Editing graphs with *dotty*

Eleftherios Koutsofios
Stephen C. North

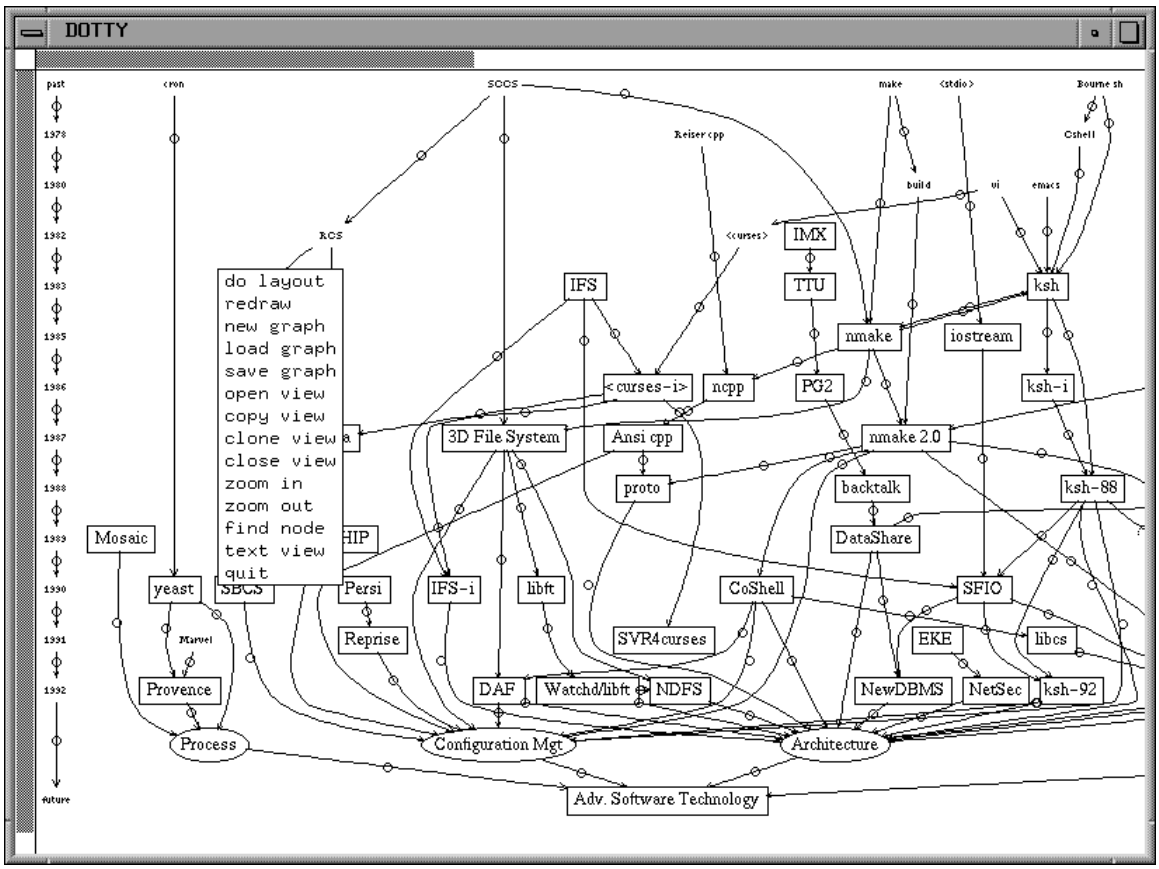
96b (06-24-96)

Abstract

dotty is a graph editor for the X Window System. It may be run as a standalone editor, or as a front end for applications that use graphs. It can control multiple windows viewing different graphs.

dotty is written on top of *dot* and *lefty*. *lefty* is a general-purpose programmable editor for technical pictures. It has an interpretive programming language similar to AWK and C. The user interface and graph editing operations of *dotty* are written as *lefty* functions. Programmer-defined graph operations may be loaded as well. Graph layouts are made by *dot*, which runs as a separate process that communicates with *lefty* through pipes.

The screen dump below shows a snapshot of a typical *dotty* session.



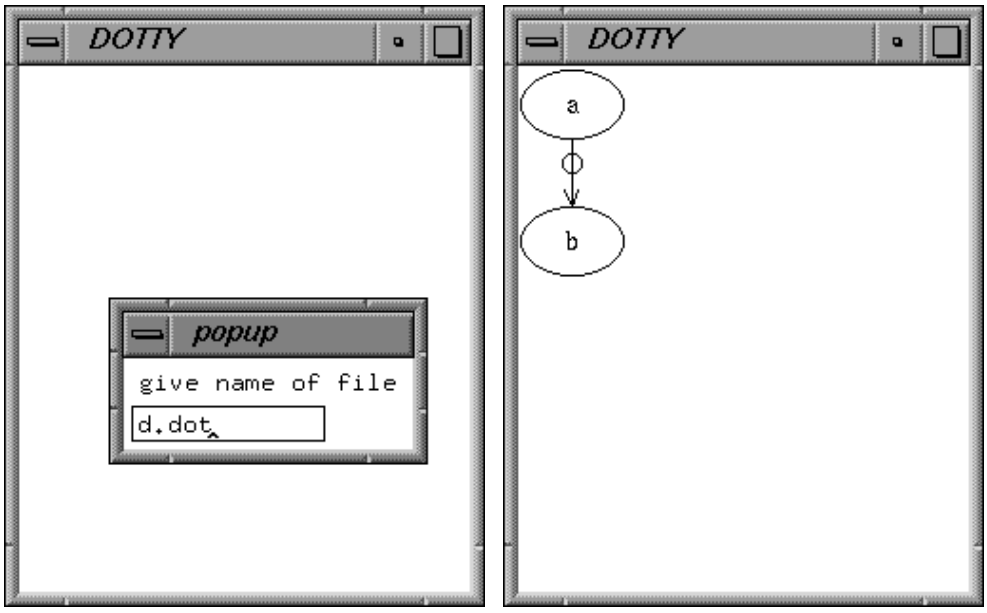
Chapter 1. Overview

dotty is a graph editor built by combining the programmable graphics editor *lefty* [KD91] and the graph layout tool *dot* [GKNV93]. *lefty* has been programmed to operate on internal representations of graphs, and to allow the user to edit graphs. The *lefty* program that implements *dotty* starts up *dot* as a separate process to compute layouts. When the user asks for a new layout, *lefty* sends the graph to *dot*. *dot* computes the layout and outputs the graph (in the graph language notation) with coordinate and size information as graph attributes. *lefty* then redraws the picture using the new layout. *dotty* can manage several windows, displaying different graphs. A future version will support multiple views.

dotty can be customized to handle graphs for specific applications. For example, if a graph is supposed to not have cycles, the user can edit the *lefty* function that inserts edges to check if inserting an edge would create a cycle. *dotty* can also be programmed to communicate with other processes. This allows it to be a front end for other tools. In this case, a tool can download its state to *dotty*, as a graph, and whenever the user changes the graph, *dotty* sends a message to the tool, which changes its state accordingly.

Upon startup, *dotty* opens a window labeled `DOTTY`. The window is empty, unless a graph file name was specified on the command line, in which case the graph is displayed. The user can then add and delete nodes and edges or change attributes such as color and shapes of nodes and edges. *dotty* does not generate new layouts automatically. The user has to ask for a new layout explicitly.

