A SIMPLE GRAPH BROWSER

by

John Alan McDonald

TECHNICAL REPORT No. 219
October 1991

Department of Statistics, GN-22
University of Washington
Seattle, Washington 98195 USA
A simple graph browser

JOHN ALAN MCDONALD *
Dept. of Statistics, University of Washington

October 1991

Abstract

The purpose of this document is to describe a minimal interface to a graph browser that is part of the Clay user interface toolkit, a component of Arizona Clay is an object of current research and development; the idea here is to provide a useful interface to its graph browser which places as few constraints as possible on future changes.

* This work was supported in part by the Dept. of Energy under contract FG0685-ER25006 and by NIH grant LM04174 from the National Library of Medicine.
Contents

1 Introduction
   1.1 Declarative graph layout using nonlinear programming .................. 3

2 A protocol for Graph data structures ....................................... 4
   2.1 Graph protocol ................................................................... 4
   2.2 Edge protocol .................................................................. 5
   2.3 A sample implementation ................................................... 5

3 Clay ...................................................................................... 5

4 Customizing behavior ............................................................. 5
   4.1 Updating the display ......................................................... 5
   4.2 Menus ............................................................................. 5
   4.3 Layout ............................................................................. 6

5 Examples .................................................................................. 7

6 Reference Manual ...................................................................... 10

List of Figures

1 A class inheritance graph for some classes related to graph browsers. ........ 7
2 A examiner graph for some objects used to make a class browser. ............ 8
3 Horse pedigree layout, treating the pedigree as an undirected graph. ........ 8
4 Horse pedigree layout using inheritance. .............................................. 9
5 Horse pedigree layout using kinship and inheritance. ............................ 9
1 Introduction

The purpose of this document is to describe a minimal interface to a graph browser that is part of the Clay user interface toolkit, a component of the Arizona system [14, 17, 16, 19, 20, 18, 15]. Clay is an object of current research and development; the idea here is to provide a useful interface to its graph browser which places as few constraints as possible on future changes.

I've separated :Clay (the user interface package) into 3 packages (and 3 modules or systems), to better expose the interface. The 3 packages are :Graph, :Clay, and :Browser. The corresponding systems are defined in az/az.system (along with a number of other systems not discussed here). The latest version of the three systems is on both belgica.stat.washington.edu and eowyn.rad.washington.edu, under my home directory. All three systems are compiled by loading az/browser/make.lisp and loaded by loading az/browser/load.lisp.

1.1 Declarative graph layout using nonlinear programming

Automatic layout of graphs is a problem that has a sizable literature; for recent surveys see [26]. Roughly speaking, methods for layout can be divided into procedural and declarative (also known as constraint-based [9]).

By procedural layout, we mean simply that there is a procedure, which, given a graph, computes its layout. A typical layout procedure (such as that given by Rowe et al. [24]) traverses the graph, assigning each node to an unoccupied site in a regular grid of allowed positions. The order of traversal and the method of site assignment are chosen in the hope of producing a "readable" layout: a minimal number of crossed edges, short (but not too short) edges, almost straight paths between important pairs of non-adjacent nodes, node positions whose y (or x) coordinates are consistent with the partial ordering of a directed graph, and so on. There are often additional passes over the graph during which the nodes are re-arranged in an attempt to improve the layout.

Declarative layout could be thought of as a special case of procedural layout, where the procedure definition is based on three basic abstractions:

- a domain of possible layouts (for a given graph),
- a specification "language" that allows the user to express essential and desirable features of the layout, and
- a generic solver, that takes a graph and a specification, and returns an element of the domain that meets or well approximates the specification.

For example, the domain might be all possible assignments of nodes to positions in a discrete lattice in the plane. The specification language might allow the user to ask to minimize some loss function (eg. the number of edge crossings) subject to constraints on the node positions (eg. no two nodes can occupy the same position in the lattice).

The distinction between procedural and declarative layout is, of course, very similar to the distinction between procedural and declarative programming languages. In particular, our notion of declarative layout is intentionally close to constraint programming languages [2, 27, 10, 11].

The advantage of procedural layout is speed. A procedural method makes one or a few passes over the graph; the declarative example given above is an expensive integer programming problem, whose solution would require the equivalent of many iterations over the graph.

