nodable Language Reference Manual

Karen Shi
Manager
ks3650@columbia.edu

Elena Sotolongo
Language Guru
es3693@columbia.edu

Ajita Bala
System Architect
ab4420@columbia.edu

Ariel Goldman
System Architect
apg2164@barnard.edu

Naviya Makhija
Tester
nm3076@columbia.edu

February 2021

Contents

1 Introduction

nodable is an imperative, statically-typed graph programming language designed to help users create, use, manipulate, and search graphs. Graphs are essential for data representation in computer science, but the most popular programming languages require users to implement graphs with low-level data structures. *nodable*'s aim is to simplify the implementation of graphs and their algorithms by featuring built-in data types for graphs, trees, and edges. Users will be able to define node types and attributes themselves. Several graph algorithms, including various shortest-path algorithms and search algorithms, will be part of *nodable*'s standard library. *nodable*'s syntax contains elements of the Java and C programming languages.

2 Lexical Conventions

2.1 Character Set

Our language's character set is **`a-zA-Z0-9,-;<>*+/=!&|{}()[]_`**

2.2 Comments

Single line comments are denoted by **`//`**

```
// this is a single line comment
```

Multi-line comments are denoted by **`/* */`**

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

2.3 Identifier (Names)

An identifier in our language must begin with an alphabetic letter (**`a-zA-Z`**), is made up only of the characters **`a-zA-Z0-9`**, and is a arbitrarily long string.

2.4 Keywords

These keywords are reserved words that should not be used as identifiers:

```
int float char string boolean true false if else while for node graph
    tree bintree eM
```

2.5 Constants

There are four types of constants in our language **`int`**, **`float`**, **`char`**, and **`boolean`**.

2.6 Elementary Operations and Spacing

Each expression is delimited by a semicolon and sets of expressions are marked by curly braces {}. Whitespace and indentation do not play a role in the flow of the code.

3 Syntax Notation

Syntactic categories/non-terminals will be written in *italics*. Literal words will be written in **`courier new`**.

4 Objects and Types

In *nodable*, object types have been created for graph objects along with the fundamental object types to allow for easy manipulation of graphs. All types below are valid types if they are elementary types or made up of another valid type.

4.1 Fundamental Objects

nodable has the following elementary types:

Type	**Description**
`int`	4 bytes, integer value
`char`	8 bytes, decimal value
`float`	1 byte, character value
`boolean`	1 byte, True or False
`string`	An array of characters

4.2 Lists

Lists in *nodable* are derived from Java's Arrays and Python's lists. They are ordered collections of elements of the same type. A list declaration requires the user to specify the type of list being created from the types in Sections 4.1 and 4.3. Lists are mutable and do not have a fixed size so elements can be added (at the end or at an index), removed or replaced.

Lists are indexed beginning at 0 and elements can be retrieved, altered or added using the below functions on a specified index.

4.3 Graph Objects

nodable also has a series of graph-specific object types as follows:

Type	Description
graph	A set of nodes (nodeset) and an adjacency array of edges (eM) that connect pairs of nodes
tree	A set of nodes (nodeset) and an adjacency array of edges (eM) that connect pairs of nodes with the condition that two vertices are connected by *exactly* one path
bintree	A set of nodes (nodeset) and an adjacency array of edges that connect pairs of nodes with the condition that a node has at most two children
node	Similar to a struct in C, nodes contain attributes that uniquely identify the node. Users are able to define their own nodes so the size and attributes are customizable. Nodes must have at least one attribute.

These are all valid types because they are comprised of the fundamental types which are themselves valid.

5 Conversions

5.1 Integer and Floating

Integers and floating point numbers can be converted between each other by creating a variable of the desired data type and assigning it to the variable of the opposing data type that is wished to be converted.

To convert an integer to a float:

```
int i = 2;
float f = i;
// f = 2.0
```

To convert a float to an integer, the decimal portion gets truncated:

```
float f = 2.0;
int i = f;
// i = 2
```

5.2 Arithmetic Conversions

Any basic arithmetic operation must involve two operands of the same type. It is possible to perform operations on ints and floats if one of the operands is cast to another type so that the types match.

