GraphQuil
Language Reference Manual
COMS W4115

Steven Weiner (Systems Architect), Jon Paul (Manager),
John Heizelman (Language Guru), Gemma Ragozzine (Tester)

Chapter 1 - Introduction

This is a reference manual for the GraphQuil language. GraphQuil is a specialized programming language designed to easily manage and manipulate data stored in graphs. It is simultaneously a data-definition language as well as a data manipulation language. Users operating GraphQuil will have the ability to create vertices that represent objects while also being able to link them together and define their relationships within a graph.

Chapter 2 - Types

2.1 Explicit Typing

Data in GraphQuil requires explicit typing. In naming any variable type, an explicit specification of type is required.

2.2 Primitive Types

GraphQuil provides four primitive types that make up the basis for arranging data in your programs: *int*, *double, char*, and *bool*. These are the building blocks that define what type of data can be included in each Node.

2.2.1 Ints

Integers, denoted by the keyword *int* are 32-bit numbers with a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).

2.2.2 Doubles

Doubles, denoted by the keyword *double*, are implemented as 64-bit numbers using the IEEE-754 standard.

2.2.3 Chars

Chars, denoted by the keyword *char*, is an 8-bit data type used specifically for storing ASCII characters.

2.2.4 Booleans

Booleans, denoted by the keyword *bool*, accept only *true* or *false* values and are used in comparisons of equality.

2.3 Non-Primitive Types

2.3.1 Arrays

An array is a data structure of user-defined types that holds variables and other data structures. Here is an example of the initialization of an array called “intArray” that would store *ints*:

```
int[] intArray = new int[5];
```

Arrays are dynamically sized according to number of elements stored within the Array. *intArray* has been initialized as being able to store five *chars*, and so its size is at least 160 bits. Arrays are accessed by index, with indexing starting at 0. The syntax for doing though uses the same brackets as the construction, and returns the value at that index. The same works for assigning to an array. An example of each:

```
int x = intArray[0];      #this accesses the array and copies the value at 0 to x
intArray[1] = x;#this sets the spot at index 1 in the array to x
```

Finally, arrays store one additional piece of data, their length as an int, which can be accessed using the operator '.' followed by the identifier *length*. For example, int len = intArray.length; would set len to 5.

2.3.2 Strings

Strings are implemented as arrays of *chars*. Whitespace is not ignored. Here is an example of the initializing of a String called "string":

```
String string = "string";
```

The string stores six *chars*, and so its size is at least 48 bits.

2.3.3 Nodes

A *Node* is a data structure specifically for use in a *Graph*. Nodes are required to store one array, *edges*, that stores all the outward edges from the node. This array is special in that it may not be modified directly; it is modified automatically on the linking of two nodes, as seen in section 4.2.6. Additionally, nodes can only be present in one graph at a time, and a duplicate node with the same data must be created before adding it to a different graph. To meet this requirement, the graph must be specified when creating the node.

Each node must also be given its own type before creation, and there is no base implementation of Node within GraphQuil that stores certain variables. Before declaring a node, there must be a defined NodeType to denote what that specific kind of node will hold (see section 3.4 for more information on NodeTypes). However, links can be made irrespective of type, and the type of node does not affect the way links are created between them in the graph.

2.3.4 Edges

An edge is a link created between two nodes. There are two different kinds of edge types, weighted and unweighted. Unweighted edges create links that connect two nodes, but provide no information about those links. Weighted edges, however, in a way similar to nodes, can contain all kinds of data about the connections between different nodes. To create a weighted edge between two nodes, there must first be a defined EdgeType (see section 3.5 for more info), then a declaration of the edge that specifies what type of edge to be created and the data to populate said edge (see section 4.2.7 for the syntax for doing so).

2.3.5 Graphs

Graphs are an aggregate of nodes that allow certain operations to be run upon them and are identified by the keyword *Graph*. A graph contains one piece of data, which is the array of all of the nodes contained within it, named with the identifier *nodes*. Because graphs are simply aggregations of different nodes, the syntax used to define them only requires creating an identifier, so the following is sufficient to define a graph: Graph g1;

Chapter 3 - Lexical Conventions

3.1 Identifiers

Identifiers are the names used to identify individual variables within a given function. Each variable must be given a unique identifier within its scope of the function, and each identifier is case-sensitive. These identifiers must begin with an ascii letter (a-z or A-Z), but each following letter can be either a letter, a digit (0-9), or an underscore (_). The only additional restriction on what strings can be used as identifiers is that they must not match one of the GraphQuil's reserved keywords.

