Galant User's Guide

This document describes how to use the Galant interface. The **G**raph **Al**gorithm **An**imation **T**ool, or Galant, provides facilities to manage graphs and algorithms and view animations.

Contents

1. Overview

Galant provides three major components across two screens:

1. Textual Editor screen
    a. Algorithm text editor
    b. Graph text editor
2. Visual Graph screen

Figure 1.1 gives a reference for windows and their components which will be referenced through this document.

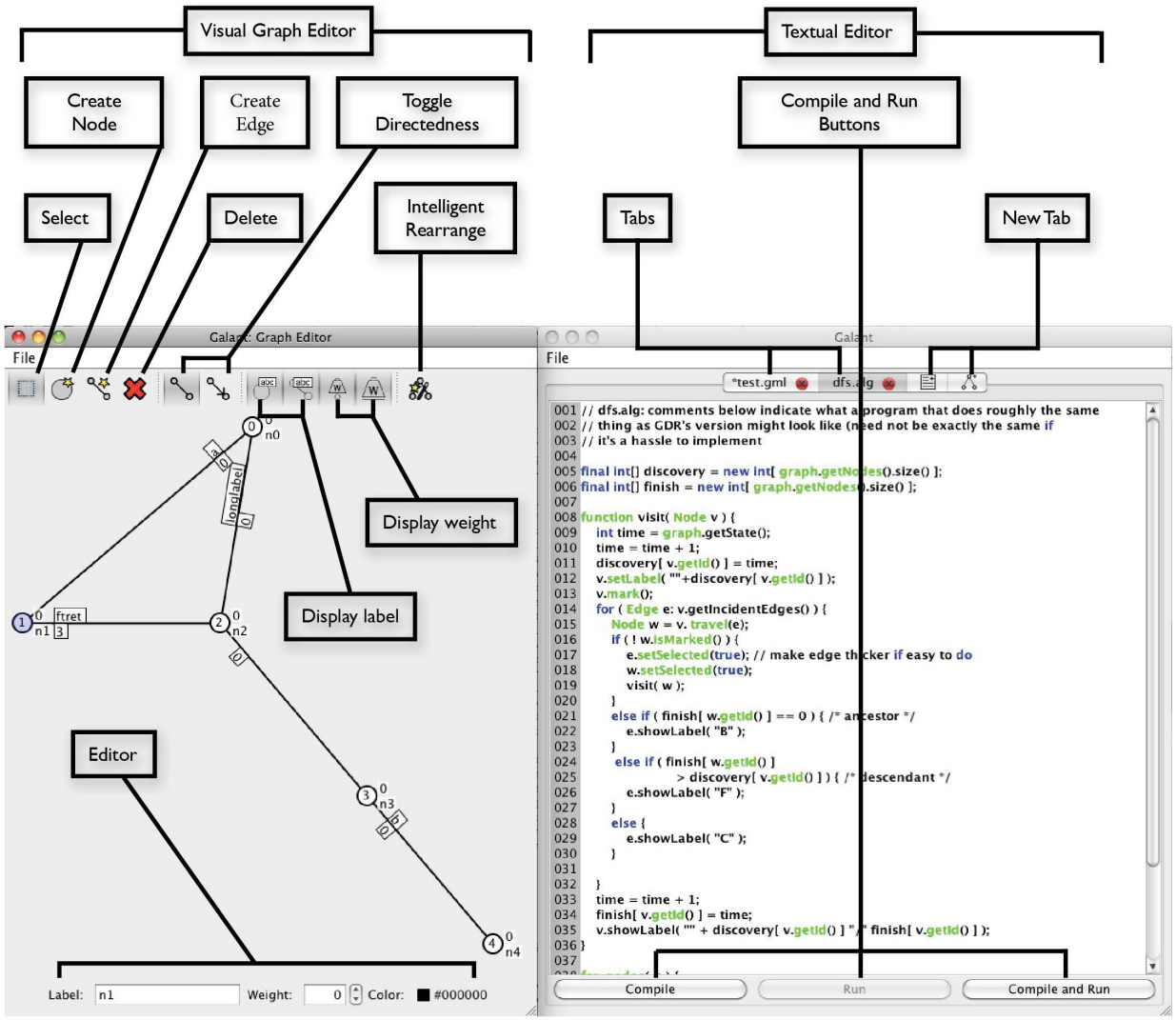

Figure 1.1

These components operate in two modes: Edit mode and Animation mode.