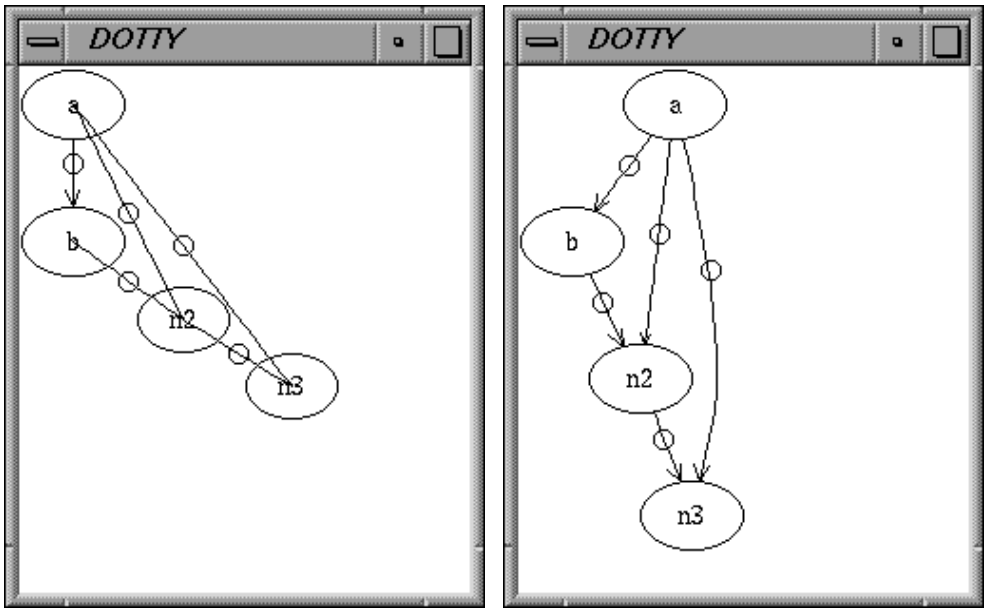
As an example, the user can start up *dotty* and select `load graph` from the menu. Figure 1.1a shows the `DOTTY` window and the dialog window that asks for the graph file name. In this example, the user asks for file `d.dot`. Figure 1.1b shows the result of the `load graph` action.



(a) (b)

Figure 1.1: Loading a graph file

The user can then insert more nodes and edges as shown in Figure 1.2a. Nodes can be inserted by clicking the left mouse button over white space. Edges can be inserted by pressing the middle button over the tail node and then, with the button held down, moving the mouse to the head node and releasing the button. Figure 1.2b shows the graph after the user asks for a new layout.

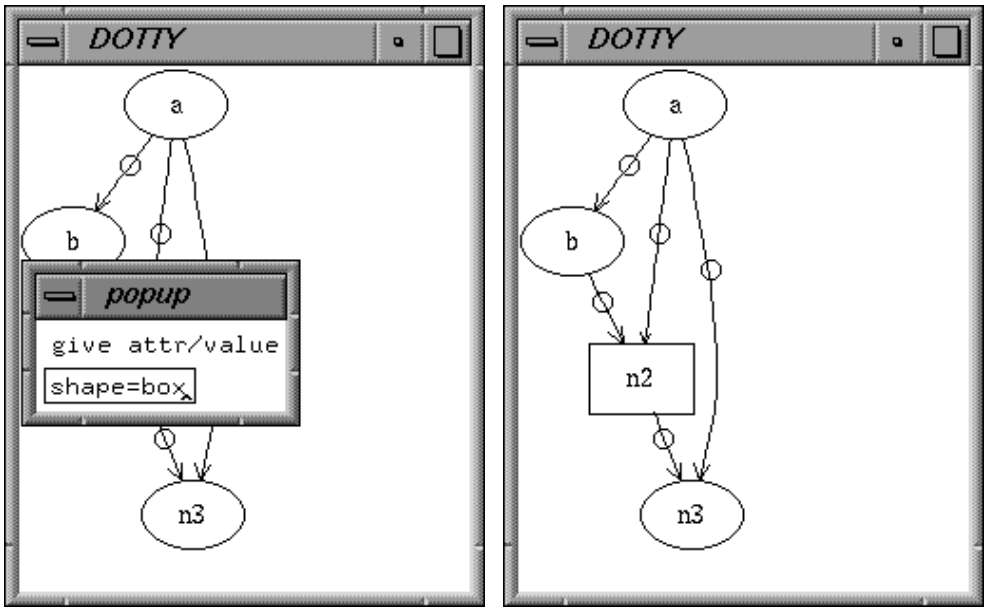


(a) (b)

Figure 1.2: Inserting nodes and edges

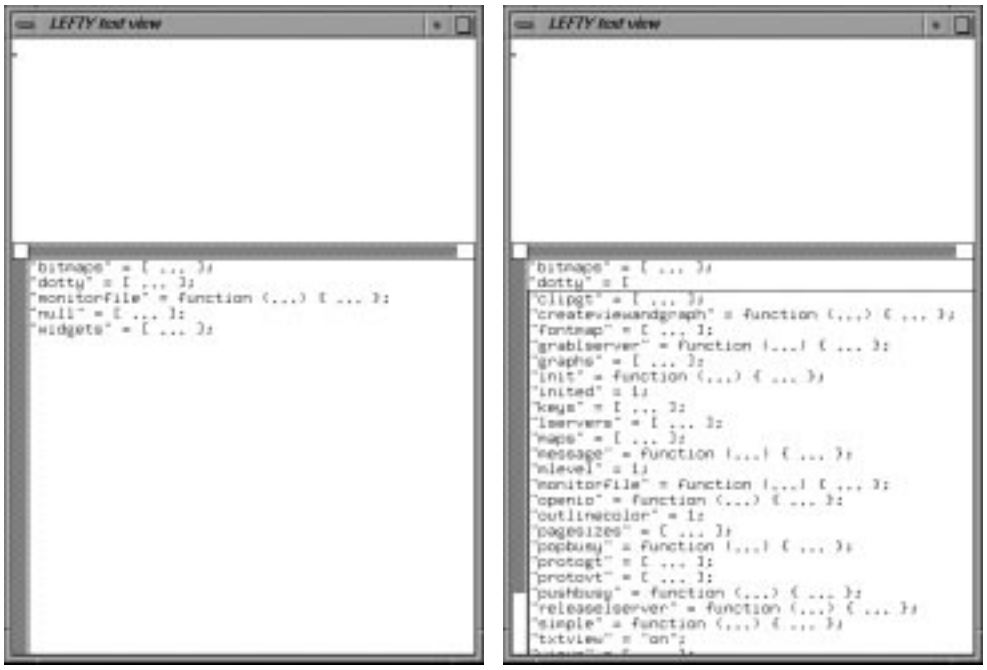
The user can also change node and edge attributes. When the user presses the right mouse button over a node or an edge, a node/edge-specific menu appears. One of its options is `set attr`. Figure 1.3a shows the dialog box that pops up when the user selects that option over node `n2`. In this case, the user is specifying that the shape of node `n2` should be changed to a box. Figure 1.3b shows the result.

The user can examine and change the *lefty* program that implements *dotty* by selecting the option `text view` from the global menu. This opens a window labeled `LEFTY Text View`. The top part of the window can be used to run *lefty* commands. The bottom part shows the current program. Most entries are shown as ellipses (. . .). Clicking on an entry expands it one level. Figure 1.4a shows the text view and Figure 1.4b shows the text view with the `dotty` entry expanded. `dotty` contains all the functions and data structures for *dotty*.



(a) (b)

Figure 1.3: Changing attributes



(a) (b)

Figure 1.4: The text view

Chapter 2. System Description

The user can perform actions from either the WYSIWYG view or the program view. The WYSIWYG view is more intuitive. The program view is more useful for making global changes and for customizing *dotty*. Customizations can be done online, by typing *lefty* expressions at the program view, or by creating a *lefty* script in a file and loading it in.

2.1 Graphical Interface

dotty supports multiple types of WYSIWYG views. Each type of view can have its own way of displaying a graph and its own set of mappings from user actions to graph / picture operations. By default, *dotty* provides two types of views: a normal view and a bird's-eye view.

A normal view displays the graph at 1:1 scale (1 *dot* point to 1 pixel). The widget containing the graph grows or shrinks as the graph grows or shrinks. The user can use the scrollbars of the outer widget to navigate through the graph. The user can also change the scale ratio to zoom in or out.

A bird's-eye view keeps adjusting the scale ratio so that the graph always fits entirely inside the outer widget. As the graph grows or shrinks the scale ratio changes too. When the user clicks on a location on this view *dotty* adjusts the viewing position of all the other views of the same graph to center them on that location.