For example:

```
int i = 2;
float f = 2.5;
// i + f will not compile
// i * f will not compile
```

These code samples will compile. Note that when casting a float to an int, the number will round down.

```
int i = 2;
float f = 2.5;
// i + (int) f = 4
// (float) i * f = 5.0
```

5.3 String Conversions

To concatenate string and another data type, the other type must be explicitly cast to a string.

6 Expressions

6.1 Operator Precedence and Association

In *nodable*, operations of higher precedence are always evaluated before operations of lower precedence. Multiplicative operators (multiplication, division, and modulus) have higher precedence than additive operators (addition and subtraction), which has higher precedence than relational, logical, and assignment operators. Addition, subtraction, multiplication, division, and modulus are left-associative.

6.2 Literals

Literals represent fixed boolean, numeric, string, and character data and are represented directly in the code. Literals can be used in arguments to functions.

6.3 Primary Expressions

Primary expressions include identifiers, constants, and parenthesized expressions.

primary-expression :
 identifier
 constant
 (expression)

6.3.1 Identifiers

Identifiers are names - unique within their own scope - given to refer to data types or functions. Reserved key words/names are not eligible to be used as identifiers.

6.3.2 Constants

Constants are strings, characters, numbers (integer, float), and boolean constants.

6.3.3 Parenthesized Expressions

These are simply expressions surrounded by parentheses, and this implies precedence over other arithmetic expressions.

6.4 Unary Operators

Unary operators require only one operand and are grouped from right to left and include incrementation/decrementation operators and logical negation.

unary-expression :
 `not` *unary-expression*

6.5 Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and on literals. These operators are grouped left to right.

arithmetic-expression :
 *expr * expr*
 expr / expr
 expr % expr
 expr + expr
 expr - expr

Type	**Description**	**Example**
`+`	Addition, left-associative	`a + b;`
`-`	Subtraction, left-associative	`a - b;`
`*`	Multiplication, left-associative	`a * b;`
`/`	Division, left-associative	`a / b;`
`%`	Modulus, left-associative	`a % b;`

Each of these operators can be performed on integers and floating point expressions. Additionally, the + operator can be used for string concatenation. For the operators % and /, if the right operand is 0 the result is undefined.

6.6 Assignment Operators

Assignment operators are used to assign values to variables. The only assignment operator in *nodable* is =. The value to the right of the operator is assigned to the variable on the left of the operator.

assignment-expression :
  unary-expr = *expr*

Type	**Description**	**Example**
`=`	Assignment, right-associative	`int a = 5;`

6.7 Relational Operators

Relational operators are used to check the relationship between two operands (literals or variables). The type on each side of these operators should be either an integer or float. All relational operators return a boolean value **true** or **false**.

relational-expression :
  expr > *expr*
  expr < *expr*
  expr >= *expr*
  expr <= *expr*

Type	**Description**	**Example**
`>`	Greater than, left-associative	`3 > 1;`
`<`	Less than, left-associative	`3 < 5;`
$\geq$	Greater than or equal to, left-associative	**1** $\geq$ **1;**
$\leq$	Less than or equal to, left-associative	**3** $\leq$ **5;**

6.8 Logical Operators

Logical operators are used to assert whether multiple values or expressions, or the negation of a value or expression, are true. Logical operators are used with boolean values, and return a boolean value (**true** or **false**).

logical-AND-expression :
expr && *expr*

logical-OR-expression :
expr || *expr*

Type	**Description**	**Example**
`&&`	logical AND, non-associative	`a && b;`
`||`	logical OR, non-associative	`a || b;`

7 Declarations

Declarations specify the interpretation (value and type) given to a set of identifiers.

