Report Number: WUCS-88-19

1988-08-01

A Graph Browser with Zoom and Roam for Allegro Common Lisp

Steve B. Cousins and J. Andrew Fingerhut

This report describes an object-oriented tool that has been developed for viewing graphs on a Macintosh II computer using Allegro Common Lisp. The tool is useful for visualizing data which can be represented in tree or graph form. The graphs can be viewed for far away to get a global view, and from close up so that the labels on the vertices can be discerned. Scrolling can be performed at a nearly infinite number of resolutions, and a search feature makes it easy to find any node rapidly. Although the 'information space' on which the graph is logically plotted... **Read complete abstract on page 2.**

A Graph Browser with Zoom and Roam for Allegro Common Lisp

Steve B. Cousins and J. Andrew Fingerhut

Complete Abstract:

This report describes an object-oriented tool that has been developed for viewing graphs on a Macintosh II computer using Allegro Common Lisp. The tool is useful for visualizing data which can be represented in tree or graph form. The graphs can be viewed for far away to get a global view, and from close up so that the labels on the vertices can be discerned. Scrolling can be performed at a nearly infinite number of resolutions, and a search feature makes it easy to find any node rapidly. Although the 'information space' on which the graph is logically plotted is 2-dimensional, nodes can be arbitrarily large or small, and this combined with the zooming feature gives somewhat the illusion of 3-dimensions. The tool is implemented using an object-oriented paradigm, in which the window containing the graph rendering is an object, and so are each of the vertices in the graph. Operations on vertices of the graph are implemented as messages to objects, so that specifics objects can choose to respond in individual ways to various events. This approach makes the tool useful for a wide range of tasks, including browsing the object system inheritance lattice of the lisp system itself, displaying dependency nets in a belief maintenance system, and browsing a hypertext network.

A GRAPH BROWSER WITH ZOOM AND ROAM FOR ALLEGRO COMMON LISP

Steve B. Cousins and J. Andrew Fingerhut

WUCS-88-19

August 1988

Medical Informatics Group and
Center for Intelligent Computer Systems
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

This work was supported by a grant from Apple Computer Corporation and by the Center for Intelligent Computer Systems and the Medical Informatics Group at Washington University.

Revised 9/6/88

Abstract

This report describes an object-oriented tool that has been developed for viewing graphs on a Macintosh II computer using Allegro Common Lisp. The tool is useful for visualizing data which can be represented in tree or graph form. The graphs can be viewed from far away to get a global view, and from close up so that the labels on the vertices can be discerned. Scrolling can be performed at a nearly infinite number of resolutions, and a search feature makes it easy to find any node rapidly. Although the 'information space' on which the graph is logically plotted is 2-dimensional, nodes can be arbitrarily large or small, and this combined with the zooming feature gives somewhat the illusion of 3-dimensions.

The tool is implemented using an object-oriented paradigm, in which the window containing the graph rendering is an object, and so are each of the vertices in the graph. Operations on vertices of the graph are implemented as messages to objects, so that specific objects can choose to respond in individual ways to various events. This approach makes the tool useful for a wide range of tasks, including browsing the object system inheritance lattice of the lisp system itself, displaying dependency nets in a belief maintenance system, and browsing a hypertext network.

1 Introduction

This report describes an object-oriented tool that has been developed for viewing graphs on a Macintosh II computer using Allegro Common Lisp.* A directed graph is a set of vertices and a set of edges, where edges consist of pairs of vertices. In most applications of graphs, vertices have information associated with them such as labels. Edges may also have labels. Formally, a graph is just the sets of vertices and edges, but graphs are often embedded in a plane so that they can be visualized. This is done by mapping each vertex in the graph to a location on the plane, and then drawing lines between vertices corresponding to edges.

A useful application of computer graphics is to render an embedding of a graph on a computer screen so that the graph can be viewed interactively by a computer user. For example, a programmer might find it helpful to see a graph of subroutine calls, but would probably be unwilling to spend the time to draw one by hand. A tool like the one described in this report can be used to take the graph and render it on the computer screen. It may not be possible to render the graph legibly on the available screen space, so the tool must have the ability to treat the screen as a virtual window on the graph rendering.

