GRAIL Language Reference Manual

Aashima Arora (aa3917) - system architect
Rose Sloan (rns2144) - manager
Jiaxin Su (js4722) - tester
Riva Tropp (rtt2114) - language guru

February 22, 2017

1 Introduction

This manual describes GRAIL, a language optimized for building and performing computations on graphs. The syntax is streamlined to facilitate easily constructing, accessing, searching, and modifying graphs, nodes, and edges.

2 Tokens

GRAIL tokens are separated by one or more whitespace characters. Comments delimited by /* and */ or single-line comments beginning with // are also ignored. Comments may not be nested.

2.1 Identifiers

An identifier is a sequence of characters, all of which must be either alphanumeric or the underscore (_) character. The first character must be a letter. Uppercase and lowercase letters are considered distinct but the choice of case in identifiers holds no significance to the compiler.

2.2 Keywords

The following identifiers are reserved as keywords and may not be used elsewhere: graph, digraph, edge, string, num, char, void, boolean, if, else, while, for, return, weight, to, from, sort, break, continue, display, type, empty, free, print, size, true, and false

2.3 Constants

Numbers, denoted num, are sequences of digits that can contain a single decimal point. The decimal point must be followed by one or more digits.

Characters, denoted char, are single characters enclosed by single quotation marks.

Boolean constants are represented by the keywords true and false. Booleans may take on only these two values and void (and the last only on initialization).

String literals are a series of characters delimited by double quotation marks. Strings cannot be nested, though a double quotation mark can appear inside a string by using the escape sequence \".

3 Types

3.1 Primitive Types

GRAIL has 4 primitive types: boolean, char, num, and void. A boolean is a Boolean value. A char is a single member of the ASCII character set. (More complex character sets are not supported.) A num is an number. The void type is a null type, used in functions that return no variables and uninitialized primitives.

3.2 Derived types

Lists are arrays of primitives or objects of the same type. The type of a list is the type of the first element inserted into a list. If all the elements of a list are removed, it maintains its type.

Strings are arrays of chars. While strings are stored internally as arrays, they act similarly to primitive types, as they can be declared with the keyword string and can be nodes of graphs.

Edges are types consisting of two nodes, which must be primitives or strings of the same type, which is the type of the graph or digraph they belong to, as well as a weight, which is a num.

Graphs are collections of nodes and edges. Each graph has a type, which is either a string or a primitive type, and every node is an object of that type. Internally, the graph is represented as an ordered list of edges, and which can be iterated over and sorted by edge features.

Digraphs are identical to graphs except that the edges in digraphs are directed. That is, while the choice of which node is stored as the source and which is the destination in an edge of a graph is arbitrary, in a digraph it is not.

4 Objects and LValues

An object is a named region of storage; an lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier with suitable type and storage class. The name "lvalue" comes from the assignment expression E1 = E2 in which the left operand E1 must be an lvalue expression.

Edges:

Edges are a built-in object with three features, to, from, and weight, which can be accessed thus:

```
my_edge.weight
```

Examples:

```
num name = 3; //num is an lvalue
graph g = {e, f, g}; //g is an object
```

5 Expressions

Expressions, consisting of type-compatible operators or groups of operators separated by operands, are outlined below in descending order of precedence.

5.1 Subset Operators

di/graph*:

a*[i] yields the node (char, string, or int) stored at index i in di/graph* a.

a*[x : statement] (e.g. a*[x : x == 3]) yields a di/graph* of nodes x that match the condition/s. (The statement must return a boolean.) If no conditions are supplied, yields all nodes.

graph:

g[i] yields the edge stored at index i in di/graph g.

g[x -(</>/-) y: statement] (e.g. [x – y: x == 3 && y > 4]) yields a graph of edges x, y with the appropriate connection (–, ->, or <-), where x and y match the conditions specified. If a condition is left out, yields all nodes that fit the remaining conditions. If all conditions are left out, yields the whole graph. In a digraph, – is equivalent to connections in both directions. For example, the graph of all nodes going to z could be called using g[x -> y: y == z].

5.2 Unary Operators

expr* (di/graph): Refers to the node collection of the graph.

!expr (booleans, boolean expr): Logical not. Yields the opposite value of the expression.

-expr (num:) Numeric negation. Yields the expression multiplied by negative one.

