nodable Programming Language Proposal
February 2021

Karen Shi	(ks3650@columbia.edu)	**Manager**
Elena Sotolongo	(es3693@columbia.edu)	**Language Guru**
Ajita Bala	(ab4420@columbia.edu)	**System Architect**
Ariel Goldman	(apg2164@barnard.edu)	**System Architect**
Naviya Makhija	(nm3076@columbia.edu)	**Tester**

1. Overview

nodable is an imperative, statically-typed graph programming language designed to help users create, use, manipulate, and search graphs. The goal is to simplify the implementation of commonly used graph algorithms by eliminating the need for lower-level data structures to represent graphs. Instead, *nodable* will feature built-in data types for **graphs** and **edges**, and allow for user-defined nodes. Similarly to structs in C, users of the language will be able to store any type of data in a node and customize and modify the information held within it. We anticipate that the language will be able to meet the needs of programmers who routinely use graphs to represent data and networks.

nodable syntax will contain elements of C, Java, and Python. For example, the way a user builds a type of node is modeled on how you build structs in C, the notation of calling functions on nodes is inspired by Java, and the structure of a 'for loop' is inspired by a combination of Java and Python.

2. Language Details

2.1 Data Types

Type	**Description / notation**
`graph`	A set of nodes (nodeset) and a set of edges (edgeset) that connect pairs of nodes
`node`	Similar to struct in C, contains attributes that uniquely identify the

	node. Users define their own nodes, so the size is customizable
`edge`	Similar to struct in C, contains information on the source and destination nodes and whether it is directed or undirected
`int`	4 bytes, integer value
`double`	8 bytes, decimal value
`char`	1 byte, character value
`float`	4 bytes, floating point value
`string`	An array of characters
`boolean`	1 byte, True or False
`list`	A resizable array with a built-in functionality, including add(), remove(), and contains(); can support any data type `list<int> example = [4, 6, 2, 9, 0];`
`dict`	A collection of unordered and unique key-value pairs, can support any data type `dict<city, int> values = {[key1:value1], [key2:value2]};`

2.2 Operators and Syntax

Operator	**Description**
`=`	Simple assignment operator. Assigns the value of the right operand to the left operand
`+`	Adds two operands
`-`	Subtracts the second operand from the first
`*`	Multiplies operands
`**`	Exponents
`/`	Divides first operand by second operand

`==`	Checks if the values of two operands are equal. Returns True if yes, False if not
`!=`	Checks if values of two operands are not equal. Returns True if they are not equal, False if they are
`>`	Checks if the value of the left operand is greater than the value of the right operand. Returns True if this condition is met
`<`	Checks if the value of the left operand is less than the value of right operand. Returns True if this condition is met
`>=`	Checks if the value of the left operand is greater than or equal to the value of the right operand. Returns True if this condition is met
`<=`	Checks if the value of the left operand is less than or equal to the value of right operand. Returns True if this condition is met
`++`	Increment operator; increases integer value of operand by 1
`--`	Decrement operator; decreases integer value of operand by 1
`+=`	Add AND operator. Adds the value of the right operand to the value of the left operand and assigns the result to the left operand
`-=`	Subtract AND operator. Subtracts the value of the right operand from the value of the left operand and assigns the result to the left operand
`&&`	Logical AND operator. Returns True if both of the operands are true/non-zero
`||`	Logical OR operator. Returns True if either of the operands are true/non-zero
`!`	Logical NOT operator. Reverses logical state of given operand; if the operand is logically true, the NOT operator will make it return False
`<->`	Undirected/bidirectional edge (default weight of 1)
`->`	Directed edge (default weight of 1)
`<-n->`	Undirected/bidirectional edge with weight n (n is an int)
`-n->`	Directed edge with weight n (n is an int)

2.3 Keywords/Reserved words

The reserved words in *nodable* are based on our source languages as well as graph theory.