This tool has been used to create object system hierarchy browsers, hypertext browsers, and dependency network browsers. Of course, any application which has data organized as a graph or tree could use this tool to help to visualize the data. Another potential application is a graph of function calls, indicating which functions are called by which other functions.

In this work, we refer to the virtual space on which the graph resides as *information space* and the screen is viewed as a discrete, imperfect window onto this space. Information space is a plane with real number axes, in which objects have an arbitrary but definite size. The window onto information space is a square which has sides of length *scale* (in information space coordinates). The window is mapped onto a computer screen by taking the ratio of the number of pixels in either direction to the scale (giving a number of pixels per unit of real space).

The operation of **roaming** corresponds to moving the window around in information space. We use both of the terms scroll and roam to indicate this motion. Scrolling the screen up corresponds to moving the window down in information space, scrolling right corresponds to moving left, etc. Roaming is accomplished using the arrow keys on the keyboard, as the mouse is used for other operations like choosing objects.

The concept of **zooming** in and out on the screen corresponds to adjusting the size of the window on information space. Making the window smaller corresponds to zooming in, while making the window larger corresponds to zooming out. Zooming way out allows the user to visualize the graph as a whole, while zooming in shows details of specific nodes. As the user zooms out, the vertices get smaller and the vertex labels begin to get clipped until at a certain threshold they are no longer drawn (it is also possible for some to be drawn and some not to be drawn, since nodes can be

*Allegro Common Lisp is also called Coral Lisp, and is marketed by Franz Inc.

different sizes).

The rendition of the graph can be modified interactively (useful for uncovering partially covered nodes). By simply clicking on a node and dragging it to the desired location (a standard Macintosh technique), nodes can be placed exactly where the user wants them. The browser also has a layout algorithm which treats the graph as a tree and rearranges all nodes which are descendents of a root node. The layouts are not perfect, of course, but they do give a starting point from which the user can arrange the nodes to his taste.

With all of the zooming and roaming (scrolling) that is available, it is possible for a user to get lost. For example, holding down the zoom-in button for a long time causes the window on information space to become very small. Roaming at that point could take a long time in just the right direction to find the graph again. Similarly, roaming off in some direction could leave the window on information space far from any part of the graph. To help keep the user from getting lost, three functions are included: Home, Center, and Find. The **home** command sets the window on information space to be a rectangle that approximately bounds all of the nodes in the graph. It is the best way to get a big picture of the whole graph quickly. The **center** command moves the window on information space so that the selected item (if any) is in the center of the screen, big enough so that its label can be read. The **find** command searches all nodes in the graph for one containing a search string, and centers the first node found which qualifies. Find is implemented similarly to a search mechanism in a text editor: calling find repeatedly finds the next instance (of course, the order is somewhat random because of the 2-dimensional nature of the graph), and eventually find will loop around to the first instance again.

The browser implements functions to allow the user to add and delete nodes and edges graphically. Adding a node is as simple as positioning the mouse where you would like a corner of the node to be, and then dragging until the outline on the screen is the size you want the node to be. Deleting a node is as simple as selecting it and hitting the delete key. Edges are similarly added and deleted by dragging from a source node to a destination node. Nodes and edges can also be edited, so that attributes such as their label or size or default action (in the case of nodes) can be modified. The functions for adding and deleting can be disabled by the applications programmer for applications that require that the graphs not be changed.

Nodes can have actions associated with them, which are arbitrary lisp expressions. For example, in a hypertext network, choosing a vertex could cause the associated hypertext window to be displayed. Similarly, in an object hierarchy browser double-clicking on a node calls the inspector on the object class named by the node.

The network can be saved in its current form and reloaded. This is useful since users can modify the graph's layout dynamically. When saving the graph, lisp code is generated which will reproduce the graph when evaluated. This also allows a programmer to include the graph directly in his lisp programs, or to modify it using a text editor.

Finally, the code is in place for graphs to be drawn in color, though using color

causes the system performance to deteriorate severly. Simply drawing the graph in color can be performed relatively rapidly, but zooming and roaming become a bit slow. Therefore, the implementation allows color to be turned on and off on the fly so that browsing can be done in black and white with color used for the final view if an application calls for it.

