GRAPHER

The *Grapher* library package contains a collection of functions and an interface for laying out, displaying, and editing graphs (i.e., networks of nodes and links). Graphs have node labels but not link labels. Links are drawn by default as straight lines without arrowheads, but you can control the appearance of individual links. Node labels can be single lines of text, bit maps of arbitrary size, or image objects. Facilities exist for having actions triggered by selecting nodes in graphs, and image objects containing graphs can be constructed, so users can include graphs in documents and other image structures.

A typical way to use *Grapher* is to implement a function that creates a partially specified graph that represents some user data (or control) structure. For instance, the *Browser* Lisp Library package uses graphs to represent function-calling structures (from Masterscope). Such a partially specified graph need have only the graph labels and the links specified. It is given to the LAYOUTGRAPH function along with some formatting information. LAYOUTGRAPH assigns a position to each node. There are formats for laying out trees, lattices, and cyclic graphs. LAYOUTGRAPH returns a GRAPH record, which is usually given to the function SHOWGRAPH. SHOWGRAPH displays a graph in a window. Displayed graphs are often used as menus: selecting a node with the left or middle button can cause user-provided functions to be called on that node. Displayed graphs can be edited using the right button. Nodes can be added, deleted, moved, enlarged, or shrunk. Links can be added or deleted.

DATA STRUCTURES

The GRAPH Record

A graph is represented by a GRAPH record, which has the following fields: GRAPHNODES, DIRECTEDFLG, SIDESFLG, GRAPH.MOVENODEFN, GRAPH.ADDNODEFN, GRAPH.DELETENODEFN, GRAPH.ADDLINKFN, GRAPH.DELETELINKFN, and GRAPH.FONTCHANGEFN.

GRAPHNODES is a list of graph nodes, which are described below.

DIRECTEDFLG and SIDESFLG are flags that control how links are drawn between the nodes. If DIRECTEDFLG is NIL, *Grapher* draws each link in such a way that it does not cross the node labels of the nodes it runs between. Often, this leaves some ambiguity, which is settled by SIDESFLG. If

SIDESFLG is NIL, *Grapher* prefers to draw links that go between the top and bottom edges of nodes. If SIDESFLG is non-NIL, *Grapher* prefers to draw links between the sides of the nodes.

If DIRECTEDFLG is non-NIL, the edges are fixed (e.g., always to the left edge of the To node). This can cause links to cross the labels of the nodes they run between. In this case, if SIDESFLG is NIL, the From end of the link is attached to the bottom edge of the From node; the To end of the link is attached to the top edge of the To node. If DIRECTEDFLG is non-NIL and SIDESFLG is non-NIL, the From end of the link is attached to the right edge of the From node; the To end of the link is attached to the left edge of the To node.

The remaining fields give you hooks into the graph editor, which is described below.

The GRAPHNODE Record

The GRAPHNODE record has the following fields of interest:

NODELABEL is the thing that gets displayed as the node. If this is a bit map, BITBLT is used; if it is an image object, its *ImageBoxFn* and *DisplayFn* are used. Anything else is printed with PRIN3.

NODEID is a unique identifier. NODEIDs are used in the link fields instead of pointers to the nodes themselves, so that circular Lisp structures can be avoided. NODEIDs are often used as pointers to the structure represented by the graph.

TONODES is a list of NODEIDs: a link runs from this node to each node in TONODES. Entries in this field can be used to specify properties of the lines drawn between nodes, as described below.

FROMNODES is a list of NODEIDs: a link runs to this node from each node in FROMNODES.

NODEPOSITION is the location of the center of the node (a POSITION).

NODEFONT specifies the font in which this node's label is displayed. It can be any font specification acceptable to FONTCREATE, including a FONTDESCRIPTOR. NODEFONT is changed by the graph edit operations Larger and Smaller. When this happens, the font family may be changed as well as the size. Default is the value of DEFAULT.GRAPH.NODEFONT (initially NIL).

NODEBORDER specifies the shade and width of the border around a node via the following values:

NIL,0	No border (equals border of width zero)
T	Black border, one pixel wide, as before

1,2,3. . . Black border of the given width

-1,-2. . . White border of the given width

(*W S*) Where *W* is an integer and *S* is a texture or a shade; yields a border *W* pixels wide filled with the given shade *S*

Default is the value of DEFAULT.GRAPH.NODEBORDER (initially NIL).

