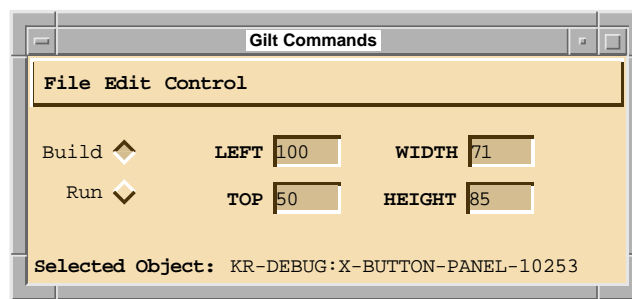
Figure 12.3: The Gilt Command window. The "Edit" menu from the menubar provides control over all properties of the gadgets in the work window and provides dialog boxes for precise positioning. Switching between "Build" and "Run" mode allows you to test the gadgets as you build the interface. Text boxes display the position and dimension of the selected gadget, whose name appears at the bottom of the command window.
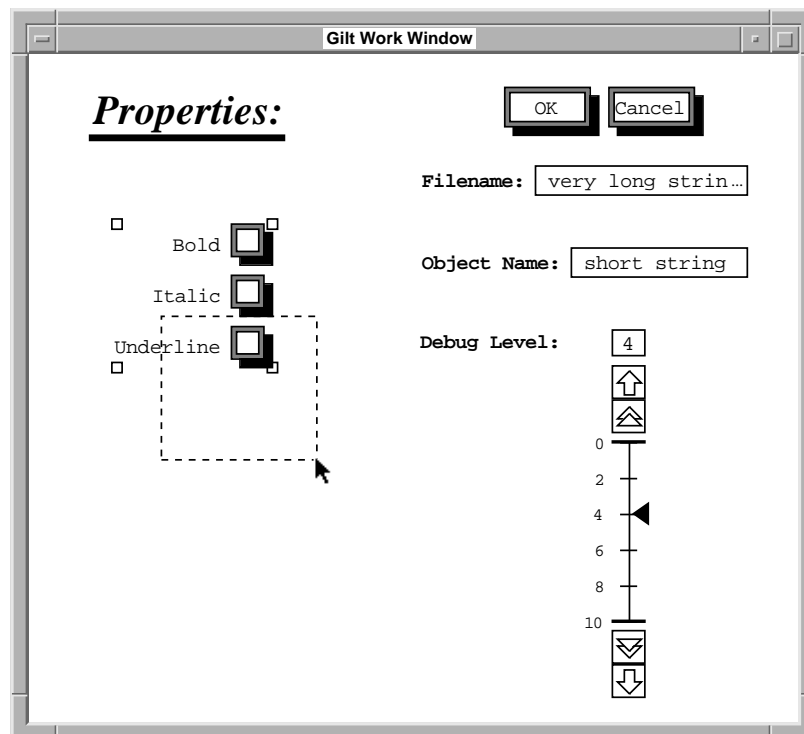
Figure 12.4: The Gilt Work window showing a sample dialog box being created using the Garnet look and feel.

12.4.1 Gadget Palettes

The single version of Gilt allows you to place Motif or Garnet look-and-feel gadgets 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 of the Gilt menubar. You can only see one gadget window at a time. The dialog boxes for Gilt itself use only the Motif look and feel.

12.4.2 Placing Gadgets

When you press with any mouse button on a gadget in the gadgets palette window (see Figures [GadgetWindow], page 618, or [motifgadgetwindow], page 619), that gadget becomes selected. Then, when you press with the *right* mouse button in the work window, an instance of that gadget will be created. Some gadgets, such as the scroll bars, have a variable size in one or more dimensions, so for those you need to press the right button down, drag out a region, and release the button.

The gadgets supplied for the *Garnet* look and feel are (from top to bottom, left to right in Figure [GadgetWindow], page 618):

Menubar: a pull-down menu,

Text-button-panel: for commands,

Scrolling-menu: when there are many items to choose from,

Option-button: a popup-menu which changes the label of the button according to the selected item,

Popup-menu-button: a popup-menu which does not change labels,

OK-Cancel: A special gadget to be used when you want the standard OK and Cancel behavior (see section [okcancel], page 639),

X-button-panel: for settings where more than one is allowed,

Radio-button-panel: for settings where only one is allowed,

Menu: a menu with an optional title,

H-scroll-bar: for scrolling horizontally,

H-slider: for entering a number in a range,

V-scroll-bar: for scrolling vertically,

V-slider: for entering a number in a range,

OK-Apply-Cancel: Similar to OK-Cancel, but supports Apply (like OK, but don't remove window),

Gauge: another way to enter a number in a range,

Trill-device: enter a number either in a range or not,

Labeled-box: enter any string; box grows if string is bigger,

Scrolling-labeled-box: enter a string; box has a fixed size and string scrolls if too big,

Text: for decoration,

Multifont-text: for decoration,

Rectangle: for decoration,

Line: for decoration,

Bitmap: for decoration,

Pixmap: for decoration.

The gadgets supplied for the *Motif* look and feel are (from top to bottom, left to right in Figure [motifgadgetwindow], page 619):

Motif-Menubar: a pull-down menu

Motif-Text-button-panel: for commands,

Motif-Menu: a menu with an optional title,

Motif-Scrolling-Menu: when there are many items to select from,

Motif-Check-button-panel: for settings where more than one is allowed,

Motif-Radio-button-panel: for settings where only one is allowed,

Motif-OK-Cancel: A special gadget to be used when you want the standard OK and Cancel behavior (see section [okcancel], page 639),

Motif-OK-Apply-Cancel: similar to OK-Cancel, but supports Apply,

Motif-Option-Buton: a popup-menu whose button's label changes according to the selection

Motif-V-scroll-bar: for scrolling vertically,

Motif-V-slider: for entering a number in a range,

Motif-H-scroll-bar: for scrolling horizontally,

Motif-trill-device: for selecting from a range of numbers,

Motif-Gauge: another way to enter a number in a range,

Motif-Scrolling-labeled-box: enter a string; box has a fixed size and string scrolls if too big,

Pixmap: for decoration

Bitmap: for decoration

Rectangle: for decoration,

Line: for decoration,

Motif-Box: A gadget that resembles a raised (or depressed) rectangle, used to achieve a Motif style effect. Set the `:depressed-p` parameter.

Text: for decoration,

Multifont-text: for decoration,

Motif-Background: a special rectangle that helps achieve the Motif effect. It always moves to the back of the window, and can only be selected at the edges.

In addition to the standard gadgets, Gilt supplies a text string, a line, a rectangle and a bitmap. These are intended to be used as decorations and global labels in your dialog boxes. They have no interactive behavior.

The Motif version also provides a background rectangle. This is a special rectangle which you should put behind your objects to make the window be the correct color. Note: to select the motif-background rectangle, press at the edge of the window (the edge of the background rectangle). You might want to select the rectangle to delete it or change its color (using the properties menu).

12.4.3 Selecting and Editing Gadgets

When you press with the left mouse button on a gadget in the work window, it will become selected, and will show four or twelve selection handles. The objects that can change size (such as rectangles and scroll bars) display black and white selection handles, and the objects that cannot change size (such as buttons) only show white selection handles.[1] If you press on a *white* handle and drag, you can change the object's position. If you press on a *black* handle, you can change it's size (see Figure ⟨undefined⟩ [handlesfig], page ⟨undefined⟩).

---

[1] You can indirectly change the size of buttons by setting offsets and sizes in the property sheet, however.
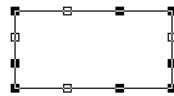
Figure 12.5: Pressing on a black selection handle causes the object to grow, and pressing on a white one causes it to move.

If you press over an object with either the *middle* mouse button or hold down the keyboard shift key while hitting the left button, then that object is added to the selection set (so you can get multiple items selected). If you press with middle or shift-left over an

item that is *already* selected, then just that item becomes de-selected. If you press with the left button in the background (over no objects), all objects are deselected. While multiple objects are selected, you can move them all as a group by pressing on the selection handle of any of them. Since Gilt uses the multi-selection gadget, it supports selecting all objects in a region (hold down the left button and sweep out a region), and growing multiple selected objects (if they are growable, then press on a black handle at the outside of the set of objects).

To explicitly set the size or position of the selected object (when only one is selected), you can use the number fields in the Command Window (see Figure [CommandWindow], page 621). Simply press with the left button in one of these fields and type a new number. When you hit `return`, the object will be updated. These fields are a handy way to get objects to be evenly lined up (but also see the `"Align"` command).

12.5 Editing Strings

Editing the strings of most gadgets is straightforward: 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 string labels, simply click to select them, and then click again with the left button to begin editing. The fonts of multifont strings can be edited using the keyboard commands described in the "Multifont" section of the Opal Chapter.

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.

The editing operations supported for regular text (and labels) are:

> `^h, delete, backspace`: delete previous character.
>
> `^w, ^backspace, ^delete`: delete previous word.
>
> `^d`: delete next character.
>
> `^u`: delete entire string.
>
> `^b, left-arrow`: go back one character.
>
> `^f, right-arrow`: go forward one character.
>
> `^a`: go to beginning of the current line.
>
> `^e`: go to end of the current line.
>
> `^y`: insert the contents of the X cut buffer into the string at the current point.
>
> `^c`: copy the current string to the X cut buffer.
>
> `enter, return, ^j, ^J`: Finished.

        `^n`: Finished, but add a new item (if a list).

        `line-feed`: Start a new line (if editing a multi-line text).

        `^g`: Abort the edits and return the string to the way it was before editing started.

If the item is a member of a list, such as a menu item or a radio button, then if the string is empty, that item will be removed. If the string is terminated by a `^n` (control-n) instead of by a return, and if this is the last item, then a new item will be added. The items can also be changed in the properties dialog box for the gadget (see below).

Some strings cannot be edited directly, however. This includes the labels of sliders and gauges, and the indicators in scroll bars. To change these values, you have to use the property sheets. Also, for gadgets that have strings as their *values*, such as the text input field and scrolling-text input field, you can only set the value strings by going into Run mode. Note, however, that the values are not saved with the gadget (see section [usinggiltdbs], page 638).

To change the bitmap picture of a bitmap object, specify the name of the new bitmap using the `"Properties..."` command.

12.6 Commands

There are many commands in Gilt, and the command menu is a menubar at the top of the main window. The menubar implementation allows you to give 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.

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. (See section [duplicating-objects], page 629.)

        `Delete` — delete the selected objects and don't put them into clipboard. This operation can be undone with the `Undo Last Delete` command. (See section [deleting-objects], page 631.)

        `Delete All` — delete all the objects in the window. This operation can be undone with the `Undo Last Delete` command. (See section [deleting-objects], page 631.)

        `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. (See section [deleting-objects], page 631.)

        `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. (See section [to-top], page 629.)

To Bottom — make the selected objects be covered by all other objects. (See section [to-top], page 629.)

Properties... — bring up the properties window. (See section [giltpropertiessec], page 631).

Align — bring up the dialog box to allow aligning of the selected objects with respect to the first of the objects selected. (See section [align], page 629.)

Many of these commands are now implemented with the functions in the Standard-Edit mechanism, described in the Gadgets Chapter.

12.6.1 To-Top and To-Bottom

The selected object or objects can be made so they are not covered by any objects using the "To Top" command in the Gilt Command Window. The objects can be made to be covered by all other objects by selecting the "To Bottom" command.

12.6.2 Copying Objects

The "Duplicate" command in the Command Window causes the selected object or objects to be duplicated. The new object or objects will have all the same properties as the original, but the original and new objects can be subsequently edited independently without affecting the other object (the new object is a copy, not an *instance* of the original). The copy is placed at a fixed offset below and to the right of the original, and is selected, so it can subsequently be moved.

12.6.3 Aligning Objects

The Align function allows you to neatly line up a set of objects, and to adjust their sizes to be the same. Figure [alignfig], page 630, shows the dialog box that appears when the "Align..." command is selected. Align adjusts the present positions of objects only; it does not set up constraints. Therefore, you can freely move objects after aligning them.
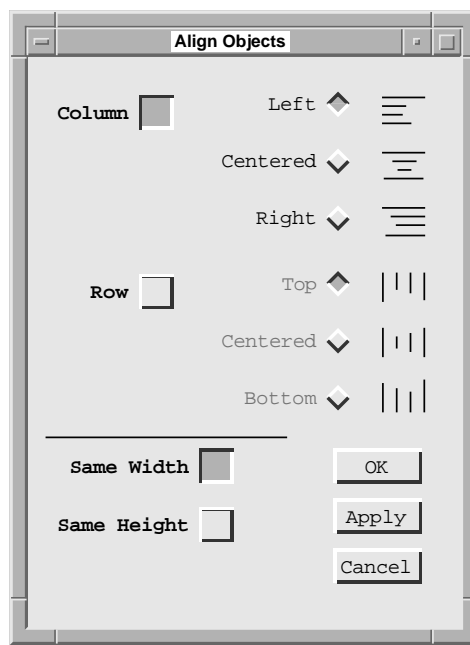
Figure 12.6: The Align dialog box, after the user has specified that the selected objects should be aligned and centered in a column and be adjusted to be the same width.

To use Align, you first select two or more objects in the workspace window (remember, to select more than one object, press on the objects with the middle mouse button or hold down the shift key while hitting the left button). The *first* object you select is the

reference object, and the other objects will be adjusted with respect to that first object. For example, if you want to make objects be the same width, then the width will be that of the first selected object. You should not change the selection while the Align dialog box is visible.

Aligning in a column or row also adjusts the spacing between objects to be all the same. The spacing used between the objects is the average space between the objects before the command is given.

If a line is selected, then it is made to be exactly horizontal if "Column" is specified, or vertical if "Row" is selected. The size of lines can also be adjusted using the width and height buttons. If both "Same Width" and "Same Height" are selected for a line, then an error message is given.

If the "Same Width" and/or "Same Height" buttons are pressed, and one of the selected objects other than the first cannot change size, then an error message is presented. All other selected objects are still adjusted, however.

12.6.4 Deleting Objects

Choosing the `"Delete Selected"` command in the Gilt Command Window will remove the selected object or objects from the work window. Selecting the `"Undo Last Delete"` command will bring the object back. Selecting `"Delete All"` removes all the objects from the work window. `"Undo Last Delete"` will bring all of the objects back. All of the deleted objects are kept in a queue, so the undo command can be executed repeatedly. Note that this is not a general Undo; only undoing of deletes is supported.

12.6.5 Properties

Each type of gadget has a number of properties. First select the object in the workspace window, and then select the `"Properties..."` command (in the Gilt Command Window). The window for the properties will appear below the selected object, but then can be moved. You should not change the selection while the properties dialog box is visible.

If the selected object is a rectangle, line or string, then special dialog boxes are available so you can change the color, filling-style, font, etc. These were created using Gilt.

The general property sheet lists all of the properties that you can change, and will look something like Figure [propsheet], page 633. You can press in the value (right) side of any entry and then type a new value (using the same editing commands as in section [editingcommands], page 627). You can move from field to field using the tab key (after pressing with the left mouse button in a field to start with). When finished setting values, hit the "OK" button to cause the values to be used and the property sheet to disappear, or hit the "Apply" button to see the results and leave the property sheet visible. If you hit "Cancel", the changes will not affect the object, and the property sheet will go away.

You can 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 edit-
ing the foreground-color property. If you start to edit a property but change your mind, hit
`Control-G` if text editing or `Cancel` in 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.