The advantage of the declarative approach is flexibility. In general, the only way for a user to modify the result of a procedural layout is to reprogram the procedure; because of the ad hoc nature of most layout procedures, it is difficult to know how to change the procedure to get the desired
effect. The three abstractions of the declarative approach — the domain, the specification, and the solver — gives a user immediate access to a whole class of layouts. The key research issue is to design abstractions to provide a large and useful class of layouts, without giving up too much in performance compared ad hoc procedural methods.

Our proposal for declarative layout is an extension of the spring model [3, 9]. The spring model is based on a physical analogy: Imagine a spring placed between each pair of nodes in the graph. Let the rest length of the spring be the desired distance between the nodes, e.g., proportional to the length of the shortest path in the graph connecting the two nodes. Then simply minimize the energy. (This is a minor variation on multidimensional scaling [1, 28].)

The domain of the layout is the space of node positions, $\mathbb{R}^2$, that is, the n-fold direct sum of the plane. The specification language is the set of rest lengths (and spring constants). The solver is whatever minimization routine is used.

Our extension to the spring model is to minimize the energy subject to affine inequality (and equality) constraints on the node positions, in other words, nonlinear programming [4, 8, 12, 13]. For our solver, we are using standard packages for constrained minimization, in particular MINOS and NPSOL [6, 7, 22, 21, 23]. The possibility of affine constraints gives us a much richer specification language, making it easy, for example, to preserve the directedness of a graph in the vertical or horizontal ordering of the nodes, or to adjust a layout for the size and shape of the window in which the graph is being displayed.

The use of modern optimization methods, as represented by MINOS and NPSOL, gives us still greater flexibility. We can experiment with other energy functions than the spring model and eventually provide users with more general ways to specify the energy than the rest lengths and spring constants. For example, we are considering energy functions that depend on the relationships of pairs of edges as well on pairs of nodes — with the goal of minimizing or at least controlling edge crossings.

2 A protocol for Graph data structures

The Graph package is in directory az/graph. It defines a simple functional protocol for data structures that are used to represent graphs and that are to be displayed by Clay's graph browser.

2.1 Graph protocol

A graph object must have an immutable identity, for example, you cannot represent a graph by a list and cons new nodes or edges on the front.

A graph object must provide methods for two generic functions

- `gr:nodes` returns a list of the nodes in the graph. Nodes may be anything. Nodes are considered identical if they are eql.
- `gr:edges` returns a list of the edges in the graph. Edges must obey the edge protocol defined below. Edges are considered identical if they are eql.

If an object has methods for `gr:nodes` and `gr:edges`, then it is considered an instance of the abstract `gr:Generic-Graph` type and satisfies `(typep object 'gr:Generic-Graph).

Graphs that want to be considered directed and/or acyclic should define appropriate methods for the following two generic functions: `gr:directed-graph`, which returns `T` or `Nil` depending on whether the graph should be considered directed, and `gr:acyclic-graph?`, which returns `T` or `Nil` depending
on whether the graph can be assumed to be acyclic. The default method for both generic functions returns nil.

2.2 Edge protocol

An object representing a graph edge must have methods for `gr:edge-node0` and `gr:edge-node1`, which return the nodes the edge connects. If the edge is considered directed, then it goes from `gr:edge-node0` to `gr:edge-node1`.

If an object has methods for `gr:edge-node0` and `gr:edge-node1`, then it is considered an instance of the abstract `gr:Generic-Edge` type and satisfies `typetype object `gr:Generic-Edge`).

2.3 A sample implementation

The file implementation.lisp provides sample implementation classes `gr:Graph`, `gr:Digraph`, and `gr:Dag` that obey the graph protocol and may be useful base classes for more specialized graphs.

3 Clay

The Clay package, in az/clay and az/layout, provides a generic graph browser that can be used with any object that obeys the graph protocol.

A graph browser is created by a call to `clay:make-graph-diagram`, which has one required argument, the graph to be presented, and keyword arguments: `:diagram-name`, which is used for the label of the window in which the graph is displayed, and `:x-order-f`, `:y-order-f`, `:node-pair-distance-f`, and `:rest-length`, which are used to customize the layout and are discussed in section 4.3.

4 Customizing behavior

4.1 Updating the display

Currently, consistency between graphs and their presentations is maintained using my implementation of Announcements [15] (in az/tools/announcements.lisp), based on Mark Niehaus's implementation in Prism of Sullivan-Notkin ideas about events/mediators [25]. A full discussion of events, mediators, announcements, etc., is beyond the scope of this document.