NODELABELSHADE contains the background shade of the node. If this field is non-NIL, then when a node is displayed, the label area for the node is first painted as specified by this field, then the label is printed in INVERT mode. This does not apply to labels that are bit maps or image objects. The legal values for the field are: NIL (same as WHITESHADE), T (same as BLACKSHADE), a texture, or a bit map. Default is the value of DEFAULT.GRAPH.NODELABELSHADE (initially NIL).

NODEWIDTH and NODEHEIGHT are normally set by *Grapher* to be the width and height of the node's NODELABEL. However, if you provide integers for these fields, their values are used instead. This feature can be used to give a node a larger-than-normal "margin" around its label.

Entries in the TONODES field can be used to specify the appearance of the lines drawn between nodes. If an item in the TONODES of a node N1 is not a NODEID but rather a list of the form:

(*Link% Parameters ToNodeID . ParamList*)

then *ParamList* is interpreted as a property list specifying properties of the link drawn from N1 to *ToNodeID*.

Properties of *ParamList* currently noticed are LINEWIDTH, DASHING, COLOR, and DRAWLINKFN. The first three are passed directly to DRAWLINE. For example, if the TONODES for A is:

((*Link% Parameters* B LINEWIDTH 4 DASHING (3 3))

(*Link% Parameters* C DASHING (5 1) COLOR 12))

then two dashed lines will emanate from A, with the one to B having width 4 and dashing (3 3), and the one to C having the default width 1, dashing (5 1), and color (if implemented) 12.

If the property DRAWLINKFN is on the list, then its value must be a function to be called instead of DRAWLINE. It is passed all the arguments of DRAWLINE plus the *ParamList* as a last argument.

For convenience, the variable LINKPARAMS is set to the (constant) value *Link% Parameters*. When DISPLAYGRAPH scales the graph to the units of a particular output stream, the properties whose names are found on the SPECVAR *ScalableLinkParameters* are also scaled.

Graph nodes can be created by the CREATE operator from the record package or with the following function:

(NODECREATE *ID Label Position ToNodeIDs FromNodeIDs Font Border LabelShade*) [Function]

This function returns a GRAPHNODE with NODEID= *ID*, NODELABEL=*Label*, NODEPOSITION= *Position*, NODEFONT=*Font*, NODEBORDER=*Border*, and NODELABELSHADE= *LabelShade*.

THE LAYOUT FUNCTIONS

(LAYOUTGRAPH *NodeLst RootIDs Format Font MotherD PersonalD FamilyD*) [Function]

LAYOUTGRAPH "lays out" a partially specified graph by assigning positions to its graph nodes. It returns a GRAPH record suitable for displaying with SHOWGRAPH. The input *NodeLst* is a list of GRAPHNODES that partially specifies a graph: only their NODEID, NODELABEL, and TONODES fields need to be filled in. Optional fields are: NODEFONT, NODEWIDTH, and NODEHEIGHT. These optional fields are filled in appropriately if they are NIL. All other fields are ignored and/or overwritten. *RootIDs* is a list of the node identifiers of the nodes that become the roots. The other arguments are optional and control the format of the layout.

Format controls the geometry of the layout. It is an unordered list of atoms. There are four basic formats.

COMPACT is the default format. The graph is laid out as a forest (a set of trees) in such a way that the minimal amount of screen space is used.

In the FAST format, the graph is laid out as a forest, sacrificing screen space for speed.

In the LATTICE format, the graph is laid out as an acyclic directed graph (a lattice).

In the REVERSE/DAUGHTERS format, the graph is laid out with the To nodes in reverse order.

These basic formats come in two flavors:

HORIZONTAL [the default] has roots at left and links run left-to-right.

VERTICAL has roots at the top and links run top-to-bottom.

The directions can be reversed by including the atom REVERSE in *Format*. Thus, for example, *Format*=(LATTICE HORIZONTAL REVERSE) lays out horizontal lattices that have the roots on the right, with the links running right-to-left. *Format*=(VERTICAL REVERSE) lays out vertical trees that have the roots at the bottom, with links running bottom-to-top. *Format*=NIL lays out horizontal trees that have the roots on the left.