For a complete explanation of what the fields of each gadget do, see the Gadgets Chapter.

```
┌─────────────────────────────────────────────────┐
│ ═  □    X BUTTON PANEL           □  □            │
│ ┌─────────────────────────────────────────────┐ │
│ │ ┌──────┐                                     │ │
│ │ │ Done │                                     │ │
│ │ └──────┘                                     │ │
│ │ PIXEL-MARGIN: NIL                            │ │
│ │ H-ALIGN:  │:LEFT  :CENTER  :RIGHT│           │ │
│ │ FIXED-WIDTH-SIZE: 72                         │ │
│ │ FIXED-WIDTH-P: │T  NIL│                      │ │
│ │ V-SPACING: 5                                 │ │
│ │ VALUE: NIL                                   │ │
│ │ FONT: ▤                                      │ │
│ │ GRAY-WIDTH: 3                                │ │
│ │ SHADOW-OFFSET: 5                             │ │
│ │ BUTTON-WIDTH: 20                             │ │
│ │ KNOWN-AS [keyword]: NIL                      │ │
│ │ SELECT-FUNCTION: NIL                         │ │
│ │ BUTTON-HEIGHT: 20                            │ │
│ │ TEXT-OFFSET: 5                               │ │
│ │ TEXT-ON-LEFT-P: │T  NIL│                     │ │
│ │ ITEMS: ("Bold" "Italic" "Underline" )        │ │
│ │ DIRECTION:  │:VERTICAL  :HORIZONTAL│         │ │
│ │ H-SPACING: 5                                 │ │
│ │ FIXED-HEIGHT-P: │T  NIL│                     │ │
│ │ FIXED-HEIGHT-SIZE: NIL                       │ │
│ │ RANK-MARGIN: NIL                             │ │
│ │ INDENT: 0                                    │ │
│ │ VISIBLE: │T  NIL│                            │ │
│ └─────────────────────────────────────────────┘ │
└─────────────────────────────────────────────────┘
```

Figure 12.7: The property sheet that appears for a particular X-Button-Panel.

Some of the fields of these property sheets are edited in a special way. The `DIRECTION` field must be either `:VERTICAL` or `:HORIZONTAL`, so the field shows these names, and you can press with the left button to pick the desired value. Fields that represent fonts show a special icon, and if you click on it, the special font dialog box will appear. However, the

font is not changed in the object until the "OK" or "Apply" buttons are hit on *both* the font dialog box and the main property sheet.

The field named KNOWN-AS should be set for all gadgets that programs will want to know the values of, and will be the name of the slot that holds the object (so it should be a keyword, e.g., :myvalue). The SELECT-FUNCTION slot can contain a function to be called at run time when the gadget is used. Note that you might want to specify the package name on the front of the function name. However, if you are going to have OK-Cancel or OK-Apply-Cancel in the dialog box, you probably do not want to supply selection functions, since selection functions are called when the gadget is used, not when OK is hit (see section [using], page 640).

If the property sheet thinks any value is *illegal*, the value will be displayed in italics after a return or tab is hit, and Gilt will beep. You can edit the value, or just leave it if the value will become defined later (e.g., if the package is not yet defined).

Unfortunately, however, the error checking of the values typed into the property sheets is not perfect, so be careful to check all the values before hitting OK or Apply. If a bad value is set into the gadget, Gilt will crash. You can usually recover from this by setting the field back to a legal value in the Lisp window. For example, if :gray-width got a bad value, you might type:

```
(kr:s-value user::*gilt-obj* :gray-width 3)
(opal:update-all)
(inter:main-event-loop)
```

12.6.6 Saving to a file

When the "Save..." command is selected from the Command window, Gilt pops up the dialog box shown in Figure [Savedialogbox], page 635.
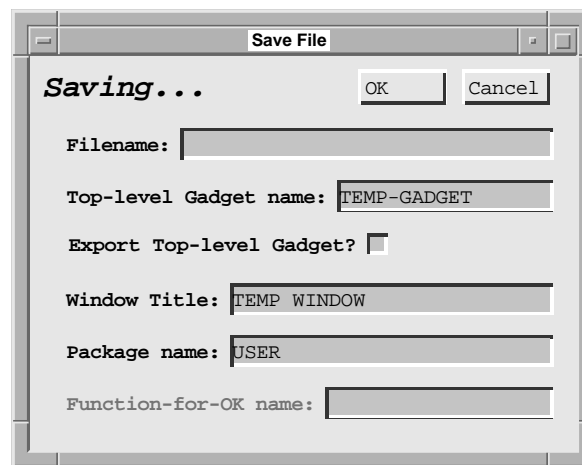
Figure 12.8: The dialog box that appears when the Save command is chosen.

The only field you need to fill in is the `"Filename"` field, which tells the name of the file that should be written. Simply press with the left button in the field and begin typing. This is a scrollable field, so if the name gets too long, the text will scroll left and right. You might also want to use the window manager's cut buffer (^Y) if you can select the string for

the file in a different window. Pressing with the mouse button again will move the cursor, so you need to hit `return` or `^G` to stop editing the text field.

All the objects in the work window will be collected together in a single Garnet "aggregadget" when written to the file. The `"Top-level Gadget name"` field allows you to give this gadget a name. This is usually important if you want to use the gadget in some interface, so you can have a name for it. If you press the `"Export Top-level Gadget"` button, then an export line will be added to the output file.

As described below in section ⟨undefined⟩ [Using], page ⟨undefined⟩, there is a simple function for displaying the created gadget in a window. If you want this window to have a special title, you can fill this into the `"Window Title"` field. The current position and size of the workspace window is used to determine the default size and position of the dialog box window when it is popped up, so you should change the workspace window's size and position (using the standard window manager mechanisms) before hitting OK in the Save dialog box.

If you want the gadget to be defined in a Lisp package other than `USER`, then you can fill this into the `"Package name"` field.

Finally, if you have included the special OK-Cancel gadget in your workspace window, then the `"Function-for-OK name"` field will be available. Type here the name of the function you want to have called when the OK button is hit. The parameters to this function are described in section ⟨undefined⟩ [Using], page ⟨undefined⟩.

After filling in all the fields, hit `"OK"` to actually save the file, or `"Cancel"` to abort and not do the save.

If you have already read or saved a file, then the values in the Save dialog box will be based on the previous values. Otherwise, the system defaults will be shown.

Note: There is no protection or confirmation required before overwriting an existing file.

12.6.7 Reading from a file

You can read files back into Gilt using the `"Read..."` command. This displays the dialog box shown in Figure [readdialogbox], page 637. Press with the left mouse button in the `"Filename"` field and type the name of the file to be read, then hit return.
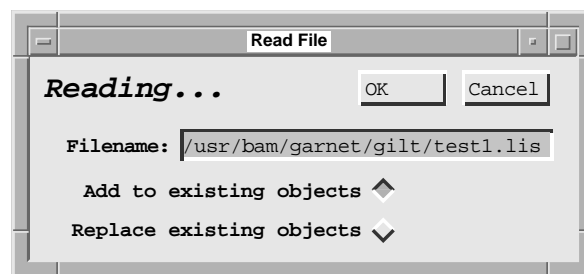
Figure 12.9: The dialog box that appears when the Read command is hit.

If there are objects already in the workspace window, then you have the option of adding the objects in the file to the ones already in the work window using the "Add to existing objects" option, or else you can have the contents of the workspace window deleted first using the "Replace existing objects" option. If you use the "Replace" option, then the

window size is adjusted to the size specified when the file was written. Also, reading a file using the replace option puts the previous contents of the workspace window in the delete stack so that they can be retrieved using the "Undo Last Delete" command.

Output produced from the GarnetDraw utility program can be read into Gilt, which would allow more elaborate decorations to be added to a dialog box. But in general, only files written with Gilt can be read with Gilt.

12.6.8 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 [GiltDemo]. The paper includes examples of how to use this feature, but a full set of documentation is still pending. If there is sufficient demand for documentation of this module, we will supply an addendum to this chapter (direct requests to garnet@cs.cmu.edu).

12.7 Run Mode

To try out the interface, just click on the button in the command window labeled "Run". This will grey out most of the commands, and allow the gadgets in the work window to execute as they will for the end user (except that application functions will not be called). To leave run mode, simply press on the "Build" button.

12.8 Hacking Objects

Gilt does not provide all options for all objects and gadgets. If you want to change other properties of objects that are not available from the property sheets, you could hit the HELP key while the mouse is positioned over the object to bring up the Inspector (see the Debugging chapter, starting on page [No value for "debug"] for details).

You can also access the selected object directly from Lisp. If one object is selected, its name is printed in the command window. Also the variable user::*gilt-obj* is set with the single selected object. If multiple objects are selected, then user::*gilt-obj* is set with the list of objects selected. You can go into the Lisp listener window, and type Garnet commands to affect the selected object (e.g., s-value some slots), and call (opal:update-all). This technique can also be used to add extra slots to objects. The changes you make will be saved with the object when it is written.

12.9 Using Gilt-Created Dialog Boxes

There are various ways to use Gilt-created collections of gadgets in an application.

The file that Gilt creates is a normal Lisp text file that creates the appropriate Garnet objects when loaded. The file should be compiled along with your other application files, in order to provide better performance.

12.9.1 Pop-up dialog box

Probably the easiest way to use a set of gadgets is as a pop-up dialog box. The application should be sure to *load* the file that Gilt created before calling the functions below.

When Gilt writes out the gadgets, it does *not* save the values as the initial defaults. Therefore, if you want to have default values for any gadgets, you need to set them from your program. This should be done *before* the window is displayed using the function:

    `gilt:Set-Initial-Value` *top-gadget gadget-name value*`[function], page 90`

The `top-gadget` is the top-level gadget name specified in the `"Top-level Gadget name"` field of the Save dialog box. The `gadget-name` is the name of the particular gadget to be initialized. This name will be a keyword, and will have been specified as the `KNOWN-AS` property of the gadget using the gadget's property sheet (which appears when you hit the `"Properties..."` command). The `value` is the value to be used as the default, in the appropriate format for that gadget.

Next, the Gilt function `show-in-window` can be used to display the dialog box in a window:

    `gilt:Show-In-Window` *top-gadget* `&optional` *x y modal-p*`[function], page 90`

The *top-gadget* is the gadget name used in the Save dialog box. The size of the window is determined by the size of the workspace window when the file was written. The position of the window will either be the position when written, or it can be specified as the *x* and *y* parameters, which are relative to the screen's upper-left corner. When the *modal-p* parameter is T, then interaction in all other Garnet windows will be suspended until the window goes away (e.g., when the user clicks the "OK" button). If you want the window relative to a position in another window, the function `opal:convert-coordinates` is useful.

The function `show-in-window-and-wait` performs the same function as `show-in-window`, but it waits for the user to click on an OK or Cancel button before returning (`show-in-window` returns immediately after bringing up the window).

    `gilt:Show-In-Window-And-Wait` *top-gadget* `&optional` *x y modal-p*`[function],`
    `page 90`

When the user clicks on the OK button, this function will return the values of all the gadgets in the dialog box in the form of `gilt:gadgets-values`, which is:

    `((:FILENAME "/usr/bam/garnet/t1.lisp") (:VAL 49) (:BUTTON "Start"))`

where the keywords are the names (`:known-as` slot) of the gadgets. If the user hits Cancel, then `show-in-window-and-wait` returns `nil`. Apply does not cause the dialog box to go away, so you might want to supply an OK-Function for the dialog box.

If selection functions were specified in the gadget's select-function slot using the `"Properties..."` command, then these functions are called immediately when the gadgets are used.

If the dialog box has an OK-Cancel or OK-Apply-Cancel gadget in it, then the function specified in the `"Function-For-OK name"` field of the Save dialog box will be called when the user hits the OK or Apply buttons. This function is parameterized as:

    `(lambda (top-gadget values)`

The `top-gadget` is the same as above. The `values` parameter will be a list containing pairs of all the gadget names of gadgets which have names, and the value of that gadget.

Again, the names are the keywords supplied to the KNOWN-AS property. For example, values might contain:

```
((:FILENAME "/usr/bam/garnet/test1") (:REINITIALIZE NIL))
```

The function

> gilt:Value-Of *gadget-name* *values*[function], page 90

can be used to return the value of the specific gadget named gadget-name from the values list values. For example, if v is the above list, then (gilt:value-of :filename v) would return "/usr/bam/garnet/test1".

After the Function-For-OK is called, the dialog box window is made invisible if OK was hit, and left in place if Apply was hit. If Cancel was hit, then the window is simply made invisible. If show-in-window is called again on the same dialog box, the old window is reused, saving time and space.

To destroy the window associated with a gadget, use the function:

> gilt:Destroy-Gadget-Window *top-gadget*[function], page 90

This does *not* destroy the top-gadget itself. Note that destroying the top-gadget while the window is displayed will not destroy the window. However, destroying the window explicitly (using opal:destroy) *will* destroy both the window and the gadget.

12.10 Using Gilt-Created Objects in Windows

If you want to use the gilt-created gadgets inside of an application window, you only need to create an instance of the top-gadget, which is the top-level gadget name specified in the "Top-level Gadget name" field of the Save dialog box. The instance will have the same position in the application window as it had in the Gilt workspace window. If you use the standard Gilt OK-Cancel gadget, it will make the application window be invisible when the OK or Cancel buttons are hit. If you do not want this behavior, then you need to create your own OK-Cancel buttons.

The set-initial-value described above can still be used for gilt gadgets in application windows. In addition, the function

> gilt:Gadget-Values *top-gadget*[function], page 90

can be used to return the values list of all gadgets with names. The return value is in the form of the values parameter passed to the Ok-Function.

12.11 Hacking Gilt-Created Files

Since the file that Gilt creates is a normal text file, it is possible to edit it with a normal text editor. Some care must be taken when doing this, however, if you want Gilt to be able to read the file back in for further editing. (If you do not care about reading the files back in, then you can edit the file however you like.)

The simplest changes are to edit the values of slots of the objects. These edits will be preserved when the file is read in and written back out. Be sure not to change the value of the :gilt-ref slot.

When Gilt saves objects to a file, it sets the :constant slot of all the gadgets to T. If you expect to ever change any properties of widgets in the dialog boxes when they are being used by an application, then you should hand-edit the Gilt-generated file to change the

constant value (typically by :excepting the slots you plan to change dynamically). (Gilt reads in the file using `with-constants-disabled`, so the defined constant slots will not bother Gilt.)

If you want to create new objects, then these can be put into the top level aggregadget definition. You should follow the convention of having a :box (or :points) slot and putting the standard constraints into the :left and :top fields (or :x1, :y1, :x2 and :y2). For example, to add a circle, the following code might be added into the top-level aggregadget's :parts list:

```
(:mycircle ,opal:circle
  (:box (50 67 30 30))
  (:left ,(o-formula (first (kr:gvl :box))))
  (:top ,(o-formula (second (kr:gvl :box))))
  (:width 30)
  (:height 30))
```

If you do not supply a :gilt-ref field, Gilt will allow the user to move the object around, but not change its size or any other properties. For some objects, it might work to specify the :gilt-ref slot as "TYPE-RECTANGLE", "TYPE-LINE" or "TYPE-TEXT".

If you add extra functions or comments to the file, they will *not* be preserved if the file is written out and read in. Similarly, interactors added to the top level gadget will not be preserved.

⟨undefined⟩ [References], page ⟨undefined⟩,

13 C32 Reference: A Constraint Editor

by Dario Giuse

10 May 2020

13.1 Abstract

C32 is an object and constraint editor for Garnet objects. It allows Garnet objects to be viewed and edited using a spreadsheet-like interface. New values can be installed in the slots of an object directly, or constraints can be defined among the objects.

13.2 Overview of C32

C32 [C32] is an object and constraint editor for Garnet objects. It allows Garnet objects to be viewed and edited using a spreadsheet-like interface. Each object is viewed in a panel, where each row corresponds to a slot in the object. The value of a slot can be changed directly, by editing the value shown in the corresponding row. Constraints can be edited textually in separate formula-editing windows.

C32 can be used as a stand-alone tool to create and edit Garnet objects. It is possible, for example, to edit an existing object by typing its name in the title of a C32 panel, or by pointing and clicking on an object with the mouse. In addition, C32 is integrated with Lapidary, which uses it for several editing tasks that used to be handled specially.

Because it uses the spreadsheet paradigm, C32 is highly structured. Each object is represented as a panel, and all panels are organized horizontally in one long window. A horizontal scroll bar allows different panels to be displayed in the window. Panels that contain more than 12 slots are displayed with a vertical scroll bar, so more slots can be made visible as desired.

C32 keeps the display of each panel up to date. When a slot is modified using C32, the values displayed for other dependent slots are modified accordingly. Even if objects are changed from outside C32 (using the Lisp listener or the mouse, for example), their corresponding panels are always kept up to date.