2.1.1 Normal View

dotty implements the following mouse and keyboard actions for the normal view.

left button. If the button is pressed over a node, that node moves to track the mouse. If the mouse is over white space (not a node or edge) a new node is created at that position.

middle button. If the down event occurs over a node, a rubber-band line is displayed while the mouse tracks over the picture. If the up event occurs when the mouse is over a node, a new edge is created between the two nodes.

right button. This activates one of the next two menus, depending on the selection under the mouse.

global menu. This menu is activated when the right mouse button is pressed over white space.

`undo`	undo the last insert / delete operation.
`paste`	merge the subgraph previously constructed through the `cut` and `copy` operations with the current graph, centering it at the current position.
`do layout`	generate a new graph layout through *dot*.
`cancel layout`	abort the current graph layout operation. Works only if `layoutmode` is set to `async` for the current graph.
`redraw`	clear the window and redraw the graph.
`new graph`	erase the current graph.

`load graph` load a graph from a *dot* file. A requester is displayed, asking for the file name.

`reload graph` reload the current graph from its associated file.

`save graph` save a graph in its associated *dot* file.

`save graph as` save a graph in a *dot* file. A requester is displayed, asking for the file name.

`open view`
`copy view`
`clone view`
`birdseye view`
`close view` create or destroy views. `open` creates a new—empty—view, while `close` closes the current view. `copy` and `clone` make copies of the current view. Views created with `clone` share a single graph; changing the graph from any view results in all views being updated. `birdseye view` is similar to clone, except that the new view is of type bird's-eye.

`zoom in`
`zoom out` make the graph picture bigger or smaller. If the X server supports scalable fonts, the node and edge labels are also scaled appropriately. Otherwise, a font close in size to the desired one is chosen. If the picture does not fit in the window, scrollbars appear to let the user pan on the graph.

`find node` find a node by name. A requester is displayed, asking for the node's name. If a node is found, whose label attribute matches the user response, the graph is redrawn so that this node *appears* centered.

`print graph` prints or stores in a file a printable form of the current graph. Under UNIX, if the user selects to store a file, the file is saved in PostScript format. Under MS Windows, the graph is stored in the metafile format. As a side-effect, the graph is also pasted on the Clipboard.

`text view` toggle the display of the *lefty* text view.

`quit` terminate *dotty*.

node/edge-specific menu.

`cut`
`Cut`
`copy`
`Copy` the lowercase versions of these commands cut or copy the selected node or edge. The uppercase versions cut or copy the selected node or edge and recursively any other nodes and edges connected to the selected one. For example, if there is the path: `A -> B -> C` and the user selects Copy on node `B`, *dotty* will construct a clip graph that contains nodes `B` and `C` and edge `B -> C`.

`group`
`Group` both of these commands ask for an attribute name. If the selected node does not have that attribute, nothing happens. If it does, then `group` replaces all the nodes that have that attribute and also have the same value assigned to

that attribute with a single node. All the edges starting or terminating on one of the nodes to replace, are replaced by edges that start or terminate on the new node.

`delete`
`Delete`
`remove`
`Remove` — `delete` removes the seleced node or edge. `remove` asks for an attribute name. If the selected node does not have that attribute, nothing happens. If it does, then all the nodes that have the same attribute / value pair are removed. The uppercase versions of the commands recursively remove all objects that are only connected to the selected objects.

`set attr` — set an attribute for a node or an edge. Refer to the *dot* manual for a list of its attributes. *dotty* understands most but not all of *dot*'s picture specific attributes. For example, *dotty* understands `shape=`*box* but not `shape=`*polygon*.

`print attr` — print on `stdout` all the attributes associated with the selected object.

keyboard events. Several keys are bound to actions. Currently, all of them are programmed as short-cuts to menu actions. Similarly to menus, some keys work when pressed with the mouse positioned over a node or edge while other work anywhere.

`u`	`undo`
`p`	`paste`
`l`	`do layout`
`k`	`cancel layout`
`space`	`redraw`
`L`	`reload graph`
`s`	`save graph`
`z` or `Z`	`zoom in` or `zoom out` but at a slower rate than the menu commands.
`c` or `C`	`copy` or `Copy`
`g` or `G`	`group` or `Group`
`d` or `D`	`delete` or `Delete`
`r` or `R`	`remove` or `Remove`
`a`	`set attr`

Resizing a `DOTTY` window resizes the window frame only; the layout within maintains its size.

2.2 Programming Language Interface

The *lefty* program that controls *dotty* is contained in several files. These scripts define a set of functions and a set of data structures. All functions and variables are stored under a table called `dotty`.

2.2.1 Data Structures

dotty provides two main classes of objects, graphs and views. All graphs are stored in table `dotty.graphs` and all views in table `dotty.views`.

`protogt`
`graphs`

`protogt` contains the default prototype graph. It is a table. One of its fields is `graph` which contains default values for graph, node, and edge attributes. `protogt` also contains all the functions that operate on the graph, such as `insertnode`. `protogt` is used as an argument to the `creategraph` function. The graph that this function creates uses the graph attribute values specified in `protogt.graph`. All operations on the new graph are performed through the functions specified in `protogt`. *dotty* provides one prototype graph. The programmer can create new prototype graphs as well as partial prototypes that simply override some functions or data values. `graphs` is a table that contains all the currently active graphs. The naming convention we use is that each sub-table in `graphs` is called `gt`. This is done to distinguish the table containing all the graph data and functions from `gt.graph`, the table that contains just the graph data.

`protovt`
`views`

`protovt` contains the default prototype view. Like `protogt`, this table contains both the data and the functions that manage a view. For example, one of the data items is `vsize`. This is a table that contains the x,y size of the view in pixels. One of the function items is `leftdown` that is the function called when the user presses the left mouse button inside the view. `protovt` is used in function `createview` to specify the size, position, and display characteristics of the new view. *dotty* provides two prototype views, `dotty.protovt.normal` and `dotty.protovt.birdseye`. The programmer can create new prototype views as well as partial prototypes that simply override some functions or data values. `views` is a table that contains all the currently active views for all graphs. The naming convention we use is that each sub-table in `views` is called `vt`.

`gt.graph` contains several tables. The three most imprortant are:

`nodedict`
`nodes`
`edges`

`nodedict` contains a mapping of node names to node ids. `nodes` is the global table of nodes, indexed by node id. `edges` is the global table of edges, indexed by edge id. Node and edge ids are small unique integers. Each node and edge entry is a table. Each entry has a subtable called `attr` that contain all the key-value attribute pairs. Each node table has an edges sub-table containing all the edges that start or terminate on that node. Each edge table has two entries called `tail` and `head` that are references to the two nodes that this edge connects.

2.2.2 Graph Functions

`dotty.init function ()`

Initializes *dotty*.

```
gt.creategraph (protogt)
gt.copygraph (ogt)
gt.destroygraph (gt)
gt.loadgraph (gt, name, type, protograph, layoutflag)
gt.savegraph (gt, name, type, savecoord)
gt.setgraph (gt, graph)
gt.erasegraph (gt, protogt, protovt)
gt.layoutgraph (gt)
dotty.monitorfile (data)
```

`creategraph` creates and returns a new graph based on the `protogt` prototype graph. If `protogt` is `null`, `dotty.protogt` is used instead. If `protogt.mode` is set to `'replace'`, then `protogt` must contain all the entries for a prototype graph. Otherwise, any entries not specified in `protogt` are taken from `dotty.protogt`. `copygraph` creates and returns a new graph by copying an older graph (`ogt`). `destroygraph` removes a graph and all its views. `loadgraph` reads in a graph. `type` specifies whether to read the graph from a file (value `'file'`) or a pipe (`'pipe'`). `protograph` is a prototype for the graph data. If set to `null`, `dotty.protogt.graph` is used. If `layoutflag` is `1` then this function runs `gt.layoutgraph (gt)` before returning. `savegraph` saves a graph to a file or pipe. If `savecoord` is `1`, the node and edge coordinates are saved in the graph. `setgraph` sets the graph data of `gt` to `graph`. `erasegraph` replaces the graph data of `gt` with `protogt.graph` and resets all the views for `gt` to the values of `protovt`. `layoutgraph` initiates a graph layout for graph `gt`. This involves writing out the graph to a pipe connected to a graph layout process, usually *dot*. If `gt.layoutmode` is set to `'sync'`, the function waits for the *dot* process to return the layout information, updates the graph coordinates and redraws the graph in all its views. If `gt.layoutmode` is set to `'async'` then the function returns. To complete the layout, function `dotty.monitorfile` must be called. By default, *dotty* assigns `dotty.monitorfile` to `monitorfile`. This instructs *lefty* to call this function whenever there is input from a registered file descriptor. `layoutgraph` registers the file descriptor of the *dot* pipe. If *dotty* is used in another application this application must arrange for the same effect. If the application needs to have its own `monitorfile`, it must arrange for that function to call `dotty.monitorfile` if the file descriptor is a *dot* file descriptor. `dotty.monitorfile` returns `1` if it was one of the file descriptors it handles, `0` otherwise.

```
gt.createview (gt, protovt)
gt.destroyview (gt, vt)
```

