Aggregadgets, Aggrelists, & Aggregraphs Reference Manual

Andrew Mickish
Roger B. Dannenberg
Philippe Marchal
David Kosbie
A. Bryan Loyall

December 1994

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.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

1. Aggregadgets

1.1. 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 manual 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.

1.2. 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.

1.2.1. 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. As with aggregates, components are listed in display order, that is, *from back to front*.

A short example of an aggregadget definition is shown in figure 1-1, and the picture of this aggregadget is in figure 1-2.

```
(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 1-1: A simple CHECK-MARK aggregadget.

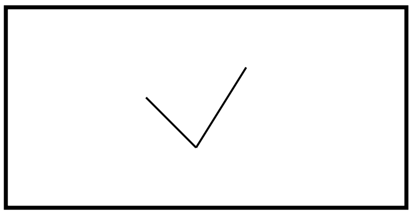


Figure 1-2: The picture of the CHECK-MARK aggregadget.

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 1.2.6).

1.2.2. 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 1-1, 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 1-3 and 1-4.

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 1.2.4 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 1-3: 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 1-4: The `:right-line` component of CHECK-MARK.

As shown in figure 1-3, 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.

1.2.3. Destroying Aggregadgets

`opal:Destroy` *gadget* [*Method*]

`opal:Destroy-Me` *gadget* [*Method*]

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`.

1.2.4. Constants and Aggregadgets

The ability to define constant slots is an advanced feature of Garnet that is discussed in detail in the KR manual. 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 Manual:

```
(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.

1.2.5. 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.

1.2.6. 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 1.2.1. In figure 1-1, 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 1-5 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 1-5: 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 1-5 may be given the equivalent value

```
    ,(o-formula (opal:gvl-sibling :left-line :x2))
```

1.2.7. 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 1-6.

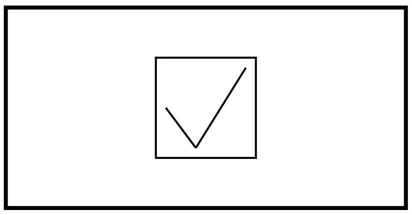


Figure 1-6: A picture of a check-box.

The check-box can be considered a hierarchy of objects: the CHECK-MARK object defined in figure 1-1, and a box. This hierarchy is illustrated in figure 1-7.

```
             check-box
               /   \
              /     \
            box  check-mark
                     /   \
                    /     \
            left-line   right-line
```

Figure 1-7: The hierarchical structure of a check-box.

The CHECK-BOX hierarchy is implemented through aggregadgets in figure 1-8. 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 1-1). The aggregadget definition for the CHECK-MARK part could have been written out explicitly, as in the more complicated CHECK-BOX schema of figure 1.9.1. 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 1-8: The definition of a check-box.

See section 1.9.2 for another example of a modularized multi-level aggregadget, and see section 1.4.1 for information about inheriting structure from other multi-level aggregadgets.

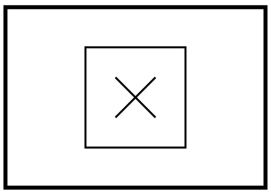
1.2.8. Nested Part Expressions for Aggregadgets

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 1-9 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)
               frac))))

(create-instance 'X-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
              ((:line1 ,opal:line
                       (:x1 ,(o-formula (horiz-prop 0.3)))
                       (:y1 ,(o-formula (vert-prop 0.3)))
                       (:x2 ,(o-formula (horiz-prop 0.7)))
                       (:y2 ,(o-formula (vert-prop 0.7))))
               (:line2 ,opal:line
                       (:x1 ,(o-formula (horiz-prop 0.7)))
                       (:y1 ,(o-formula (vert-prop 0.3)))
                       (:x2 ,(o-formula (horiz-prop 0.3)))
                       (:y2 ,(o-formula (vert-prop 0.7))))))))))
```

Figure 1-9: A box with an X, illustrating nested parts.

1.2.9. 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 1-10 all have the same prototype.

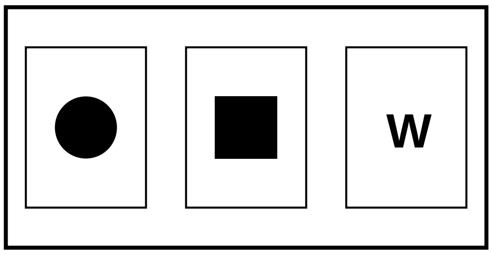


Figure 1-10: Aggregadgets that generate a part through a function.

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 <u>not</u> 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 <u>instance</u> 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 1-10, 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 1.4.2. For example, the labels in figure 1-10 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 1.5.2.7).

1.2.10. 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 1-11.

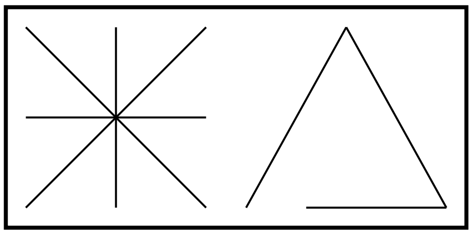


Figure 1-11: The multi-line picture.

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 1.5.

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 1.2.6). (***Note:*** The standard lisp function `values` may be used to return two arguments from the generating function.)

Figure 1-12 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 1-11.