13.3 Loading C32

To load C32, you load the file "garnet-c32-loader" and then type

```
(c32:do-go)
```

Note that if you are using Lapidary, you do not need to load C32 explicitly; Lapidary does it automatically.

The function (c32:do-go) creates two windows: the C32 Commands window, which contains the main commands used to control C32, and the spreadsheet window. Initially, the spreadsheet window contains a single, empty panel whose title reads "Object name:". The title of this panel can be edited to the name of an object, which is then displayed in the panel.

The full syntax for do-go is as follows:

`c32:do-go` **&key** (*startup-objects* **nil**) (*test-p* **nil**) (*start-event-loop-p*   [Function]
    t)

> If <startup-objects> is specified, it should be a list of Garnet objects. When C32 is
> started, it creates a panel for each object in the list, plus the empty panel. If <test-p>
> is specified, a little test window is created with a few objects in it. C32 can then
> be used to edit the objects in the window. Setting <start-event-loop-p> to `nil` cause
> C32 to start up with the main-event loop not running.

13.4 The Spreadsheet Window

The spreadsheet window contains a list of panels, each displaying a Garnet object. Figure
[c32-spreadsheet], page 644, shows the spreadsheet window with a panel for a `label-text`
object and the empty panel. Each panel consists of a vertical scroller (using for displaying
more slots), a title, and up to 12 rows. Each row displays the contents of a slot.
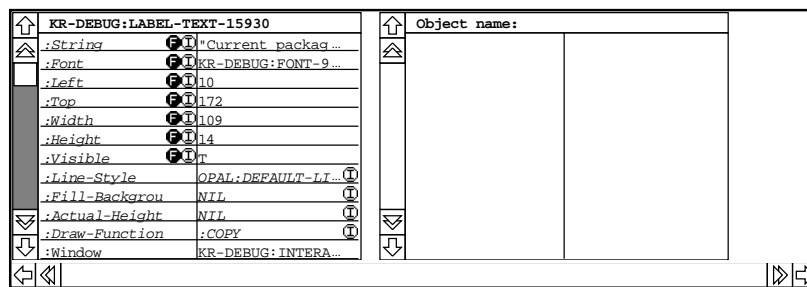
Figure 13.1: C32 Spreadsheet Window with two panels.

The title of each panel shows the name of the object displayed in the panel. The title can be edited by clicking the left mouse button over it, and then using the normal Garnet text editing commands. Type Return to go ahead, and ^G if you do not want to make the change. Entering the name of a different object in a panel's title causes the panel to display

the new object. If the object does not exist, C32 will ask you if you wish to actually create a new object. Setting the title of a panel to be empty causes the panel to be removed from the spreadsheet window; the object being displayed is unaffected, however. Setting the title of the empty panel to an object's name causes a new panel to be created for the object.

The left half of each row displays the name of the slot. The names of local slots are shown in a roman face; the names of inherited slots are shown in italics. Slots that contain a formula are indicated by an "F" on a dark circle. If the formula was inherited, this is indicated by an "I" inside a circle, next to the formula symbol. The right half of each row displays the value of the slot. If the value is inherited, an "I" inside a circle is shown at the far right of the row.

At any time, you can have a primary selection and a secondary selection. The primary selection is shown by a dark background, and is used for the majority of C32 operations that require a slot or an object. You may change the primary selection by clicking the left mouse button over a slot name, i.e., in the left side of a row. The secondary selection is shown by a gray outline around a slot, and is used for operations that require two slots. You may change the secondary selection by clicking the middle mouse button over a slot name. Both primary and secondary selections are toggles, and can be eliminated by clicking over them.

Panels allow you to modify objects, as well as displaying them. The value of a slot can be edited by editing its text: first click on the value (in the right half of the row) to get a text cursor, and then use the normal Garnet text editing commands. When you type Return, the slot is set to the new value and the object is modified (type ^G if you do not want to make the change). Note that the package shown in the Commands Window is used when reading the value you type in. Setting the package appropriately ensures that you do not have to type package qualifiers for every function and symbol.

The last row of a panel is always empty. Clicking in its left-hand half allows you to add a slot to the object, or to display a slot that is currently not shown. When you click, you start editing an (initially empty) slot name. If the slot is currently not shown, its current value is displayed. If the slot is not present, it is created. Its initial value is inherited, if possible; otherwise, it is set to `nil`. You may then edit the value (in the right-hand side). As a special shortcut, it is also possible to enter a slot name and a value together; just type the slot name, a space, and then the value.

The formula associated with a slot can also be edited using the spreadsheet window. Click on the "F" symbol; this pops up a window that allows the formula to be edited, as explained below. This mechanism also let you create formulas for slots that do not yet contain one: simply click on the place where the "F" symbol would be, and a new formula window will appear. You may then type the text for the new formula. Note that editing the value of a slot that contains a formula is equivalent to doing an `s-value` on the slot with the new value: the old value is temporarily replaced, but the formula is unaffected.

13.5 Editing Formulas

Formulas can be edited in special editing windows. When you click on the "F" symbol of a slot in the spreadsheet window, a window is created in which you can edit the textual representation of the formula. If you click on the "F" symbol and a window already displays that formula, the window is simply moved to the front. If you click on the "F" symbol of a

slot that contains a value, the value is shown as the initial text for the (yet to be created) formula.

Figure [c32-formula], page 647, shows the C32 formula window for the `:left` slot of the object shown in Figure [c32-spreadsheet], page 644. A formula window contains a header, a vertical scroller, and a text window. The header displays the name of the object and the slot upon which the formula is installed, and contains five buttons. The scroller allows you to examine different portions of long formulas.

```
KR-DEBUG:LABEL-TEXT-15930                          OK   Cancel
slot :Left
   Insert Function     Insert From Spread    Insert From Mouse

  (KR:GVL :PARENT :LEFT)
```
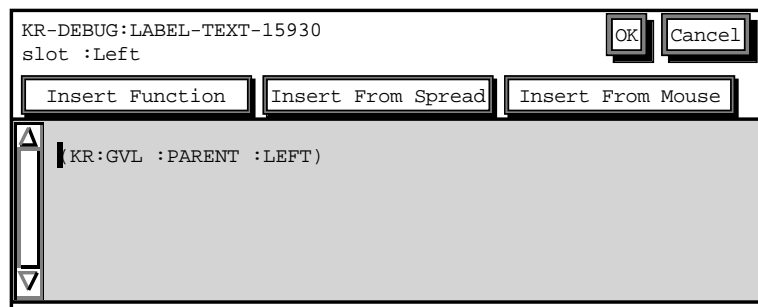
Figure 13.2: The C32 formula window for the :left slot.

When the formula window is created, the text cursor is initially positioned at the top left of the text. The cursor can be moved, and the text can be edited, using the normal Garnet text editing operations. Note that typing the abort character (^G) in a formula window

has no effect; click the Cancel button if you really want to abort the current changes to the formula's text.

The five buttons in the header include the OK and Cancel button, plus three buttons that can be used to reduce typing when the formula is being edited. The "OK" button installs the expression currently displayed in the formula window into the slot of the object, and hides the formula window. If errors are detected (for example, because the syntax in the formula expression is illegal), you will see an error message and the formula window will remain on the screen. The "Cancel" button removes the window without modifying the formula that is currently installed on the slot.

The "Insert Function" button pops up a menu with the names of functions that are commonly used in formulas. In addition to `floor`, `max`, and the like, the menu includes the functions from the `opal:gv-` family. Select a function from the menu by clicking the left mouse button over it; this will show the function's name in reverse video. Clicking on the "Insert Function" button will now insert the selected function, enclosed in parentheses, at the cursor position in the formula window.

The "Insert From Spread" button inserts a reference to the object and slot that are currently selected in the spreadsheet window into the formula being edited. C32 detects whether the selected object is the same as the one that contains the formula, and if so, it generates a simple reference using `gvl`.

The "Insert From Mouse" button is similar, except that it allows you to select the target object using the control-left mouse button. The button pops up a dialog box through which you may select an object. C32 will attempt to guess a slot; for example, if you click `control-left` on the left part of a string, it will insert the `:left` slot. If C32 cannot guess any slot, it leaves the slot name blank. When the appropriate object (and possibly slot) is shown in the dialog box, clicking the "Apply" button will insert a reference into the formula window. Clicking the "OK" button will do the same, and hide the dialog box as well.

13.6 The Commands Window

This window contains a menu with C32 commands that apply to the spreadsheet window, and is shown in Figure [c32-commands], page 649. Many of the buttons in the Commands Window operate on the slot that is currently selected in the spreadsheet window. The window also displays the current package that is used by C32 to interpret Lisp values and expressions (for example, when you type a value or a formula). The package can be changed by editing the string in the box.
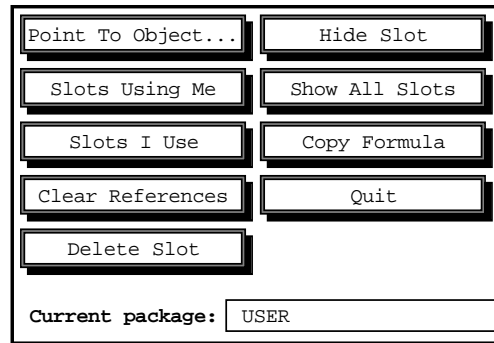
Figure 13.3: The C32 Commands Window.

The meaning of each group of buttons is explained below.

13.7 [Point To Object]

This button pops up a dialog box that allows an object to be added to the spreadsheet window by pointing and clicking with the mouse. The dialog box explains how to select objects (by clicking control-left) and how to move from one object to another underneath it. Clicking control-left on any Garnet object inserts the name of the object in the dialog box.

Once the name of the desired object is displayed in the dialog box, you can click the Apply button (which creates a new panel displaying the object), the OK button (which does the same thing and then hides the dialog box), or the Cancel button.

13.8 [Showing references to other slots]

Three buttons are used to show references, i.e., dependencies among slots. Clicking on the "Slots Using Me" button displays green arrows that indicate what formulas (in objects currently shown in C32) use the selected slot. The arrows originate on the Formula symbol of the dependent slots, and point to the value portion of the selected slot. If a dependent slot belongs to another object, and that object is shown in a panel, the arrow is drawn to the other panel.

Clicking on the "Slots I Use" button of a slot that contains a formula shows red arrows from the formula symbol to all the slots (currently shown in C32) upon which the formula depends. These arrows are heavier than the ones discussed previously, and they point to the slot side, rather than the value side.

Clicking on the "Clear References" button eliminates all currently displayed reference arrows. This operation does not alter any internal value, of course.

13.9 [Deleting, hiding, and showing slots]

The "Hide Slot" button causes the selected slot to be eliminated from the C32 panel. The value of the slot in the object is unaffected. The "Show All Slots" button causes all slots in the selected object to be displayed in the panel.

The "Delete Slot" is used to delete the currently selected slot from the actual object. Care should be taken, because this is a destructive operation. Trying to delete some of the most important slots prompts for confirmation.

13.10 [Copy Formula]

The "Generalize Formula" button allows a function to be created by generalizing the formula associated with the current slot. Generalizing a formula means that a function is created in which hard-wired references to objects and slots are replaced by parameters. The function can then be used as a more general expression, for example inside other formulas. This button pops up a dialog box that allows you to type the name of the function to be created, and to select names for object and slot references. When you click OK, the definition of the new function is printed in the Lisp listener window.

The "Copy Formula" button copies the formula installed on the secondary-selected slot to the slot that corresponds to the primary selection. A box pops up to make sure this is what you want to do.

13.11 [Quit]

This button causes C32 to destroy all its windows and exit. If C32 was started up from Lapidary, however, the windows are simply temporarily hidden, so that C32 can start up faster the next time.

13.12 C32 Internals

The list of slots to be displayed in a panel is kept in the `:slots-to-show` slot of Garnet objects. In most cases, this slot is inherited. If you are creating new types of objects, you may want to set the `:slots-to-show` slot appropriately in the prototype, so that C32 will display only relevant slots when it shows an instance in a panel.

It is probably a good idea not to try to edit the contents of the `:slots-to-show` slot using C32 itself.

The current version of C32 is not very optimized; redisplaying and scrolling panels, in particular, is rather inefficient. We hope to make its performance better in the next release.

⟨undefined⟩ [References], page ⟨undefined⟩,

14 Lapidary Reference

by Brad T. Vander Zanden, David Bolt

10 May 2020

14.1 Abstract

This document describes the features and operations provided by Lapidary, a graphical interface builder that allows a user to pictorially specify all graphical aspects of an application and interactively create much of the behavior. Lapidary allows a user to draw most of Opal's objects, combine them into aggregadgets, align them using iconic constraint menus or custom constraints, and create behaviors by entering appropriate parameters in dialog boxes representing each of Garnet's interactors, or by demonstrating the appropriate behavior for feedback objects.

14.2 Getting Started

To load Lapidary, type ((load garnet-lapidary-loader)) after Garnet has been loaded, or type ((defvar load-lapidary-p t)) before Garnet is loaded, and Garnet will automatically load Lapidary when the Garnet loader file is invoked. To start Lapidary, type ((lapidary:do-go)). This will cause Lapidary to come up in its initial state with the following windows:

editor-menu: This menu contains a set of functions that deal with aggregadgets, constraints, saving and restoring objects, deleting objects, and setting properties of objects.

shapes menu: This menu allows the designer to create opal graphical objects and windows.

box-constraint menu: This menu allows the designer to attach constraints to an object that control its left, top, width, and height.

drawing window: This window allows the designer to create new objects or load objects from existing files.

14.3 Object Creation

Lapidary allows new objects to be created from scratch, loaded from pre-defined gadgets files, or created directly in Garnet and then linked to a Lapidary window. The shapes menu displays the primitive graphical objects that can be created in Lapidary.
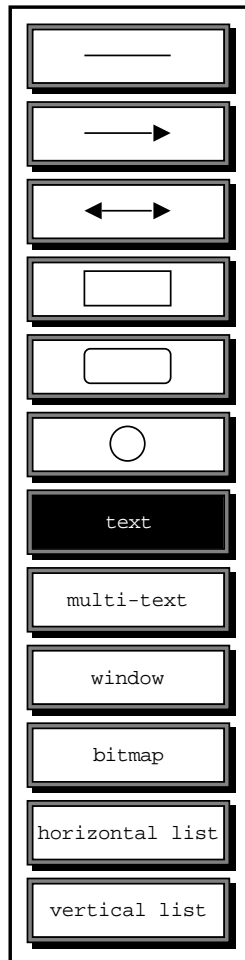
Figure 14.1: Shapes menu

The first six geometric shapes can be created by selecting the appropriate menu-item and sweeping out the item in a drawing window with the right mouse button down. Feedback corresponding to the selected shape will be shown as the object is swept out. Properties

such as line-style, filling-style, and draw-function can be set from the corresponding property menus (see section Section 14.10 [Properties], page 681).

To create a single line of text, select text and then click where you want the text to start. A cursor will appear and one line of text can be entered from the keyboard. For more than one line of text use multi-text. Single-line text can be terminated with either a mouse click or by hitting RETURN but, multi-line text can only be terminated by a mouse click.

To create a window, select the ⟨window⟩ menu-item and Lapidary will create a new window. Since, new windows initially have the same size and location as the draw window, they must be moved in order to expose the original draw window.

Bitmaps can be loaded by selecting the ⟨bitmap⟩ menu-item. Lapidary brings up a dialog box that allows the user to enter the name of an image file and the window that the bitmap should be placed in. The window name is obtained from the title border that surrounds a window or the name that appears in the icon for the window.

To create a horizontal or vertical list, first select a prototype object. Then select horizontal or vertical list and sweep out the list. A property sheet will appear that can be used to set parameters that control the list's appearance. A description of the parameters can be found in the chapter on aggregadgets and aggrelists.

14.4 Selecting Objects

Lapidary permits two types of selections: primary selections and secondary selections. Primary selections are denoted by black grow boxes that sprout around the perimeter of an object; secondary selections are denoted by white grow boxes. Most operations do not distinguish between these two types of selections and will operate in the same way on both types of selections. However, two operations, attaching a constraint to an object and defining parameters for an object, do make this distinction.