`createview` creates and returns a new view based on the `protovt` prototype view. If `protovt.mode` is `null`, `dotty.protovt` is used instead. If `protovt.mode` is set to `'replace'`, then `protovt` must contain all the entries for a prototype view. Otherwise, any entries not specified in `protovt` are taken from `dotty.protovt`. `destroyview` destroys a view.

```
gt.zoom (gt, vt, factor, pos)
```

zooms in or out in a view. If `factor` is less than `1`, the graph becomes larger. If `factor` is greater than `1` the graph becomes smaller.

```
gt.findnode (gt, vt)
```

asks for a node name or label. If such a node exists, the view is repositioned so that this node appears at the center.

```
gt.setattr (gt, obj)
gt.getattr (gt, node)
```

`setattr` asks for a `key=val` response from the user then sets the object's attribute `key` to `val`. `obj` must be a node or edge object. `getattr` asks for a `key` name. If the specified node has an attribute called `key`, this function returns a table with two fields, `key` and `val`.

```
dotty.createviewandgraph (name, type, protogt, protovt)
dotty.simple (file)
```

`createviewandgraph` is a utility function that calls `creategraph`, `createview`, and `loadgraph`. `simple` calls `createviewandgraph` with arguments `(file, 'file', null, null)`.

```
dotty.pushbusy (gt, views)
dotty.popbusy (gt, views)
```

These functions implement a stack. As long as there is something on the stack, the shape of the mouse pointer is the hourglass icon. When the stack becomes empty, the pointer reverts to its default icon.

```
gt.printorsave (gt, vt, otype, name, mode, ptype)
```

This function prints out a graph or saves it in a file. `otype` can be `printer` or `file`. If it is `file`, the UNIX version of *dotty* will generate a PostScript file. The MS Windows version will generate a Metafile and also post the picture to the Clipboard. `name` is the name of the file when `otype` is `file`, otherwise it's ignored. `mode` can be `portrait`, `landscape`, or `best fit`, to specify the orientation of the drawing. `ptype` can be `8.5x11`, `11x17`, or `36x50` to select the paper size. All arguments after `vt` may be `null` in which case *dotty* will prompt the user for values.

```
gt.getnodesbyattr (gt, key, val)
gt.reachablenodes (gt, node)
```

`getnodesbyattr` return a table of nodes that contain the `key=val` attribute pair. This table is indexed by node id. `reachablenodes` returns a table of nodes that are reachable through one or more levels of edges from `node`.

```
gt.insertsgraph (gt, name, attr, show)
gt.removesgraph (gt, sgraph)
gt.mergegraph (gt, graph, show)
gt.insertnode (gt, pos, size, name, attr, show)
gt.removenode (gt, node)
gt.insertedge (gt, nodea, porta, nodeb, portb, attr, show)
gt.removeedge (gt, edge)
gt.swapedgeids (gt, edge1, edge2)
gt.removesubtree (gt, obj)
gt.removenodesbyattr (gt, key, val)
gt.removesubtreesbyattr (gt, key, val)
gt.groupnodes (gt, nlist, gnode, pos, size, attr, keepmulti, show)
gt.groupnodesbyattr (gt, key, val, attr, keepmulti, show)
gt.cut (gt, obj, set, mode, op)
```

```
gt.paste (gt, pos, show)
gt.undo (gt, show)
gt.startadd2undo (gt)
gt.endadd2undo (gt)
```

These functions manipulate the graph structure. `insertsgraph` inserts a subgraph in graph `gt`. `removesgraph` removes a subgraph. `mergegraph` merges `graph` into `gt.graph`. Each pair of nodes with the same name in both graphs are merged into a single node. `insertnode` inserts a new node at position `pos` with size `size`. `pos` and `size` may be `null`. `removenode` removes a node. `insertedge` inserts an edge between the two nodes referenced by `nodea` and `nodeb`. `porta` and `portb` are strings that correspond to *dot* edge ports. `removeedge` removes an edge. `swapedgeids` swaps the edge ids of the edges referenced by `edge1` and `edge2`. Nodes and edges are sent to *dot* sorted by their ids. `removesubtree` removes the node or edge referenced by `obj` and recursively all nodes that are only connected to nodes deleted in the previous phase. For example, if we call `removesubtree` on node `A` in the graph `A -> B -> C, D -> C`, it will delete nodes `A` and `B` but not `C` because `C` is also connected to `D`. `removenodesbyattr` and `removesubtreebyattr` remove all nodes (or all nodes and their connected subtrees) that contain the `key=val` attribute pair. `groupnodes` replaces the list of nodes in table `nlist` with a single node. This node is `gnode` if not `null`, or a new node created by `groupnodes`. `keepmulti` may be `0` or `1` to indicated whether to eliminate multiple edges to and from the group node. With `keepmulti` set to `1`, if we group nodes `B` and `C` together in the graph `A -> B, A -> C` we get the graph `A -> NEW, A -> NEW`. `groupnodesbyattr` calls `groupnodes` with `nlist` set to the list of nodes that contain the `key=val` attribute pair. `cut` cuts or copies pieces of a graph to the clipgraph. `obj` is a reference to a node or edge. `set` may be `one` or `reachable` to select just the object, or the object and its reachable set. `mode` can be `support` or `normal`. If set to `support`, the selection will include edges that have only one of their endpoints attached to a node in the selection (by default only edges with both endpoints in the selection are included). To accomplish this, new *support* nodes are created to replace the nodes not in the selection. `op` may be `'cut'` or `'copy'`. `paste` merges the current clipgraph with graph `gt`. `undo` undoes the most recent insertion or deletion of nodes or edges. *dotty* keeps an undo list per graph going back to the last major operation, such as `loadgraph`. `startadd2undo` and `endadd2undo` can be used to group multiple insert and delete operations in one undo entry. For example, `removesubtree` calls `startadd2undo` before removing any nodes and `endadd2undo` afterwards. When undo is called to undo the subtree removal, all the nodes and edges of the subtree will be re-inserted at once. The `attr` argument in all these functions is a list of key-value pairs to be attached to the node or edge specified in the call. `show` can be `0` or `1` to indicate that the inserted object must be displayed at once or not. Removed objects are immediately removed from the display.

```
gt.startlayout (gt)
gt.finishlayout (gt)
gt.cancellayout (gt)
```

`startlayout` sends a graph to a layout process, usually *dot*. `finishlayout` receives the layout results and updates the positions and sizes of the nodes and edges. It does not redraw the graph. If `gt.layoutmode` is `'sync'`, `gt.layoutgraph` calls `startlayout` and then `finishlayout`. If the mode is `'async'` `gt.layoutgraph` calls just `startlayout` and `finishedlayout` is called from `dotty.monitorfile`. `cancellayout` cancels the layout pending for the graph. This is only possible when `gt.layoutmode` is `'async'`.

```
gt.drawgraph (gt, views)
gt.redrawgraph (gt, views)
gt.setviewsize (views, r)
gt.setviewscale (views, factor)
gt.setviewcenter (views, center)
gt.getcolor (views, name)
```