3.2 Reserved Keywords

The following is a comprehensive list of reserved keywords inGraphQuil:

add	bool	char
continue	dest	do
double	edges	else
false	for	Graph
has	if	in
int	new	Node
num	print	return
static	String	true
void	while	

3.3 Literals

3.3.1 Integer Literals

Integer literals are made up of digits without any decimal points or other characters separating the digits. They are solely to be used for whole numbers, but they can be positive or negative, distinguished by an included or omitted '-' sign. Additionally, integer literals may not begin with a 0.

3.3.2 Double Literals

Double literals include digits with a single decimal point, followed by at least one number. A 0 can be the first number if and only if it is immediately followed by a decimal point, although in such a case the 0 is not required. Beyond that, any sequence of numbers followed by a decimal point and another sequence of numbers of length >= 1 can be a double.

3.3.3 Bool Literals

The bool type can hold two different values, *true* or *false*, and those exact literals are used to declare which value the bool takes.

3.3.4 Char Literals

Char literals are a single character surrounded by a single quote (‘) on each side. This character can be any character in the 16-bit Unicode character set, with a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

3.3.5 String Literals

String literals are sequences of chars anywhere from length 0 to the maximum integer value (around 2 billion characters). They must begin and end with quotation marks (“) and can be filled with any character within the same character set as chars. Additionally, strings support a number of special characters to insert codes into the string, as listed below:

\'	Single quotation mark (‘)
\"	Double quotation mark (“)
\\	Backslash (/)
\t	Tab
\b	Backspace
\r	Carriage return
\n	Newline

3.4 Node Types

Before declaring a specific node, a node type must be first established, as there is no general type node to be instantiated by the user. To do so, there must be a line defining what type of node the identifier will be associated with. The type includes a listing of all the values to be stored in that node type, as well as the type of values the node stores. These values can be of any valid primitive or non-primitive type, with the exception of graphs, edges, and nodes. Using nodes to store other nodes removes the purpose behind a graph structure, so all data contained within a node must be of some other type.

Additionally, each value must be associated with an identifier to indicate what the name of that value is, and each identifier must be unique for that node type. For example, the following is a valid declaration of a NodeType:

```
#assume there is a defined Graph g1
NodeType Student has (String name, int age, String[] courses);
```

To create a particular instance of a node, the type must be specified in its construction, along with the graph the node is inserted into:

```
Node john = new Student(“John”, 21, {“PLT”, “CS Theory”, “Aristotle”}) in g1;
```

3.5 Edge Types

Creating a weighted edge requires an edge type to define what kind of data the edge will hold. This structure is very similar to defining a NodeType, and requires the identifier of the EdgeType, as well as the types and identifiers for each piece of data to be stored. Also, the same requirement applies in that these values can take any type other than nodes, graphs, or other edges. The convention is as follows:

EdgeType relationship has (String type, int closeness, bool positive);

3.6 Operators

The following denotes the operators of the language and their intended function:

Operator	Use	Associativity
+	Adds two values	Left
-	Substracts right from left	Left
*	Multiplies two values	Left
/	Divides right into left	Left
%	Modular division	Left
=	Assignment	Right
==	Equal to	Left
!=	Not equal to	Left
<	Less than	Left
>	Greater than	Left
<=	Less than or equal to	Left
>=	Greater than or equal to	Left
->	Link from left to right	Right
<->	Bi-directional link	Right
!	Logical negation	Right
&&	Logical and	Left
||	Logical or	Left

The order of precedence is as follows, from greatest to least important:

* / %

+ -

< > <= >=

!

== !=

&&

||
= -> <->

3.7 Separators

The following indicates the separators of the language and their usage.

“;”
Signals the end of a line.

“,”
Separates one argument from another argument in a declaration statement.

“(...)”