7.1 Type Specifiers

nodable has 5 primary type specifiers:

typ:
```
  int
  float
  char
  string
  boolean
```

nodable has one type-specifier that requires a specific element type (either primitive or a user-defined node):

```
list
```

nodable also has 4 graph-related type specifiers that require element types:
```
  graph
  tree
  bintree
  node
```

7.2 List Declarations

A list, as described in section 4.2, is a mutable ordered collection of elements of the same type. Lists must be declared with a specific element type.

list <*typ* >*identifier*;

Lists can be initialized as an empty or with values.

list <*typ* >*identifier* = []; **list** <*typ* >*identifier* = [*expr_list*];

7.3 Node Declarations

Nodes are modeled on C structs and are created and defined by the user. Nodes can store various data types and values and can be customized to the user's and graph's needs. Before declaring a node, a specific node type must be defined with a node prototype. The node prototype includes a list of attributes associated with that node and their data types (int, char, string, float, or boolean). Nodes must contain at least one attribute. Attributes cannot be of type graph, tree, or bintree.

node *identifier* {*vdecl_list*};

The sample code below creates a node 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;
}
```

To create a particular instance of a node, the node type must be specified in the construction as well as the values to all of the attributes of that node type.

nodetype identifier = (expr-list);

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 = ("New York City", 8000000, "NY");
```

Node equality can be compared using `==` and `!=`. This equality checks if the two nodes are located in the same place in memory, not if they store the same content.

7.4 Graph Declarations

Graph declarations in *nodable* act the same as declarations of other basic types. Graphs must have a size (number of nodes) specified when they are instantiated. Graphs and trees can be declared in the same way once a type of node has been created by the user.

`graph` *identifier* = **`graph`** <*nodetype* >(*int-literal*);

Creating a graph will create an empty list of nodes of the specified nodetype and length. It will also create a square adjacency edge matrix with length equal to the number of nodes in the graph. The matrix will be initialized with null values.

To modify the nodes contained in the graph by their index use this context free grammar:

identifier **`[`** *int-literal* **`]`** **`=`** *typ-instance* **`;`**

```
Graph<city> cityGraph = Graph<city>(3); //creates a graph with room
                                        //for three nodes
cityGraph[0] = ("nyc", 8000, "NY");
cityGraph[1] = ("boston", 500, "MA");
cityGraph [2]= ("philadelphia", 800, "PA");
```

The nodes are stored as a list where the ID of each node is their index in this list. They can be accessed by indexing the Graph object which will return all the data of that node struct as a tuple.

7.4.1 Edge Matrix Declaration

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.

The edges in a graph are represented by an edge matrix (eM), which is essentially a 2D array that is immutable in size. When a graph is declared with n nodes, a corresponding n×n edge matrix is generated. Edges whose weights have not been declared are automatically set to **`null`** in the edge matrix.

The sample code below shows the edge matrix for the Graph cityGraph declared earlier. Here, we can see that there are no loops since the edge weights are set to 0 along the diagonals of the matrix. The edge between the "nyc" node (index 0) and the "boston" node (index 1) has a weight of 8, and the edge between the "nyc" node (index 0) and the "philadelphia" node (index 2) has a weight of 6. There has not been a weight declared for the edge between the "boston" node (index 1) and the "philadelphia" node (index 2) so it still remains as the default **null** value.

```
cityGraph.eM = [[0, 8, 6],
           [8, 0, null],
             [6, null, 0]]; //eM stands for edge matrix
```

Notice that this follows the CFG format:

id **.eM =** [*list-of-list-of-expr*]

Edge values in the edge matrix can be accessed and subsequently modified by simply using array indexing.

```
cityGraph.eM[1][0] = 4; //changes the distance from boston to nyc to 4
```

Notice that this follows the CFG format:

id **[** *int-literal* **] [** *int-literal* **] =** *int-literal*

If a node is removed from the graph, the edge matrix will set its values in the corresponding row and column of the index of the removed node back to **null**. The size of the edge matrix will not change.

```
cityGraph.removeNode(0); // remove the "nyc" node in index 0

cityGraph.eM = [[null, null, null],
           [null, 0, null],
             [null, null, 0]];
             // values in row 0 and column 0 set to null
```

7.5 Tree Declarations

nodable trees are declared and implemented in the same way as graphs. However, there are changes and restrictions on the edge matrix to account for their specific conditions. Edge matrices for trees consist of booleans, not ints, where the boolean alue represents whether there is a parent-child relationship between two nodes or not.

tree *identifier* = **tree** <*nodetype* >(*int-literal*);

This creates a list of nodes of the given nodetype as well as an empty adjacency matrix. When nodes are added to the node list, users will have to specify the structure of the tree by updating the appropriate entries in the edge matrix.