Edit mode allows the user to modify both graphs and algorithms through the means described in sections 3 (Graph Editing) and 4 (Algorithm Editing).

Animation mode disables all forms of modification except the ability to moves nodes on the screen. Animation mode allows the user to step through animation scenes through the means described in section 5 (Animating Algorithms). Exiting animation mode returns the program to Edit mode.

2. Workspace

Opened graphs and algorithm files are displayed in the textual editor in a single tabbed editor. These elements make up the Galant workspace. This workspace maintains the following properties.

The last selected graph tab becomes the active graph in Galant.

Galant's workspace always keeps at least one graph and one algorithm open.

File > Open allows the user to open Algorithm files (.txt, .alg) and Graph files (.gml) into the workspace. Opening a graph sets it to the active graph.

- Algorithms are Java-like text files. For more information on developing algorithms, please refer to the **Galant Programmer's Guide**
- Graphs are GraphML files. For more information on this standard, please see http://graphml.graphdrawing.org/

Clicking the new algorithm icon (figure 2.1) opens a new blank algorithm editor

figure 2.1: New algorithm icon

Clicking the new graph icon (figure 2.2) opens a new blank graph editor. Clicking anywhere in the visual graph editor will automatically add the default GraphML formatting.

figure 2.2: New graph icon

3. Graph Editing

Graphs can be edited textually in the “Textual Editor screen” or graphically in the “Visual Editor screen.” These editors are linked, and changes in one will update the other in the defined cases.

- Editing in the visual editor push instantly to the textual editor.
  WARNING: Any unsaved changes in the textual graph will be overwritten.
- Editing in the textual editor pushes to the visual graph on save.

Graphs are written in GraphML (extension .gml), and the textual editor implements keyword highlighting. For more information about this format, please see http://graphml.graphdrawing.org/

If an opened GraphML file is improperly formatted, the Visual Graph screen will appear blank. Fixing the GraphML and saving will cause the correct graph to appear. Making an edit in the Visual Graph screen before fixing the file will overwrite the GraphML.

The Visual Graph screen has a toolbar with four sections, seen in figure 3.1. The following descriptions refer to the sections labeled in figure 3.1.

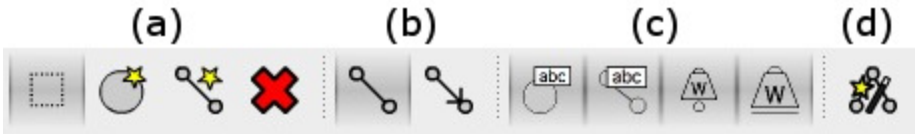

figure 3.1: Visual Editor toolbar

1. **Graph Edit mode (a)**. The selected option determines what happens when the user interacts with the screen. Only one of these is selected at a time. When interacting with the screen, the node with the highest id takes precedence, and nodes have precedence over edges. **Clicking and dragging nodes always moves then selects the node in focus and ignores any other click functions of the current mode.**
   - **Select**: Selects the nearest clicked graph component. This results in a dark overlay over nodes and a reddening of edges to indicate selection. If a component is selected, an inline editor will appear in the Visual Editor to allow editing of the component’s label, weight, and color.
   - **Create Node:** Creates a node at the location of the cursor if there isn’t another node at the cursor location. If there is another node beneath the cursor, it is instead selected to avoid accidentally stacking multiple nodes in the same position.
   - **Create Edge:** Creates an edge through a simple two-click procedure.
     1. Click the desired source node. It becomes selected, and an edge will

follow the cursor. If anything besides a node is clicked, edge creation is not started.
    2. Click the desired destination node. An edge is established between the source and destination, and all components are deselected. Clicking anywhere else on the screen cancels the creation of an edge.
- **Delete:** Deletes the topmost component at the cursor's clicked position. If the deleted component is a node, all edges with the selected node as a source or destination are also deleted.

2. **Directedness Toggle (b)**: Changes the graph between an undirected graph and a directed graph. All edges store their endpoints as a source and destination regardless of directedness mode, and undirected simply treats both nodes the same. When creating an edge, the first node clicked will be stored as the source and the last node clicked will be stored as the destination. The direction of an edge can be changed in the textual editor or by deleting and re-adding the edge in the opposite direction.