5.3 Numeric Binary Operators

Binary operators group left to right.

expr * expr (num, graph, digraph): Multiplication operator.

expr / expr (num, graph, digraph): Division operator.

	num	graph	digraph
num	Yields product (*) or quotient (/) (num)	NS	NS
graph		Yields a graph with the edges (and nodes) from both graphs. Duplicate edges are multiplied (*) or divided (/)	NS
digraph			Yields a graph with the edges (and nodes) from both graphs. Duplicate edges in the same direction are multiplied (*) or divided (/)

expr + expr (num, di/graph, di/graph*, edge) (commutative): Addition Operator.

	num	di/graph	di/graph*	edge
num	Yields sum			Yields a di/graph* list with the num node added, if not already present
di/graph	Yields a graph with the edges (and nodes) from both (di)graphs. Duplicate edges (in the same direction) are summed		Yields a graph with all the edges and from the di/graph and the edge and its associated nodes. If the edge already exists in the graph, they are summed.	
di/graph*		Yields a di/graph* list containing the nodes in both di/graph*s		
edge				

expr - expr (num, graph, digraph) Weight: Subtraction operator.

	num	di/graph	edge	di/graph*
num	Yields difference			Subtracts a num node and all associated edges from the di/graph list, if a num type di/graph
di/graph		Yields a di/graph with the edges (and nodes) from both di/graphs. In the case of duplicate edges, the edges weights from the second graph are subtracted from the first	Yields a graph with all the edges and from the di/graph and the edge and its associated nodes. If the edge already exists in the graph, the one on the right is subtracted from the one on the left.	Not Supported
digraph		Not Supported		Yields a di/graph* list containing the nodes in the first and not in the second

expr .+ expr (graph, digraph, edge): Left join operator.

	di/graph	edge
digraph	Yields a (di)graph with the edges (and nodes) from both graphs. In the case of duplicate edges (in the same direction), only those in the left are taken.	Yields a (di)graph with the edges (and nodes) from graph and the edge. In the case of duplicate edges (in the same direction), only those on the left are taken.

expr .- expr (graph, digraph): Subtraction Operator.

	di/graph	di/graph - edge
digraph	Yields a (di)graph with the edges (and nodes) from the first graph and without the edges of the second.	Yields a (di)graph with the edges (and nodes) from graph without the edge.

expr & expr (graph, digraph) Intersection Operator: Yields a (di)graph with only the edges (and nodes) that appear in both graphs.

5.4 Relational Operators

expr < expr (num), less than

expr > expr (num), greater than

expr <= expr (num), less than or equal to

expr >= expr (num), greater than or equal to

Return true if the relation is true and false otherwise.

5.5 Equality Operators

expr == expr (num) returns true if the numbers are equal and false otherwise.

expr != expr (num) returns false if the numbers are equal and true otherwise

expr == expr (graph, digraph) returns true if both di/graphs contain the same nodes and edges.

5.6 Logical Operators

expr && expr (boolean) returns true if both booleans or boolean expressions are true and false otherwise.

expr || expr (boolean) returns true if at least one of the booleans or boolean expressions is true and false otherwise.

5.7 Assignment

lvalue = expr replaces the value of the lvalue with the value of the expr if they are the same type.

5.8 Edge Operators

expr –(w) expr yields an edge connecting both expr's (num, char, or string) with optional weight (in optional parens) w. An unset weight defaults to 1.

expr ->(w) expr yields directed edge from the first expr (num, char, or string) to the second with similarly optional weight conventions.

expr <-(w) expr yields directed edge from the second expr (num, char, or string) to the first with similarly optional weight conventions.

6 Declarations

Declarations define and specify the properties of an identifier in a structured format.

6.1 Type specifiers

The type-specifiers in our language are:

- void
- char
- string
- num
- edge
- graph
- digraph
- list
- node

These type specifiers define the type of the variables, parameters, and function return types. Num includes not only integers but also floating point numbers. Void is only allowed in the function return type.

6.2 Object Declarators

Each type has its own declarator, formatted in the following way:

```
num variableName;
char variableName;
string variableName;
edge variableName;
graph variableName;
digraph variableName;
```

The variableName is the identifier of the variable.

6.3 Function Declarators

The Function declarator is formatted in the following way:

```
functionName(param, param, ... )
```

FunctionName is the identifier of the function, followed by a left parenthesis, a list of function parameters (optional), and a right parenthesis. In the function parameter list, these parameters need to be separated by commas.

6.4 Initialization

The general format of the initialization of variables is

```
lvalue = assignment expression / {initializer-list}
```

More specifically, it can be applied to num, edge, and graph:

```
num|edge|graph variableName //initialized to void
num variableName = numValue
num variableName = numValue binaryOperator assignment_expression
edge variableName = (node->node) | (node--node) | (node<-node)
graph variableName = {edge, edge, ... } | {list of edges}
```

The following examples show the declaration and initialization of graphs and nodes:

```
num graph g = {1->2,2->3,3->4};
g* += 5; //Adding an element to a node list
g += 4<-(2.5)5; //Adding weighted edge to a graph
```

7 Statements

Statements execute in sequence. They do not have values and are executed for their effects. The statements in our language are classified in the following groups:

- Assignment statement
- Function-call statement
- Sequence statement
- Control-Flow statement
- Loop Statement
- Jump Statement

7.1 Assignment Statement

Assignment statement is used to assign identifier with the value of the expression. It is formatted in the following format:

```
identifierName = expression;
```

This statement is commonly used for initialization of variables or expressions.