LAYOUTGRAPH creates "virtual" graph nodes to avoid drawing a tangle of messy lines in cases where the graph is not a forest or a lattice to begin with. It modifies the nodes of NODELST, which may involve changing some of the TONODES fields to point to new nodes. The modified NODELST is set into the GRAPHNODES field of a newly created GRAPH record, which is returned as the value of LAYOUTGRAPH. The creation of virtual nodes depends on whether LATTICE is a member of FORMAT. The forest (i.e., non-LATTICE) case will be described first.

Nodes are laid out by traversing the forest top-down, depth-first. Whenever an already laid-out node is found, instead of drawing a link to it that might cut across arbitrary parts of the graph, LAYOUTGRAPH creates a copy of the node (same NODELABEL, different NODEID, no TONODES), lays it down, and "marks" both it and the original node (by setting their NODEBORDER fields and NODELABELSHADE fields). Hence, a marked node occurs at least twice in the forest. The default is to leave the shade alone and set the border to one, but the appearance of marked nodes can be controlled by adding (MARK . . . .) to the FORMAT argument. The tail of (MARK . . . .) is a property list. If it is NIL, marking is suppressed altogether. If it contains BORDER or LABELSHADE properties, those values are used in the corresponding fields of marked nodes. *Format* adds a few twists to this basic marking strategy. It can include one or both of these atoms:

COPIES/ONLY Only the new "virtual" nodes are marked. The original is left unmarked.

NOT/LEAVES Marking is suppressed when the node has no daughters.

For example, *Format* = (COPIES/ONLY NOT/LEAVES) marks nodes that are copies of nodes that have daughters (i.e., if you see a mark, the node has daughters that aren't drawn). *Format* = (NOT/LEAVES) marks both copies and originals, but only when they have no daughters. *Format* = NIL marks originals and copies regardless of progeny.

If *Format* includes LATTICE, then a node that is the daughter of more than one node is not marked. Instead, links from all its parents are drawn to it. No attempt is made to avoid drawing lines through nodes or to minimize line crossings. However, in HORIZONTAL format, nodes are positioned so that "From" is always left of "To." Similar conventions hold for the other formats. In VERTICAL

format, for instance, the TONODES of a node are positioned beneath it, and the FROMNODES are positioned above it. Cyclic graphs cannot be drawn using this convention, of course (how can a node be left of itself?). When LAYOUTGRAPH detects a node that points to itself, directly or indirectly, it creates a "virtual" node, as described above, and marks both the original and the copy. If *Format* includes COPIES/ONLY, then only the newly created node is marked.

This ends the discussion of the *Format* argument to LAYOUTGRAPH. The other arguments are not so complicated.

Font is a font specification for use as the default NODEFONT.

The remaining three arguments control the distances between nodes. NILs cause "pretty" defaults based on the size of *Font*. *PersonalD* controls the minimum distance between any two nodes. *MotherD* is the minimum distance between a mother and her daughters. *FamilyD* controls the minimum distance between nodes from different nuclear families. The closest two sister nodes can be is *PersonalD*. The closest that two nodes that are not sisters can be is *PersonalD+FamilyD* .

LAYOUTGRAPH reads but does not change the fields NODEBORDER and NODELABELSHADE of the nodes given it (except for the marked nodes, of course). Thus, if you plan to install black borders around the nodes after the nodes have been laid out (e.g., by RESET/NODE/BORDER, described below), it is a good idea to give LAYOUTGRAPH nodes that have white borders. This causes the nodes to be laid out far enough apart that when you blacken the borders later, the labels of adjacent nodes are not overwritten.

THE DISPLAY FUNCTIONS

(SHOWGRAPH *Graph W LeftButtonFn MiddleButtonFn TopJustifyFlg AllowedITFlg CopyButtonEventFn*) [Function]

SHOWGRAPH displays the nodes in the *Graph*. If *W* is a window, the graph is displayed in it. If the graph is larger than the window, the window is made a scrolling window. If *W* is NIL, the graph is displayed in a window large enough to hold it. If *W* is a string, the graph is displayed in a window large enough to hold it, and the window uses the string for the window title. SHOWGRAPH caches some information in the GRAPHNODE fields to make scrolling faster. The graph is stored on the GRAPH property of the window. SHOWGRAPH returns the window.