```
(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 1-12: An aggregadget with a function to create the parts.

Figure 1-13 shows how to create the same aggregadgets as in figure 1-11, 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.

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.

```
(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 1-13: An aggregadget with a function to create named parts.

1.3. 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—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 1.2.10.

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 1-14, 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 1.2.6.

Figure 1-14 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 1-15 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 1-16.

```
(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 1-14: 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 1-15: The printouts of an aggregadget and its attached interactor.

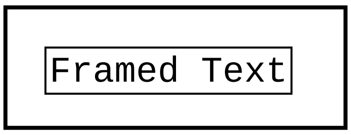


Figure 1-16: A picture of the FRAMED-TEXT aggregadget.

1.4. 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.

1.4.1. 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 1-17 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’.

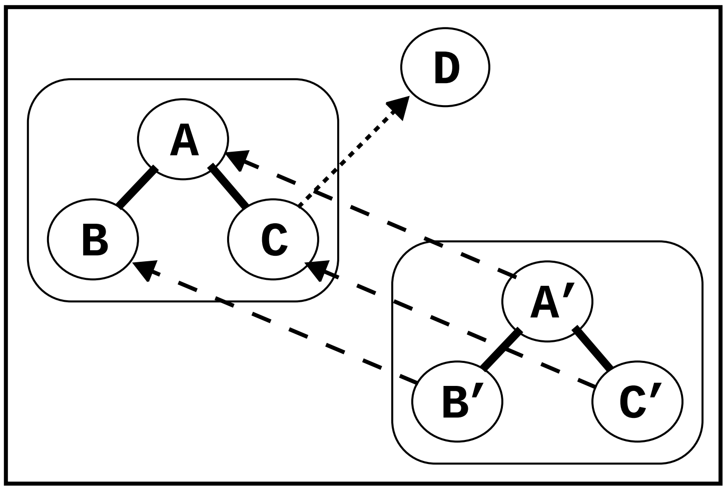


Figure 1-17: A prototype aggregate and one instance. The dashed lines go from instances to their prototypes, solid lines join children to parents, and the dotted line from C to D represents a formula dependence which is inherited by C’.

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.

1.4.2. 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 1.4.5:

1. 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.

2. 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))
   ```

3. 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))
   ```

4. 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))
   ```

5. Any element of the list of parts may be a list of the form (*name prototype slots*), as described in section 1.2.1. 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.

1.4.3. 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.

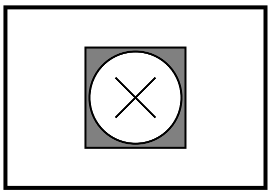
The `:parts` syntax has a special variation to accomplish this form of multiple inheritance. If the keyword `:inherit` occurs at the top level in the *slots* list, then the next element of *slots* must be a list of slot names. All the slots not mentioned in the `:inherit` clause are inherited from the 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))
```

1.4.4. Instance Examples

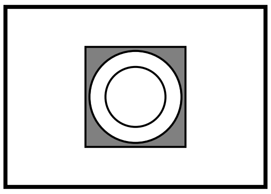
Figure 1-18 illustrates how to override and inherit parts from an aggregadget. The prototype aggregadget is the `x-box` aggregadget shown in figure 1-9. In the instance named CIRCLE-X-BOX, a circle has been inserted between the box and the ‘‘X’’ mark, and the box has a gray fill.