7.2 Function-Call Statement

The function-call statement is used when a defined function is called. It is formatted in the following format. Parameters can be any of the primitive types or objects or expressions that evaluate to those types or objects.

```
type functionName (parameter, parameter, ...);
```

7.3 Sequence Statement

Statements can be written one after another. This is seen as the sequence statement and is formatted in the following format:

```
statement; statement; statement;...
```

7.4 Control-Flow Statement

The control-flow statements use the expression as conditional test to decide which block of statements will get executed. They have the following formats:

```
if (expression) { statement(s) }
if (expression) { statement(s) } else { statement(s) }
if (expression) { statement(s) } else if (expression) { statement(s) } else { statement(s) }
```

7.5 Loop Statement

while loop and for loop are available in the GRAIL in the following format:

```
while (expression) { statement(s) }
for (initialization expression; conditional expression; execution expression) { statement(s) }
```

The while loop takes one expression as the conditional expression to check if the available variables or expressions qualify, which determine if the body statement(s) will be executed or not. The for loop takes three expressions: initialization expression, conditional expression, and execution expression. The initialization expression will be executed when the for loop is initiated. The conditional expression is the test expression to check if the condition is satisfied. Once it is satisfied, the loop terminates. The execution expression will be executed after every time the body statement(s) is executed.

7.6 Jump Statement

break can be used to break the first outer loop if a certain condition is reached.

```
for (expression; expression; expression) {
    ...
    if (expression) {
        break;
    }
    ...
}
```

continue can be used to continue to the next round of the loop if a certain condition is reached.

```
while(expression) {
    ...
    if (expression) {
        continue;
    }
    ...
}
```

8 Scope

A declared entity is a class type, member (class, field, or function) of a reference type, type parameter (of a class, function or constructor), parameter (to a function, constructor), or local variable.

Every declaration that introduces a name has a scope, which is the part of the program text within which the declared entity can be referred to by a simple name.

The scope of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name.

A declaration is said to be in scope at a particular point in a program if and only if the declaration's scope includes that point.

The scope of a formal parameter of a function or constructor is the entire body of the function or constructor.

The scope of a function's type parameter is the entire declaration of the function, including the type parameter section, but excluding the function modifiers.

9 Library Functions

9.1 Print

Prints strings to standard output.

```
print(string)
```

9.2 Display

Function for displaying graphs and digraphs

```
display(graph)
display(digraph)
```

9.3 Sort

Returns a graph with its contents sorted by a feature of the edges, with an optional ascending/descending field. Default is ascending.

```
sort(graph, feature, [asc/desc])
```

9.4 Size

Returns the size of a derived type, for example:

```
size(my_graph) //returns the number of edges in the graph my_graph
size(my_graph*) //returns the number of nodes in the graph my_graph
```

10 Examples

The following program implements Djikstra's Algorithm for finding a shortest path.

```
list getclosestpaths(graph g, num s){
    g = sort(g, weight, asc);
    list dist = [size(g)];
    list prev= [size(g)];
    list visited = [size(g)];

    num inf = Grail.Math.INF;

    for(num i = 0; i < size(dist); i+=1){
        dist[i] = inf;
        visited[i] = false;
    }
    dist[s] = 0;

    for(num i = 0; i < size(g); i+=1){
```

```
        next = closestNode(dist, visited);
        visited[next] = true;
        neighbors = g[x --> y : x == next]; //Get all nodes from x

        for(num j = 0; j < size(neighbors); j+=1){
            num n = neighbors[j];
            num d = neighbors[j].weight + dist[next];

            if(dist[n] > d){
                dist[n] = d;
                prev[n] = next;
            }
        }
    }
return pred;
}

num closestNode(list dist, list visited){
    num d = Grail.Math.INF;
    num n;
    for(num i = 0; i < size(dist); i++){
        if(v[i] == false && dist[i] < d){
            n = i;
            d = dist[i];
        }
    }
return n;
}

//initialize a graph, add a node and an edge, and run closest paths algorithm

graph g = {1->2,2->3,3->4};
g* += 5;
g += 4->5;

list allPaths = getClosestPaths(g,1);
```

11 Recapitulation

GRAIL is intended to be a language for the sorting, searching, and manipulating graphs, nodes, and edges. GRAIL is possessed of several types, denoted by whitespace-separated tokens. Some, such as num and char, are primitive types, while others, like graph and list, are derived types. Graphs are the central types of our language, defined as a collection of nodes and edges, where nodes can be any primitive type or string. GRAIL offers a robust library of operators for merging and subsecting graphs, searching and modifying nodes and edges, and standard logical and assignment operators. Functions and primitives have stated types, derived types require all their elements to be the same as the first type inserted. Control flow is similar to C, with the omission of the pure block scoping. A standard library also includes functions such as print and display.