`drawgraph` draws the graph in all the `views`. `views` is a table of views. It can be `gt.views`, i.e. all the views that this graph has, or a subset of them. `redrawgraph` first clears the canvases then draws the graph. `setviewsize` sets the size of the canvases that contain the graph. `finishlayout` calls this function to adjust the size of the canvases after a layout is complete. `setviewscale` sets the scale factor of the views. `setviewcenter` adjusts the relative position of the canvases to their outer widgets so that point `center` appears in the middle of the outer widget. `getcolor` returns a color id corresponding to the color `name`. `name` is a string. It may contain a color name such as `blue`, or a triplet of HSV values. If such a color was used before in a graph, `getcolor` simply returns the color id assigned to the color, otherwise it allocates a new id, sets its color value to `name` and returns the id.

```
gt.drawsgraph (gt, views, sgraph)
gt.undrawsgraph (gt, views, sgraph)
gt.drawnode (gt, views, node)
gt.undrawnode (gt, views, node)
gt.movenode (gt, node, pos)
gt.drawedge (gt, views, edge)
gt.undrawedge (gt, views, edge)
```

`drawsgraph` and `undrawsgraph` draw or erase subgraph `sgraph` in all the `views`. Only cluster subgraphs are currently drawn. `drawnode` and `undrawnode` draw or erase `node` in `views`. They use the node's `shape` attribute to select a function in `gt.shapefunc`. `movenode` repositions `node` to location `pos`. All the edges attached to the node move too. `drawedge` and `undrawedge` draw or erase `edge`.

```
gt.shapefunc.record (gt, canvas, node)
gt.shapefunc.plaintext (gt, canvas, node)
gt.shapefunc.box (gt, canvas, node)
gt.shapefunc.Msquare (gt, canvas, node)
gt.shapefunc.ellipse (gt, canvas, node)
gt.shapefunc.circle (gt, canvas, node)
gt.shapefunc.doublecircle (gt, canvas, node)
gt.shapefunc.diamond (gt, canvas, node)
gt.shapefunc.parallelogram (gt, canvas, node)
gt.shapefunc.trapezium (gt, canvas, node)
gt.shapefunc.triangle (gt, canvas, node)
```

These are the shape functions provided by *dotty*. `canvas` is the widget id of the canvas widget to draw in. The programmer can replace any or all of these functions with different versions. The programmer could also add functions for new shapes. In this case, however, the programmer needs to arrange for *dot* to recognise these shapes.

```
gt.unpacksgraphattr (gt, sgraph)
gt.unpacknodeattr (gt, node)
gt.unpackedgeattr (gt, edge)
gt.unpackattr (gt)
```

These functions convert attributes from their string representations in the `attr` sub-table of the graph objects, to values that can be used by *lefty*. Some attributes such as `fontsize` are just converted from strings to integers. Other need more complex translations. For example, function `getcolor` is used on all `color` and `fontcolor` attributes. The first three functions operate on a single object, while the fourth operates on all the objects of the graph. `unpackattr` is called by `loadgraph` after a graph is read in. The first three functions are usually called to change the appearance of an object after one of its attributes has changed. This is done with a sequence like: `gt.undrawnode (gt, node); node.attr.color = 'blue'; gt.unpacknodeattr (gt, node); gt.drawnode (gt, node);`. These functions are not used to convert layout information received from a layout process, which is also received as *dot* string attributes. `finishlayout` does that immediately.

```
gt.doaction (data, s)
```

This function takes a string specifying an action (`s`) and a table (`data`) specifying the graph object and a screen location. If the object `data.obj` is a node or an edge, `doaction` calls the appropriate action function with arguments: `(gt, vt, data.obj, data)`. If `data.obj` is `null`, it calls a global action function with arguments `(gt, vt, data)`. Each graph contains a table `gt.actions` with three sub-tables: `general`, `node`, and `edge`. Each sub-table contains key-value pairs where the key is the name of an action and the value is the corresponding function.

```
gt.actions.general["undo"] (gt, vt, data)
gt.actions.general["paste"] (gt, vt, data)
gt.actions.general["do layout"] (gt, vt, data)
gt.actions.general["cancel layout"] (gt, vt, data)
gt.actions.general["redraw"] (gt, vt, data)
gt.actions.general["new graph"] (gt, vt, data)
gt.actions.general["load graph"] (gt, vt, data)
gt.actions.general["reload graph"] (gt, vt, data)
gt.actions.general["save graph"] (gt, vt, data)
gt.actions.general["save graph as"] (gt, vt, data)
gt.actions.general["open view"] (gt, vt, data)
gt.actions.general["copy view"] (gt, vt, data)
gt.actions.general["birdseye view"] (gt, vt, data)
gt.actions.general["clone view"] (gt, vt, data)
gt.actions.general["close view"] (gt, vt, data)
gt.actions.general["zoom in"] (gt, vt, data)
gt.actions.general["zoom out"] (gt, vt, data)
gt.actions.general["zoom in slowly"] (gt, vt, data)
gt.actions.general["zoom out slowly"] (gt, vt, data)
gt.actions.general["find node"] (gt, vt, data)
gt.actions.general["print graph"] (gt, vt, data)
```

```
gt.actions.general["text view"] (gt, vt, data)
gt.actions.general["quit"] (gt, vt, data)
```

These are the default global actions. They are invoked by the default global menu and key bindings, but the programmer can also call them directly.

```
gt.actions.node["cut"] (gt, vt, obj, data)
gt.actions.node["Cut"] (gt, vt, obj, data)
gt.actions.node["copy"] (gt, vt, obj, data)
gt.actions.node["Copy"] (gt, vt, obj, data)
gt.actions.node["group"] (gt, vt, obj, data)
gt.actions.node["Group"] (gt, vt, obj, data)
gt.actions.node["delete"] (gt, vt, obj, data)
gt.actions.node["Delete"] (gt, vt, obj, data)
gt.actions.node["remove"] (gt, vt, obj, data)
gt.actions.node["Remove"] (gt, vt, obj, data)
gt.actions.node["set attr"] (gt, vt, obj, data)
gt.actions.node["print attr"] (gt, vt, obj, data)
```

These are the default node actions. They are invoked by the default node-specific menu and key bindings, but the programmer can also call them directly.

```
gt.actions.edge["cut"] (gt, vt, obj, data)
gt.actions.edge["Cut"] (gt, vt, obj, data)
gt.actions.edge["copy"] (gt, vt, obj, data)
gt.actions.edge["Copy"] (gt, vt, obj, data)
gt.actions.edge["group"] (gt, vt, obj, data)
gt.actions.edge["Group"] (gt, vt, obj, data)
gt.actions.edge["delete"] (gt, vt, obj, data)
gt.actions.edge["Delete"] (gt, vt, obj, data)
gt.actions.edge["set attr"] (gt, vt, obj, data)
gt.actions.edge["print attr"] (gt, vt, obj, data)
```

These are the default edge actions. They are invoked by the default edge-specific menu and key bindings, but the programmer can also call them directly.

2.2.3 View Functions

A view table contains a set of user interface functions. These are called with argument `data`, which is the standard *lefty* table passed to a user interface function. It contains subfields `widget`, `obj`, and `pos`. If the user event is a mouse button or keyboard key up event, the table also contains `pobj` and `ppos`, corresponding to the fields `obj` and `pos` of the matching down event. *dotty* provides two set of functions, one for the normal view and one for the bird's-eye view.

```
vt.uifuncs['leftdown'] (data)
vt.uifuncs['leftmove'] (data)
vt.uifuncs['leftup'] (data)
vt.uifuncs['middledown'] (data)
```

```
vt.uifuncs['middlemove'] (data)
vt.uifuncs['middleup'] (data)
vt.uifuncs['rightdown'] (data)
vt.uifuncs['keyup'] (data)
vt.uifuncs['redraw'] (data)
vt.uifuncs['closeview'] (data)
```

Both the normal and the bird's-eye view sets contain these functions. The `left*` functions perform different operations in the two sets but the rest are identical. All but the last two functions perform the operations described at the beginning of this chapter. `redraw` is called by *lefty* when it receives a re-paint event from the window system. `closeview` is called when *lefty* receives an event to close one of its top-level widget. If this function does not exist, *lefty* will exit when the user tries to close one of the views. This function needs to call the `destroyview` function if it decides to allow the closing. Function `rightdown` uses the table `vt.menus` to construct a menu to display. `vt.menus` has three sub-tables called `general`, `node`, and `edge`. Function `keyup` uses a similar table (`vt.keys`) to map keys to actions.