If either *LeftButtonFn* or *MiddleButtonFn* is non-NIL, the window is given a *ButtonEventFn* that, in effect, turns the graph into a menu. Whenever the left or middle button is depressed and the cursor is over a node, that node is displayed inverted, indicating that it is selected. Letting up on the

button calls either the *LeftButtonFn* or the *MiddleButtonFn* with two arguments: the selected *GraphNode* (this is NIL if the cursor was not over a node when the button was released) and the window (from the window, the functions can access the graph via WINDOWPROP).

The graph's initial position is determined by *TopJustifyFlg*. If T, the graph's top edge is positioned at the top edge of the window; if NIL, the graph's bottom edge is positioned at the bottom edge of the window.

THE EDITING FUNCTIONS

If *AllowEditFlg* is non-NIL, the right button can be used to edit the graph. (The normal window commands can be accessed by right-buttoning in the border or title regions.) Right-buttoning while the control shift is down allows positioning of nodes by tracking the cursor. Right-buttoning with the control shift up brings up a menu of edit operations. The edit operations allow moving, adding, and deleting of nodes and links. Adding a node prompts for a NODELABEL, creates a new node with that label, adds it to the graph, and allows you to position it. Deleting a node removes it (using DREMOVE) from GRAPH after deleting all of the links to and from it. When the STOP menu command is selected, the graph window is closed.

CopyButtonEventFn is a function to be run when you copy-select from the *Grapher* window. If this is not specified, the default simply COPYINSERTs a *Grapher* image object.

Certain fields of the GRAPH record are functions that get called by the graph editor whenever an action is performed on an element in the displayed graph. They allow the graph to serve as a simple edit interface to the structure being graphed. Below are the fields of GRAPH and the arguments that the corresponding function calls. In all cases *Graph* is the graph being displayed, and *Window* is the window in which it is displayed.

(GRAPH.MOVENODEFN *Node NewPos Graph Window*) [Function]

is called after you have stopped moving a node interactively (i.e., is not called as the node is being moved).

(GRAPH.ADDNODEFN *Graph Window)* [Function]

is called when you select "Add a node" and returns a node or NIL if no new node is to be added. A node-moving operation is called on the new node after it is created to determine its position.

(GRAPH.DELETENODEFN *Node Graph Window*) [Function]

is called when a node is deleted. Before this function is called, all of the links to or from the node are deleted.

(GRAPH.ADDLINKFN *From To Graph Window*) [Function]

is called when a link is added.

(GRAPH.DELETELINKFN *From To Graph Window*) [Function]

is called when a link is deleted, which can be either directly or from deleting a node.

(GRAPH.FONTCHANGEFN *How Node Graph Window*) [Function]

is called when you ask for the label on a node to be made larger or smaller. How is one of LARGER or SMALLER.[1]

(DISPLAYGRAPH *Graph Stream Clip/Reg Trans*) [Function]

puts the specified graph on *Stream* (which can be any image stream) with coordinates translated to *Trans*. Some streams might also implement *Clip/Reg* as a clipping region. This is primarily to improve efficiency for the display.

(HARDCOPYGRAPH *Graph/Window File ImageType Trans*)

produces a file containing an image of *Graph* (e.g., like SHOWGRAPH, only for files). If *Graph/Window* is a window, HARDCOPYGRAPH operates on its GRAPH window property. The *File* and *ImageType* arguments are given to OPENIMAGESTREAM to obtain a stream that the graph will be displayed on. *Trans* is a position in screen points (i.e., it is scaled by the image stream's DSPSCALE) of the lower-left corner of the graph from the lower-left corner of the piece of paper.

GRAPHER IMAGE OBJECTS

A graph data structure can be encapsulated in a *Grapher* image object, so that it can be inserted in a *TEdit* document or other image structure. *Grapher* image objects are constructed by the following function.