The next section of this document describes how a user sees an application of the browser, including what the specific commands are for zooming, roaming, saving, etc. The following section documents the browser from a programmer's perspective, and follows the style of the Allegro Common Lisp manual, defining each relevant class and operation and variable of that class. Finally, the last section contains a few comments about the applications that have been developed using the browser, and some comments on the efficiency, usefulness, and a wish-list for the tool.

2 User's Manual

The browser package is designed so that users familiar with the Macintosh user interface will be comfortable with it. The only major exception to this is that in the current release arrow keys are used for scrolling instead of scroll bars (see the section on Roaming). This section describes how to access the features of the browser (and naturally, of applications which use the browser). Please note that some of the features can be disabled by an applications programmer, and that applications programmers may even change the way some of the features work or add new features. This manual can only serve as a starting point for applications which use the browser.

2.1 Browsing features

One of the main problems with a browser that allows a very wide range of zooming and roaming is that it is possible to get lost. This section describes how zooming and roaming are done, and then describes some additional features that help traverse the graph in other ways and help to avoid becomming lost in [information] space.

All features can be accessed through the browser menu, but typing shortcuts are almost always available and when they are, they are noted below.

2.1.1 Zoom

Zooming is done by hitting the plus and minus keys. Typing a '+' while the browser is the top window will cause the browser to zoom in on the area of the graph that is centered in the window. Typing a '-' will cause the browser to zoom out.

The operation of zooming corresponds to changing the size of the window on information space. It may also slightly adjust the position of the window on information space in order to center the zoom with respect to the screen window.

2.1.2 Roam

The roam or scrolling feature is accessed by typing the arrow keys. Typing the down arrow moves the window on information space down and causes the screen window to scroll up. Typing the left arrow corresponds to looking left, etc.

Unlike standard Macintosh applications, scroll bars are not used. The reason for this is that scroll bars imply a finite field of view, while the browser does not restrict the distance one can browse in any direction (or at any scale).

2.1.3 Find

The find feature searches the titles of all of the nodes and edges in the graph looking for a specific string. When find is activated, a dialog window appears and the user is asked for a string to find. The first node or edge found containing the string is selected and centered in the window (the appropriate zooming and roaming is done automatically). If find is activated again, the search begins from the selected object,

so repeated searches for the same string will not always find the same object. Note: the concept of order in a two dimensional graph is not well defined, but the browser keeps track of the graph objects internally in a flat list. It is not necessarily true that objects next to each other internally will be near each other on the screen.

The find function can be activated by typing F, or from the browser menu.

2.1.4 Center

The center function automatically zooms and roams to bring the selected objects to front and center of the screen window. Center is activated with the C key or from the browser menu. It will not work if no object is selected (see selecting nodes in the next section). If multiple objects are selected, center works on the whole set.

2.1.5 Home

The home function puts the whole graph inside of the screen window, attempting to use all of the space in at least one direction. The function is useful for finding the graph if you have roamed or zoomed and gotten lost in information space.

2.2 Node Features

This section describes operations on specific nodes. As a user, you can move nodes around, change their attributes such as name, color, size, or action, and delete nodes (unless the application has specifically disallowed any of these actions). Finally, it is also possible to add nodes to the graph.

2.2.1 Move

Moving nodes is the simplest operation. To move a node, simply drag it to its desired location. As you are dragging it, an outline of the node is moved with the mouse cursor. When you release the mouse button, the node and all links associated with it will be redrawn to the new location. It is also possible to move multiple nodes – see the next section.

2.2.2 Select

Selecting is modeled after the Macintosh finder. The following explanation attempts to explain in words the concepts, but working with the mouse is best learned by doing.

Clicking on a node with no modifier keys down will select the node, causing it to be highlighted and causing any items previously selected to become deselected. Following the conventions of the finder, holding the shift key down while clicking on a node will toggle its selectedness and leave any other selected items alone. If the mouse cursor is not over any node, clicking will deselect the selected nodes and dragging will

form a rectangle. When dragging is finished, any nodes inside of the rectangle will become selected.

Moving multiple nodes Clicking on a selected node will move it; if multiple nodes are selected, all will be moved together.

