Grape Language Reference Manual

2018 Fall Programming Languages and Translators

James Kolsby	*jrk2181*
Nick Krasnoff	*nsk2144*
HyunBin Yoo	*hy2506*
Wu Po Yu	*pw2440*

October 13, 2018

1. Introduction

Graph algorithms are an extremely ripe domain for networks and relationships of data. Graph algorithms can be very useful in a wide range of applications, including databases, network flow problems, and even language parsing using finite automata. Grape is a language that is designed to make the assembly and manipulation of graphs much more visually comprehensible and syntactically convenient. It should allow its user to implement programs like the Minimum mean cycle canceling algorithm or Kruskal's algorithm easily and concisely. Quick graph descriptions and pattern searching are among the optimizations of the language.

2. Data Types

2.1. Primitive Types

Int - A 32-bit signed integer designated by a series of digits

Bool - A 1-bit boolean designated by ***True*** or ***False***

Float - An signed double precision flowing number designated by a sign, a decimal and an exponent.

String - A series of characters that are enclosed in double quotes

2.1.1. Examples of Primitives

```
Int i = 1;
Bool b = False;
Float pi = 3.14;
String name = "Stephen";
```

2.2. Reference Types

List - A collection which is ordered and mutable. It is designated by a series comma-delimited expressions enclosed in square brackets, like so:

```
List a = [1,2,3];
List b = ["Hello", "World"];
```

Dict- An unordered and mutable dictionary of values mapped to keys. It does not allow duplicate keys. A dictionary is designated by a comma-delimited set of key value pairs, mapped by colons, enclosed in curly brackets, like so:

```
Dict d = {foo: 1, bar: 10}
```

Dictionary values can be set or accessed using their key, like so:

```
d.city = "Charleston"
```

2.3. Graph Types

Node - A node is a container representing a vertex in a graph, designated by an expression enclosed in single quotes, like so:

```
Node a = '3';
Node b = '"Hello"';
```

Edge - An edge is an object that represents a directed relationship between two nodes, designated by an expression enclosed in hyphens with a closing bracket representing its directionality. As with a node, the expression contained in an edge can be of any type, for instance an integer containing a cost of traversing that edge. For instance: ***`Edge a = -3->;`***

An edge contains references to two nodes, a source and a destination. These nodes can be set or accessed as follows:

```
Node a = '1';
Node b = '42';
Edge c = -3->;
c.to = a;
c.from = b;
```

Graph - A graph is a collection of Nodes and Edges that can be interconnected or disjoint. Graph initialization is designated by a space-delimited path of nodes and edges, enclosed in double angle brackets, like so:

```
Graph x = <<'"Atlanta"' -4-> '"New York"'>>;
```

Graphs can contain any Nodes of any type, and can mix types. More complicated graphs can be described using a comma-delimited series of paths. Reference names can be passed into the graph.

```
Node a = '"Atlanta"';
Graph cities = <<a -5-> '"Charleston"', a -30-> '"New York"',
   a -100-> '"San Francisco"'>>;
```

Graph initialization can be used to describe paths wherein two edges share a common node between them, for instance:

```
Graph path = <<'1' -30-> '2' -40-> '3'>>
```

These paths are evaluated from left to right, where the ***`from`*** of each subsequent Edge is the same as the ***`to`*** of the Edge preceding it. In the above example, the Nodes containing Integers 1 and 3 are both connected to the node containing '2' via the Edges containing 30 and 40 respectively.

Undirected edges can be expressed in the context of a graph using the `--` shorthand. They are evaluated as a pair of directed edges pointing to both of the nodes, like so: *`<<'1' -- '2'>>`*

3. Operators and Expressions

3.1. Variable Assignment

The `=` operator is used for a variable assignment. The right-hand expression is evaluated and its value is assigned to the left-hand typed ID. LHS and RHS must have the same type. This operator will be evaluated right-to-left.

3.2. Arithmetic Operators

The arithmetic operators are `%` (Modulo), `**` (Exponent), `*` (Multiplication), `/` (Division), `+` (Addition) and `-` (Subtraction). They are all binomial operators. The minus sign can also be used as a unary operation to invert a number's sign (Negation).

3.3. Relational Operators

The relational operators are `<` (Less than) `>` (Greater than) `<=` (Less than or equal to) `=>` (Greater than or equal to) `==` (Equal to) `!=` (Not equal to). They are evaluated from left to right. They each require two values which are to be compared and will return ***True*** if the comparison is truthful and ***False*** otherwise.