All a client of the graph browser needs to know is to `(az:announce graph :graph-changed)` when it wants displays of that graph to update in response to addition or deletion of nodes or edges to or from graph.

4.2 Menus

The behavior of the graph browser can be specialized by adding items to the menus that are produced when the right (righthand) button is pressed. If the mouse is within a node, then the node menu appears. If the mouse button is not within a node, and is within the rectangle enclosing an edge, then the edge menu appears. Otherwise the whole graph menu appears.

If your graph, nodes, and edges are instances of classes or structures that you have defined, then you can add menu items by defining new methods for the three generic functions: `clay:graph-node-menu`, `clay:graph-edge-menu`, and `clay:graph-menu`. Your methods must return a (newly constructed) list of menu items. A menu item is a list whose first item is the string that is printed in the menu,
and whose second item is a function that is applied to the remaining items. Note that methods for these generic function must be defined with append methods combination. For example, here is method for clay:graph-node-menu defined in az/browser/class.lisp:

```lisp
(defmethod clay:graph-node-menu append ((graph Class-Graph) (node Class))
  (list (list (format nil "Add direct subclasses of " node) #'add-subclass-nodes node graph)
        (list (format nil "Add direct superclasses of " node) #'add-superclass-nodes node graph)))
```

4.3 Layout

The function make-graph-presentation takes several keyword arguments than can be used to customize the layout of the graph.

- :x-order-f, :y-order-f
  The graph layout is computed subject to the constraint that all nodes remain in the window and optionally subject to constraints that require node A to be above node B or node A to be to the left of node B.
  If supplied, :x-order-f must be a function of two arguments that can be applied to any pair of nodes in the graph. It should return T if its first argument should be constrained to be to the left of its second argument and Nil otherwise.
  If supplied, :y-order-f must be a function of two arguments that can be applied to any pair of nodes in the graph. It should return T if its first argument should be constrained to be above its second argument and Nil otherwise.
  It is better for performance reasons to use a minimal, non-transitive ordering function. For example, if A is above B and B is above C, it is not necessary to also constrain A to be above C.
  If neither :x-order-f or :y-order-f are supplied then default constraints will be imposed. For graph objects that return T for both gr:directed-graph? and gr:acyclic-graph?, the default constraints are that parent nodes must be left of their children. Otherwise, there are no default constraints (other than the bounds constraints that all nodes must remain in the window).

- :node-pair-distance-f
  If supplied, :node-pair-distance-f must be a function of two arguments that can be applied to any pair of nodes in the graph. It must return a positive Double-Float number.
  The layout algorithm tries to place nodes so that the distance between them on the screen is proportional to the value returned by :node-pair-distance-f.
  If :node-pair-distance-f is not supplied, then it defaults to the graph distance, that is, the number of edges between the two nodes in the shortest path in the graph, ignoring any directedness of the edges.

- :rest-length
:Rest-length determines the average length of the displayed edges in the graph, in a somewhat complicated fashion. The distance between the centers of two nodes will be roughly the rest-length times the geometric average of the widths or heights of the nodes times a given desired distance (see below). The overall scale of the layout will change linearly with :rest-length, as long as all nodes remain in the interior of the window. The exact result of a given :rest-length with a particular graph is hard to predict ahead of time. 10 seems to be a good value for many graphs.

5 Examples

The Browser module, in az/browser/, provides three examples of how to use the graph browser. Script.lisp has examples of how to create all three kinds of browsers.

The first two examples show how to use the graph browser with minimal specialization — essentially just adding menu items. The first example is a CLOS class hierarchy browser, in class.lisp (see figure 1 for an example). The second example is a Lisp object browser, in examiner.lisp (see figure 2 for an example).

The third example is demonstrates alternate layouts, using custom order and distance functions. It is contained in two files, pedigree.lisp and horses.lisp. These files create a small database for a small number of Przewalski's horses [5]. The following figures show 3 alternative layouts: (figure 3) treats the pedigree as an undirected graph, (figure 4) uses inheritance to place parents above children, and (figure 5) uses distance function based on kinship, the probable genetic similarity between two individuals.
Figure 2: A examiner graph for some objects used to make a class browser.

Figure 3: Horse pedigree layout, treating the pedigree as an undirected graph.
Figure 4: Horse pedigree layout using inheritance.