Edit Choosing the edit option with the menu or by typing E when a node is selected allows you to modify some attributes of the selected node. If more than one node is selected, an arbitrary one is edited.

- *Name:* The text that appears in the node (and that is searched on).
- *Shape:* Currently a choice of Rect (rectangle), Round-Rect (rectangle with rounded corners), or Oval.
- *Location:* Exactly where the node is in information space, for fine adjustments of position.
- *Action:* A lisp expression to be evaluated when the node is double-clicked on.
- *Size:* The size of the node in information space.

Delete Removes the selected nodes, and *all edges in or out of them*, from the graph. Be careful: undo is not implemented yet.

Set Color The color of an individual node can be set by choosing the set color option from the menu. The set color option sets the color of the selected nodes (but edges do not have color currently). This feature is only useful if the color option is enabled.

2.2.3 Add

The option key is the add key. Holding down the option key while the mouse is not over a node and dragging will outline a node. When the mouse is released, the new node exists and can be given a label with edit, removed with delete, or moved by dragging.

2.3 Edge Features

This section describes the operations available to a user of a browser on edges. Edges are lines between nodes, which can be drawn with different patterns and may have labels and/or arrowheads. Currently, arrowheads are implemented as boxes 4/5ths of the way from the source to the destination. Edges can be added in much the same way that nodes are added (by holding the option key). Edges are automatically deleted when either node they are adjacent to is deleted, or they can be explicitely deleted.

2.3.1 Move

Moving an edge causes the two nodes connected to the edge to be moved also. Currently, there is no way to click on an edge directly (see selecting an edge, below), so an edge can only be moved as part of a multiple-item move: Select the edge and some other node, and drag the node.

2.3.2 Add

An edge is added by holding the option (add) key down and dragging from the source node to the destination node. While dragging, an outline of the edge is shown from the source node to the mouse. When the mouse button is released, the edge is in place. The operation is cancelled if the mouse button is released while the mouse cursor is over the source node or over no node.

2.3.3 Select

Edges can be selected in one of three ways:

1. By using the command key to draw a box completely around them
2. By using the option key as though to re-add them
3. By holding down the command key and dragging from the source to the destination.

The last way is the preferred way. A selected edge is wide and black; it should be easy to visually identify the selected edges.

Edit Editing edges is done by selecting the edge and activating the edit operation (typing E or choosing Edit from the browser menu). The source and destination of a node cannot be edited; only the label and appearance of the edge may be changed.

The label is the text that appears near the destination of the edge when the option to draw labels is turned on. The appearance is very wide if the edge is selected. When not selected, the edge can currently have one of the following forms:

- a plain edge pattern, the default
- a black edge pattern, slightly thicker than plain but less thick than selected
- a gray edge pattern
- a light edge pattern

Delete Deleting an edge is done in exactly the same way as deleting a node: the edge is selected and then the delete operation is activated with either the menu option or the delete key.

2.4 Miscellaneous

2.4.1 Layout

The layout operation changes the way the graph appears on the screen. It moves the nodes in information space and attempts to make the whole look nice. Sometimes it succeeds. It is impossible to lay a general graph out in such a way that no edges cross (think about a complete graph on 4 nodes). Graphs can take many forms, and often there are multiple (or no) good ways to display them.

The default layout algorithm starts with a single selected node as the root, and holds it constant while moving all nodes which are descendents of that node. A level graph is created to determine how far from the root each node will be placed, and then the nodes are placed in rows based on their distance.

If the graph to be layed out is a tree or simple DAG, laying out from the root may do a reasonable job. The layout can be enhanced by dragging interior nodes to other positions and calling layout again with the interior node selected.

2.4.2 Speed options

Some features, such as color, are simply too expensive to use while browsing. However, rather than simply not provide features which may be expensive, we have chosen to provide them and allow them to be disabled. The options submenu of the browser menu contains features that may be enabled or disabled for speed.

Node Labels Drawing node labels takes extra time, especially if the number of nodes is large. Node labels are not drawn when the nodes are very small, but this option allows you to turn them off all of the time. Default: enabled.

Color As mentioned above, color is expensive. This option allows you to disable color for speed. Default: disabled.