Lapidary provides two types of selection modes (Figure [lapidary-editor-menu], page 655): a "leaves" mode which causes Lapidary to select leaf elements of an aggregate, and a "top-level objects" mode which causes Lapidary to select top-level aggregates (objects that do not belong to an aggregate will be selected in either mode). To aid the user in determining whether they have selected a leaf or aggregate element, Lapidary uses different types of selection handles, rectangular handles for leaf elements and circular handles for aggregates (Figure [selection], page 659). If the object is too small to accomodate 8 selection handles, either a thin or thick-lined arrow is used to highlight the selection, depending on whether the object is a leaf or aggregate object (Figure [selection], page 659).

In either mode, additional clicks over the selected object will cause Lapidary to cycle through the aggregate hierarchy. For example, when the user clicks on the label shown in Figure [aggregate-hierarchy], page 656.a, and Lapidary is in "top-level objects" mode, the entire list element is selected (Figure [selection-techniques], page 657.a). If the user clicks on the label again, the label is selected (Figure [selection-techniques], page 657.b). Clicking once more with the mouse causes the key-box to become selected (Figure [selection-techniques], page 657.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.
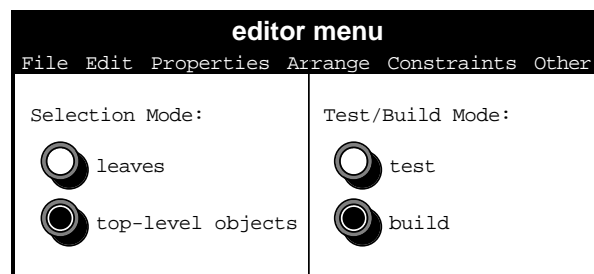
Figure 14.2:

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.
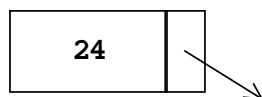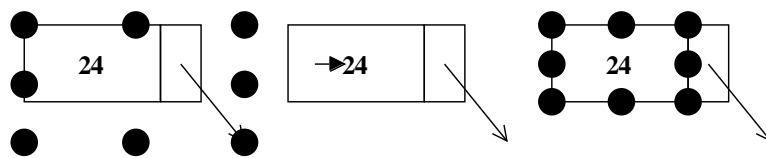
(a) (b) (c)

Figure 14.4:

As the user repeatedly clicks the mouse button over an object, the selection cycles through the aggregate hierarchy shown in Figure [aggregate-hierarchy], page 656. 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).

Lapidary provides the usual range of selection operations found in drawing editors, select, add to selection, deselect, remove from selection, select/deselect in region. These operations are supported for both primary and secondary selections. In addition, Lapidary allows the user to select covered objects by pointing at an already selected object and requesting that the object directly covered by the selected object be selected.
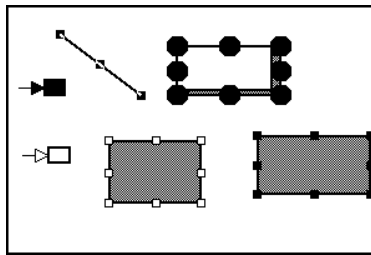
Figure 14.5: Different types of selection objects. Squares are for primitive graphical objects, circles are for aggregadgets, and arrows are for objects too small to accommodate grow boxes.

14.5 Mouse-Based Commands

Lapidary is primarily a mouse-based system so it is important to know which mouse buttons correspond to which operation. These bindings are set in the file mouse-bindings.lisp and may be edited. Currently the following operations can be bound to mouse buttons (the pair following each entry shows the default and the variable that must be changed to modify the default):

Primary Selection (leftdown, *prim-select-one-obj*): The user can either point at a particular object and make it the primary selection, or sweep out a rectangular region of the screen and make all objects that *intersect* the region be primary selections. This operation causes the previous primary selections to be deselected. If the mouse is not pointing at any objects, all primary selections are deselected. Each successive mouse click over the same object moves the selection one level higher in the aggregate hierarchy, until the top-most level is reached, at which point the selection process cycles back to a leaf (if the selection is initially a top-level object, the next click cycles to the leaf).

Secondary Selection (middledown, *sec-select-one-obj*): Same as primary selection but makes a secondary selection.

Deselect Primary Selection (control-leftdown, *primary-deselection-button*): This operation allows the user to deselect primary selections. The user can either point at a specific object or sweep out a rectangular region, in which case all objects that intersect this region will be deselected (if they are primary selections).

Deselect Secondary Selection (control-middledown, *secondary-deselection-button*): Same as Deselect Primary Selection except secondary selections are deselected.

Primary Select Covered Object (shift-control-leftdown, *prim-push-sel-under-button*): This operation allows the user to select covered objects. When the user points at a particular area of the screen, Lapidary determines which object is currently selected, and then deselects it and primary selects the first object that it covers. If no object under the mouse is selected, Lapidary primary selects the top object. If multiple objects under the mouse are selected, Lapidary finds the first unselected object which is under a selected object, selects the unselected object, and deselects the topmost selected object.

Secondary Select Covered Object (shift-control-middledown, *sec-push-sel-under-button*): Same as Primary Select Covered Object except a secondary selection is made.

Add to Primary Selection (shift-leftdown, *prim-add-to-select*): Same as Primary Selection except previously selected objects remain selected. Covered objects that are selected are automatically added to a selection, rather than causing previously selected objects to be deselected. Multiple clicks with the (Add to Primary Selection) button over the *selection handles* of a covered object will cause the selection to cycle through the aggregate hierarchy (Figure [covered-selection], page 662).

Add to Secondary Selection (shift-middledown, *sec-add-to-select*): Same as add to primary selection but adds to secondary selection.

Move Object (leftdown, *move-button*): This operation allows the user to move an object around the window. The user must point at one of the eight "grow" boxes around the perimeter of box objects, or one of the three "grow" boxes attached to line

objects or the arrow if the object is too small to contain grow boxes. If the object is
a box object and the user points at one of the corner boxes, the object can move in
any direction, if the user points at one of the side boxes, the object can move in only
one direction (along the x-axis if the left or right side is chosen and along the y-axis if
the top or bottom side is chosen). If the object is a line object, Lapidary will attach
the mouse cursor to the point designated by the grow box (either an endpoint of the
line or its midpoint) and move the line in any direction. If the object is undersized so
that the object does not have grow boxes but instead is pointed at by an arrow, then
pointing at the arrow will cause the cursor to be attached to the northwest corner of
the object and the object can be moved in any direction.

Grow Object (middledown, *grow-button*): This operation allows the user to resize
an object. The user must point at one of the eight "grow" boxes around the perimeter
of the object if the object is a box, one of the endpoint "grow" boxes attached to the
object if the object is a line, or the arrow that points at the object if the object is too
small to contain the grow boxes. If the object is a box object and the user points at one
of the corner boxes, both the object's width and height can change, if the user points
at one of the side boxes, only one of the object's dimensions will change (the width if
the left or right side is chosen, the height if the top or bottom side is chosen). If the
object is a line object, Lapidary will attach the mouse cursor to the point designated
by the grow box and move that endpoint while holding the other endpoint fixed. If the
object is undersized so that the object does not have grow boxes but instead is pointed
at by an arrow, then pointing at the arrow will cause the cursor to be attached to the
northwest corner of the object and the object's width and height will both change.

Object Creation (rightdown, *obj-creation-button*): The user sweeps out a region of
the screen and Lapidary creates the object selected in the shapes menu.

Copy Object (shift-rightdown, *copy-button*): This operation allows the user to create
a copy of an object and position it in a window (copies of an object can also be
created using the (make copy) command in the editor menu window, see Section [edit-
commands], page 669, for details). The user must point at one of the eight "grow"
boxes around the perimeter of box objects, or one of the three "grow" boxes attached
to line objects or the arrow if the object is too small to contain grow boxes. The selected
"grow" box constrains the initial movement of the new object (see (Move Object) for
a description of how the "grow" boxes constrain movement).

Instance Object (control-rightdown, *instance-button*): This operation allows the user
to create an instance of an object and position it in a window (instances of an object
can also be created using the (make instance) command in the editor menu window,
see Section [edit-commands], page 669, for details). The user must point at one of the
eight "grow" boxes around the perimeter of box objects, or one of the three "grow"
boxes attached to line objects or the arrow if the object is too small to contain grow
boxes. The selected "grow" box constrains the initial movement of the new object (see
(Move Object) for a description of how the "grow" boxes constrain movement).

Text Editing (rightdown, *obj-creation-button*): The user can edit a selected text
object by pointing at it and clicking with the object creation button. The user can use
any text editing command described in the interactors chapter and clicks down on the
mouse button to indicate that editing is complete.

(a) (b) (c)

Figure 14.6: 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).

Drawing Window 4
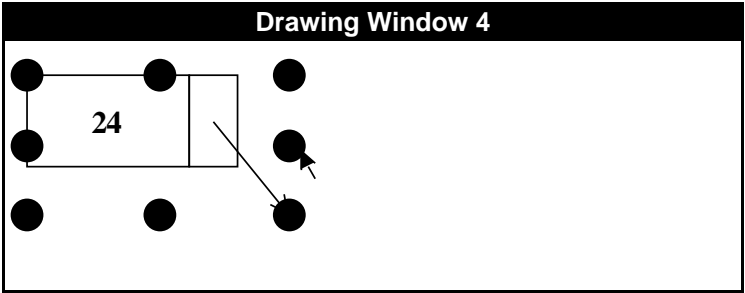
14.6 Editor Menu Commands

The commands in Lapidary's pull down menu (Figure [lapidary-editor-menu], page 655) provide a set of commands for saving and restoring objects, manipulating aggregadgets, applying constraints, and editing properties.

14.7 File

(Save Gadget:) Objects are written out using (opal:write-gadget), so the file contains a series of create-instance calls. The value in the object's (:known-as) slot is passed as the name parameter to create-instance. For example, if the object's (:known-as) slot is (:white-rect) and the object is a rectangle, the first line of the create-instance would be

```
(create-instance 'white-rect opal:rectangle)
```

Primary selections are saved before secondary selections, so it is best to make prototypes primary selections and instances of these prototypes secondary selections. The user can also save an entire window by having no objects selected and typing in the string that appears in a window's title bar or icon in the corresponding area of the dialog box.

Lapidary looks at each saved object to determine if the object has any links which Lapidary thinks should be parameters. If Lapidary finds any such links, it pops up the link parameters dialog box and asks the user if these links should be made into parameters (see Section [parameters], page 674). Pressing either the (OK) or (CANCEL) buttons in the link parameters dialog box allows Lapidary to continue. The (CANCEL) button in the link parameters dialog box will not cause Lapidary to discontinue the save operation, it will simply cause Lapidary to proceed to the next object.
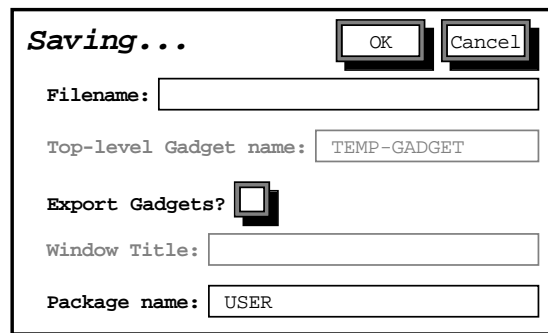
Figure 14.8: Save file dialog box

(Load Gadget:) Requests the name of a file and then loads it (Figure [load-dialog-box], page 666). Lapidary expects a variable named *Garnet-Objects-Just-Created* to be initialized in the user package which contains the names of the created objects. If the user selects the option Replace existing objects, then the objects in the loaded file

will replace the current objects in the drawing window. If the user selects the option
`Add to existing objects`, then the objects in the file will be added to the existing
objects in the window.
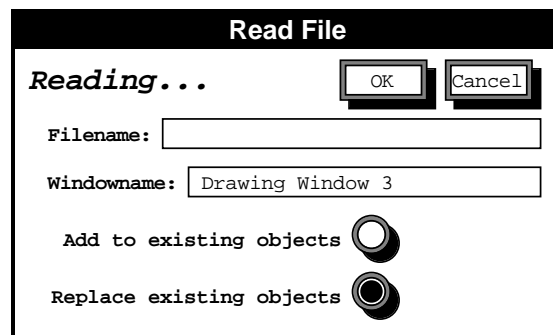
Figure 14.9: Load file dialog box

(Add Gadget:) Users may create objects in the lisp listener and then link them to a Lapidary window. (add-gadget) pops up a dialog box that requests the name of the object to be added and the name of a window to place the object in (Figure [add-gadget-fig], page 668). The name of the object should be the one used in the call to create-instance. For example, the object created by `(create-instance 'my-gadget opal:rectangle)` is named "my-gadget". The name of the window should be the name that appears in the window's title bar or in its icon.
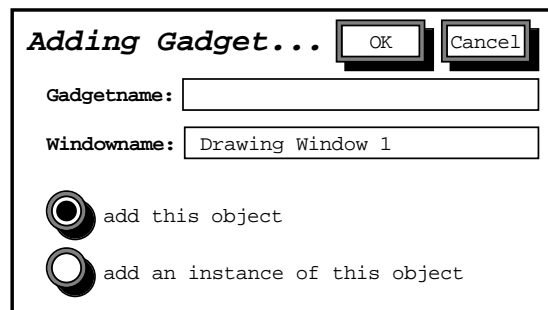
Figure 14.10: Add gadget dialog box

The user has the option of either adding the object itself or an instance of the object to Lapidary. If the user decides to add the object itself and the object has instances, Lapidary will pop up a warning box indicating that editing this object could have unintended consequences on other applications that use this object. For example, it

is better to add an instance of a garnet gadgets text button rather than the actual button defined in the gadgets package, since editing the actual button is likely to cause Lapidary to fail (Lapidary uses garnet gadgets text buttons).

(Quit:) Allows the user to exit Lapidary. It is suggested that before rebooting Lapidary, that the user create a new lisp listener and reload Garnet.

14.8 Edit

(Make Instance:) Creates an instance of the selected object. The selected object is the new object's prototype.

(Make Copy:) Creates a copy of the selected object. The value of each slot in the selected object will be copied to the new-object. The new object will have the same prototype as the selected object, and thus will inherit from the selected object's prototype rather than the selected object.

(Delete Object:) Destroys all selected objects.

(Delete Window:) Pops up a dialog box and asks the user to input the name of a window that appears in a window's title bar or icon. Lapidary then destroys the window.

14.9 Properties

Lapidary contains four property menus that control an object's line-style, filling-style, draw-function, and font. The line-style and filling-style menus (Figures [shade-menu], page 670, and [line-menu], page 671) provide a set of commonly used styles, an "Other" option which prompts the user for the name of a style, and a "Constraint" option that allows the user to enter a custom constraint that defines the style (see Section [constraints], page 683, for information on how to enter a custom constraint). The color button pops up a color menu that allows the user to select a pre-defined color or create a new color by mixing hues of red, green, and blue.

(Filling Style:) Allows the user to set the filling style of selected objects.
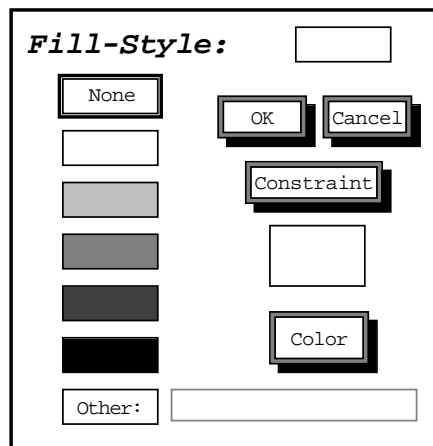
Figure 14.11: Filling styles that can be attached to objects in Lapidary

(Line Style:) Allows the user to set the line style of selected objects.
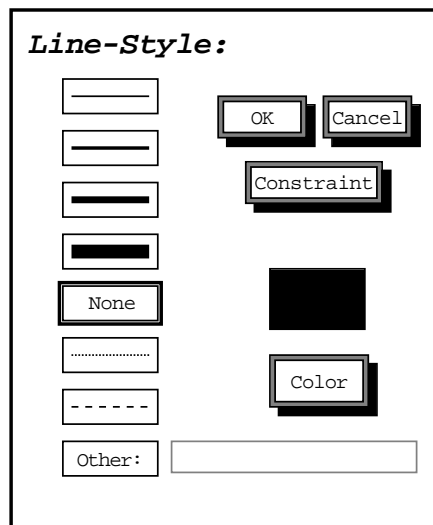
Figure 14.12: Line styles that can be attached to objects in Lapidary

(Draw Function:) Allows the user to set the draw function of all selected objects. The Opal chapter describes draw functions in more detail.