3. **Display Toggles (c)**: The four display toggles turn on/off the displaying of node labels, edge labels, node weights, and edge weights respectively. The first time Galant is opened, all of these are turned on by default. Any changes to these toggles persists through Galant sessions.

4. **Intelligent Rearrangement Toggle (d)**: Toggles on and off Galant's automatic graph rearrangement. When it is on, adding edges and deleting components triggers an automatic repositioning of nodes which is then scaled and centered to fill the screen.

4. Algorithm Editing

Algorithms can be edited textually in the "Textual Editor screen."

Algorithms are written in stripped-down java (extension .alg/.txt), and the textual editor implements keyword highlighting. Line numbers are displayed in the editor for help debugging.

In-depth information about developing new algorithms can be found in the **Galant Programmer's Guide.

5. Animating Algorithms

To animate an algorithm, an algorithm file must be compiled, then run via the Algorithm controls (figure 5.1). The algorithm runs on the active graph in Galant, i.e. the last in-focus graph in the workspace. If there are errors in either compilation or execution on a graph, an exception dialog will appear with error details.

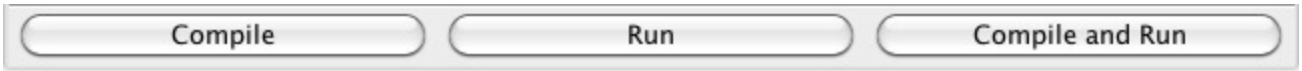

figure 5.1: Algorithm controls

After algorithms are compiled, their step-by-step animation can be iterated through by clicking on the Step Forward and Step Backward buttons in the animation window (figure 5.2). The Step Forward and Step Backward buttons become disabled when the animation is at the beginning and end of the queue, respectively.

figure 5.2: Animation controls

As the animation is stepped through, the graph displays highlighting that corresponds with the algorithm's actions at that time. For example, a node that is being visited will become highlighted.

Closing an animation via the exit button (figure 5.2) returns a graph to its original state.

Since graphs revert to their original state when graphs are closed, Galant provides an export feature to save graphs as they appear in a step of an algorithm. Click File > Export, then enter a filename for the exported GraphML.

6. Preferences

Galant Preferences can be accessed by clicking File > Preferences, or by pressing Ctrl + Shift + P (For Mac: Cmd + Shift + P).

From this menu, the default directory that is accessed when opening a graph or algorithm can be customized, as well as the default output directory for compilation.

Under Editors > Algorithm Editor, Java Keywords and API call colors can be specified. To change these colors, click anywhere on the color swatch or hex color representation to open up a color selection menu.

Under Editors > Textual Graph Editor, the highlighting color for GraphML keywords can be specified. To change these colors, click anywhere on the color swatch or hex color representation to open up a color selection menu.

Under Editors > Visual Graph Editor, the line width for edges can be specified. To increase the line width, either type in the appropriate width, in pixels, for the edges, or click the up and down arrows to increase or decrease, respectively, the edge width. This value has a max of 10.

Galant Programmer's Guide

This document describes how to write a graph algorithm for Galant. Graph algorithm code is written using Java syntax through the Algorithm Editor panel, and requires no knowledge of classes and minimal knowledge of functions. This is handled by Galant and the programmer need only write the function itself.

Contents

1. Getting Started

Provided by Galant and prepended to the written code is a Graph object, which is accessible through the `graph` variable or the `getGraph()` function.

Graph States

When a graph is modified, a new state is created which is used by Galant to step forward and backwards through steps in the animation. Each individual API, unless marked otherwise, will increment this graph state by one. If this doesn't make sense for the algorithm (e.g. several modifications should be changed at once), the user can lock the graph state and unlock it when a set of atomic updates have been completed.

WARNING: If a state is left locked, all executed code following the lock will appear as one step in the graph animation.