In figure 1-19, the CIRCLE-X-BOX aggregadget is further modified by replacing the ‘‘X’’ with a circle.



```
(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)))
               (:filling-style ,opal:white-fill))
      :mark))))
```

Figure 1-18: Adding a circle and changing the filling style in an instance of the X-BOX aggregadget.



```
(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)))
             (:top ,(o-formula (+ (gvl :parent :top) 10)))
             (:width ,(o-formula (- (gvl :parent :width) 20)))
             (:height ,(o-formula (- (gvl :parent :height) 20))))))))
```

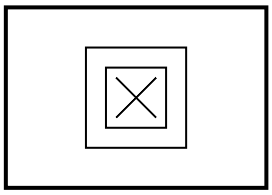
Figure 1-19: Omitting the ‘‘X’’ and adding an inner circle to the CIRCLE-X-BOX aggregadget.

1.4.5. 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 1-20 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
              (:left ,(o-formula (horiz-prop 0.2)))
              (:width ,(o-formula (- (horiz-prop 0.8)
                                     (horiz-prop 0.2))))
              (:top ,(o-formula (vert-prop 0.2)))
              (:height ,(o-formula (- (vert-prop 0.8)
                                      (vert-prop 0.2)))))))))))))
```

Figure 1-20: Extending the x-box prototype with a new rectangle in the mark part.

1.5. 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 1.7.4.)

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 1.3).

1.5.1. 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/column
 (: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` – The leftmost coordinate of the aggrelist (default is 0).

`:top` – The topmost coordinate of the aggrelist (default is 0).

`:items` – 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` – Either a schema or a description of a schema (see section 1.5.2.1).

`:direction` – 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` – Vertical spacing between elements (default is 5).

`:h-spacing` – Horizontal spacing between elements (default is 5).

`:fixed-width-p` – 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` – The width of all components, if `:fixed-width-p` is T (default is NIL).

`:fixed-height-p` – 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` – The height of all components, if `:fixed-width-p` is T (default is NIL).

`:h-align` – 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` – 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` – 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` – 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` – The amount to indent upon starting a new row/column (in pixels) (default is 0).

1.5.2. 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 1.5.3.)

To cause an aggrelist to generate its components from a prototype, the `:item-prototype` and the `:items` slot may be set.

1.5.2.1. 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 1.6).

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 1.5.3).

1.5.2.2. 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 *n*th string in the `:items` list to the *n*th 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.

1.5.2.3. 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* [*Method*]

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.

1.5.2.4. 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 Manual. However, just as with aggregadgets (discussed in section 1.2.4), 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               :v-spacing          :h-align            :fixed-height-size
:top                :h-spacing          :fixed-width-p      :rank-margin
:items              :indent             :fixed-height-p     :pixel-margin
:direction          :v-align            :fixed-width-size
```

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.

1.5.2.5. 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 1-21. 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)))))))
```

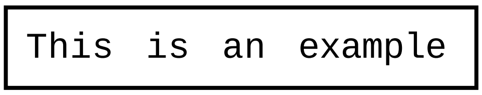

This is an example

Figure 1-21: The picture of an itemized aggrelist.

1.5.2.6. An Aggrelist with an Interactor

As another example of an itemized aggrelist, consider the schema FRAMED-TEXT-LIST defined in figure 1-22. A picture of the FRAMED-TEXT-LIST aggrelist appears in figure 1-23.

This aggrelist explicitly defines an aggregadget as the `:item-prototype`. This aggregadget is similar to the FRAMED-TEXT schema defined in figure 1-14, but there is an additional `:final-function` slot (see figure 1-22). The purpose of the `:final-function` is to keep the strings in the `:items` list consistent with the strings in the components.

```
(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 1-22: An aggrelist using an aggregadget as the `:item-prototype`.

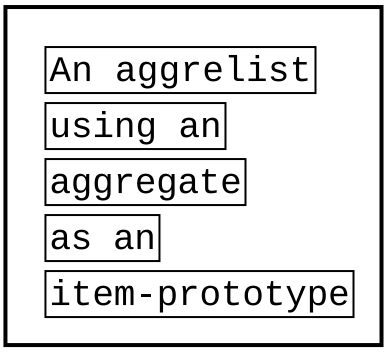


Figure 1-23: A picture of the FRAMED-TEXT-LIST aggrelist.