Edge labels The labels on the edges will probably often crowd the display, in addition to being pretty slow. Default: disabled

Edge arrows Edge arrows are implemented as small rectangles four-fifths of the way from the source to the destination. They indicate the direction of the edge. Default: enabled.

Multiple Edge Patterns This option determines whether or not the edges are all displayed as simple lines or as lines with patterns. In practice, enabling this option does not seem to have much effect on the system performance. Default: enabled.

3 Programmer's Manual

Browser

A *browser* is a specialized kind of window that contains *browser-items* connected by lines representing a graph. Browser-items are implemented as objects, and have the property that when clicked on they can perform some action.

Browsers inherit from windows and can perform all window operations. However, some window operations are redefined or shadowed by the browser. Browsers can be saved and loaded similar to text files (the saved form is lisp code to recreate the browser), and save some window information such as the location and size along with the other information more specific to browsers. The save functions are not compiled along with the browser. Instead, source code is provided so that it can be customized for an application.

This manual describes the functions available to an applications programmer to create a browser for a specific application. Each available function and variable is described here. Of course, other functions and variables may have been used in the implementation, but if they are not documented here they are subject to change and should not be used. This manual attempts to follow the style of the Allegro Common Lisp manual as closely as possible.

browser [variable]
Contains the browser class object.

exist *init-list* [*browser* function]
Initializes the browser. The browser will take up space on the Macintosh heap until it is destroyed (with the function self-destruct).

keyword	type	default	meaning
:window-title	string	"Untitled Browser"	the window title of the browser
:window-position	point	#@(5 50)	initial position
:window-size	point	#@(500 150)	size of browser
:window-show	boolean	t	Whether the browser is initially shown. It is useful to set this to nil if many items will be added to the graph, to avoid the overhead of redrawing each item.
:browser-wx	number	0.0	initial x position of the window on information space
:browser-wy	number	0.0	initial y position of the window on information space
:scale	number	20	size of window on information space
:zoom-factor	number	1.5	amount to multiply scale by on each zoom in or out
:scroll-steps	number	20	number of scrolling steps it would take to cross the screen
:window-filename	pathname	nil	Place window should be saved to with window-save
:allow-additions	boolean	t	whether the user can add nodes and edges to the graph
:allow-deletions	boolean	t	whether the user can delete nodes and edges from the graph
:default-item-action	expression	nil	default action to take when browser-items are double-clicked on
:my-menu	menu		menu displayed when browser is on top

add-browser-items *items &key :check-permissions :notify :allow-redraw* [*browser* function]
browser-item-addable *item* [*browser* function]
These functions allow you to add to the graph. Add-browser-items adds objects of class *browser-item* (or a sub-class) to the graph. The keyword parameters are used to decide whether or not to call browser-item-addable, browser-item-added and window-redraw. The default is not to check permissions but to notify through browser-item-added. Browser-item-addable is called each time an item is being added to the browser. If it returns nil, the item is not added. The default version always returns t, but can be shadowed.

make-browser-item *x y size-x size-y* [*browser* function]
This function is called by the browser when the user creates a node. The usual version returns an instance of *browser-item*, but it may be shadowed by an application to return an instance of a subclass of *browser-item*. The shadowing function is responsible for setting the location and size according to the parameters (since the usual make-browser-item would create a *browser-item* different from the item being created by the application program).

delete-selected-items *&key :check-permissions :notify :allow-redraw* [*browser* function]
browser-item-deleted *item* [*browser* function]
Delete-selected-item provides a mechanism for getting rid of nodes (select them and call delete-selected-items). The default for the keyword parameters is to not check permissions but to notify. Browser-item-deleted is called once for each item deleted. It normally does nothing.

browser-scale [*browser* function]
set-browser-scale *new-scale &key :allow-redraw* [*browser* function]
browser-scale-in *&optional zoom-factor* [*browser* function]
browser-scale-out *&optional zoom-factor* [*browser* function]
These functions manipulate the current scale of the browser. Scale is just the size of the window on information space, in information space. The default scale is 20, but scale can be set to any real number. Scale-in and scale-out adjust scale by dividing or multiplying it by the zoom-factor (if none is given, the default is used).