Graph Components

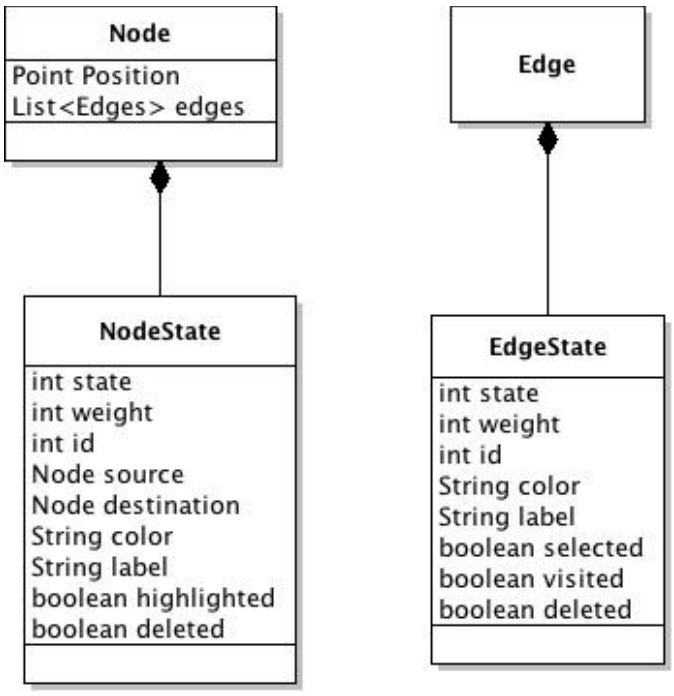

figure 1.1: Component properties

A Graph is a collection of Node and Edge objects. Each of these maintains a set of states representing each modification of the component. Larger modifications (done through the locking mechanism) will overwrite states appropriately with all the changes. Figure 1.1 shows all of the

properties associated with each component.
The natural ordering of Nodes and Edges is by ascending order of weight in their latest state. If these are stored in a collection, e.g. List<Node> nodeList, sorting them can be achieved by executing Collections.sort(nodeList).

2. API

Algorithms are used to create graph animations by manipulating components through the following APIs. Unless otherwise indicated, all of the following methods that set values will create a new state.

Algorithm

beginStep()	Forces the animation to consider all graph changes one step until endStep() or another beginStep() is called.
endStep()	Ends a step in the graph state. If no beginStep() has been called previously, does nothing.

Graph

isDirected()	Returns TRUE if the graph is directed, FALSE otherwise.
setDirected(boolean directed)	Sets the graph to directed for True and undirected for False
getNodes()	Gets a list of all non-deleted nodes in the latest state of the graph.
getEdges()	Gets a list of all non-deleted edges in the latest state of the graph.
getNodeById(int id)	Gets the node whose id matches the input. Returns null if the node doesn’t exist or has been deleted.
getEdgeById(int id)	Gets the edge whose id matches the input. Returns null if the edge doesn’t exist or has been deleted.
select(Node n)	Sets the specified node as the selected node in the

	graph and deselects all other selected nodes. Note: If the desired effect is to leave other Nodes selected, see the Node class's setSelected method.
addNode()	Adds a Node with default settings to the graph and returns a pointer to the new Node.
addEdge(Node source, Node target)	Adds an edge to the graph between the two specified nodes, and also stores the edge in each Node.
addEdge(idSource, idTarget)	Adds an edge to the graph between two nodes with specified indices, and also stores the edge in each node.
writeMessage(String message)	Stores a message to be displayed on the screen when the current state executes. Note that when multiple messages are written in a single state, only the latest one will be written.
getMessage():String	Gets the message string stored to display on-screen in the current state

Node

isSelected():boolean	Returns True if the node is selected in the latest state and False otherwise.
setSelected(boolean selected)	Sets the node as selected. In contrast with the Graph's select function, this does not deselect other selected nodes in a graph. Note: Selected nodes will be displayed with a red ring, overriding the default color set for a node.
isVisited():boolean	Returns the true if the Node has been marked as visited and False otherwise.
setVisited(boolean selected)	Sets the Node's visited/marked property to the input value. Note: Marked nodes' interior color changes from white to light grey.
isMarked():boolean	Returns the true if the Node has been marked