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 1.8.1), 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 1-14, 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 1.9.3 for an example of a menu made with an itemized aggrelist.

1.5.2.7. An Aggrelist with a Part-Generating Function

Section 1.2.9 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 1-24. 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.

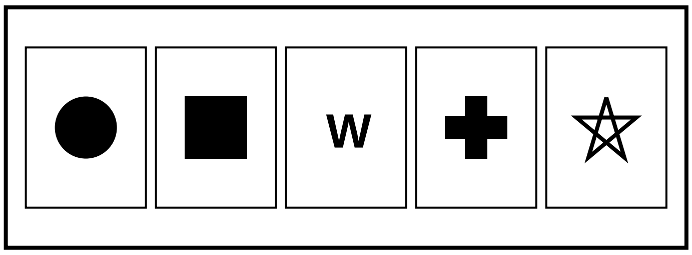


Figure 1-24: An aggrelist that uses a part-generating function in its :item-prototype

```
(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 manual for further details.

1.5.3. 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 1-25 creates an aggrelist with three components, and a picture of this aggrelist is shown in figure 1-28.

```
(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 1-25: Example of an aggrelist with a parts slot.

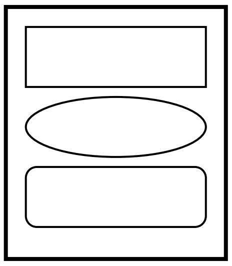


Figure 1-26: The picture of an aggrelist with three components.

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 1.4.

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 1.5.2.4 (including the `:items` slot) must be declared constant.

1.6. 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.

1.6.1. 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 1-27 illustrates a variation on the text list in figure 1-22. 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.

1.7. 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.

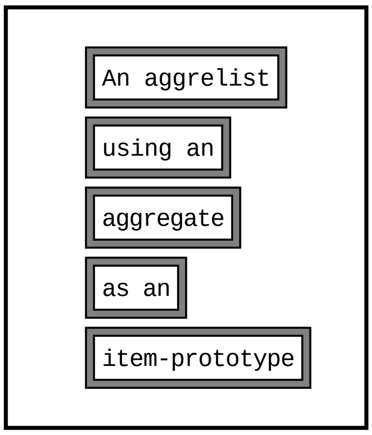
The philosophy behind structural inheritance is simply stated: *changing a prototype and then making an instance should be equivalent to making an instance and then changing the prototype.* In practice, this equivalence is difficult to achieve completely; exceptions will be noted.

1.7.1. Copying Gadgets

`opal:Copy-Gadget` *gadget* [*Function*]

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.



```
(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
           (:filling-style ,opal:gray-fill)
           ; make the frame wider
           (:width ,(o-formula (+ (gvl-sibling :text :width) 16)))
           ; make the frame taller
           (:height ,(o-formula (+ (gvl-sibling :text :height) 16))))
        (:white-box ,opal:rectangle
           (:filling-style ,opal:white-fill)
           (:left ,(o-formula (+ (gvl :parent :left) 4)))
           (:top ,(o-formula (+ (gvl :parent :top) 4)))
           (:width ,(o-formula (+ (gvl-sibling :text :width) 8)))
           (:height ,(o-formula (+ (gvl-sibling :text :height) 8))))
        (:text :modify  ; move the text to allow for the border
           (:left ,(o-formula (+ (gvl :parent :left) 8)))
           (:top ,(o-formula (+ (gvl :parent :top) 8)))))))))
```

Figure 1-27: An aggrelist that overrides parts of an inherited `:item-prototype`. The prototype FRAMED-TEXT-LIST was defined in figure 1-22.

1.7.2. Aggregadget Manipulation

1.7.2.1. Add-Component

`opal:Add-Component` *gadget element* [[`:where`] *position* [*locator*]] [*Method*]

This function behaves just like the `add-component` method for aggregates (see the *Opal Reference Manual*) 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 1.2.5). 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.

1.7.2.2. Remove Component

`opal:Remove-Component` *gadget element* [*destroy?*] [*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.

1.7.2.3. Add-Interactor

Interactors can be added by calling

`opal:Add-Interactor` *gadget interactor* [*Method*]

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.