Draw Function [OK] [Apply] [Cancel]

Copy ⦿   Xor ○   And ○   Or ○

Others
Clear ○	Set ○	No-Op ○
Copy-Inverted ○	Invert ○	Equiv ○
Nand ○	Nor ○	And-Inverted ○
And-Reverse ○	Or-Inverted ○	Or-Reverse ○
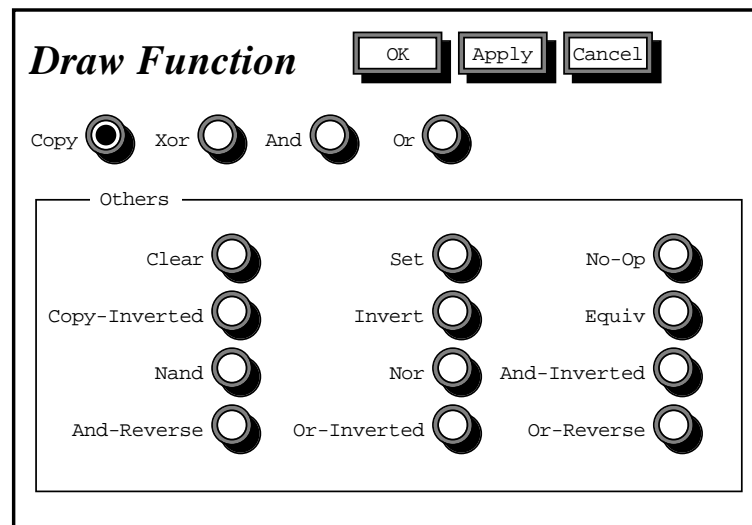
Figure 14.13: Draw functions that can be attached to objects in Lapidary

(Name Object:) Requests a name from the user (no quotes should be used), converts it to a keyword, and stores it in the :known-as slot of the selected object (if there is more than one selected object, Lapidary will rename the last object the user selected; name object does not distinguish between primary and secondary selections). Lapidary also

creates a link with this name in the object's parent that points to this object. When an object is saved, it will be assigned this name.

(List Properties:) Brings up a property list for horizontal and vertical lists. This property list allows the user to modify any of the customizable slots of an aggrelist. The list of customizable slots can be found in the Aggrelists chapter.

(Text Properties:) Allows the user to choose a standard Opal font, to request a font from one of the directories on the user's font path, to request a font from an arbitrary directory, or to enter a custom constraint that determines the font (Figure [text-menu], page 674). It also allows the user to enter a custom constraint that determines the string of a text object.

Figure 14.14: Lapidary's text properties menu

(Parameters:) Allows the user to specify that one or more slots in an object should be parameters (Figure [parameters-fig], page 676). A slot that is a parameter will have its value provided at run-time by the application. To create parameters, the user must make both a primary and a secondary selection. The primary selection is the

object whose slots are being made into parameters and the secondary selection is the object that the parameters will retrieve their values from. Typically the secondary selection will be the top-level aggregadget that contains the object, since the top-level aggregadget is the only object that the application should know about (an application should not be required to know the parts of an aggregadget). For example, if a label text object belongs to an aggregadget, the user might make the label the primary selection and the aggregadget the secondary selection. If the object is already at the top-level, then the object should be both the primary and secondary selection.

```
                          Parameters

   If a slot should be a parameter, enter the name of
   the slot in the secondary selection that it should
   retrieve its value from. If you do not want a slot
   to be a parameter, just leave the text box next to
   it blank (or make it blank it it currently contains
   the name of a slot.


   :FONT              [ :font ]

   :STRING            [ :value ]

   :LINE-STYLE        [  ]

   :DRAW-FUNCTION     [  ]


   Select the following button to get a list of objects
   that are referenced by the primary selection, and
   which lapidary thinks should be parameters. If there
   are no such objects, nothing will appear.


   [ link parameters ]


              [ OK ]   [ Cancel ]
```

Figure 14.15: Parameters dialog box

To turn a slot into a parameter, select the text box next to the slot and enter the name of
the slot in the secondary selection that the slot should retrieve its value from. In Figure
[parameters-fig], page 676, the label's (string) slot retrieves its value from list element's
(value) slot, and the (font) slot retrieves its value from the list element's (font) slot.

To make the slot no longer be a parameter, make the slot's text box be blank. Lapidary maintains a list of slots for each objects that can be turned into parameters. If the user wants to parameterize a slot that is not displayed in the parameters dialog box, the user can bring up C32 and place a formula in the desired slot that retrieves its value from the top-level aggregadget.

link parameters

object: TRANSPARENT-THIN-ARROW-LINE-25038 [OK] [Cancel]

parameters object: TRANSPARENT-THIN-ARROW-LINE-25038

Some of the slots in the object reference objects that might be parameters. These objects
are listed below, along with the slots that reference them. If you make any of these objects
into parameters, Lapidary will create formulas that cause the slots to retrieve the parameters
from the object labeled 'parameters object'.

If you want to see the object labeled 'object:', select the object's name with the left mouse
button and the object will be flashed on the display.

If you want to make an object a parameter, enter the parameter name in the blank box next to each
object. This name must be a slot name (e.g., :string). If the user has already provided a name,
Lapidary puts it in the box.

If you do not want an object to be a parameter, just leave its parameter box blank (or make it
blank if Lapidary has put a name in there).

If you want to see one of the referenced objects, select the object's name with the left mouse
button and the object will be flashed on the display.

For more information on this operation, see the Lapidary reference manual.

object referenced by	slots	parameter name
TRANSPARENT-THIN-LINE-RECTANGLE-25026	:X1 :Y1	:from-obj
TRANSPARENT-THIN-LINE-RECTANGLE-25028	:X2 :Y2	:to-obj

Figure 14.16: Link parameters dialog box

The link parameters button in the parameters dialog box allows the user to specify
links that should be parameters. Links are used by Lapidary-generated constraints
to indirectly reference other objects. For example, when the user creates a constraint
that attaches the endpoint of a line, say (arrow1) to a rectangle, say (rect1), Lapidary

generates a link in (arrow1) that points to (rect1). When a link references an object that is not part of the primary selection's top-level aggregadget, Lapidary guesses that this link should be a parameter and displays it in the link parameters dialog box (Figure [link-parameters], page 678). For each such link, Lapidary displays the value of the link, the slots that reference the link, and a parameter name, if any, that the user has assigned to this link. The user can change this parameter name by editing it, or can indicate that this link should not be a parameter by making the parameter name blank.

```
                              Parameters

       If a slot should be a parameter, enter the name of
       the slot in the secondary selection that it should
       retrieve its value from. If you do not want a slot
       to be a parameter, just leave the text box next to
       it blank (or make it blank it it currently contains
       the name of a slot.


       :FONT                   :items

       :STRING                 :items

       :LINE-STYLE

       :DRAW-FUNCTION


       Select the following button to get a list of objects
       that are referenced by the primary selection, and
       which lapidary thinks should be parameters. If there
       are no such objects, nothing will appear.


       link parameters



                   OK      Cancel
```

Figure 14.17: Parameters dialog box for an Aggrelist

To make a slot depend on an :items list in an aggrelist, make any object in the aggrelist be a primary selection and make the aggrelist be the secondary selection. Then enter :items in the labeled box for any slot on the parameters menu that should get its value from :items. For example, suppose the prototype object for a list is a text object

and the string and font slots of the text object should retrieve their values from the aggrelist's (:items) slot. To do this the user makes the aggrelist the secondary selection and one of the text objects in the aggrelist a primary selection. The user then selects the (parameters) option, which causes Lapidary to pop up the parameters menu. Typing :items in the type-in fields next to the (font) and (string) slots creates the necessary formulas that link these slots to the (:items) slot in the aggrelist (Figure [paramItems-fig], page 680). The :items slot of the aggrelist will now contain a list of the form ((string1 font1) (string2 font2)...(stringN fontN)).

If the prototype object is an aggregadget (such as a labeled box that contains a rectangle and a piece of text), then any of the parts of the aggregadget, and the aggregadget itself, can have slots that depend on the aggrelist's :items slot. This is done by parameterizing the parts one at a time. For example, if the string slot of the text object and the filling-style slot of the rectangle should be parameters, the user could first select the rectangle and parameterize it, then select the text object and parameterize it. Lapidary does not follow any easily described rules in constructing the :items list (e.g., the string and font values could easily have been reversed in the above list), so users should look at the :items list Lapidary constructs before writing their own.

If a slotname besides :items (e.g., :string) is entered in a type-in field, then the slot is treated as an ordinary parameter, and all items in the list will have a formula that accesses this slot in the aggrelist. For example, if a list consists of rectangles, and the rectangles should all have the line-style that is passed to the aggrelist, then the user would select one of the rectangles and enter the an appropriate name, such as (:line-style), next to the :line-style slot of the rectangle.

14.10 Arrange

(Bring to Front:) Brings the selected objects to the front of their aggregadget (i.e., they will cover all other objects in their aggregadget). If multiple objects are selected, it brings the objects to the front in their current order.

(Send to Back:) Sends the selected objects to the back of their aggregadget (i.e., they will be covered by all other objects in their aggregadget). If multiple objects are selected, it sends the objects to the back in their current order.

(Make Aggregadget:) Creates a new aggregadget and adds all selected objects (both primary and secondary selections) to it. The selected objects must initially belong to the same aggregadget or else Lapidary will print an error message and abort the operation. The (:left) and (:top) slots of the objects added to the aggregadget are constrained to the aggregadget unless they were already constrained (if the object is a line, the (:x1), (:y1), (:x2), and (:y2) slots are constrained). The constraints tie the objects to the northwest corner of the aggregadget and use absolute offsets based on the current position of the objects. Thus if an object is 10 pixels from the left side of the aggregadget (the bounding box of the aggregadget is computed from the initial bounding boxes of the objects), the object's (:left) slot will be constrained to be 10 pixels from the left side of the aggregadget. If the object is a line, the object's endpoints will be tied to the aggregadget's northwest corner by absolute fixed offsets. These constraints cause the objects to move with the aggregadget when the aggregadgets moves. If the user wants different constraints to apply, the user can primary select an object,

secondary select the aggregadget, and attach a different constraint. The aggregadget derives its width and height from its children, so the :width and :height slots of the children are not constrained to the aggregadget. Because the aggregadget computes its width and height from its children, it is not permitted to resize an aggregadget.

(Ungroup:) Destroys selected aggregadgets and moves their components to the aggregadgets' parents.

14.11 Constraints

(Line Constraints:) Brings up the line constraints dialog box (Figure [line-constraint], page 687).

(Box Constraints:) Brings up the box constraints dialog box (Figure [box-constraint], page 684)

(C32:) Brings up C32. Each primary and secondary selection is displayed in the spreadsheet, and additional Lapidary objects can be displayed using the (Point to Object) command. While Lapidary is running, only objects in Lapidary's drawing windows can be displayed in the spreadsheet. Nothing will happen if the user attempts to execute the (Point to Object) command on an object which is not in a Lapidary drawing window. The C32 chapter describes how to use C32 and Section [custom-constraint], page 688, describes a number of modifications Lapidary makes to C32.

14.12 Other

(Interactors:) Displays a menu of interactors that the user can choose to look at. Once the user selects an interactor, the information from that interactor will be displayed in the appropriate interactor dialog box (see Section [interactors], page 693) and the user is free to change it. In addition, menu items are provided for the five Garnet-defined interactor types: choice (encompassing both menu and button interactors), move/grow, two-point, text, and angle. If the user has selected a set of objects, then the interactors menu will contain all interactors associated with these objects. Lapidary will display all interactors whose (:start-where) slot references these objects, or whose (:feedback-obj) or (:final-feedback-obj) points at these objects. If no objects are selected, then the interactors menu will contain all interactors that have been created in Lapidary.

(Clear Workspace:) Deletes all objects from Lapidary but does not destroy any of the drawing windows.

14.13 Test and Build Radio Buttons

(Test:) Deactivates the Lapidary interactors that operate on the drawing windows and activates all user-defined interactors. This allows the user to experiment with the look-and-feel that the user has created.

(Build:) Deactivates all user-defined interactors and reactivates the Lapidary interactors, allowing the user to modify the look-and-feel.

14.14 Creating Constraints

Lapidary provides two menus for creating constraints, one that deals with "box" constraints (constraints on non-line objects) and one that deals with line constraints. In addition, several of the property menus provide a custom constraint option that allows the user to input a constraint that determines the property. Each of the menus contains buttons labeled with tiny rectangular boxes that indicate how an object will be positioned if the constraint associated with that button is chosen. The rectangular boxes in the buttons are colored black to indicate that the primary object is the object that will be constrained, and the white rectangular boxes positioned at the four corners of the rectangle in the box constraint menu indicate that the secondary selection is the object that will be referenced in the constraint.

The Box and Line Constraint dialog boxes, can be used separately from Lapidary (see section [constraint-gadget], page 689).

The constraint menus can display the current position and size of a primary selection. By pressing the (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.

14.15 Box Constraints

The box constraint menu allows constraints to be attached to the (:left), (:top), (:width), and (:height) of an object (see Figure [box-constraint], page 684). The user attaches constraints by first selecting the object to be constrained (a primary selection) and the object to be referenced in the constraint (a secondary selection). The user then selects the appropriate buttons in the box constraint menu. The possible constraints for the (:left) slot are:
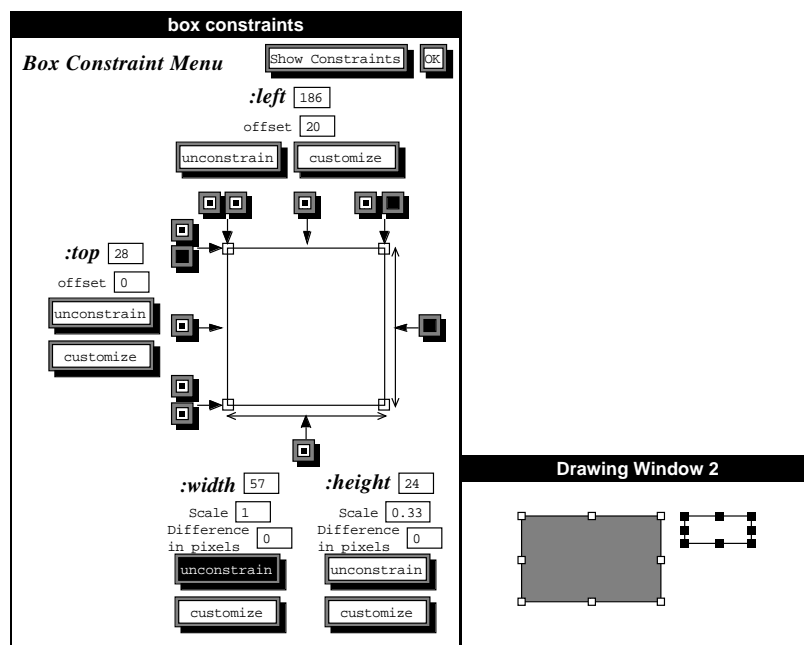
Figure 14.18:

The constraint 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.

left-outside: The right side of the primary selection is aligned with the left side of the secondary selection.

left-inside: The left side of the primary selection is aligned with the left side of the secondary selection.

center: The center of the primary selection is aligned with the center of the secondary selection.

right-inside: The right side of the primary selection is aligned with the right side of the secondary selection.

right-outside: The left side of the primary selection is aligned with the right side of the secondary selection.