The compiler will prevent certain edges from being added if they violate the definition of the tree.

The code below declares a tree with *city* nodes as well:

```
Tree cityTree = Tree<City>(2);
```

7.6 Binary Tree Declarations

nodable binary trees **`bintrees`** are a specialized type of tree. When a binary tree is initialized, a node list and adjacency matrix are created. Unlike in graphs and trees, the adjacency matrix of a binary tree will be automatically populated upon initialization.

`bintree` *identifier* = <*nodetype* >(*int-literal*);

This creates a list of nodes of the given nodetype as well as an empty adjacency matrix. When nodes are added to the node list, users will have to specify the structure of the tree by updating the appropriate entries in the edge matrix. The compiler will prevent certain edges from being added if they violate the definition of the tree.

```
bintree cityTree = new bintree<City>(2);
```

7.7 Function Declarations

nodable declares functions in a similar way to the standard Python style, but uses curly braces to determine scope. The keyword **`def`** must be used at the beginning of all function definitions.

`def` *identifier* (*args-list*) ->*type-specifier*$_{opt}$ { *function definition* };

The args-list is a list of arguments inside of parentheses. This list must include type specifiers and an identifier for each argument. An argument can be of any type, including graph types.

The function definition can include a list of statements and expressions of unlimited length. If the function has a return type, the definition must contain a return statement with a value indicated by the type-specifier. Otherwise, if there is no return type, the function does not require a return statement.

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

Here are some written examples of function definitions:

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

Functions must be written above the main function when they're called:

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

The function signature must specify the data types of the parameter list, as well as the return type:

```
int count_empty(list<int> values){
    count = 0;
    for v in values{
        if (v == 0) {
            count = count + 1;
        }
        return count;
    }
}
```

8 Statements

Statements in *nodable* are executed in sequence with a series of statements acting as a `list` of statements.

Statements can be expression statements, conditional statements, and iteration statements. All statements are delimited by semicolons.

stmt:

  expr;
  return *expr*;
  { *stmt-list*; }
  if (*expr*) *stmt elif-list* **else** *stmt*;
  if (*expr*) *stmt elif-list*;
  while (*expr*) *stmt*;
  for (*expr*$_{opt}$; *expr*; *expr*) *stmt*;

8.1 Expression Statements

An expression statement is the most basic form of statement. These include assignments, declarations, operations, and function calls.

Here are some examples:

```
int x = 5;
x = x + 1;
int y = add(4, 5) + add(9, 10);
```

8.2 Iterative Statements

Iterative statements in *nodable* include while loops and for loops.

8.2.1 While Loop

Based on Java syntax, this loop executes the inner statements if the outer boolean expression is satisfied. It terminates once the condition is false.

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

8.2.2 For Loop

Based on Java syntax, For Loops allow iteration through lists and other data structures/objects so that expressions can be executed on that data. *nodable* has two types of for loops. There is a loop that iterates through the data structure:

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

nodable also has a generic for loop that allows iteration through a set of integers as a counter:

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

8.3 Conditional Statements

These are conditional statements established based on Java syntax to allow users to selectively execute expression and other statements. *nodable* includes if statements (with no else), if-elif-else statements, and if-else statements.

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

In the above, if the first condition (temp >= 90) evaluates to be **True** then the first statement (weather = "hot") executes. Else it evaluates another condition (temp > 40 temp <= 90). If this is **True**, it executes the second statement (weather = "nice"). Finally, if neither condition 1 nor condition 2 are true, the third statement (weather = "cold") is executed.

9 Scope

nodable uses curly brackets {} to determine scope (functions, conditional statements, and iterative statements). Any variable declared within a particular code block cannot be referenced outside of it, and any variable declared outside of code blocks are regarded as global variables and can be used anywhere in the program.