1.7.2.4. Remove-Interactor

`opal:Remove-Interactor` *gadget interactor* [*destroy?*] [*Method*]

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.

1.7.2.5. Take-Default-Component

`opal:Take-Default-Component` *gadget name* [*destroy?*] [*Method*]

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*.

1.7.3. Itemized Aggrelist Manipulation

1.7.3.1. Add-Item

`opal:Add-Item` *aggrelist* [*item*] [[`:where`] *position* [*locator*] [`:key` *function-name*]] [*Method*]

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 1.2.4.

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 1.5.2.5:

```
    (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)
```

will 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.

1.7.3.2. Remove-Item

`opal:Remove-Item` *aggrelist* [*item* [`:key` *function-name*]] [*Method*]

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 1.2.4.

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.

1.7.3.3. Remove-Nth-Item

`opal:Remove-Nth-Item` *aggrelist n* [*Method*]

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 1.2.4.

1.7.3.4. Change-Item

To change just one item in the `:items` list, call

`opal:Change-Item` *aggrelist item n* [*Method*]

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.

1.7.3.5. Replace-Item-Prototype-Object

`opal:Replace-Item-Prototype-Object` *aggrelist item-proto* [*Method*]

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.

1.7.4. Ordinary Aggrelist Manipulation

1.7.4.1. Add-Component

The `add-component`, defined in section 1.7.2.1 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` – 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` – 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` – This contains the previous component in the list, regardless of what is visible.

`:next` – 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 1-25 is shown in figure 1-28. 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 1-28: Example of an aggrelist built using add-component.

1.7.4.2. Remove-Component

See section 1.7.2.2 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.

1.7.4.3. Remove-Nth-Component

`opal:Remove-Nth-Component` *aggrelist n* [*Method*]

The n^{th} component of *aggrelist* is removed by invoking `remove-local-component`. Instances of *aggrelist* are *not* affected.

1.7.5. 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*]] [*Method*]

`opal:Remove-Local-Component` *gadget element* [*destroy?*] [*Method*]

`opal:Add-Local-Interactor` *gadget interactor* [*Method*]

`opal:Remove-Local-Interactor` *gadget interactor* [*destroy?*] [*Method*]

`opal:Add-Local-Item` *aggrelist* [*item*] [[`:where`] *position* [*locator*] [`:key` *function-name*]] [*Method*]

`opal:Remove-Local-Item` *aggrelist* [*item* [`:key` *function-name*]] [*Method*]

1.8. 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:

1.8.1. Write-Gadget

`opal:Write-Gadget` *gadget file-name* &optional *initialize?* [*Function*]

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.

1.8.1.1. Avoiding Deeply Nested Parts Slots

One would expect an instance of a standard gadget (see the *Garnet Gadgets Reference Manual*) 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 – The default; write out all slots and parts that differ from the prototype.
- `:me` – Assume that all components and 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`– 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`.

1.8.1.2. 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 1.2.10), 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*` [*Variable*]

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.

1.8.1.3. 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))
```

1.8.1.4. 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*` [*Variable*]

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*` [*Variable*]

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*` [*Variable*]

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.

1.8.1.5. 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))))
```

1.8.1.6. 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.

1.9. More Examples

1.9.1. A Customizable Check-Box

Figure 1-29 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 1-30.

1.9.2. Hierarchical Implementation of a Customizable Check-Box

Figure 1-31 shows the definition of a customizable check-box as in figure 1-29. However, this second CHECK-BOX definition exploits the hierarchical 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.

1.9.3. Menu Aggregadget with built-in interactor, using Aggrelists

The figure 1-33 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 1-34 shows how to create an instance of the menu. A picture of these menus (the prototype and its instance) is shown in figure 1-32.

```
(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 1-29: The definition of a customizable check-box.

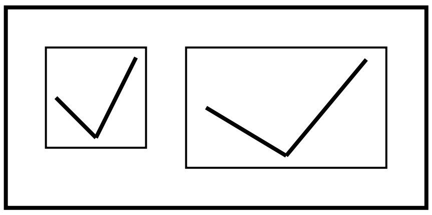


Figure 1-30: Instances of the customizable check-box.

```
(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 1-31: A hierarchical implementation of a customizable check-box.