The possible constraints for the (:top) slot are:

top-outside: The bottom side of the primary selection is aligned with the top side of the secondary selection.

top-inside: The top side of the primary selection is aligned with the top side of the secondary selection.

center: The center of the primary selection is aligned with the center of the secondary selection.

bottom-inside: The bottom side of the primary selection is aligned with the bottom side of the secondary selection.

bottom-outside: The top side of the primary selection is aligned with the bottom side of the secondary selection.

The only option for the (:width) slot is to constrain the width of the primary selection to the width of the secondary selection and the only option for the (:height) slot is to constrain the height of the primary selection to the height of the secondary selection. In addition, each of the four slots may have a custom constraint attached to them (see Section [custom-constraint], page 688). Each of the four slots also has an "Unconstrain" option that destroys the constraint attached to that slot.

The constraints in the box constraint menu can be fine-tuned by entering offsets for the constraints, and in the case of the size slots (width and height), scale factors as well. When an object is centered with respect to another object, the offset field changes to a percent field denoting an interval where 0% causes the center point of the constrained object to be attached to the left or top side of the object referenced in the constraint and 100% causes the center point of the constrained object to be attached to the right or bottom side of the object referenced in the constraint. By default this percentage is 50. The (Difference in pixels) and (Scale) factors cause the width and height constraints to be computed as *Scale * Dimension + Difference in pixels*.

Finally, each of the slots has a labeled box next to its name that allows the user to type in an integer that will be placed in that slot. If there is already a constraint in the slot, the constraint will not be destroyed so the value will only temporarily override the value computed by the constraint (the next time the constraint is recomputed, the value will be lost). This operation works only when there is one primary selection and no secondary selections.

14.16 Line Constraints

The line constraint menu allows the endpoints of a line to be attached to other objects or the (:left) and (:top) slots of a box object to be constrained to the endpoint of a line (Figure [line-constraint], page 687). The buttons on the box and line object in Figure [line-constraint], page 687, indicate the various locations where the endpoint of a line can be attached to a box or line object or where a point of a box can be attached to a line. Thus the two endpoints of a line can be attached to any of the corners, sides, or center of a box object and any of the corners, sides, or center of a box object can be attached to the endpoints or center of a line.
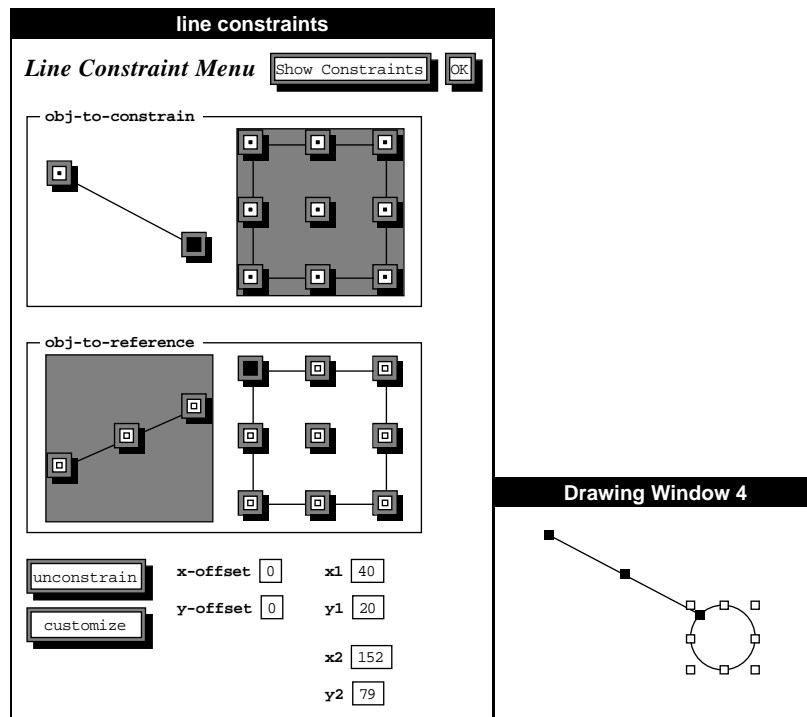
Figure 14.19:

The constraint 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.

The line object in the constraint menu is oriented in the same direction as the selected line in the drawing window, so that the user knows which endpoint is being constrained. The buttons with the blackened rectangles indicate the points that can be constrained in the primary selection. Similarly, the buttons with the white rectangles indicate the points in the secondary selection that the primary selection can be attached to.

The (Unconstrain) button at the bottom of the menu allows the user to destroy the constraint on the selected point and the (Customize) button allows the user to input a custom constraint (described in Section [custom-constraint], page 688). Finally the (x-offset) and (y-offset) labeled boxes allow the user to enter offsets for the constraint. All offsets are added to the value computed by the constraint. For example, an x-offset of 10 causes an endpoint constrained to the northwest corner of a box object to appear 10 pixels to the right of that corner.

The end points of a line can be set directly by typing in values for x1, y1, x2 and y2. This function is only active if there is a single line as the primary selection and no secondary selections.

14.17 Custom Constraints

When the user selects the custom constraint option in any of the constraint or property menus, Lapidary 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.

Information on C32 can be found in the C32 chapter. However, Lapidary 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 Lapidary will automatically generate indirect references and create appropriate link names. The user can edit these link names by bringing up the parameters menu and hitting the link parameters button (see Section [parameters], page 674). 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, Lapidary 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, Lapidary may not be able to generalize it, so the formula may behave strangely if it is inherited.

The second change Lapidary makes is in copying formulas. Lapidary 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 chapterly 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).

14.18 The Constraint Gadget

The constraint menus have been bundled into a constraint gadget that can be used independently of Lapidary. The constraint gadget provides two menus: a box constraint menu for specifying box-type constraints (Figure [box-constraint], page 684) and a line constraint menu for specifying line-type constraints (Figure [line-constraint], page 687). These menus operate as described in Sections [box-constraint-section], page 683, and [line-constraint-section], page 686. The menus also provide access to C32 through (customize) buttons. The module can be loaded independently of Lapidary with `(garnet-load "lapidary:constraint-gadget-loader")` and is exported from the gadgets package (garnet-gadgets).

14.18.1 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 [Functions], page 690. Certain slots of the gadget, described in section [cg-parameters], page 689, 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.

14.18.2 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 ⟨undefined⟩ [functions], page ⟨undefined⟩, for details on the c32-custom-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.

14.18.3 Exported Functions

The following functions are exported from the constraint gadget module:

    gg:Box-Constraint-Do-Go [No value for ''function'']

    gg:Line-Constraint-Do-Go [No value for ''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:

    gg:Show-Box-Constraint-Menu [No value for ''function'']

    gg:Show-Line-Constraint-Menu [No value for ''function'']

These functions 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], page 90
           &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.

prompt - A text string that should be displayed in the query box that appears when C32 is invoked.

gg:CG-Destroy-Constraint *object   slot* [function], page 90

This function destroys a constraint created by the constraint gadget. Required parameters are an object and a slot.

```
@emph{destroy-constraint-support-slots}: destroys the slots in the :links,
:offset, and :scale slots of a formula (these are the
standard support slots created by the constraint gadget).
A slot is destroyed only if the formula is the only formula
that depends on this slot. Required parameters are:

                object: The object which contains the formula.
                formula: A formula.

                An optional parameter is:

                destroy-meta-info-p: If this parameter is true, the meta
                        slots :links, :offset, and :scale are destroyed in
                        the formula. This parameter defaults to @code{nil}.
```

gg:Valid-Integer-P *gadget string* [function], page 90

`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.

c32::Install-Links *formula   obj* [No value for ``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 *obj* parameter names the object which the links should be installed in.

14.18.4 Programming with Links

Each constraint contains indirect references to objects rather than direct references. The set of link names 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`.

```
The rationale for storing slot names rather than actual values in a
formula's meta-slots is as follows:

@itemize

link slots: Storing the real objects pointed to by the constraint would
be difficult because of inheritance. When a formula was inherited,
it would have to change the object it was pointing to. If the names
of link slots are stored, it can use the link slot to get the real
object. The link slot presumably will have been made to point to
the appropriate object. Also, storing the name of the link slot
makes it easier for an application to change the link names in the
formula, if the application is so inclined.

offset and scale slots: Storing the names of the offset and scale slots
serves two purposes: 1) the application can immediately find out
what the constraint is calling the offset and scale slots and
change them if necessary (the user may not want to call an offset
slot, :left-offset); and 2) when an offset or scale changes, the
change only has to be made in one place rather than two places.
For feedback purposes, the application can retrieve the offset and
scale indirectly using the names of these slots.
```

14.18.5 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 chapterly 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).

14.18.6 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.

14.19 Interactors

Lapidary provides a set of dialog boxes that allow a user to define new interactors or modify existing ones. To create or modify an interactor, select the (Interactors) command from the Lapidary editor menu. Lapidary will display a menu listing Garnet-defined and user-defined interactors that may be viewed. Select the desired interactor and Lapidary will display the appropriate interactor dialog box.

All interactor dialog boxes have a number of standard items, including a set of action buttons, a name box, a (:start-where) field, and buttons for events. In addition, each dialog box allows the user to set the most commonly changed slots associated with that interactor. Other slots may be set using C32 (see section [custom-constraint], page 688).

The name field allows the user to type in a name for the interactor. The name is not used to name the interactor, but instead is converted to a keyword and stored in the interactor's :known-as slot. If the interactor is saved, the user-provided name will be placed in the name parameter field for create-instance.

14.20 Action Buttons

The action buttons permit the following types of operations:

Create Instance: This operation creates an instance of the displayed interactor and, if the user has modified any of the slot values, overrides the values inherited from this

interactor with the modified values. In addition, Lapidary examines the (:start-where) field of the new interactor and if the start-where includes an aggregadget, adds the interactor to the aggregadget's behavior slot.

Modify: This operation stores any changes that the user has made to the interactor's slots in the interactor.

Destroy: This operation destroys the interactor.

Save: This operation prompts the user for a file name and then writes out the interactor.

C32: This operation brings up C32 and displays the interactor in the spreadsheet window. The user can then edit any slot in the interactor. Any changes the user makes will not be discarded by the (Cancel) button. It is generally advisable to bring up the C32 menu only *after* the interactor has been created. (the one exception to this rule is when C32 appears as the result of pressing a formula button. If the user enters a formula in the formula window, the formula will be installed in the instance). Otherwise the user will end up editing the prototype for the interactor to be created, instead of the interactor itself. The C32 chapter describes how to operate C32 and Section [custom-constraint], page 688, describes the modifications Lapidary makes to C32.

Cancel: This operation discards any changes the user has made to the dialog box since the last create-instance or modify command.

14.21 Events

Lapidary allows the user to define the start, stop, and abort events of an interactor using event cards. Each card defines one event and a list of events can be generated from a deck of cards. Each interactor dialog box contains buttons that pop up a window for each event that defines a start, stop or abort event. A sample event card is shown in Figure [cards], page 695. Selecting (Delete this event) will cause this event to be deleted. However, Lapidary will not allow you to delete an event card if it is the only one that exists. (Add an event) causes a new event to be created. (OK) makes the window disappear and generates the event list for the desired event.

Figure 14.20: A sample event card deck

Any combination of ⟨shift⟩, ⟨control⟩, and ⟨meta⟩ can be selected, but if the ⟨any modifier⟩ button is selected then the other modifier buttons will become unselected. The mouse actions and keyboard items are all mutually exclusive, so selecting one will cause

the previously selected item to be deselected. Events like #\Return can be generated by simply typing "Return" in the (Specific keypress) box (quotes are not needed).

14.22 :Start Where

Every interactor dialog box displays two commonly used start-wheres for an interactor and allows the user to select an alternative one using the (other) button (Figure [choice-inter], page 697). If (other) is selected, a dialog box will appear which lists all possible :start-where's. Once the desired start-where is selected, Lapidary will incorporate the selected object in the drawing window into the start-where if it is appropriate (which it is in all cases but (t) and (nil)). If the start-where requires a slot (which the (list) start-where's do), Lapidary will request the name of a slot.

If the user wants a type restriction, then pressing the (type restriction) button will cause Lapidary to request a type restriction. A type restriction can be either an atom (e.g., opal:text) or a list of items (e.g., (list opal:text opal:rectangle)). The type restriction button is a toggle button so if it is already selected, selecting it again will cause the type restriction to be removed. Also, selecting a new start-where will cause the type restriction to be removed.

14.23 Formulas

Selecting a formula button in any of the interactor dialog boxes causes the interactor to be displayed in the C32 spreadsheet window and the current value of the slot associated with the formula button to be displayed in a C32 formula window. This value can then be edited into a formula. When the (OK) button is pressed in the formula window, C32 disappears and the formula is batched with the other changes that have been made to the interactor since the last (Create Instance) or (Modify) command. The formula is not actually installed until the (Create Instance) or (Modify) buttons are selected. If the user selects (Cancel) in the interactor dialog box, the formula will be discarded. The formula will also be discarded if the user selects (Cancel) in the C32 formula window.

14.24 Specific Interactors

14.24.1 Choice Interactor

The choice interactor dialog box allows the user to create either a button interactor or menu interactor, depending on whether the (menu) or (button) radio button is selected (Figure [choice-inter], page 697). The other slots that can be set using this dialog box are:

Figure 14.21: Choice interactor dialog box

:start-where. If the user selects either (aggregadget of items) or (single item) and there
is a least one selection in a drawing window (it may be either a primary or secondary

selection), then start-where's with (:element-of) and (:in-box) are generated with the selected object.

:feedback-obj. Selecting the radio button associated with (interim feedback) will cause the selected object in the Lapidary drawing windows to become the interim feedback for this interactor. If this object is constrained to one of the objects that satisfies the start-where or to a component of one of these objects, Lapidary will automatically generalize the constraints so that the object can appear with any of the objects in the start-where.

The user can also use the (by-demo) option to demonstrate interim feedback. Lapidary will pop up an OK/Cancel box when an object that satisfies the start-where is selected. The user can then use the various Lapidary menus to modify this object so that it looks as it should when the object's (:interim-selected) slot is set. Once the desired look is achieved, the user selects OK and the changes will be installed so that the object looks like its original self when it is not interim selected, and will look like the by-demo copy when it is interim selected.

Lapidary implements the by-demo operation by comparing the values of the following slots in the original object and the copied object: `:left`, `:top`, `:width`, `:height`, `:visible`, `:draw-function`, `:font`, `:string`, `:line-style`, `:filling-style`, `:x1`, `:x2`, `:y1`, `:y2`.

The last option the user can choose is (none) in which case `nil` will be stored in the (:feedback-obj) slot. This will not undo the effects of a by-demo operation since by-demo also places `nil` in the (:feedback-obj) slot.

:final-feedback. The options for final feedback are identical to those for (:feedback-obj). The by-demo changes will appear when the object's (:selected) slot is set to (t). Multiple final feedback objects can be created by selecting several objects and pressing the final feedback button. Lapidary will then bring up C32 and prompt the designer for a constraint that determines when to use each kind of feedback object at run-time.

:final-function. The user can type in the name of a function that should be called when the interactor completes.

:how-set. The user can set the (:how-set) slot by selecting a radio button or entering numbers in the (increment-by) and (optionally) (max value) fields.

14.24.2 Move/Grow Interactor

The move/grow interactor dialog box (Figure [lapidary-move-inter], page 700) allows the user to specify a move/grow interactor. The slots that can be set using this dialog box are:

:start-where. If the user selects either (Object to Press Over) or (One of This Aggregate) and there is at least one selection in a drawing window (it may be either a primary or secondary selection), then start-where's with :in-box and :element-of are generated with the selected object.

:line-p. This slot is set by the (Line) and (Box) buttons. If a formula is selected, the formula should return (t) if the interactor is moving/growing a line, and (nil) if it is moving/growing a box object.

:grow-p. This slot is set by the (Grow) and (Move) buttons. If a formula is selected, the formula should return (t) if the interactor is growing an object, and (nil) if it is moving an object.

:min-length. Specifies a minimum length for lines.

:min-width. Specifies a minimum width for box objects.

:min-height. Specifies a minimum height for box objects.

:obj-to-change. The user can let the move/grow interactor modify the object that satisfies the start-where, present an example object to change to Lapidary or use a formula to compute the object to change. This slot would be set if the interaction should start over a feedback object such as selection handles, but should actually move the object under the feedback object.