browser-zoom-factor [*browser* function]
set-browser-zoom-factor *new-zoom-factor* [*browser* function]
These functions allow you to adjust the zoom factor. The zoom factor is a real value, but should be greater than 1. Its initial value is 1.5.

browser-wx [*browser* function]
browser-wy [*browser* function]
set-browser-wx *new-wx* [*browser* function]
set-browser-wy *new-wy* [*browser* function]
These functions all you to directly examine or set the position of the window on information space. The point (browser-wx, browser-wy) in information space corresponds to the upper left hand corner of the screen window.

browser-scroll-steps [*browser* function]
set-browser-scroll-steps *new-scroll-steps* [*browser* function]
These functions allow you to adjust the scroll-steps. The scroll-steps is a real value, which indicates how many steps it would take to scroll across the screen. Its initial value is 20.

browser-scroll-left *&optional scroll-steps* [*browser* function]
browser-scroll-right *&optional scroll-steps* [*browser* function]
browser-scroll-up *&optional scroll-steps* [*browser* function]
browser-scroll-down *&optional scroll-steps* [*browser* function]
These functions force the browser to scroll in the appropriate direction.

selected-items [*browser* function]
select *choices &key :allow-redraw* [*browser* function]
deselect *choices* [*browser* function]
These functions allow the programmer to determine which things (nodes or edges) are selected, and to select or deselect any nodes or edges. Of course, nodes should be selected before being deselected.

home [*browser* function]
center-selected-items [*browser* function]
find-item [*browser* function]
center-items *items* [*browser* function]
These functions adjust the window on information space to give a new perspective on the graph. Home sets the window on information space so that it includes the whole graph. It is useful to avoid getting lost in [information] space. Center-selected-items moves the selected items to the center of the screen, and adjusts the scale so that the label on the item can be read. Find item displays a dialog box to get a string from the user, and then selects and centers the first node it finds which contains the string in its label (if one exists). These functions all call center-items, which is the generic item-centering function. The argument *items* is a list of nodes and/or edges.

info-to-screen *x y* [*browser* function]
screen-to-info *p* [*browser* function]
Mechanism for translating between information space coordinates and screen coordinates. These functions are probably not necessary for the programmer, but then we don't know what you may want to do someday...

info-to-screen returns a point unless either of the screen coordinates would be out of the range [-32768, 32767], in which case NIL is returned.

Given a screen point *p*, screen-to-info converts this point into its equivalent place in information space. These coordinates are returned as multiple values from this function, with the x-coordinate first and y-coordinate second. If *p* is nil, then (0,0) is returned.

self-destruct [*browser* function]
Window-close is shadowed to call window-hide, so that clicking in the close box does not destroy the browser. Therefore, self-destruct is included to call window-close and reclaim the space taken up by the window.

browser-colored? *default=nil* [*browser* variable]
browser-draw-labels? *default=t* [*browser* variable]
browser-fancy-edges? *default=t* [*browser* variable]
browser-draw-arrows? *default=t* [*browser* variable]
browser-draw-edge-labels? *default=nil* [*browser* variable]
Variables which control the browser efficiency options (see the user's manual).

default-item-action [*browser* variable]
This variable is funcall'ed with the node double-clicked as the argument. It therefore defines the default action when a node is double clicked. The default default is to print a message to the listener saying that no default is defined. For example, to make the default action for browser b to print the browser item text, you could do:

```
(ask b (setq default-item-action
#'(lambda (x) (print (ask x (browser-item-text))))))
```

In the future, this may be replaced with a *browser-item* function which could be shadowed.

my-menu [*browser* variable]
This variable is bound to the browser menu that appears whenever the browser is the front window.

browser-items [*browser* function]
Returns a list of all of the browser-items in the browser.

edge-exists-p *source dest* [*browser* function]
Provides an easy way to determine whether or not an edge is in the graph. If the edge exists, it is returned.

layout *root &key :notify* [*browser* function]
Layout potentially modifies the positions of any nodes which are descendents of the root node. root must be a *browser-item*. The default layout algorithm is tailored for acyclic graphs, and is not claimed to be optimal by any means. If you don't like it, shadow it. In the usual version, when the notify key is T (the default), layout will call browser-item-moved for each node that is moved.

window-redraw [*window* function]
This function forces a call to window-draw-contents, and is used any time the screen needs to be redrawn.