Figure 5: Horse pedigree layout using kinship and inheritance.
6 Reference Manual

These reference manual entries were produced automatically, using the Definitions module described in [16].

gr:acyclic-graph?  Generic Function

Documentation:
   Is this graph guaranteed to have no cycles (in its directed edges)?

Usage:
   (gr:acyclic-graph? graph)

Arguments:
   graph — T

Returns:
   (Member T Nil)

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

gr:acyclic-graph? T  Primary Method

Documentation:
   The default is that a graph is not guaranteed to be acyclic; note that this does not mean that
   the graph is guaranteed to have cycles.

Usage:
   (gr:acyclic-graph? graph)

Arguments:
   graph — T

Returns:
   nil

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

gr:acyclic-graph? Gr:Dag  Primary Method

Documentation:
   Dags are assumed to have no cycles.

Usage:
gr:acyclic-graph

(gr:acyclic-graph? g)

Arguments:
  g — Gr:Dag

Returns:
  t

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

g:acyclic-graph

(gr:add-nodes-and-edges nodes edges graph)

Arguments:
  nodes — List
  edges — List
  graph — T

Returns:
  graph

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

gr:add-nodes-and-edges  T  T  Gr:Graph

Documentation:
  The after method announces that the graph has changed.

Usage:
  (gr:add-nodes-and-edges nodes edges graph)

Arguments:
nodes — T
dges — T
graph — Gr:Graph

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

| gr:add-nodes-and-edges T T Gr:Graph | Primary Method |

Documentation:
This is not as trivial as you might think, because we need to maintain consistency between the graph's nodes and edges.

Usage:
(gr:add-nodes-and-edges nodes edges graph)

Arguments:

| nodes — List |
| edges — List |
| graph — Gr:Graph |

Returns:

| graph |

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

| :Browser | Package |

Usage:
(in-package :Browser)

Source: #p"/belgica-2g/jam/az/browser/package.lisp"

| Br:Class-Graph | Class |

Documentation:
The purpose of having Class-Graph objects is to enable us to focus our attention on a subset of the inheritance relations in a set of classes. The nodes in a Class-Graph are the classes themselves. The edges represent direct-subclass relations.

Usage:
(typep x 'Br:Class-Graph)

Parents:
Gr:Dag

Source: #p"/belgica-2g/jam/az/browser/class.lisp"
Using a Dag implies there are no cycles, but doesn’t enforce it.

Usage:
```
(typep x 'Gr:Dag)
```

Parents:
- Gr:Digraph

Children:
- Br:Class-Graph

Source: `#p"/belgica-2g/jam/az/graph/implementation.lisp"`

### `gr:delete-nodes-and-edges`

#### Documentation:
Destructively modify `<graph>` by deleting `<nodes>` and `<edges>`. Note that nodes are the more primitive objects. Adding a node to a graph does not imply adding any edges, even though some node objects may contain references to edges. Adding an edge, however, does imply ensuring that its node are contained in the graph. Also, deleting a node implies deleting the corresponding edges, but deleting an edge has no effect on the nodes.

Methods are responsible for maintaining consistency in the graph’s nodes and edges. After the operation is complete, any node referred to by an edge in `(edges graph)` must be in `(nodes graph)`.

In addition to deleting the edges in the argument `<edges>`, a method must delete all edges corresponding to deleted `<nodes>`.

#### Usage:
```
(gr:delete-nodes-and-edges nodes edges graph)
```

#### Arguments:
- `nodes` — List
- `edges` — List
- `graph` — T

#### Returns:
- `graph`

Source: `#p"/belgica-2g/jam/az/graph/protocol.lisp"

### `gr:delete-nodes-and-edges T T Gr:Graph`

#### Documentation:
The after method announces that the graph has changed.
Usage:
(gr:delete-nodes-and-edges nodes edges graph)

Arguments:

- nodes — T
- edges — T
- graph — Gr:Graph

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

**Gr:Delete-nodes-and-edges**

<table>
<thead>
<tr>
<th>Primary Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>T T Gr:Graph</td>
</tr>
</tbody>
</table>

Documentation:
This is not as trivial as you might think, because we need to maintain consistency between the graph's nodes and edges.

Usage:
(gr:delete-nodes-and-edges nodes edges graph)

Arguments:

- nodes — List
- edges — List
- graph — Gr:Graph

Returns:

graph

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

**Gr:Digraph**

<table>
<thead>
<tr>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Documentation:
Using a Digraph implies that the direction of the edges is meaningful.