If the user presents an example object to change to Lapidary by selecting an object and pressing the (Change this object) button, Lapidary will automatically construct a formula so that the interactor changes the correct object at run-time. For example, in Figure [lapidary-move-inter], page 700, 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. 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.

:final-function. The user can type in the name of a function that should be called when the interactor completes.

:feedback-obj. An interim feedback object can be created by creating the desired object and pressing the interim-feedback button. Constraints will be automatically attached to the feedback object that cause it to move or grow appropriately, and that make it visible/invisible at the appropriate times. If multiple objects are selected, Lapidary will bring up C32 and prompt the designer for a constraint that determines when to use each kind of feedback object at run-time.

:attach-point. Controls where the mouse will attach to the object.

The grow and move parameters allow the user to control which slots in the object that is being grown or moved will actually be set. If a formula is entered, it must return a value that can be used by the slot :slots-to-set (see the Interactors chapter for more details on this slot).

Example: To create an interactor that moves a box,

Create the box and leave it selected

Select interactors from the editor menu and then select move/grow

In :start-where click on "Object to Press Over". This will cause the selected rectangle's KR name to be displayed.

Press the CREATE INSTANCE action button

To test the interactor press test button in the editor menu and drag it around.

objects to start over

interim feedback

obj-to-change
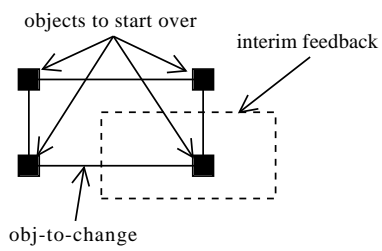
Move/Grow Interactor	

Move/Grow Interactor

Interactor Name: MYMOVEINTER

:start-where

Object to Press Over

or

One of This Aggregate   BOX-SELECTION-HANDLES

or

Other   Type restriction:

CREATE INSTANCE

MODIFY

DESTROY

SAVE

C32

CANCEL

14.24.3 Two Point Interactor

The two point interactor dialog box (Figure [two-point-inter], page 702) allows a user to create a two point interactor. The slots that can be set using this dialog box are:

:start-where. If the user selects (Start Anywhere in Window) then a start-where with (t) is generated. if the user selects (Start in Box) and there is at least one selection in a drawing window (it may be either a primary or secondary selection), then a start-where with :in-box is generated with the selected object.

:line-p. This slot is set by the buttons (Create Line) and (Create Non-Line). If a formula is selected, the formula should return (t) if the interactor is creating a line, and (nil) if it is creating a box object.

:min-length. Specifies a minimum length for lines.

:min-width. Specifies a minimum width for box objects.

:min-height. Specifies a minimum height for box objects.

:flip-if-change-side. Indicates whether a box may flip over when it is being created.

:abort-if-too-small. Indicates whether the operation should be aborted if the object is too small or whether an object of the minimum size should be created.

:feedback-obj. An interim feedback object can be created by creating the desired object and pressing the interim-feedback button. Constraints will be automatically attached to the feedback object that cause it to sweep out as the mouse cursor is moved, and that make it visible/invisible at the appropriate times. If multiple objects are selected, then Lapidary will bring up C32 and prompt the designer for a constraint that determines when to use each kind of feedback object at run-time. If the standard feedback option is selected, a box or line feedback object is automatically created according to whether a line or box is being created.

:final-function. The user can type in the name of a function that should be called when the interactor completes.

Example: To create a two-point interactor with line feedback

1) select the interactors option from (other) in the editor menu and then select two point interactor in the menu that pops up

2) click on Start Anywhere in Window

3) click on Create Line

4) click on Standard Feedback

5) click on CREATE INSTANCE

To test this interactor, enter test mode, press down on the left mouse button, and sweep out a line. No line will be created because a final function was not provided.

Figure 14.23: Two point interactor dialog box

14.24.4 Text Interactor

The text interactor dialog box (Figure [text-inter], page 703) allows the user to create or modify a text interactor and to edit the following slots:

Figure 14.24: Text interactor dialog box

:start-where. If the user selects either (object to press over) or (one of this aggregadget) and there is at least one selection in a drawing window (it may be either a primary or

secondary selection), then start-wheres with (:in-box) and (:element-of) are generated with the selected object.

:obj-to-change. The user can either let the text interactor modify the object that satisfies the start-where or use a formula to compute the object to change. Lapidary can construct a formula for this slot if necessary (see section [move-grow-sec], page 698).

:feedback-obj. An interim feedback object can be created by creating a text object and pressing the interim-feedback button. Constraints will be automatically attached to the feedback object that cause it to appear at the selected text object and that make it visible/invisible at the appropriate times. If multiple objects are selected, then Lapidary will bring up C32 and prompt the designer for a constraint that determines when to use each kind of feedback object at run-time.

:cursor-where-press. This slot is set by the buttons (where pressed) and (at end of string). If (where pressed) is selected, the text editing cursor will appear under the mouse cursor. If (at end of string) is selected, the text editing cursor will always appear at the end of the string when editing starts.

:final-function. The user can type in the name of a function that should be called when the interactor completes.

14.24.5 Angle Interactor

The angle interactor dialog box (Figure [angle-inter], page 705) allows the user to create and modify an angle interactor. The slots that can be set by this dialog box are:

:start-where. If the user selects (object to press over) and there is a least one selection in a drawing window (it may be either a primary or secondary selection), then a start-where with (:in-box) is generated. If the user selects (start anywhere in window), then a start-where of (t) is generated.

:obj-to-change. The user can either let the angle interactor modify the object that satisfies the start-where or use a formula to compute the object to change.

:feedback-obj. An interim feedback object can be created by creating an object, selecting it, and pushing the interim feedback button. The :angle slot of the object will be set as the interactor is operated and the object will be made visible/invisible as appropriate. To make the feedback object or the object that gets the final angle change in response to changes in the :angle slot, custom constraints must be created for the position and size slots. See the angle interactor section in the Interactors chapter for sample constraints. If multiple objects are selected, then Lapidary will bring up C32 and prompt the designer for a constraint that determines when to use each kind of feedback object at run-time.

:final-function. The user can type in the name of a function that should be called when the interactor completes.

:center-of-rotation. This is the center of rotation for the interaction. The user can either enter a list of (x,y), enter a formula that returns a list of (x,y) or select one of the standard locations for the center of rotation by selecting the appropriate button.

Figure 14.25: Angle interactor dialog box

14.25 Getting Applications to Run

Lapidary-generated files consist of a set of create-instance calls. The objects created are stored in a list and assigned to the variable *Garnet-Objects-Just-Created*. The top of a Lapidary-generated file contains code to load the lapidary-functions.lisp file, which provides functionality to support the created objects.

15 Hints on Making Garnet Programs Run Faster

Brad A. Myers

10 May 2020

15.1 Abstract

This chapter discusses some hints about how to make Garnet programs run faster. Most of these techniques should not be used until your programs are fully debugged and ready to be shipped.

15.2 Introduction

An important goal of Garnet has been to create a system that is as efficient as possible. For example, users should notice that version 2.2 is about two or three times faster than 2.1. Now that people are writing large-scale systems using Garnet, a number of things have been learned about how to make Garnet programs run faster. This chapter collects a number of hints about how to write efficient Garnet code. If you have ideas about how to make the underlying Garnet system run faster, or new hints to add to this section, please let us know.

The ideas in this chapter are aimed at producing the final production version of your system. Therefore, we feel that you should not worry about the comments here during early development. For example, turning off the debugging and testing information is likely to make your development more difficult. Also, declaring constants makes changing code more difficult. Generally, you should get your system to a fairly well-debugged state before applying these ideas.

Of course, the easiest way to make Garnet run faster is to get a faster machine and/or more physical memory. With SPARC IIs and HP Snakes becoming more prevalent, and 100 mip machines like the DEC Alpha around the corner, we see expect that the next generation of applications will have much less of a problem with achieving adequate performance.

15.3 General

Ideas in this section are relevant to any code written in Lisp, not just Garnet code. Some of these may seem obvious, but we have seen code that violates many of them.

Be sure to compile all your files.

The variable `user::*default-garnet-proclaim*`, which is defined in `garnet-loader.lisp`, provides some default compiler optimization values for Allegro, Lucid, CMU, LispWorks, and MCL lisp implementations. The default gives you fast compiled code with verbose debugging help. You can `setf` this variable before loading (and compiling) Garnet to override the default proclamations, if you want to sacrifice debugging help for speed:

```
(PROCLAIM '(OPTIMIZE (SPEED 3) (SAFETY 0) (SPACE 0)
                     (COMPILATION-SPEED 0)))
```

Fundamental changes in underlying algorithms will often overcome any local tweaking of code. For example, changing an algorithm that searches all the objects to one that

has a pointer or a hash table to the specific object can make an application practical for large numbers of objects.

Use a fast Lisp system. We have found that Allegro Version 4.2 is much faster than Allegro V3.x. Also, Allegro and Lucid are much faster than KCL and AKCL on Unix machines.

Most systems have specialized commands and features for making smaller and faster systems. For example, if you are using Allegro, check out PRESTO, which tries to make the run-time image smaller. One user reported that the "reorganizer" supplied with Lucid, the CPU time used decreased about 10-20%, and the overall time for execution dropped by about 30%. We have found that the tracing tools supplied by vendors to find where code is spending its time are mostly worthless, however.

Beware of Lisp code which causes CONS'ing. Quite often, the most natural way to write Lisp code is the one that creates a lot of intermediate storage. Unfortunately, this may result in severe performance problems, as allocating and garbage-collecting storage is among the slowest operations in Lisp. The recommendations below apply to all of your code in general, but in particular to code that may be executed often (such as the code in certain formulas which need to be recomputed many times).

As a rule, mapping operations (like `mapcar`) generate garbage in most Lisp implementations, because they create temporary (or permanent) lists of results. Most mapping operations can be rewritten easily in terms of DO, DOLIST, or DOTIMES.

Handling large numbers of objects with lists is generally expensive. If you have lists of more than a few tens of objects, you should consider using arrays instead. Arrays are just as convenient as lists, and they require much less storage. If your application needs variable numbers of objects, consider using variable-length arrays (possibly with fill pointers).

Declare the types of your variables and functions (using DECLARE and PROCLAIM).

Some Lisp applications will give you warnings or notes about Lisp constructs that are potentially inefficient. In CMU Common Lisp, for example, setting `speed` to 3 and `compilation-speed` to 0 generates a number of messages about potentially inefficient constructs. Many such inefficiencies can be eliminated easily, for example by adding declarations to your code.

Wrap all lambdas in `#'` rather than just ' (in CLtL2 the # is no longer optional). This comes up in Garnet a lot in final-functions for interactors and selection-functions for gadgets. Note, in the `:parts` or `:interactors` parts of aggregadgets or aggrelists, use `,#'` (comma-number-quote) before lambdas and functions.

You can save an enormous amount of time loading software if you make images of lisp with the software already loaded. For example, if you start lisp and load Garnet, you can save an image of lisp that can be restarted later with Garnet already loaded. We have simplified this procedure by providing the function `opal:make-image`. If you want to make images by hand, you will have to use `opal:disconnect-garnet` and `opal:reconnect-garnet` to sever and restore lisp's connection with the *X11* server. All of these functions are documented in the Opal chapter.

It may help to reboot your workstation every now and then. This will reset the swap file so that large applications (like Garnet) run faster.

15.4 Making your Garnet Code Faster

This section contains hints specifically about how to make Garnet code faster.

The global switch `:garnet-debug` can be removed from the `*features*` list to cause all the debugging and demo code in Garnet to be ignored during compiling and loading. This will make Garnet slightly smaller and faster. The `:garnet-debug` keyword is pushed onto the `*features*` list by default in `garnet-loader.lisp`, but you can prevent this by setting `user::Garnet-Garnet-Debug` to `nil` before compiling and loading Garnet. Garnet will need to be recompiled with the new `*features*` list, so that the extra code will not even get into the compiled binaries. Of course, you will lose functions like `inter:trace-inter`.

Turn off KR's type-checking by setting the variable `kr::*types-enabled*` to `nil`. Note: the speed difference may be imperceptible, since the type system has been implemented very efficiently (operations are only about 2% slower with type-checking).

If you have many objects in a window, and an interactor only works on a small set of those objects, then the small set of objects should be in their own aggregate or subwindow. This will cause Opal's `point-in-gob` methods run faster, which identify the object that you clicked on. When objects are arranged in an orderly aggregate hierarchy, then the `point-in-gob` methods can reject entire groups of objects, without checking each one separately, by checking whether a point is inside their *aggregate*'s bounding box. For example, in `demo-motif` the scroll bars are in their own aggregate. Putting objects in a seperate subwindow is even faster, since the coordinates of the click will only be checked against objects in the same window as the click.

Use `o-formula`s instead of `formula`s. O-formulas are compiled along with the rest of the file, whereas formulas are compiled at load- or run-time, which is much slower.

Try not to use formulas where not really needed. For example, if the positions of objects won't change, use expressions or numbers instead of formulas to calculate them.

Try to eliminate as many interactors as possible. Garnet must linearly search through all interactors in each window. To see how many interactors are on your window, you can use `(inter:print-inter-levels)`. If this is a long list, then try to use one global interactor with a start-where that includes lots of objects, rather than having each object have its own interactor. This can even work if you have a lot of scattered gadgets. For example, if you have a lot of buttons, you can use a button-panel and override the default layout to individually place each button.

The `fast-redraw` property of graphical objects can be set to make objects move and draw faster. This can be used in more cases than with previous versions of Garnet, but it is still restricted. See the fast-redraw section of the Opal chapter.

Aggrelists are quite general, and have a lot of flexibility. If you don't need this flexibility, for example, if your objects will always be in a simple left-aligned column, it will be more efficient to place the objects yourself, or create custom formulas.

If you are frequently destroying and creating new objects of the same type, it is more efficient to just keep a list of objects around, and re-using them. Allocating memory in Lisp is fairly expensive.

If you are deleting a number of objects at the same time, first set the window's `:aggregate` slot to `nil` and update the window. Then, when you are done destroying,

set the aggregate back and update again. For example, to destroy 220 rectangles on a Sparc, removing the aggregate reduced the time from 11.8 to 2.4 seconds (80%)! So your new code should be:

```
;; Code fragment to quickly destroy all the objects within an aggregate.█
(let ((temp-agg (kr:gv my-window :aggregate)))
  (when temp-agg
    ;; First, temporarily remove the aggregate:
    (kr:s-value my-window :aggregate NIL)
    (opal:update my-window)
    ;; Now do the actual destroying:
    (dolist (object (kr:get-values temp-agg :components))
      (opal:destroy object))
    ;; Finally, restore the aggregate:
    (kr:s-value my-window :aggregate temp-agg)
    (opal:update my-window)))
```

If you have objects in different parts of the same window changing at the same time, it is often faster to call update explicitly after one is changed and before the other. (This is only true if neither of the objects is a fast-redraw object. Many of the built-in gadgets are fast redraw objects for this reason, so this usually is not necessary for built-in gadgets.) The reason for this problem is that Garnet will redraw everything in a bounding box which includes all the changed objects. If the changed objects are in different parts of a window, then everything in between will be redrawn also. Ways around this problem include calling update explicitly after one of the objects changes, making one of the objects be a fast redraw object if possible, moving the objects closer together if possible (so there aren't objects in between), or putting the objects in separate subwindows if possible (subwindows are updated independently).

Conventional object-oriented programming relies heavily on message sending. In Garnet, however, this technique is often less efficient than the preferred Garnet programming style, which relies on slots and constraints. Rather than writing methods to get values from certain slots in an object, for example, consider accessing those slots directly and having a formula compute their value. The Garnet style is more efficient, since it avoid the message-sending overhead. Because Garnet provides a powerful constraint mechanism, the functionality that would normally be associated with a method can typically be implemented in a formula.