Chapter 3. Customizing *dotty*

An important aspect of *dotty* is that it can readily be customized to handle application-specific graphs. A simple type of customization is to change the user interface functions such as `leftup` to do something different. More complex customizations could involve adding new editing operations and setting up communication channels with other processes.

The rest of this section provides some general guidelines for customizing *dotty* and some examples.

3.1 General Guidelines

The scripts in the `dotty*.lefty` files do not perform any actions (besides loading several functions and data structures). Therefore, it should not be necessary to change these files. The simplest approach is to create a new *lefty* script, say `new.lefty`. This script could load `dotty.lefty`, define new functions and overide predefined functions, and finally start up the main event loop:

```
load ('dotty.lefty');
new.protogt = [
    'actions' = copy (dotty.protogt.actions);
    # new actions are added later in this file
];
new.protovt = [
    'name' = 'NEW';
    'type' = 'normal';
    # other entries are added later in the file
];
new.protogt.actions.general['play fwd'] = function (gt, vt, data) {
    while (new.next (gt))
        ;
};
new.protovt.uifuncs.leftup = function (data) {
    local gt;
    gt = dotty.graphs[dotty.views[data.widget].gtid];
    if (new.next (gt) == 0)
        echo ('at end of log');
};
new.main = function () {
    dotty.init ();
    dotty.createviewandgraph (null, 'file', new.protogt, new.protovt);
    ...
};
new.main ();
```

dotty uses the `LEFTYPATH` environment variable to find files to load. This is a colon separated list (like `PATH`). Thus, any files you use for customization should be in directories pointed to by `LEFTYPATH`.

In general, data structures can be read directly. For example,

```
a = dotty.views[10].colors.blue;
```

assigns the color id that is mapped to 'blue' to variable `a`. Updating values, however, must be done through functions. For example, to make a new color mapping, one should call

```
gt.getcolor ([10 = dotty.views[10];], 'pink');
```

instead of doing the assignment

```
dotty.views[10].colors['pink'] = 13;
```

since more is required than just adding another entry to the `colors` table.

A graph contains the following fields.

```
graphattr = [ ... ];
graphdict = [ ... ];
graphs = [ ... ];
edgeattr = [ ... ];
edgedict = [ ... ];
edges = [ ... ];
nodeattr = [ ... ];
nodedict = [ ... ];
nodes = [ ... ];
```

The structure of these fields is very similar to that of *dot*. `graphs` is an array of subgraphs. `nodes` is an array of the nodes in the graph. `edges` is the array of edges. Each node contains an `edges` table that holds all the edges attached to the node.

Node- or edge-specific information can be added to one of two places:

under the object's attr table, e.g. `node.attr.state = 'NJ';`

under the object's main table, e.g. `node.state = 'NJ';`

Information added to the `attr` table is sent to *dot* when the graph is drawn, and is also saved in the graph file. This means that one should not store table objects in `attr` since *dot* does not understand them.

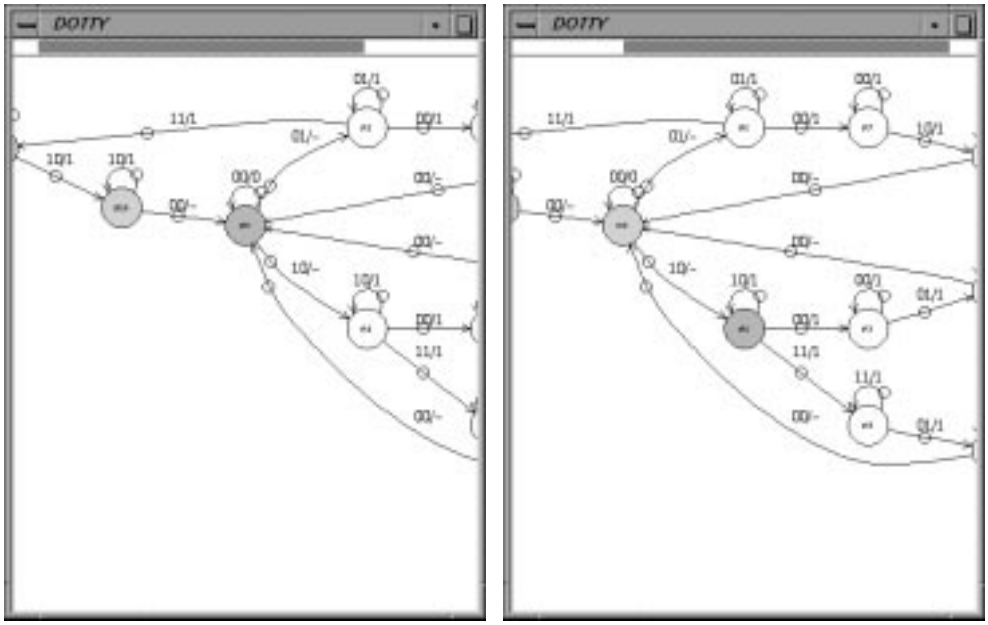
If an attribute has to be mapped in some way (between *dot*'s representation, and *dotty*'s internal values) the mapping is stored in the node's main table. For example, if `node.attr.color` is `'blue'`, `node.color` may contain the internal color index `2`.

3.2 Examples

Appendix B contains the complete source for the first example.

3.2.1 Finite Automaton Simulator

In this tool, an automaton is displayed as a directed graph. A sequence of state transitions can be loaded in and animated either in single step or in continuous mode, forwards or backwards. If the graph does not fit in the window, the window is scrolled to always keep the current node visible. Figure 3.1 shows two consecutive snapshots.



(a) (b)

Figure 3.1: Finite Automaton Simulator

After loading the graph that represents an automaton, `fa.lefty` also loads a sequence of state transitions from a file.

```
fa.loadtrans = function (filename) {
    local fd, i;
    if (~((fd = openio ('file', filename, 'r')) >= 0)) {
        echo ('cannot open transition file: ', filename);
        return;
    }
    echo ('reading transition file');
    i = 0;
    while ((fa.trans[i] = readline (fd)))
        i = i + 1;
    closeio (fd);
    fa.trani = 0;
    fa.states[0] = fa.currstate;
    fa.currstate.count = 1;
};
```

Array `fa.states` stores all the states the automaton goes through while executing the specified transitions. `fa.next` executes the *next* transition (pointed to by `fa.transi`). First, it finds the edge in the graph that corresponds to the transition. If such an edge exists, the node at the head of the edge becomes the *current* node and is highlighted. Any nodes that the automaton has been through also use a distinctive color. All other nodes are white.

```
fa.next = function (gt) {
    local label, eid, edge, tran;
    if (~(label = fa.trans[fa.trani]))
        return 0;
    for (eid in fa.currstate.edges) {
        edge = fa.currstate.edges[eid];
        if (edge.attr.label == label & edge.tail == fa.currstate) {
            tran = edge;
            break;
        }
    }
    if (tran) {
        fa.trani = fa.trani + 1;
        fa.setcolor (gt, fa.currstate);
        fa.currstate = (fa.states[fa.trani] = tran.head);
        if (fa.currstate.count)
            fa.currstate.count = fa.currstate.count + 1;
        else
            fa.currstate.count = 1;
        fa.setcurrcolor (gt, fa.currstate);
        if (fa.trackcur)
            fa.focuson (gt, fa.currstate);
    }
    return 1;
};
```

`fa.prev` moves backwards through the transition sequence. `fa.leftup` advances one step forward in the transition log by calling `fa.next`. `fa.middleup` moves one step backwards.

```
fa.protovt.uifuncs.leftup = function (data) {
    local gt;
    gt = dotty.graphs[dotty.views[data.widget].gtid];
    if (fa.next (gt) == 0)
        echo ('at end of log');
};
```

`fa.setcolor` sets the color of the node that used to be the current node. `node.count` keeps track how many times the node has been visited and sets its color accordingly

```
fa.setcolor = function (gt, node) {
    if (node.count) {
        node.attr.style = 'filled';
        node.attr.color = fa.highlightcolor;
    } else {
        gt.undrawnode (gt, gt.views, node);
        remove ('style', node.attr);
        node.attr.color = fa.normalcolor;
    }
    gt.unpacknodeattr (gt, node);
    gt.drawnode (gt, gt.views, node);
};
```