Usage:
(typep x 'Gr:Digraph)

Parents:
Gr:Graph

Children:
Br:Examiner-Graph Gr:Dag

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"
Does this graph have directed edges?

Usage:
(gr:directed-graph? graph)

Arguments:
graph — T

Returns:
(Member T nil)

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

The default is that a graph’s edges are not considered directed.

Usage:
(gr:directed-graph? graph)

Arguments:
graph — T

Returns:
nil

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

Usage:
(gr:directed-graph? g)

Arguments:
g — Gr:Digraph

Returns:
t
Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

```
(gr:directed-graph? Br:Examiner-Graph)  Primary Method
```

Documentation:
Examiner-Graphs are directed.

Usage:
```lisp
(gr:directed-graph? g)
```

Arguments:
- `g` — Br:Examiner-Graph

Returns:
- `t`

Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"

```
gr:edge-node0
```

Documentation:
If the edge is directed, then it goes from `edge-node0` to `edge-node1`.

Usage:
```lisp
(gr:edge-node0 edge)
```

Arguments:
- `edge` — T

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

```
gr:edge-node0 Cons
```

Documentation:
A simple edge representation is a list of length 2. The methods specialize for Cons, rather than List, so that they don't match treat () as equivalent to (nil nil).

Usage:
```lisp
(gr:edge-node0 edge)
```

Arguments:
- `edge` — Cons

Returns:
node0

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

| gr:edge-node1 | Generic Function |

Documentation:
If the edge is directed, then it goes from edge-node0 to edge-node1.

Usage:
(gr:edge-node1 edge)

Arguments:
edge — T

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

| gr:edge-node1 Cons | Primary Method |

Documentation:
A simple edge representation is a list of length 2. The methods specialize for Cons, rather than List, so that they don't match treat () as equivalent to (nil nil).

Usage:
(gr:edge-node1 edge)

Arguments:
edge — Cons

Returns:
node1

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

| gr:edges | Generic Function |

Documentation:
Returns a list of the edges in the graph.
For performance reasons, this may be the actual list of edges used by the graph. If this list is modified, the results are unpredictable. It should be copied before any changes are made. To change the graph itself, use one of the editing operations, eg. <add-nodes-and-edges>.

Usage:
(gr:edges graph)

Arguments:
graph — T

Returns:
List

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

## Usage:
(gr:edges x)

## Arguments:
- x — Gr:Graph

## Returns:
List

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

## Usage:
(setf (gr:edges x) new-value)

## Arguments:
- new-value — T
- x — Gr:Graph

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

## Documentation:
The nodes in an Examiner-Graph are random Lisp objects. The edges represent references to a Lisp object that are contained in another Lisp object.

## Usage:
(typep x 'Br:Examiner-Graph)

## Parents:
Gr:Digraph

Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"
**br:examiner-references**

**Generic Function**

**Documentation:**
A generic function for navigating among related objects.

**Usage:**
```
(br:examiner-references object)
```

**Arguments:**
- `object` — T

**Source:**
```
#p"/begica-2g/jam/az/browser/examiner.lisp"
```

**br:examiner-references T**

**Primary Method**

**Documentation:**
Default is no references.

**Usage:**
```
(br:examiner-references object)
```

**Arguments:**
- `object` — T

**Returns:**
- `nil`

**Source:**
```
#p"/begica-2g/jam/az/browser/examiner.lisp"
```

**br:examiner-references Sequence**

**Primary Method**

**Documentation:**
Make a list of all the unique objects in the sequence.

**Usage:**
```
(br:examiner-references object)
```

**Arguments:**
- `object` — Sequence

**Returns:**
- `List`
Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"

<table>
<thead>
<tr>
<th>br:examiner-references</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Method</td>
<td></td>
</tr>
</tbody>
</table>

Documentation:
Examiners skip references in Arrays for the present.

Usage:
(br:examiner-references object)

Arguments:
object — Array

Returns:
il

Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"

<table>
<thead>
<tr>
<th>br:examiner-references</th>
<th>Standard-Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Method</td>
<td></td>
</tr>
</tbody>
</table>

Documentation:
Examiners look at all the references in the <az:interesting-slots> of a CLOS object.