Browser-items

browser-item [variable]
The browser-item class. *browser-item*'s include all of the information needed by the browser of "things to be browsed".

exist *init-list* [*browser-item* function]
Initializes a browser-item. The browser-item still needs to be added to a browser after being created.

keyword	type	default	meaning
:browser-item-name	atom	(gentemp "BI-")	The name used to refer to the object. The atom will be bound to the object, so it should not be already bound.
:browser-item-text	string	""	Text that will appear as a label on the node. Also used by the find command of the browser.
:browser-item-x	number	0	X location of the item in information space.
:browser-item-y	number	0	Y location of the item in information space.
:browser-item-size-x	number	0.5	The width of the object in information space.
:browser-item-size-y	number	0.5	The height of the object in information space.
:browser-item-shape	atom	'rect	The shape of the item, one of ('rect 'round-rect 'oval).
:browser-item-action	expression	nil	The action to be taken when this item is double-clicked
:browser-item-color	list	(0 0 0)	The color of the item (only used when color is turned on).

draw [*browser-item* function]
The function used to draw the node. May be allowed to be shadowed in the future, but would be risky at this point.

edit *&key :notify* [*browser-item* function]
browser-item-edited [*browser-item* function]
The function called when a node is edited. The default version displays a window allowing the text, action, size, and location to be changed. May be shadowed for specific applications. The usual version examines the value of notify (default t) to

determine whether or not to call browser-item-edited when edit is called.

select-self [*browser-item* function]
deselect-self [*browser-item* function]
These functions are called when the item is selected or deselected; they should not be directly called. They can be shadowed, but should be called by the shadowing function (as usual-select-self or usual-deselect-self). They are provided as a hook for detecting selections.

browser-item-name [*browser-item* function]
Returns the name of the browser item. The name is different from the browser item text – the name is a unique atom. The name cannot be changed once the item is created.

browser-item-text [*browser-item* function]
set-browser-item-text *new-text &key :notify* [*browser-item* function]
browser-item-text-set [*browser-item* function]
These functions allow you to access the text field which is displayed in the graph nodes and used by the find function. When the notify key on set-browser-item-text is set to T (the default), browser-item-text-set is called by set-browser-item-text. Normally, browser-item-text-set does nothing.

browser-item-x [*browser-item* function]
set-browser-item-x *new-x &key :notify* [*browser-item* function]
browser-item-y [*browser-item* function]
set-browser-item-y *new-y &key :notify* [*browser-item* function]
browser-item-size-x [*browser-item* function]
set-browser-item-size-x *new-size-x* [*browser-item* function]
browser-item-size-y [*browser-item* function]
set-browser-item-size-y *new-size-y* [*browser-item* function]
These functions give you examine and modify the location and size of a node. The parameters are all real numbers. The notify keyword parameter determines whether or not the function will cause browser-item-moved to be called (default t).

browser-item-moved [*browser-item* function]
Called whenever the position of the item in information space is changed by dragging the node. The usual version does nothing.

browser-item-shape [*browser-item* function]
set-browser-item-shape *new-shape* [*browser-item* function]
These functions allow you to change the shape of the browser item. The new-shape should be one of the keywords :rect, :round-rect, or :oval.

browser-item-color [*browser-item* function]
set-browser-item-color *new-color* [*browser-item* function]
The color of an item is a list of three integers: red, green, and blue. An item's color can be examined or changed with these functions.

browser-item-action [*browser-item* variable]
This variable is bound to the action to be taken when the node is double-clicked.

browser-item-added [*browser-item* function]
Notifies the item that it has been added to the browser pointed to by my-browser.

browser-item-deletable [*browser-item* function]
Inquires whether the item can be deleted. The usual version just returns t.

delete-self *&key :check-permissions :notify* [*browser-item* function]
Removes this browser item from the browser, as well as all edges going into or out of it. The defaults for the keyword permissions are to notify but not to check permissions.

children [*browser-item* function]
This function returns a list of nodes which are directly connected by edges out of this node.