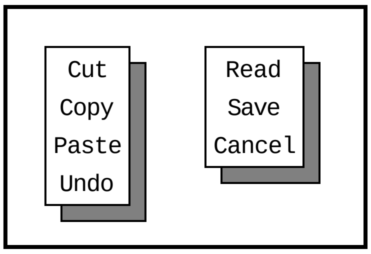


Figure 1-32: The two menus (prototype and instance) made with itemized aggrelist.

```
(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 1-33: 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 1-34: Creation of an instance of MENU.

2. 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.

2.1. 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 2.5 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.

2.1.1. 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.

2.1.2. 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 2.2.)

Instances of aggregraphs can be used as prototypes for other aggregraphs without providing values for all the required slots in the prototype.

2.1.3. 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 2.1.5 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.

2.1.4. 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)))
```

The graph pictured in figure 2-1 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.

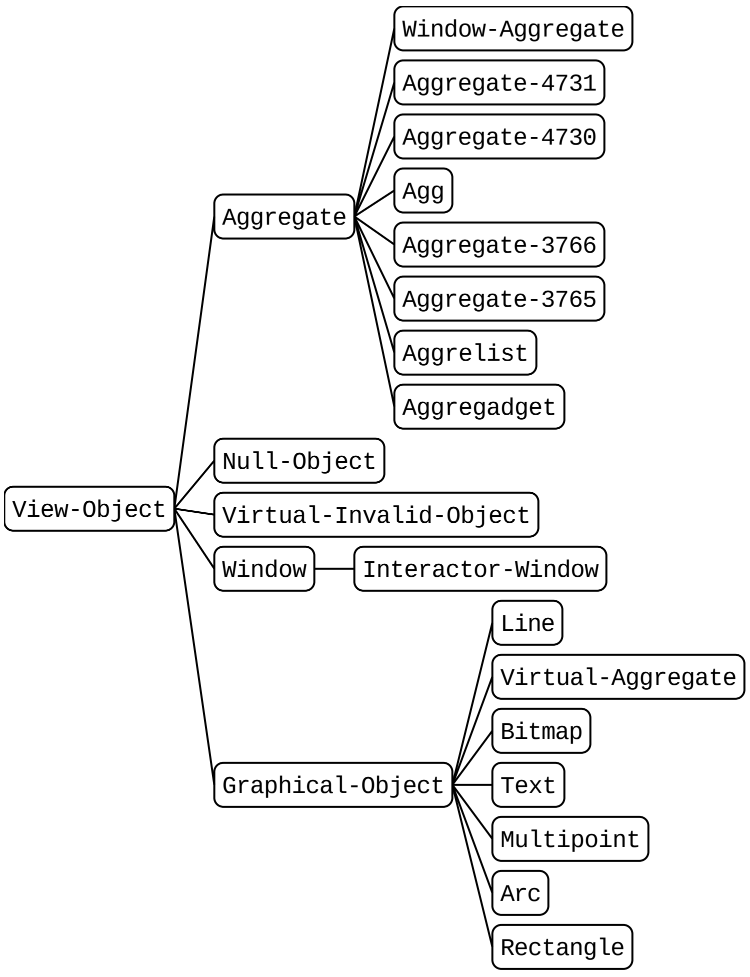


Figure 2-1: Graph generated by SCHEMA-GRAPH

2.1.5. 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))))))))
```

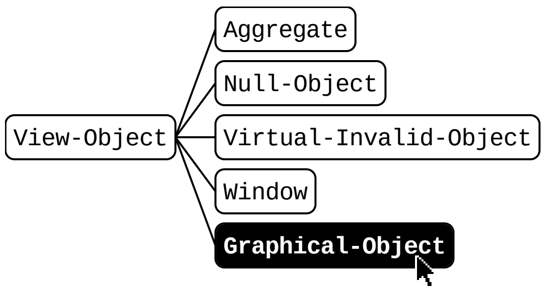


Figure 2-2: Graph generated by SCHEMA-GRAPH-2

The graph of figure 2-2 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.

2.2. 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.

2.3. 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 2.2).

Methods (can be overridden)

The same methods are available as with `aggregraph` (see section 2.2).

2.4. 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 2.2).

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.

2.5. 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.

Index

Table of Contents