(GRAPHEROBJ *Graph Halign Valign*) [Function]

returns a *Grapher*-type image object that displays *Graph*. *Halign* and *Valign* specify how the graph is to be aligned with respect to the reference point in its host (*TEdit* file, image object window, etc.). They can be numbers between zero and one, specifying as a proportion of the width/height of the

graph the point in the graph that overlays the reference point (zero means that the graph will sit completely above and to the left of the reference point; one means it will sit completely below and to the right). They can also be pairs of the form (*NodeSpec Pos*), where *NodeSpec* specifies a node that the graph is to be aligned by, and *Pos* specifies where in the node the alignment point is. The *NodeSpec* can be either a NODEID or one of the atoms *TOP*, *BOTTOM*, *LEFT*, or *RIGHT*, indicating the topmost, bottommost, etc., node of the graph. The *Pos* can be a number specifying proportional distances from the lower-left corner of the node, or the atom BASELINE, indicating the character baseline (for *Valign*, or simply zero for *Halign*). For example, to align a linguistic tree so that the baseline of the root node is at the reference point, *Valign* would be (*TOP* BASELINE). The *ButtonEventInFn* of the image object pops up a single-item menu, which if selected causes the graph editor to be run.

MISCELLANEOUS FUNCTIONS

(GRAPHREGION *Graph*) [Function]

returns the smallest region containing all of the nodes in GRAPH.

(FLIPNODE *Node DS*) [Function]

inverts a region in the stream DS that is one pixel bigger all around than NODE's region. This makes it possible to see black borders after the node has been flipped.

(RESET/NODE/BORDER *Node Border Stream Graph*) [Function]

and

(*RESET/NODE/LABELSHADE Node Shade Stream*) [Function]

reset the appropriate fields in the node. If *Stream* is a display stream or a window, the old node is erased and the new node is displayed. Changing the border may change the size of the node, in which case the lines to and from the node are redrawn. The entire graph must be available to RESET/NODE/BORDER for this purpose, either supplied as the *Graph* argument or obtained from the GRAPH property of *Stream,* if it is a window. Both functions take the atom INVERT as a special value for *Border* and *Shade*. They read the node's current border or shade, calculate what would be needed to invert it, and do so.

(LAYOUTSEXPR *Sexpr Format Font MotherD PersonalD FamilyD*) [Function]

is just like LAYOUTGRAPH, except it gets its graph as an s-expression rather than a list of GRAPHNODEs. A formal recursive definition of its first argument is: If the s-expression is an atom, its NODELABEL is itself and it has no TONODEs; else it is a list and its CAR is taken as its NODELABEL and its CDR, which must be a list of s-expressions, is taken as its TONODES. Note that circular s-expressions are allowed. For example, the following displays a parse tree for the sentence "The program displays a tree."

```
[SHOWGRAPH (LAYOUTSEXPR '(S (NP (DET the)(NOUN program))
                            (VP (VERB displays)
                                (NP (DET a)(NOUN tree))))
                           '(VERTICAL) NIL '(HELVETICA 12]
```

(DUMPGRAPH *Graph Stream*) [Function]

prints the GRAPH out on STREAM in a special, relatively compact encoding that can be interpreted by the function READGRAPH (see below). Graphs cannot be saved on files simply by ordinary print functions such as PRIN2. This is because the *Grapher* functions use FASSOC (i.e., EQ not EQUAL) to fetch a graph node given its ID, so reading it back in gives the right result only if the IDs are atomic. HPRINT resolves this problem, but it tends to dump too much information: it dumps a complete description of the node font, for example, including the character bit maps. DUMPGRAPH and READGRAPH are used in the implementation of *Grapher* image objects.

(READGRAPH *Stream*) [Function]

reads information from *Stream* starting at the current file pointer and returns a graph structure equivalent to the one that was given to DUMPGRAPH.

End Notes

1. The node labels are reprinted whenever the graph is redisplayed. If this makes scrolling of a large graph unacceptably slow, some speedup may be achieved by instructing *Grapher* to cache bit maps of the labels with the nodes so they can be rapidly BITBLTed to the screen (set the variable CACHE/NODE/LABEL/BITMAPS/FLG to T). The possible gain in time, however, may be offset by the increased storage required for the cacheD bit maps.