3.4. Boolean Operators

The logical operators are ***not***, ***and***, and ***or***. Not negates the subsequent boolean, while and and or both return the logical comparison of the values on either side of them, like so:

```
Bool t = True
Bool f = False
Bool yes = t or f
Bool no = not t
```

3.5. Graph Operations

The ***&*** operator returns the graph which is the union of two graphs.

```
Node atl = '{city: "Atlanta", state: "Georgia"}';
Node la = '{city: "Los Angeles", state: "California"}';

Graph southeast = <<atl -305- '"Charleston"',
           atl -381- '"Durham"',
           atl -662- '"Miami"',
           atl -108- '"Auburn"'>>;

Graph southwest = <<la -100- '"San Francisco"'>>;
Graph usa = southeast & southwest & <<la -3000- atl>>;
```

The *in* operator performs a search of a graph given a template. It does not alter the input graph and returns a List of all matches of that template.

```
List cities = /''/ in usa;
List trips = /--/ in usa;
```

3.5.1. Template Searching

Template searching is a key feature of Grape which provides a flexible means of finding structures within a Graph. Templates are very similar to graph constructions, however they represent a pattern against which the graph will be searched. They are enclosed in forward slashes, and can contain references to Node and Edge instances.

The resulting List will contain different types depending on the format of the template string. In the below example, there is only one Node expression in the template, and so the result will be a list of Nodes:

```
each (/''/ in usa) {
      print(this);
}
```

However, if a template contains multiple expressions, for instance an edge and a pair of nodes, we must specify which to include in the resulting List. The wildcard token * is used to omit an Edge or a Node. In this case, the results will also be a List of Nodes:

```
each (/atl -*- ''/ in usa) {
      print(this)
      print("Is near Atlanta!")
}
```

If we wish to obtain multiple items from a matched template, they must be given keys so that the resulting List contains Dictionaries with the results mapped to their identifier keys.

```
each (/'a' -*- atl -*- 'b'/ in usa) {
      print(this.a)
      print(this.b)
      print("Passes through Atlanta")
}
```

3.6. Precedence and Order of Operations

Parentheses have the highest priority in the evaluation of expressions. Logical and relational operators have lower precedence than the arithmetic operators, so statements including that include logical or relational operators alongside arithmetic operations will evaluate the arithmetic statement first and then apply relational and logical operators to them in that order. For instance this statement evaluates to ***True***:

```
Bool yes = 3 > 5 - 2 and 2 + 2 <= 4
```

4. Programming Structure

Grape programs are described as a single source file which contains a series of global statements or function declarations which are evaluated from top to bottom.

4.1. Blocks and Statements

Grape is an imperative programming language and is designed to be written in blocks, a series of statements which are executed top to bottom. Statements within a block are delimited by semicolons, and can span an arbitrary number of lines.

4.2. Comments

Single-line comments are designated by a double forward slash, and are terminated by a new line. Multi-line comments are designated by three forward slashes, and are terminated by another three forward slashes.

```
Int a = 5; // Look 'ma a comment!
///
      Welcome to the COMMENT ZONE!
///
```

4.3. Functions

Functions act as a way to compartmentalize segments of your program. Functions are defined by a return type, an ID, and zero or more comma-delimited parameters enclosed in parentheses. The function body consists of a series of statements that must contain a return statement specifying the value to be returned. A function declaration is designated as follows:

```
      fun return-type function-name(param, param) {body}
```

Here is a Grape implementation of Euclid's Algorithm using a recursive functions

```
      fun Int gcd(Int a, Int b) {
            Int r = a % b;
            if (r == 0) {
                  return b;
            }
```

```
        return gcd(b, r);
}
```

Functions can be called anywhere in the program body or in any function body, including its own. Grape supports recursion. A function call is designated by the function ID and a series of parameters enclosed in parentheses:

```
Int a = gcd(10, 15);
```

5. Control Flow

5.1. Conditionals

Grape supports if statements that may contain an optional else condition to execute if the given condition is false.

```
if (r == 0) { return 3; }
else { return 2; }
```

5.2. Loops

while loops are designated by a looping condition and a block to be executed as long as the condition is truthful. They are designated as follows:

```
while (x < 5) {
        x = x + 1;
}
```

each loops allow the user to iterate over Lists. Within the body of an each loop, there is special local variable called ***this*** which stores the current item in the List.

```
List a = [3,1,4,1,5,9,2,6];
each (a) {
      this = this + 1;
}
```

6. Standard Library

The Grape standard library provides useful built-in methods for manipulating the List, Dict, and Graph types, as well as for standard I/O:

print(a) - writes a to stdout.

6.1. List Methods

l.append(x) - append x to the end of the List and return the modified List.

l.clear() - remove all items from the List and return the modified List

l.copy() - return a deep copy of the List

l.insert(i,x) - insert variable x at index i and return the modified List

l.pop() - remove and return the last element from the List

l.remove(x) - remove the first instance of x and return the modified List

l.reverse() - reverse the entire list, and return the reversed List

l.size() - return the number of elements in the List

6.2. Dict Methods

d.size() - return the number of keys in the Dict

d.key(x) - return the key of the value x, if it exists in the Dict.

d.remove(x) - remove the key x and its value from the Dict

6.3. Graph Methods

g.size() - return the number of Nodes in the Graph

g.root() - return source Node of the Graph

g.leaves() - return a list of Nodes with only one incoming Edge

g.neighbors(x) - return a List of Nodes adjacent to Node x

g.find(x) - return the list of all Nodes that contain the value x

g.empty() - return ***True*** of the Graph is empty, otherwise return ***False***

g.switch(x,y) - switch Nodes x and y in a Graph