	and False otherwise. Note: Exhibits the same behavior as isVisited()
mark()	Sets the Node's visited/marked property to True Note: Marked nodes' interior color changes from white to light grey.
getWeight():int	Returns the Integer weight of the current node. The default weight is 0.
setWeight()	Sets the weight of the current node.
getUnvisitedPaths():List<Edge>	Returns a List<Edge> object of all the Node's incident edges connecting to unvisited Nodes in the latest state of the graph.
getVisitedPaths():List<Edge>	Returns a List<Edge> object of all the Node's incident edges connecting to visited Nodes in the latest state of the graph.
getUnvisitedAdjacentNodes():List<Node>	Returns a List<Node> object of all the Node's adjacent Nodes whose visited properties are False in the latest state of the graph.
travel(Edge e):Node	Returns the other Node endpoint of the specified edge. Returns null if neither of the edge's endpoints is this Node. If e is a loop, returns this Node.
getId():int	Returns the numerical id of the node. This will not change during Algorithm execution unless explicitly set in the user code.
getColor():String	Gets the default color of the Node in the format '#RRGGBB'
setColor(String color)	Sets the default color of the Node. This color will corresponds to the colored ring around a node and will be the default color when the node is not selected. A valid color input should be of the form '#RRGGBB'
getLabel():String	Returns the String label associated with the Node
setLabel(String label)	Sets the label associated with the Node

getPosition():Point	Gets the position of the node on the coordinate plane.
setPosition(Point p)	Sets the position of the Node on the coordinate plane. Node: This position is not associated with a State, thus changing it changes the position of the Node in all states of the animation.
setPosition(int x, int y)	Sets the position of the Node on the coordinate plane. Node: This position is not associated with a State, thus changing it changes the position of the Node in all states of the animation.
equals(Node n):boolean	Returns true if the given Node points to the same Node as the current Node and false otherwise.
setStringAttribute(String:key, value)	Stores a String in the node under the specified key. This will overwrite any String, Integer, or Double attribute already set with the same key.
getStringAttribute(String key):String	Gets the String value associated with the specified key. Returns null if the key doesn't exist or it exists but its value is not a String.
setIntegerAttribute(String:key, Integer value)	Stores an Integer in the node under the specified key. This will overwrite any String, Integer, or Double attribute already set with the same key.
getIntegerAttribute(String key):Integer	Gets the Integer value associated with the specified key. Returns null if the key doesn't exist or it exists but its value is not an Integer.
setDoubleAttribute(String key, Double value)	Stores a Double in the node under the specified key, This will overwrite any String, Integer, or Double attribute already set with the same key.
getDoubleAttribute(String key):Double	Gets the Double value associated with the specified key. Returns null if the key doesn't exist or it exists but its value is not a Double.

Edge

isSelected():boolean	Returns True if the edge is selected in the latest state and False otherwise.
setSelected(boolean selected)	Sets the edge as selected. In contrast with the Graph's select function, this does not deselect other selected edges in a graph. Note: Selected edges will be displayed with a red line, overriding the default color set for a node.
getWeight():int	Returns the Integer weight of the current edge. The default weight is 0.
setWeight()	Sets the weight of the current edge.
getSourceNode():Node	Returns the source Node of the current Edge. Note: This has no additional meaning for undirected graphs. This method will return whatever is stored as the "source" attribute in the graphml.
setSourceNode(Node n)	Sets the source Node of the current Edge. Note: This has no additional meaning for undirected graphs. This method will set the "source" attribute in the graphml.
getDestNode():Node	Returns the destination Node of the current Edge. Note: This has no additional meaning for undirected graphs. This method will return whatever is stored as the "destination" attributein the graphml.
setDestNode(Node n)	Sets the destination Node of the current Edge. Note: This has no additional meaning for undirected graphs. This method will set the "destination" attribute in the graphml.
getOtherEndpoint(Node n):Node	Returns the other endpoint of an edge, or NULL if the provided node is not either endpoint in the specified edge.
getId():int	Returns the unique ID of the edge. This will not change during Algorithm execution unless explicitly set in the user code.

getColor():String	Gets the default color of the Node in the format '#RRGGBB'
setColor(String color)	Sets the default color of the Edge. This color will corresponds to the highlighting on the edge's path and will be the default color when the node is not selected. A valid color input should be of the form '#RRGGBB'
getLabel():String	Returns the String label associated with the Node
setLabel(String label)	Sets the label associated with the Node
equals(Edge e):boolean	Returns true if the given Edge points to the same Node as the current Edge and false otherwise.
setStringAttribute(String:key, value)	Stores a String in the edge under the specified key. This will overwrite any String, Integer, or Double attribute already set with the same key.
getStringAttribute(String key):String	Gets the String value associated with the specified key. Returns null if the key doesn't exist or it exists but its value is not a String.
setIntegerAttribute(String:key, Integer value)	Stores an Integer in the edge under the specified key. This will overwrite any String, Integer, or Double attribute already set with the same key.
getIntegerAttribute(String key):Integer	Gets the Integer value associated with the specified key. Returns null if the key doesn't exist or it exists but its value is not an Integer.
setDoubleAttribute(String key, Double value)	Stores a Double in the edge under the specified key, This will overwrite any String, Integer, or Double attribute already set with the same key.
getDoubleAttribute(String key):Double	Gets the Double value associated with the specified key. Returns null if the key doesn't exist or it exists but its value is not a Double.