`fa.setcurrcolor` sets the color of the current node. `fa.focuson` adjusts the graph window so that `node` appears at the center.

```
fa.focuson = function (gt, node) {
    gt.setviewcenter (gt.views, node.pos);
};
```

3.2.2 *ldbx*: Graphical Display of Data Structures

ldbx is a prototype two-view debugger built by combining *dotty* with *dbx*, the system's standard debugger. In *ldbx*, the user can access *dbx* by typing commands as usual. At the same time, the user can also ask *dotty* to display the value of a variable graphically. Figure 3.2 shows a snapshot of *ldbx*. The top window is a *dotty* window. The bottom window is an `xterm` window. The user has typed several debugger commands in the bottom window. Some of these commands displayed values of variables. In the *dotty* window, the user has asked *ldbx* to print the value of `result` (the root of the graph). The user has then clicked over the boxes that contain pointer values (the dots) to show the structure the pointer was pointing to.

ldbx runs the system debugger as a separate process. A multiplexing process allows both the user and *dotty* to communicate with the debugger. When the user types a command in the window on the right, the multiplexor sends it to *dbx* and displays the response in the same window. When *dotty* sends a message, the multiplexor sends the message to *dbx* and collects the response. It parses the response and generates a graph. It then sends this graph to *dotty*, in the form of *lefty* statements. `ldbx.doquery` handles the communication with the multiplexing process.

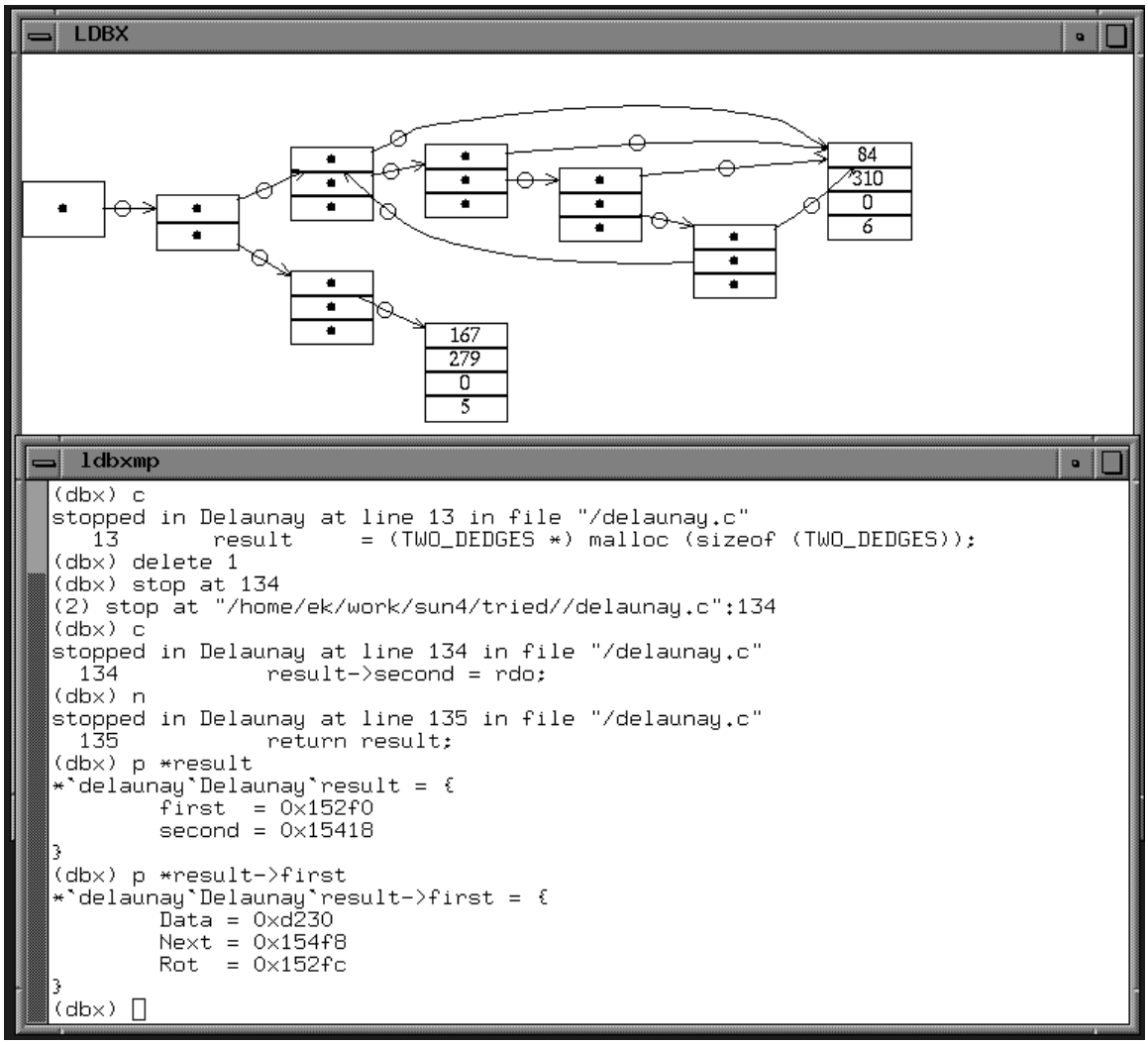


Figure 3.2: Visual Display of Data Structures

```
ldbx.doquery = function (var) {
    ...
    if (var) {
        writeline (ldbx.mpfd, concat ('print ', var));
        ldbx.gt.graph = ldbx.gt.erasegraph (ldbx.gt,
                ldbx.protogt, ldbx.protovt);
        ldbx.nodes = [];
        while ((s = readline (ldbx.mpfd)) ~= '')
            run (s);
        ldbx.gt.unpackattr (ldbx.gt);
        ldbx.gt.layoutgraph (ldbx.gt);
    }
};
```

dotty sends the message `print` *variable* and then processes any response. The response is a sequence of *lefty* statements, such as calls to `ldbx.insertnode` and `ldbx.insertedge`.

```
ldbx.insertnode = function (ident, data) {
    local node;
    if (~(node = ldbx.nodes[ident]))
        node = ldbx.gt.insertnode (ldbx.gt, null, null, null,
                ['ident' = ident;], 0);
    node.data = data;
    node.attr.label = concat ('{', ldbx.makelabel (node, node.data), '}');
    ldbx.nodes[ident] = node;
};
ldbx.insertedge = function (identa, portida, identb, portidb) {
    if (~ldbx.nodes[identa])
        ldbx.nodes[identa] = ldbx.gt.insertnode (ldbx.gt, null, null, null,
                ['ident' = identa;], 0);
    if (~ldbx.nodes[identb])
        ldbx.nodes[identb] = ldbx.gt.insertnode (ldbx.gt, null, null, null,
                ['ident' = identb;], 0);
    ldbx.gt.insertedge (ldbx.gt, ldbx.nodes[identa], concat ('f', portida),
            ldbx.nodes[identb], concat ('f', portidb), null, 0);
};
```

Chapter 4. Related Work

The first interactive graph browser as such appears to be the GRAB system created by Messinger, Rowe, et al at U.C. Berkeley [RDM+87]. GRAB can read command files or scripts but has no provision for customization. Newbery's EDGE and Himsolt's GRAPHED are more recent designs. EDGE [NT87], written in C++, is customized by deriving new classes from EDGE classes. GRAPHED [Him], a large C program, is customized by linking in user-defined event handlers. GRAPHED contains a rich set of data structures for base graphs, layouts, and graph grammars, as well as its representations of commands and events. Its orientation seems to be graph layout algorithms, not user interface, thus almost all the user-contributed applications distributed with the system are a thousand lines of code or more.