They include:

- `while, for, in, if, else, elif, continue, break, range`
- `def, main, return`
- `and, or, not`
- `graph, node, edge`
- `int, double, char, float, string, boolean, True, False`
- `list, dict, len`
- `new, const`

2.4 Control Flow

IF/ELSE Statements: Established based on Python and Java syntax to allow users to selectively execute statements.

```
if(temp >= 90)) {
      weather = “hot”;
}
elif (temp > 40 && temp <= 90) {
      weather = “nice”;
}
else {
      weather = “cold”;
}
```

WHILE Loops: As based in Python and Java, this loop executes the inner code if the boolean expression is satisfied. The loop terminates once this condition is false.

```
int i = 5;
while (i > 0) {
      print(i);
      i--;
}
```

FOR Loops: For loops allow iteration through lists and other data structures as in the example below so that operations can be performed on that data.

```
for v in d.values(){
      if (len(value) == 0) {
            count = count + 1;
      }
}
```

A generic FOR loop might look like:

```
for i in range(1, 4){
      if (arr[i]== 0){
            count = count + 1;
      }
}
```

2.5 Graphs

Graphs are an inbuilt data type in *nodable*. They contain a nodeset (a **dict** of node objects), an edgeset (a **list** of edge objects), and a **boolean** ‘directed’ that determines the types of edges that can be contained in the graph’s edgeset. All nodes in a graph must be of the same user-defined type (for example, one graph can’t contain nodes of type ‘city’ and nodes of type ‘person’).

- The *nodeset* of a graph must be a **dict** where the key is of type **string** that represents the name/ID of the node (unique for each item in the dict) and the value is a node of a certain type already created by the user.
- The *edgeset* of a graph is a list of type **edge.** All of the edges added to this edgeset must correspond to the type of graph (directed or undirected/bidirectional). An error will occur if the user attempts to add a conflicting type of edge.

```
graph<> empty = {nodeset={}, edgeset = [], directed = true};
graph<city> cities = {nodeset={[‘new york’:nyc],[‘boston’:boston]},
edgeset = [], directed = false};
```

2.6 Nodes/Structs

Nodes are modeled on C structs and are created by the user such that they can store various data types and can be customized to the user's and graph's needs. The sample code below creates a struct known as 'city'.
It has three properties: a name (String), population (int), and state (String) of varying data types, but all of which will uniquely identify each node.

```
node city {
      String name;
      int population;
      String state;
};
```

The code below creates a sample city node. Here the city is named 'nyc', it has a population of 8000000 and has a state 'NY'.

```
city nyc = ("nyc", 8000000, "NY");
```

Node equality can be compared using == and !=.

2.7 Edges

Edges are connecting objects between two node objects. Graphs can either be directed, where edges display a one-way relationship between nodes, or undirected where edges are bidirectional and can be traversed both ways. All edges have weights where the default weight, if not specified by the user, is 1 (a graph with all edges having weight 1 works as an unweighted graph). The sample code below shows the creation of an edge between two city nodes (boston and new york) that already exist in the nodeset. It is a bidirectional edge with a weight of 8.

```
g.edgeset.add(g.nodeset['new york'] <-8-> g.nodeset['boston']);
```

The example below shows another edge being created between two city nodes that is unidirectional and has a weight of 6. Note that the source node must be first and the target second.

```
g.edgeset.add(g.nodeset['new york'] -6-> g.nodeset['philadelphia']);
//allowed
```

```
g.edgeset.add(g.nodeset['new york'] <-6- g.nodeset['philadelphia']);
//not allowed
```

Edges can be accessed later on using the syntax `g.edgeset.get(node1, node2)`. This will cause an error if there is no currently existing edge between node1 and node2. Users can also access the edge weight, source, and target nodes:

```
int w = g.edgeset.get(node1, node2).weight;
node s = g.edgeset.get(node1, node2).source;
node t = g.edgeset.get(node1, node2).target;
setWeight(g, g.edgeset.get(node1, node2), 5);
```

2.8 Functions

nodable declares functions in a similar way to the standard Python style, but uses curly braces to determine scope.

nodable also requires the usage of semicolons at the end of each statement to reduce confusion and ambiguity.

```
def add(x, y) -> int{
      return x + y;
}
```

Functions have to be written above the main function where they're called

```
def main(){
      add(2,7);
}
```

The function signature can specify the data types of the parameter list, as well as the return type.

```
def count_empty(d: dict) -> int{
      count = 0;
      for v in d.values(){
            if len(value) == 0
                  count = count + 1;
      }
      return count;
}
```

2.9 Comments

- Single-line comments use `//`
- Multi-line comments use `/* */`

```
// this is a single-line comment
/* this is a
   multi-line
   comment
*/
```