Usage:
(br:examiner-references object)

Arguments:
object — Standard-Object

Returns:
List

Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"

<table>
<thead>
<tr>
<th>br:examiner-references</th>
<th>Clay:Graph-Coordinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Method</td>
<td></td>
</tr>
</tbody>
</table>

Documentation:
The interesting references in a <clay:Graph-Coordinator> are its subject and its diagrams.

Usage:
(br:examiner-references object)

Arguments:
**br:examiner-references String**

Object — Clay:Graph-Coordinator

Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"

<table>
<thead>
<tr>
<th>Primary Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>br:examiner-references String</strong></td>
</tr>
</tbody>
</table>

**Documentation:**
An examiner does not want to look at characters.

**Usage:**
(br:examiner-references object)

**Arguments:**

object — String

**Returns:**

nil

Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"

<table>
<thead>
<tr>
<th>Primary Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>br:examiner-references Symbol</strong></td>
</tr>
</tbody>
</table>

**Documentation:**
For now, Symbols just return the value and function. We might later add package and plist items.

**Usage:**
(br:examiner-references object)

**Arguments:**

object — Symbol

**Returns:**

List

Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"

<table>
<thead>
<tr>
<th>Primary Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>br:examiner-references Null</strong></td>
</tr>
</tbody>
</table>

**Documentation:**
Nil has no references.

**Usage:**
(br:examiner-references object)

Arguments:

    object — Null

Returns:

    nil

Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"

<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gr:Generic-Edge</td>
<td></td>
</tr>
</tbody>
</table>

Usage:

    (typep x 'Gr:Generic-Edge)

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gr:Generic-Graph</td>
<td></td>
</tr>
</tbody>
</table>

Documentation:

    A type for objects that have methods for the graph protocol.

Usage:

    (typep x 'Gr:Generic-Graph)

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

<table>
<thead>
<tr>
<th>Object</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gr:Graph</td>
<td></td>
</tr>
</tbody>
</table>

Documentation:

    A simple sample implementation of the Graph protocol.

Usage:

    (typep x 'Gr:Graph)

Parents:

    Az:Arizona-Object

Children:

    Br::Pedigree Gr:Digraph

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

<table>
<thead>
<tr>
<th>Object</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Graph</td>
<td></td>
</tr>
</tbody>
</table>

Usage:

    (in-package :Graph)
Clay:Graph-Diagram

Source: #p"/belgica-2g/jam/az/graph/package.lisp"

Clay:Graph-Diagram  Class

Usage:
(typep x 'Clay:Graph-Diagram)

Parents:
Clay:Root-Diagram

Source: #p"/belgica-2g/jam/az/layout/graph.lisp"

clay:graph-edge-menu  Generic Function

Documentation:
Return a list of menu items relevant to <edge> viewed as an edge in <graph>.
Each menu item must be a list whose first element is the string printed in the menu and whose second element is a function that is applied to the remainder of the list.
Append method combination is used to collect the menu items from all super classes (types).

Usage:
(clay:graph-edge-menu graph edge)

Arguments:

  graph — T
  edge — T

Returns:

  menu-spec — List

Source: #p"/belgica-2g/jam/az/layout/menus.lisp"

clay:graph-edge-menu Gr:Graph T  Append Method

Documentation:
Contributes a menu item for deleting edges.

Usage:
(clay:graph-edge-menu graph edge)

Arguments:

  graph — Gr:Graph
  edge — T
Source: #p"/belgica-2g/jam/az/layout/menus.lisp"

clay:graph-menu

Documentation:
Return a list of menu items relevant to <graph>.
Each menu item must be a list whose first element is the string printed in the menu and whose second element is a function that is applied to the remainder of the list.
Append method combination is used to collect the menu items from all super classes (types).

Usage:
(clay:graph-menu graph)

Arguments:

graph — T

Source: #p"/belgica-2g/jam/az/layout/menus.lisp"

clay:graph-menu Gr:Graph

Documentation:
Contributes no menu items.

Usage:
(clay:graph-menu graph)

Arguments:

graph — Gr:Graph

Source: #p"/belgica-2g/jam/az/layout/menus.lisp"

clay:graph-node-menu

Documentation:
Return a list of menu items relevant to <node> viewed as a node in <graph>.
Each menu item must be a list whose first element is the string printed in the menu and whose second element is a function that is applied to the remainder of the list.
Append method combination is used to collect the menu items from all super classes (types).