A parenthesised expression can be used to separate tokens. The type and value are identical to the same expression without the surrounding parentheses.

3.8 Whitespace

White space includes the space character, the tab character, and the newline character. Generally used to separate characters, it is entirely optional. No white space is required.

3.9 Comments

Multi-line comments are enclosed in #* *# characters. The #* characters introduce a multi-line comment and the *# signals the termination of the multi-line comment.

Single line comments are introduced by the character #. The single line comment terminates at the end of the line with no terminating symbol.

- Data Type Combinations (Arrays, functions, pointers(?), structs)

Chapter 4 - Syntax

4.1 Program Structure

A valid program in this language can be composed of as little or as many statements that a user desires.

4.2 Expressions

4.2.1 Unary Operators

!	The character '!' is the logical NOT operator. This operator checks the value of the one boolean operand. If the boolean operand evaluates to true, then the expression returns false. If the boolean operand evaluates to false, the expression returns true.

4.2.1 Binary Operators

The binary operators are divided into give categories: Assignment Operators, Relational Operators, Arithmetic Operators, Logical Operators, and Graphical Operators.

4.2.3 Assignment Operators

All assignment operators group right to left.

=	The simple assignment operator is the '=' character. Simple assignment takes the following form: type-declaration identifier = expression The value of the expression on the right hand side of the operator replaces the semanti value of the identifier on the right. This value must be equal in type to the declared type of the identifier.
+=	The add AND assignment operator is the two characters +=. This operator adds the semantic value of the operand on the right to the value of the left operand. This operato can only be applied to operands of primitive data types.
+-	The subtract AND assignment operator is the two characters -=. The operator subtracts the semantic value of the operand on the right from the value of the left operand. This operator can only be applied to operands of primitive data types.

4.2.3 Relational Operators

==	Two sequential equal signs denotes an equality operator. This operator checks if the values of the two operands are equal in reference. If they are equal, it returns a “true” boolean, else it returns a “false” boolean.
!=	The characters != denote a “not equal” operator. Like the equality operator, it checks to see if the two operands are equal or not. If the left and right operands are equal in reference, then the expression evaluates to “false”. If the operands are not equal in reference, then the expression evaluates to “true”.
>	The character ‘>’ denotes the “greater than” operator. This can only be used when the left and right operands are primitive types. The “greater than” operator checks if the value of the left operand is greater than the value of the right operand. If this is true, then the expression returns true. If not, the expression evaluates to false.
<	The character ‘<’ denotes the “less than” operator. This can only be used when the left and right operands are primitive types. The “less than” operator checks if the valu of the left operand is less than the value of the right operand. If this is true, then the expression returns true. If not, the expression evaluates to false.
>=	The characters “>=” denote the “greater than or equal to” operator. This operator checks if the value of the left operand is greater than or equal to the value of the right operand. If this is true, then the expression returns true. If not, the expression evaluates to false. Like the “greater than” and “less than” operators, this operator can only be used when both operands are primitive types.
<=	The characters “<=” denote the “less than or equal to” operator. This operator checks if the value of the left operand is less than or equal to the value of the right operand. I this is true, then the expression returns true. If not, the expression evaluates to false. Like the “greater than”, “less than”, and “greater than or equal to” operators, this operator can only be used when both operands are primitive types.

4.2.4 Arithmetic Operators

These operators only can be performed on primitive data types.

+	The ‘+’ character signifies the addition operator. It adds the semantic value of the operands on the left and right side.
-	The ‘-’ character is the subtraction operator. It subtracts the value of the right hand operator from the left hand operator to produce a new result.
*	The ‘*’ character is the multiplication operator. It multiplies the values of the operands on the left and right side of the operator.

/	The '/' character denotes the division operator. This divides the value of the left hand operator by the value of the right hand operator.
%	The '%' character is the modulus operator. It divides the value of the left hand operator by the value of the right hand operator, and returns the remainder of the division.

4.2.3 Logical Operators

&&	The characters "&&" signify the logical AND operator. This operator checks the value of two boolean operands. If both boolean operands evaluate to true, then the expression returns true. In any other case, the expression evaluates to false.
||	The characters "||" signify the logical OR operator. Like the logical AND operator, this checks the value of the two boolean operands, but it only returns true if any of the operands (the left operand, the right operand, or both operands) evaluate to true.