child-addable *dest* [*browser-item* function]
child-deletable *dest* [*browser-item* function]
These predicates are checked before edges are added or deleted. They always return T, but may be shadowed to inhibit some or all additions or deletions.

add-child *dest &key :check-permissions :notify :allow-redraw* [*browser-item* function]
child-added *dest* [*browser-item* function]
delete-child *dest &key :check-permissions :notify :allow-redraw* [*browser-item* function]
child-deleted *dest* [*browser-item* function]
These functions are hooks to allow the programmer to add or delete edges, and to know when the user adds or deletes edges with the mouse. An edge from this *browser-item* to some other *browser-item* can be added with add-child or deleted with delete-child. The keyword parameters determine whether the check functions (such as child-addable) or notification functions (such as child-added) are called. The defaults are not check-permissions but do notify. child-added and child-deleted are called whenever an edge is added or deleted, respectively (including when add-child or delete-child are called).

edges-out [*browser-item* function]
edges-in [*browser-item* function]

These functions return a list of objects of type *browser-edge* which are the edges out of or into the node, respectively. They are provided to give you a handle to the edge itself, instead of just to the next node.

my-browser [*browser-item* variable]
Holds a pointer to the browser this *browser-item* exists in, if any.

Browser-edges

browser-edge [variable]
The class of objects which can connect browser items in a browser. Browser edges cannot be directly added to the browser; instead, they are added indirectly with calls to add-child (*browser-item* function). The standard way to operate on edges is to create them indirectly, and then modify them with the functions below. Browser edges should not be created with calls to exist.

edge-label [*browser-edge* function]
set-edge-label *new-label* [*browser-edge* function]
These functions allow the programmer to access and modify the label of the edge from *source* to *destination*.

edge-pattern [*browser-edge* function]
set-edge-pattern *new-pattern* [*browser-edge* function]
These functions allow you to access the edge pattern of the edge. The pattern is one of :plain-edge-pattern, :black-edge-pattern, :gray-edge-pattern, or :light-edge-pattern.

select-edge [*browser-edge* function]
deselect-edge [*browser-edge* function]
Edges can be selected just as browser items can. Once selected, edges can be deleted with delete-selected-items (*browser* function). Selected edges are displayed as much thicker lines than normal edges.

edit-edge [*browser-edge* function]
The usual version of edit-edge displays a modal dialog which allows the user to modify the features of the edge such as its label or pattern. This function may be shadowed by an application program.

4 Conclusion

The browser is a useful tool for many aspects of computer science research. It allows the visualization of graphs and tree structures without spending a lot of time writing graphics code. For many applications such as the object-system inheritance lattice browser, the actual application lisp code is less than one page.

The performance of the system also seems reasonable. Screens with many nodes and edges on them can be redrawn in well under a second, and turning off some options from the menu can reduce the time necessary to render graphs with a very large number of nodes or edges.

The browser is not perfect as it is, and the following is our "wish list" of improvements for future versions:

- The draw browser-item function should be shadowable, and should allow arbitrary icons to be used for nodes.
- Nodes should be able to be resized with just the mouse instead of the editor.
- The layout routine can always be enhanced.
- Edit currently does not allow a cancel to revert to the version before changes. It probably should.
- When moving multiple objects, only the outline of the node under the mouse is drawn. Eventually, the outline of the whole block should be drawn.

The authors appreciate any comments, criticisms or enhancements. Please let us know what you think.[†]

Acknowledgements

This work was supported by a generous computer hardware grant to the Washington University Medical Informatics Group from the Advanced Technology Group at Apple Computer, Inc. We cannot say enough good things about the programming environment provided by the Macintosh and Allegro Common Lisp. We are grateful for the support of Mark Frisse, M.D. from the Medical Informatics Group and William Ball, Ph.D. from the Department of Computer Science. We also wish to thank Dan Kimura, Ph.D. from the Department of Computer Science. Dr. Kimura repeatedly asked for features which we had not yet implemented, and he was not satisfied until our program's interface worked like a Macintosh program should!

[†]Both authors can be reached at Department of Computer Science, Campus Box 1045, Washington University, St. Louis, MO 63130, or by electronic mail at sbc@wucs1.wustl.edu or jaf@wucs1.wustl.edu.