The advantage of *dotty* as compared to programming with a C or C++ graph display library is that *dotty* (*lefty*) is higher level and thus seems more appropriate for graphical user interface customization. In our experience, the graph algorithms or interaction techniques that users want to add to the base graph viewer are generally straightforward. We feel that designing these as *lefty* scripts is a good alternative to trying to compile and link a modest piece of C code into a much larger existing program. The limitation here is that to some extent the programmer must accept the user interface model supported by *lefty*. For example, at this time popup windows are supported; pulldown menu bars are not. On the other hand, though arbitrary C code can be compiled into EDGE or GraphEd, care would be needed to program these widgets in a way that is compatible with the base editor. For these reasons we feel *dotty* is a good alternative to using class libraries to create customized graph browsers.

Chapter 5. Conclusions

dotty is not only a good general-purpose graph editor, but a flexible tool for constructing graphical front ends for other tools. The programmability of *lefty* makes building front ends straightforward. *dot*'s good performance and high quality layouts result in fast response and pleasing pictures.

Appendix A. Running *dotty*

dotty can be started by issuing the command:

```
dotty [-V] [-lm sync|async] [-el 0|1] [file1 file2 ...]
```

If flag `-V` is specified, the program version is printed. If the option `-lm sync` is specified, then graph layouts will be computed online (*dotty* will block until a layout is finished). The option `-el 1` enables the printing of warning messages. If file names are specified on the command line, the graphs contained in these files will be displayed each in its own *dotty* window.

Appendix B. Program Listings

B.1 Finite Automaton Simulator

```
load ('dotty.lefty');
fa = [];
fa.normalcolor = 'black';
fa.highlightcolor = 'light_grey';
fa.currentcolor = 'tan';
fa.trackcur = 0;
fa.init = function () {
    dotty.init ();
    monitorfile = dotty.monitorfile;
};
fa.protogt = [
    'layoutmode' = 'sync';
    'actions' = copy (dotty.protogt.actions);
    # new actions are added later in the file
];
fa.protovt = [
    'name' = 'FA';
    'type' = 'normal';
    # other entries are added later in the file
];
fa.main = function () {
    local gnvt, gt;

    gnvt = dotty.createviewandgraph (null, 'file', fa.protogt, fa.protovt);
    gt = gnvt.gt;
    gt.loadgraph (gt, 'fa.dot', 'file', fa.protogt.graph, 1);
    fa.currstate = gt.graph.nodes[gt.graph.nodedict['start']];
    fa.loadtrans ('fa.trans');
    fa.setcurrcolor (gt, fa.currstate);
    fa.focuson (gt, fa.currstate);
};
fa.loadtrans = function (filename) {
    local fd, i;

    if (~((fd = openio ('file', filename, 'r')) >= 0)) {
        echo ('cannot open transition file: ', filename);
        return;
    }
    echo ('reading transition file');
    i = 0;
```

```
    while ((fa.trans[i] = readline (fd)))
        i = i + 1;
    closeio (fd);
    fa.trani = 0;
    fa.states[0] = fa.currstate;
    fa.currstate.count = 1;
};
fa.next = function (gt) {
    local label, eid, edge, tran;

    if (~(label = fa.trans[fa.trani]))
        return 0;
    for (eid in fa.currstate.edges) {
        edge = fa.currstate.edges[eid];
        if (edge.attr.label == label & edge.tail == fa.currstate) {
            tran = edge;
            break;
        }
    }
    if (tran) {
        fa.trani = fa.trani + 1;
        fa.setcolor (gt, fa.currstate);
        fa.currstate = (fa.states[fa.trani] = tran.head);
        if (fa.currstate.count)
            fa.currstate.count = fa.currstate.count + 1;
        else
            fa.currstate.count = 1;
        fa.setcurrcolor (gt, fa.currstate);
        if (fa.trackcur)
            fa.focuson (gt, fa.currstate);
    }
    return 1;
};
fa.prev = function (gt) {
    if (fa.trani == 0)
        return 0;
    remove (fa.trani, fa.states);
    fa.trani = fa.trani - 1;
    fa.currstate.count = fa.currstate.count - 1;
    fa.setcolor (gt, fa.currstate);
    fa.currstate = fa.states[fa.trani];
    fa.setcurrcolor (gt, fa.currstate);
    if (fa.trackcur)
        fa.focuson (gt, fa.currstate);
    return 1;
```

```
};
fa.setcolor = function (gt, node) {
    if (node.count) {
        node.attr.style = 'filled';
        node.attr.color = fa.highlightcolor;
    } else {
        gt.undrawnode (gt, gt.views, node);
        remove ('style', node.attr);
        node.attr.color = fa.normalcolor;
    }
    gt.unpacknodeattr (gt, node);
    gt.drawnode (gt, gt.views, node);
};
fa.setcurrcolor = function (gt, node) {
    node.attr.style = 'filled';
    node.attr.color = fa.currentcolor;
    gt.unpacknodeattr (gt, node);
    gt.drawnode (gt, gt.views, node);
};
fa.focuson = function (gt, node) {
    gt.setviewcenter (gt.views, node.pos);
};
fa.protogt.actions.general['play fwd'] = function (gt, vt, data) {
    while (fa.next (gt))
        ;
};
fa.protogt.actions.general['play bwd'] = function (gt, vt, data) {
    while (fa.prev (gt))
        ;
};
fa.protogt.actions.general['track node'] = function (gt, vt, data) {
    if (fa.trackcur)
        fa.trackcur = 0;
    else {
        fa.trackcur = 1;
        fa.focuson (gt, fa.currstate);
    }
};
fa.protovt.menus = [
    'general' = [
        0 = "undo";
        1 = "paste";
        2 = "do layout";
        3 = "cancel layout";
        4 = "redraw";
```

```
        5 = "new graph";
        6 = "load graph";
        7 = "reload graph";
        8 = "open view";
        9 = "copy view";
        10 = "clone view";
        11 = "birdseye view";
        12 = "close view";
        13 = "play fwd";
        14 = "play bwd";
        15 = "zoom in";
        16 = "zoom out";
        17 = "find node";
        18 = "track node";
        19 = "print graph";
        20 = "text view";
        21 = "quit";
    ];
    'node' = [
        0 = "cut";
        1 = "Cut";
        2 = "copy";
        3 = "Copy";
        4 = "group";
        5 = "Group";
        6 = "delete";
        7 = "Delete";
        8 = "remove";
        9 = "Remove";
        10 = "set attr";
        11 = "print attr";
    ];
    'edge' = [
        0 = "cut";
        1 = "Cut";
        2 = "copy";
        3 = "Copy";
        4 = "group";
        5 = "Group";
        6 = "delete";
        7 = "Delete";
        8 = "set attr";
        9 = "print attr";
    ];
];
```

```
fa.protovt.uifuncs.rightdown = dotty.protovt.normal.uifuncs.rightdown;
fa.protovt.uifuncs.keyup = dotty.protovt.normal.uifuncs.keyup;
fa.protovt.uifuncs.redraw = dotty.protovt.normal.uifuncs.redraw;
fa.protovt.uifuncs.closeview = dotty.protovt.normal.uifuncs.closeview;
fa.protovt.uifuncs.leftup = function (data) {
    local gt;

    gt = dotty.graphs[dotty.views[data.widget].gtid];
    if (fa.next (gt) == 0)
        echo ('at end of log');
};
fa.protovt.uifuncs.middleup = function (data) {
    local gt;

    gt = dotty.graphs[dotty.views[data.widget].gtid];
    if (fa.prev (gt) == 0)
        echo ('at start of log');
};
```

Bibliography

[GKNV93] E.R. Gansner, E. Koutsofios, S.C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE-TSE*, March 1993.

[Him] M. Himsolt. Graphed 3.0. available by anonymous ftp to forwiss.uni-passau.de (132.231.1.10) in /pub/local/graphed.

[KD91] Eleftherios Koutsofios and David Dobkin. Lefty: A two-view editor for technical pictures. In *Graphics Interface '91, Calgary, Alberta*, pages 68–76, 1991.

[NT87] Frances Newbery and Walter Tichy. Knowledge Based Editors for Directed Graphs. In H. Nichols and D. Simpson, editors, *1st European Software Engineering Conference*, pages 101–109. Springer Verlag, 1987.

[RDM+87] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Software—Practice and Experience*, 17(1):61–76, 1987.