If you use the same formula in multiple places, it is more efficient to declare a formula prototype, and create instances of it. For example:

```
(defparameter leftform (o-formula (+ 10 (first (gvl :box)))))
;; for every object
(create-instance NIL <whatever>
 ...
 (:left (formula leftform)))
```

If many objects in your scene have their own feedback objects, maybe you can replace these with one global feedback object instead. The button and menu interactors can take a :final-feedback-obj parameter and will duplicate the feedback object if necessary.

If you have a lot of objects that become invisible and stay invisible for a reasonable period if time, it might be better to remove them from their aggregate rather than just setting their `:visible` slot. There are many linear searches in Garnet that process all objects in an aggregate, and each time it must check to see if the objects are invisible.

It is slightly more efficient when you are creating a window at startup, if you add all the objects to the top level aggregate *before* you add the aggregate to the window.

The use of double-buffering doesn't make your applications run faster (they actually run a little slower), but it usually *appears* faster due to the lack of flicker. See the section in the Opal chapter on how to make a window be double-buffered.

15.5 Making your Binaries Smaller

This section discusses ways to make the run-time size of your application smaller. This is important because when your system gets big, it can start to swap, which significantly degrades performance. We have found that many applications would be fast enough if they all fit into physical memory, whereas when they begin swapping virtual memory, they are not fast enough.

Don't load the PostScript module or debugging code unless you need to. Change the values of the appropriate variables in `garnet-loader`, or set the variables before loading Garnet. The values will not be overridden, since they are defined with `defvar` in `garnet-loader`.

Declare constants where possible. This allows Garnet to throw away formulas, which saves a lot of run-time space. All the built-in objects and gadgets provide a `:maybe-constant` slot, which means that you can use `(:constant T)` to make all the slots constant. The `:maybe-constant` will contain all of the slots discussed in the chapter as parameters to the object or gadget. Of course, the slots that allow the widget to operate (e.g., the buttons to be pressed or the scroll-bar-indicator to move) are not declared constant. Remember that only slots that don't change can be declared constant. Therefore, if your gadget changes position or items or active or font after creation, then you should `:except` the appropriate slots. For example:

```
(create-instance NIL gg:motif-radio-button-panel
  ;; only the :active slot will change
  (:constant '(T :except :active))
  (:left 10)(:top 30)
  (:items '("Start" "Pause" "Quit")))
```

Several functions are discussed in the Debugging chapter (starting on page [No value for "debug"]) that are very helpful in determining which slots should be declared constant. The KR chapter describes the fundamentals of constant declarations in detail.

Don't load gadget files you don't need. Most Garnet applications (like the demos), load only the gadgets they need, if they haven't been loaded already. This approach means that lots of gadgets you never use won't take up memory.

Consider using `virtual-aggregates` if you have a lot of similar objects in an interface, such as lines in a map or dots on a graph. This will decrease storage requirements significantly.

The variable `kr::store-lambdas` can be set to `nil` to remove the storage of the lambda expressions for compiled formulas. This will save some storage, but it prevents objects from being stored to files.

16 Gem: Low-level Graphics Library

16.1 Creating New Graphics Backends

<TODO>

16.2 Using the module directly

<TODO>

16.3 Function Reference

`root-window`   [Gem Window on `:all-garnet-windows`]

    (gem-method :beep (root-window))

    (gem-method :bit-blit (window source s-x s-y width height destination d-x d-y))

    (gem-method :black-white-pixel (window))

    (gem-method :character-width (root-window font character))

    (gem-method :check-double-press (root-window state code time))

    (gem-method :clear-area (window &optional (x 0) (y 0) width height buffer-p))

    (gem-method :color-to-index (root-window a-color))

    (gem-method :colormap-property (root-window property &optional a b c))

    (gem-method :copy-to-pixmap (root-window to from width height))

    (gem-method :create-cursor (root-window source mask foreground background from-font-p x y))

    (gem-method :create-image (root-window width height depth from-data-p &optional color-or-data properties bits-per-pixel left-pad data-array))

    (gem-method :create-image-array (root-window width height depth))

    (gem-method :create-pixmap (root-window width height depth &optional image bitmap-p data-array))

    (gem-method :build-pixmap (window image width height bitmap-p))

    (gem-method :create-state-mask (root-window modifier))

    (gem-method :create-window (parent-window x y width height title icon-name background border-width save-under visible min-width min-height max-width max-height user-specified-position-p user-specified-size-p override-redirect))

    (gem-method :delete-font (root-window font))

    (gem-method :delete-pixmap (root-window pixmap &optional buffer-too))

    (gem-method :delete-window (root-window x-window))

    (gem-method :device-image (root-window index))

    (gem-method :discard-mouse-moved-events (root-window))

    (gem-method :discard-pending-events (root-window &optional timeout))

    (gem-method :draw-arc (window x y width height angle1 angle2 function line-style fill-style &optional pie-slice-p))

(gem-method :draw-image (window left top width height image function fill-style))

(gem-method :draw-line (window x1 y1 x2 y2 function line-style &optional drawable))

(gem-method :draw-lines (window point-list function line-style fill-style))

(gem-method :draw-points (window point-list function line-style))

(gem-method :draw-rectangle (window x y width height function line-style fill-style))

(gem-method :draw-roundtangle (window left top width height corner-width corner-height function line-style fill-style))

(gem-method :draw-text (window x y string font function line-style &optional fill-background invert-p))

(gem-method :drawable-to-window (root-window drawable))

(gem-method :event-handler (root-window ignore-keys))

(gem-method :flush-output (window))

(gem-method :font-exists-p (root-window name))

(gem-method :font-max-min-width (root-window font min-too))

(gem-method :font-name-p (root-window arg))

(gem-method :font-to-internal (root-window opal-font))

(gem-method :get-cut-buffer (root-window))

(gem-method :image-bit (root-window image x y))

(gem-method :image-from-bits (root-window patterns))

(gem-method :image-hot-spot (root-window image))

;;; returns three values: width, height, depth (gem-method :image-size (a-window image))

(gem-method :image-to-array (root-window image))

(gem-method :initialize-device (root-window))

(gem-method :initialize-window-borders (window drawable))

(gem-method :inject-event (window index))

(gem-method :make-font-name (root-window key))

(gem-method :map-and-wait (a-window drawable))

(gem-method :max-character-ascent (root-window font))

(gem-method :max-character-descent (root-window font))

(gem-method :mouse-grab (window grab-p want-enter-leave &optional owner-p))

(gem-method :raise-or-lower (window raise-p))

(gem-method :read-an-image (root-window pathname))

(gem-method :reparent (window new-parent drawable left top))

(gem-method :set-clip-mask (window clip-mask &optional lstyle-ogc fstyle-ogc))

(gem-method :set-cut-buffer (root-window string))

(gem-method :set-device-variables (root-window full-display-name))

(gem-method :set-draw-function-alist (root-window))

(gem-method :set-draw-functions (root-window))

(gem-method :set-drawable-to-window (window drawable))

```
(gem-method :set-interest-in-moved (window interestedp))
(gem-method :set-screen-color-attribute-variables (root-window))
(gem-method :set-window-property (window property value))
(gem-method :stippled-p (root-window))
(gem-method :text-extents (root-window opal-font string))
(gem-method :text-width (root-window opal-font string))
(gem-method :translate-character (window x y state code time))
(gem-method :translate-code (window scan-code shiftp))
(gem-method :translate-coordinates (root-window window x y &optional other-window))
(gem-method :translate-mouse-character (root-window button-code modifier-bits event-
key))
(gem-method :window-debug-id (window))
(gem-method :window-depth (window))
(gem-method :window-from-drawable (root-window drawable))
(gem-method :window-has-grown (window width height))
(gem-method :window-to-image (window left top width height))
(gem-method :write-an-image (root-window pathname image))
```

16.4 Font Handling

<TODO>

BAM: I am not sure this section is correct.

EED: This was moved from Opal to the Gem chapter where its discusion of low level graphics is perhaps more appropriate

Most users of Opal will only use the pre-defined graphical objects, and will combine them into aggregates and use formulas to attach them together. It will be rare to create new kinds of graphical objects. This should only be needed when new primitives are available, such as splines.

This chapter discusses how to create new types of graphical objects, should that be necessary.

16.5 Internal slots in graphical objects

There are numerous extra slots in all graphical objects that are used internally by Opal. This section will attempt to describe these slots and their potential uses when designing new graphical objects.

16.5.1 :update-slots

The `:update-slots` slot contains an association list of all slots in the object that affect the output picture from the object. For example:

```
* (gv opal:arc :update-slots)
((:visible) (:line-style) (:filling-style) (:draw-function) (:left)
 (:top) (:width) (:height) (:angle1) (:angle2))
```

If any of the values of these objects slots in an instance of an `opal:arc` object change, the instance will need to be redrawn at the next window update.

Anytime a slot on the `:update-slots` list is changed (either with `s-value` or by a formula being invalidated) the KR's invalidate demon is called with the object, the slot, and the slot's value on the association list.

Opal doesn't use the second value of the association pair, so it should be left as `nil`.

When creating an object that is a specialized instance of a prototype object, one should inherit all the slots on the `:update-slots` list, and then add any others as necessary. Commonly this is done by something of the form:

```
(create-instance 'opal:arc opal:graphical-object
  ...
  (:update-slots
   (append (gv opal:graphical-object :update-slots)
   '((:left) (:top) (:width) (:height)
     (:angle1) (:angle2))))
  ...)
```

By doing this, you insure that all the necessary slots are inherited, and add any new slots as necessary.

16.5.2 :drawable

The `:drawable` slot contains a structure that is the CLX drawable object that the object is to display itself into when it is sent a `draw` message. This object may not be the physical window that the object is to be displayed into, it may be a pixmap that is double buffered onto the screen somewhere in the update algorithm. All objects should trust the value in this slot, even though is may not correspond to the drawable of their window. This slot may not contain a value until the object (or one of its parents) is placed in a Garnet window.

16.5.3 :display-info

The `:display-info` slot holds information used by many of the CLX primitives for computation, and drawing. Once an object is placed in a window, this slot contains an opal structure:

```
(defstruct (display-info
             (:print-function display-info-printer))
  display
  screen
  root-window
  default-gcontext)
```

The form (`display-info-xxx` (gv *object* `:display-info`)) returns the *xxx* structure from *object*'s `:display-info` slot. These fields are useful as follows:

- `display` is the CLX structure corresponding to the current display connection to the X server. This is used in calls that affect or query the server directly, such as <xlib:open-font>, xlib:display-force-output, and <xlib:global-pointer-position>.
- `screen` is the CLX structure containing information about the window's screen. This is used most often with structure accessors to get information on values for the screen's white and black pixels, width and height in pixels or millimeters.

- **root-window** is the CLX window that corresponds to (**xlib:screen-root** *screen*) for use in calls to **xlib:create-window**.
- **default-context** is a CLX graphical context structure used all drawing requests. Opal maintains a cache on this object, so it should not be changed. It is acceptable to use this structure in an **xlib:with-gcontext** form when it is necessary to modify a gcontext outside the bounds of **with-filling-styles** [and]**with-line-styles**.

16.5.4 :x-tiles

The **x-tiles** slot contains a formula that computes a pixmap for use in drawing tiled lines, or pattern filled regions. The formula evaluates to a cons cell the car of which is the pixmap to use for tiling lines, and the cdr of which is a pixmap to use when drawing fillings. These are computed from values in the object's **:line-style** and **:filling-style** slots.

16.5.5 :x-draw-function

This slot contains a formula that is used to compute the CLX drawing function from the **:draw-function** slot. It probably won't ever be necessary to change the formula in this slot.

16.6 Methods on all graphical objects

The following methods are defined on all graphical objects and may be specialized upon when creating new classes of graphical objects.

graphical-object   [Method on **opal:draw**]
> The **draw** method on a graphical object causes the object to display itself in the window of its aggregate. This is only called by the update methods, never directly by users of Opal.

initialize   [Method on **graphical-object**]
> This method is called immediately after an instance of an object is created. It is passed the new object as its only argument.

opal:point-in-gob *x y*   [Method on **graphical-object**]
> This method should be provided for all new objects. It is used by **point-to-component** and **point-to-leaf** to query an object for a hit. The method should return **t** if the object is under the point (x, y). This function should also take into account the values in the **:hit-threshold**, **:select-outline-only**, and **:visible** slots of the object, as described in section [stdfuncs], page 156. Objects that are not visible should return **nil**.

opal::fix-properties *changed-slots*   [Method on **graphical-object**]
> This method is called on aggregates and windows at the time when the update algorithm passes them during an update. The method is called with an object, and a list of slots that have changed since it was last called. This function is often useful for calling functions that cannot easily be put into formulas.
>
> Currently **fix-properties** is not called on graphical objects, but this functionality can be added to Opal by talking to the maintainer.

16.7 Draw Methods

There are several things that are worthy of note when working on `draw` methods for new objects.

Objects *must* draw entirely within their bounding box. The redisplay algorithm will not work properly if things are drawn outside of their bounding boxes.

There are two macros for use in writing draw methods that prepare a gcontext from the gcontext cache that is appropriate for drawing outlines or fillings as described by the values in the `:line-style` and `:filling-style` slots of the object.

with-filling-styles (*variable graphical-object*) *body*   [Macro]
> This form executes the forms inside *body* with *variable* bound to a CLX gcontext structure suitable for drawing the filling of an object with respect to *graphical-object*'s filling style object in the slot `:filling-style`.

with-line-styles (*variable graphical-object*) *body*   [Macro]
> This form executes the forms inside *body* with *variable* bound to a CLX gcontext structure suitable for drawing the outline of an object with respect to *graphical-object*'s line style object in the slot `:line-style`.

> These forms are commonly used like this:

```
(define-method :draw opal:polyline (polyline)
  (let ((point-list (gv polyline :point-list))
(drawable (gv polyline :window :drawable)))
     (with-filling-styles (gcontext polyline)
       (xlib:draw-lines drawable gcontext
              point-list :fill-p t))
     (with-line-styles (gcontext polyline)
       (xlib:draw-lines drawable gcontext
              point-list))))
```

Appendix A GNU General Public License

Function Index

A

B

C

D

F

G

I

R

S

T

W

Variable Index

Keyword Index

(Index is nonexistent)

Type Index

Concept Index

Build (in Gilt)

build

butt

Button action routines

Button–Interactor

button–outside–stop?

Button

buttons

C32 panels

C32

cached values

calculator

Call–Func–On–Slot–Set

cap–style

careful–eval–formula–lambda

careful–eval

careful–read–from–string

careful–string–eval

center (justification)

center–of–rotation

Change–Active

change–cursors

change–item

Changing Label Button

character code

check–leaf–but–return–element–or–none

check–leaf–but–return–element

child vs. leaf

gest–attributes–endy

gest–attributes–initial–cos

gest–attributes–initial–sin

gest–attributes–magsq2

gest–attributes–maxx

gest–attributes–maxy

gest–attributes–minx

gest–attributes–miny

gest–attributes–path–r

gest–attributes–path–th

gest–attributes–sharpness

gest–attributes–startx

gest–attributes–starty

gesture action routines

Gesture–Interactor

gesture–interactor

Gesture

Gestures in demo–arith

gestures

get–cursor–index

get–objects

get–submenu–component

get–value

Getting Garnet

gg nickname

gilt (package)

gilt–obj

gilt–ref slot (in Gilt)

oval

overlapping

Packages in Garnet

Packages

page–trill–p

papers

parameters

parent (slot)

parent

parenthesis matching

Part–generating functions

parts in aggregadgets

parts in aggrelists

Parts of Garnet

paste–selection

pathnames

paths in formulas

pixarray

pixel–margin

pixmap

plain

Playback

point–in–gob

point–to–component

point–to–leaf

point–to–rank

pointer

Points (slot)

print–slots slot

printing schemata

printing

Priorities

priority level

priority levels

Priority Levels

priority–level–list

priority–level

procedural attachments

projecting

promote–item

prop–sheet–for–obj–go

Prop–Sheet–for–obj–With–OK

Prop–Sheet–For–Obj

Prop–Sheet–With–OK

prop–sheet

Properties... (in Gilt)

Properties

Property sheets

prototype/instance

prototypes

ps

PS

pull–down menus

purple–fill

purple

push–first–item

O

P

R

S

U

V