3. Library Functions

3.1 Graph functions

```
def removeEdge(graph g, edge e) //deletes an edge from a given graph
def removeNode (graph g, node n) //deletes a node and all associated
edges from a given graph
def getChildren(graph g, node n) -> list<node> //returns the
children of a node in a directed graph. A node p is a child of n if
there is a directed edge from n to p, and no edge from p to n.
def getDegree(graph g, node n)->int //returns the number of neighbor
nodes for a given node in a graph
def getNeighbors(graph g, node n)->list<node> //returns the
neighbors of a node in an undirected/bidirectional graph
def getNodes(graph g) -> dict<string, node> //returns a dict of all
of the nodes in a graph with their node names
def getNodeNames(graph g) -> list<string> //returns a list of the
names of all the nodes in a graph
def getEdges(graph g) -> list<edge> //returns a list of the edges
def printGraph(graph g) //prints the nodeset and edgeset of g
def getNumNodes(graph g)-> int //returns number of nodes
def getNumEdges(graph g)-> int //bidirectional edges count as two
def getNumUndirEdges(graph g) -> int //returns number of edges in an
undirected graph, divided by 2
```

3.2 Node functions

```
def getData (node n) -> dict<str, ___>
//returns a dict with all of the data of the current node
```

3.3 String functions

```
String.equals(string s) -> boolean
String.length -> int
```

4. Examples

4.1 Hello World

```
print(“Hello world!”);
```

4.2 Lists

```
//initialize list
list<int> example = [4, 6, 2, 9, 0];
print(example); //print out elements of list

for int i in example { //iterate through list
      print(i);
}

for int i in range(len(example)) { //using regular for loop
      print(i);
}

example.append(10); //appends to end
example.add(3, 10); //inserts int 10 to the 3rd index

example.get(3); //returns element at 3rd index

example.replace(3, 10); //replaces element at index 3 with 10

example.remove(3); //removes element at index 3

len(example); //returns length of list

example.contains(10); //returns boolean value
```

4.3 Dicts

```
dict<string, int> apples = {[‘Bob’: 3], [‘Alice’: 10]};
apples.add([‘Eve’: 30]); //adds key, value pair to list
apples.contains(‘Eve’); //returns boolean value
apples.replace([‘Eve’: 13]);
apples.get(‘Eve’);
apples.values(); //returns a list<int> of all values
apples.keys(); //returns a list<string> of all keys
len(apples); //returns int
```

4.4 Finding neighboring nodes in a graph

```
def getNeighbors(graph g, node src)-> list<node> {
      list<node> neighbors = [];
      for edge e in g.edgeset {
            if (getSource(e) == src) {
                  neighbors.add(getTarget(e));
            }
      }
      return neighbors;
}
```

4.5 Creating a graph, nodes, and edges

```
//Defines a node type called ‘city’
node city {
      String name;
      int population;
      String state;
};

//creates instances of ‘city’
city nyc = (“nyc”, 8000, “NY”);
city boston = (“boston”, 500, “MA”);
city philly = (“philadelphia”, 800, “PA”);

//creates graph g with nodeset containing instances of city

graph<city> g = {nodeset={[‘new york’:nyc],[‘boston’:boston]},
directed = true}; //edgeset initialized to empty list


g.nodeset.add([‘philadelphia’:philly]); //adds item to nodeset dict


g.edgeset.add(g.nodeset[‘new york’] <-8-> g.nodeset[‘boston’]);
//when working with edges and graphs, need to refer to nodes in the
context of the nodeset

g.edgeset.add(g.nodeset[‘new york’] -6-> g.nodeset[‘philadelphia’]);


g.edgeset.add(g.nodeset[‘philadelphia’] -6-> g.nodeset[‘new york’]);
```

4.6 Dijkstra's algorithm

```
//continuing code from 4.5
dict<city, int> values;
dict<city,city> prior;
list<city> unvisited;
for city c in g.nodeset
{
      values.add([c, Integer.max()]);
      unvisited.add(c);
}

city cur = g.nodeset['philadelphia']; //starting in philadelphia

while (cur != g.nodeset['new york'])
{
      unvisited.remove(cur);
      i++;
      for c in getNeighbors(g, cur)
      {
            if(unvisited.contains(c))
            {
                  if (values.get(cur) +
                  getWeight(edgeset.get(cur, c) < values.get(c))
                  {
                        values.get(c) = values.get(cur) +
                        getWeight(edgeset.get(cur,c);
                        prior.replace([c:cur]);
                  }
            }
      }
      int lowVal = Integer.max();
      city lowCity; //all vars are null/empty string
      for c in unvisited
      {
            if (val[c] < lowVal)
            {
                  lowVal = c;
            }
      }
      cur = lowVal;
}

list<city> path;
cur = g.nodeset['new york'];
```

```
while cur !=  g.nodeset[‘philadelphia’]
{
      path.appendFront(cur);
      cur = prior[cur];
}
path.appendFront(g.nodeset[‘philadelphia’]);
return path;
```