4.2.3 Graphical Operators

These operators only work when both operands are Nodes.

->	The characters "->" signify the directional link operator. It adds a directional link (edge) from the operand on the left hand side to the operand on the right hand side. The operand on the right hand side is added to the list of destination nodes of the operand on the right hand side.
<->	The characters "<->" denote the bidirectional link operator. It adds a bidirectional (and therefore, undirected) link between the left operand and the right operand. In each Node operand, the other Node operand is added to the list of destination nodes.

4.2.4 Parenthesized Expressions

Parenthesized expressions are a tool to override the built-in order of operations in GraphQuil. Parenthesis force the evaluation of the expression inside of them before using that value in whatever other expression the parenthetical is contained in. For example, if a given expression is 1 - 2 + 3, the internal order of operations would give the equivalent of (1 - 2) + 3, which equals 2. If however, parenthesis are added to make 1 - (2 + 3), which equals -4. This works for all operators in all expressions.

4.2.5 Function Creation

Functions are declared by giving a return type of the function, a unique identifier for the function, any arguments passed in, then creating and closing a block with the function body.

The syntax is as follows:

```
return_type identifier (argtype arg_identifier,…)   {   … return out;  }
```

The type of out must match the return type declared at the beginning of the function, and each function must return a type. The only exception to this rule is if the function is declared with the return type *void*, then the function should not return any type. If at any point the function should return before the end of the block, putting “return;” will break out of the function and return to wherever the function was called.

4.2.6 Linking Nodes

Given two nodes *node1* and *node2*, there are two main ways of creating edges between the nodes, weighted and unweighted. Edges that have no weight merely create a connection between the two nodes and contain no more information about those connections. They are created using the following statements:

```
node1 -> node2;      #Creates an unweighted, directed edge from node1 to node2
node1 <-> node2;     #Creates an unweighted, bidirectional edge between the nodes
```

To create a weighted node, there must first be a specified edge type (see section 3.5 which describes how to do so). Then, the syntax requires an assignment of an EdgeType to the edge created between the two nodes. In the same way a variable is assigned a value, the edge created between two nodes can be assigned a value (which requires a type) by assigning a newly constructed weighted edge to the connection between two nodes, as seen here:

```
EdgeType relationship has (String type, int closeness, bool positive);
billy -> allie = new relationship(“siblings”, 10, true);
```

Finally, either kind of edge can only be valid if both nodes are in the same graph. The compiler will give an error if the two nodes are not in the same graph.

4.2.7 Accessing Data from Non-primitive Types

To access the data stored in a non-primitive type, the operator ‘.’ is used, followed by the identifier given to the piece of data you want to access. This access has highest precedence in the order of operations, and the value will be pulled from the object before any other operations are done using the value, unless overridden by parenthesis. It works for accessing the values in nodes, edges, the length of arrays, and the node array of a graph. All of the following are valid:

```
int x = john.age;                  #where john is a Node containing an int named age
x = intArray.length;
Node[] nodes = g1.nodes;           #note that this returns all nodes in a given graph
String str = edge1.relationship;   #where edge1 is an Edge containing said String
```

4.2.8 Using Edges to Retrieve Data

Accessing all the nodes connection to a node is done through the edges directed outward from a node. To access these, each node type has an implicit array stored inside it with the identifier *edges*. This array has contains values of type Edge (which includes all EdgeTypes and unweighted edges), and can be accessed in the same way any other data can be accessed from a node by the following:

```
Edge[] edgeArray = node_indentifier.edges;
```

Then, that array can be used to iterate through and find any particular edge or node needed. Additionally, each edge, whether weighted or unweighted, contains a reference to the destination of the edge, associated with the identifier *dest*. This is done using the same syntax for accessing data for any non-primitive type, as follows: Node example = edge1.dest;

4.3 Statements

At their core, statements execute specific tasks, control the order and flow of processes being taken, and are performed in the order in which they appear.