Data Structures

Default instances of queues, stacks, and priority queues containing Nodes and Edges are available. The variables for these are: nodeQ, edgeQ, nodeStack, edgeStack, nodePQ, and edgePQ.

These variables are instances, respectively, of the classes NodeQueue, EdgeQueue, NodeStack, EdgeStack, NodePriorityQueue, and EdgePriorityQueue. These classes use the interfaces of java.util.Queue, java.util.Stack, and java.util.PriorityQueue.

3. Macros

Macros abstract away some Java code and provide a simpler interface specific to Galant and graph algorithms.
When an algorithm is compiled, macros are converted into Java code for the Java compiler. This is similar to C's preprocessor, although macros are defined within Galant rather than in the algorithm code itself.
Note: macros are not yet able to ignore text in comments or strings.

for_adjacent – iterates over the adjacent nodes and edges of a given node.

Usage: for_adjacent(*node*, *edge*, *adjacentNode*) {*code_block*}

Parameters:

node: the name of a variable of type Node. This is the starting node.

adjacentNode: a variable name. Within the code block, this can be used to refer to the current adjacent node as a Node object.

edge: a variable name. Within the code block, this can be used to refer to the current incident edge as an Edge object. *edge* connects *node* and *adjacentNode*.

code_block: a block of code that is executed for each adjacent node / incident edge of *node*. The curly braces are required.

for_nodes – iterates over all nodes in the graph.

Usage: for_nodes(*node*) *code_block*

Parameters:

node: a variable name. Within the code block, this can be used to refer to the current node as a Node object.

code_block a block of code that is executed for each node in the graph.

for_edges – iterates over all edges in the graph.

Usage: for_edges(*edge*) *code_block*

Parameters:

edge: a variable name. Within the code block, this can be used to refer to the current edge as an Edge object.

code_block: a block of code that is executed for each edge in the graph.

function – creates a function that can be called later.
Functions are objects (of type Function), and so they can be assigned to variables and passed to other functions. (Note: this is not properly implemented at the moment (and probably isn't very high priority).)

Note: currently, variables declared outside a function are not accessible in the function body unless they are declared `final`.
If you want a modifiable "global variable", one workaround is to create an final array containing the variable (e.g., `final int[] i = {`inital value`};` and within the function, `i[0] =` new value`;`).
Another, somewhat more self-descriptive workaround is to use a class called something like `Globals`, with fields corresponding to the global variables, and then to create a final instance of that class through which they can be accessed. Example: `class Globals { int i =` initial value`; } final Globals glob = new Globals();` and within the function `glob.i =` new value`;`

Usage: function *[return_type] name*(*params*) {*code_block*}

Parameters:

return_type (optional): a type. If the function returns a value, this should indicate the type of value returned.
If no value is returned, this is not necessary.

name: a variable name. Used to identify and call the function.

params: a comma-separated list of variable names, including types (e.g., "int i, String str").
Can be referenced from within the code block. May be empty, if there are no parameters.

code_block: a block of code that is executed when the function is called. The curly braces are required.

Calling a function – calls a function created by **function**.

Usage: *name*(*args*)
Returns a value of the type defined by *return_type* in **function**.

name: the same as *name* in **function**.

args: values of the types determined by *params* in **function**, which are passed to the function.

Notes
The keyword `bool` can be used in place of `boolean` if desired.

4. Imports

A set of classes, shown below, are imported by default into the algorithm. If a user requires an additional import, it may be specified at the top of the algorithm.

java.util.LinkedList
java.util.Queue
edu.ncsu.csc.Galant.algorithm.Algorithm
edu.ncsu.csc.Galant.graph.component.Graph
edu.ncsu.csc.Galant.graph.component.Node
edu.ncsu.csc.Galant.graph.component.Edge
edu.ncsu.csc.Galant.algorithm.code.macro.Function
edu.ncsu.csc.Galant.algorithm.code.macro.Pair