Usage:
(clay:graph-node-menu graph node)

Arguments:

graph — T
node — T
clay:graph-node-menu Br:Class-Graph Class

Returns:
    menu-spec — List

Source: #p"/belgica-2g/jam/az/layout/menus.lisp"

clay:graph-node-menu Br:Class-Graph Class

Documentation:
    A menu for classes.

Usage:
    (clay:graph-node-menu graph node)

Arguments:
    graph — Br:Class-Graph
    node — Class

Source: #p"/belgica-2g/jam/az/browser/class.lisp"

clay:graph-node-menu Br:Examiner-Graph T

Documentation:
    A menu for nodes in an examiner graph.

Usage:
    (clay:graph-node-menu graph node)

Arguments:
    graph — Br:Examiner-Graph
    node — T

Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"

clay:graph-node-menu Gr:Graph T

Documentation:
    Contributes a menu item for deleting nodes.

Usage:
    (clay:graph-node-menu graph node)

Arguments:
    graph — Gr:Graph
    node — T
Source: #p"/belgica-2g/jam/az/layout/menus.lisp"

```lisp
(gr:implements-graph-protocol? object)
```

**Documentation:**
Does this object have the necessary methods defined for it to be treated as a generic graph?

**Usage:**
```lisp
(gr:implements-graph-protocol? object)
```

**Arguments:**
- object — T

**Returns:**
- (Member T Nil)

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

```lisp
(br:make-class-graph classes)
```

**Documentation:**
Build the class graph from a list of classes by making edges for every direct subclass — direct superclass pair.

**Usage:**
```lisp
(br:make-class-graph classes)
```

**Arguments:**
- classes — List

**Returns:**
- Br:Class-Graph

Source: #p"/belgica-2g/jam/az/browser/class.lisp"

```lisp
(gr:make-dag edges)
```

**Documentation:**
Make a Dag from a list of edges. It is not checked for cycles.

**Usage:**
```lisp
(gr:make-dag edges)
```

**Arguments:**
gr:make-digraph

    edges — List

Returns:

  Gr:Dag

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

Function

Documentation:
    Make a Digraph from a list of edges.

Usage:
    (gr:make-digraph edges)

Arguments:
    edges — List

Returns:

  Gr:Digraph

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

Function

br:make-examiner-graph

Documentation:
    Makes an examiner graph for the list <objects>. The edges in the graph are gotten by calculating which of the objects have <examiner-references> to other objects in the list.

Usage:
    (br:make-examiner-graph objects)

Arguments:
    objects — List

Returns:

  Br:Examiner-Graph

Source: #p"/belgica-2g/jam/az/browser/examiner.lisp"
Documentation:
  Make a graph from a list of its edges.

Usage:
  (gr:make-graph edges)

Arguments:
  edges — List

Returns:
  Gr:Graph

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

clay:make-graph-diagram Function

Documentation:
  Make an interactive diagram for browsing and editing the <subject> graph.

Usage:
  (clay:make-graph-diagram subject &key coordinator slate diagram-name initial-node-configuratio:

Arguments:
  subject — Gr:Generic-Graph
  coordinator — (Or Null Clay:Graph-Coordinator)
  slate — Slt:Slate
  diagram-name — (Or String Symbol)
  initial-node-configuration — (Or Null (Simple-Array G:Screen-Coordinate (*)))
  x-order-f — (Or Symbol Function)
  y-order-f — (Or Symbol Function)
  node-pair-distance-f — (Or Symbol Function)
  rest-length — (Or Null Number)

Returns:
  Clay:Graph-Diagram

Source: #p"/belgica-2g/jam/az/layout/coordinator.lisp"

g:nodes Generic Function

Documentation:
  Returns a list of the nodes in the graph.
  For performance reasons, this may be the actual list of nodes used by the graph. If this list is
  modified, the results are unpredictable. It should be copied before any changes are made. To change
  the graph itself, use one of the editing operations, eg. <add-nodes-and-edges>. 
Usage:
   (gr:nodes graph)

Arguments:
   graph — T

Returns:
   List

Source: #p"/belgica-2g/jam/az/graph/protocol.lisp"

Usage:
   (gr:nodes x)

Arguments:
   x — Gr:Graph

Returns:
   List

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

Usage:
   (setf (gr:nodes x) new-value)

Arguments:
   new-value — T
   x — Gr:Graph

Source: #p"/belgica-2g/jam/az/graph/implementation.lisp"

References


REFERENCES

REFERENCES