4.3.1 Assignment Statements

Assignments of nodes and node values are performed in two separate ways. The first involves setting a variable as equivalent to another variable or expression. This is performed by using an equals sign, seen in the following:

```
Node node1= new CUStudent ('jrh2184', 'John', 'Heizelman', 2016) in classroom;
Node node2= new CUStudent ('jp3144', 'Jon', 'Paul', 2015) in classroom;
node2=node1;
#node2 now refers to the CUStudent node named with uni jrh2184
#the CUStudent node with uni jp3144 still exists in the classroom, though
```

To assign values into node fields, the '.' operator is used:

```
node2.uni= "jrh2184";
#updates the uni field of node1 to jrh2184
```

4.3.2 Function Calls

Function call statements are performed simply by using the desired function in any kind of expression. The function call takes highest precedence and will return its data type and use that piece of data in the expression. See the following example:

```
node2.uni = getUni(name);
```

Where getUni is defined as: String getUni(String name) { ... }

4.3.3 Conditional Statement

Usage of the *if* statement allows a certain set of steps to be taken given an explicit condition.

For example, here is the construction for an *if* statement based on an explicit condition.

```
if (node1.year== 2015) {
        #insert set of tasks to be performed here
}
```

Additionally, if a given condition is not met and the user desires to outline another condition, the *else if* construction can be used.

```
if (node1.year== 2015) {
        #insert set of tasks to be performed here
} else if (node1.year==2016) {
        #insert set of tasks to be performed here
}
```

Lastly, the *else* construction allows users to perform a set of tasks if none of the prior mentioned *if* or *else if* conditions are met.

```
if (node1.year== 2015) {
        #insert set of tasks to be performed here
} else if (node1.year==2016) {
        #insert set of tasks to be performed here
} else {
        #insert set of tasks to be performed here
}
```

4.3.4 Iteration Statement

Iteration statements are performed using *for* or *while,* in order to generate loops that execute a set of tasks.

For example, running a set of tasks on all pieces of data in an array uses a *for* loop. The syntax is simple, where each object of the type in the array can be named (in this case tmp), and the loop will apply the set of tasks in the brackets to each object in the array. See the following example

```
for ( Node tmp : classroom) {        #where classroom is an array filled with nodes
        #insert set of tasks to be performed here
}
```

Using a *while* loop is very similar. The set of task outlined within the loop are executed as long as the condition denoted in the *while* loop is met, where the condition is an expression that results in a bool. The condition is re-evaluated after each pass through the loop (given the case that the condition initially passes).

```
while (node1.year==2015){
        #insert set of tasks to be performed here
}
```

4.4 Scope

4.4.1 Scoping within blocks

Variables defined within blocks (separated by { and }) are only accessible within the block they are defined. For example, the following is invalid:

```
if(x = true)     {
        String abc = “abc”;
}
abc.length();
```

The only exception to this blocking rule is the declaration of non-primitive variables using the *new* keyword, be they graphs, nodes, or some other type. This declares a new object that is not deleted until all references to it are removed and are taken care of by behind-the-scenes garbage collection.

4.4.2 Scoping within functions

Similar to the block scoping, variables defined within a function are only applicable within that function. This applies both to the variables passed into a function and the ones named and created within it. The same exception is the declaration of non-primitives using the *new* keyword applies as before.

The main difference in the difference between function scoping and scoping within a block is in the return statement. When returning a primitive type, the value of the primitive being returned is copied directly to the expression that called the function. However, when returning a non-primitive type, a reference to the object is returned and used in the expression. The memory used to store the object remains valid until all references to it are gone, then the built-in garbage collection system removes it from the allocated memory.

Chapter 5 - Library Functions

5.1 Print

GraphQuil includes a built-in print to console feature. To print a String or primitive type *toPrint*, the statement:

```
print(toPrint);
```

Multiple Strings or primitives may be printed to console in one statement, separated by the plus sign as follows (for String or primitives *toPrint1* and *toPrint2*):

```
print(toPrint1 + toPrint2);
```

Any number of Strings or primitives may be printed in a single print statement as long as they are separated by plus signs as shown above.

Attempting to print the literal “\n” produces a newline in the console.

Only Strings and primitives may be printed; printing non-primitives directly is not allowed and elicits a compiler warning.