10 Library Functions

10.1 General Library Functions

nodable will provide a number of library functions related to graphs and trees that can perform standard and commonly-used graph and tree operations. A list of proposed standard library functions is provided here:

```
print(any_type t)
length()
```

10.2 Graph Library Functions

```
int capacity(graph g) //returns an int representing the total number of
    nodes that graph g can hold
int size(graph g) //returns an int representing the total number of
    non-null nodes in graph g
boolean expandCapacity(graph g, int newSize) //copies the nodes and edge
    matrix of g to a new graph with a larger capacity
boolean existsPath(graph g, node n1, node n2)
//returns true if there's a path connecting n1 and n2 (a valid sequence
    of edges), and returns false otherwise
list<int> shortestPath(graph g, node n1, node n2)
//returns a list of ints representing the indices of the nodes
    connecting node n1 and node2. returns null if there is no existing
    path between the nodes. list<list<int>> shortestToAll(graph g, node
    n1) //returns a list of list of ints representing the shortest
    paths to all other nodes in the graph.
boolean removeNode(graph g, int index)
//this function will set the appropriate element in the nodeset of g
    equal to null. All edges in the edge matrix that involve that node
    will be set to null.
//format: removeNode(g, 0)
int getDegree(graph g, node n) //returns the number of neighbors of node
    n
list<int> getChildren(graph g, node n) //returns a list of integers
    representing the indices of the child nodes of n (assuming g is
    directed).
list<int> getNeighbors(graph g, node n) //returns a list of integers
    representing the indices of the adjacent nodes to n.
g.bfs(node s, node g) //returns a list of ints, representing the index
    numbers of the nodes of the graph accessed from the start node to
    the goal node using breadth first traversal
g.dfs(node s, node g) //returns a list of ints, representing the index
    numbers of the nodes of the graph accessed from the start node to
    the goal node using depth first traversal
boolean containsCycle(graph g) //returns a boolean value indicating if
    there is a cycle in graph g or not
tree<nodetype> mst(graph g<nodetype>) //returns a tree object
    representing the minimum spanning tree of graph g using Kruskal's
    algorithm
boolean kColoring(graph g, int k) //returns a boolean value - true if g
    can be colored with at most k colors, and false otherwise.
list<list<int>>shortestPaths(graph g) //returns a list of lists of ints
    representing the shortest distances between every pair of vertices.
    Implemented using the Floyd Warshall algorithm
boolean isStronglyConnected(graph g) //returns a boolean value - true if
    the graph is strongly connected, false otherwise.
```

10.3 Tree Library Functions

```
int capacity(tree t) //returns an int representing the total number of
    nodes that tree t can hold
int size(tree t) //returns an int representing the total number of
    non-null nodes in tree t
boolean removeNode(tree t, int index) //will set the indicated node and
    all of its children equal to null list<int> preorder(tree t)
    //returns a list of nodes in tree t, with the root node first
    followed by the nodes in the left subtree and the right subtree
    (recursively generated)
list<int> inorder(tree t) //returns a list of nodes in tree t. Traverses
    the left subtree, the root node, then the right subtree.
list<int> postorder(tree t) //returns a list of nodes in tree t by
    recursively traversing the left and right subtrees followed by the
    root node.
int height(tree t) //returns an integer representing the height of the
    tree
boolean isEmpty(tree t) //returns whether or not a tree is empty (i.e.
    it's nodeset only contains null values)
int branchingFactor(tree t) //returns an int representing the average
    branching factor of a tree (i.e. the average number of children)
int numberOfChildren(tree t, int nodeIndex) //returns, as an int, the
    number of children of the node with the inputted index in the graph
boolean isChild(tree t, int parentIndex, int childIndex) //returns, as a
    boolean, whether the node at the childIndex is the child node of
    the node at the parentIndex
```

10.4 Binary Tree Library Functions

These functions can be run on trees that are confirmed by the compiler to be binary trees.

```
int capacity(bintree b) //returns an int representing the total number
    of nodes that tree t can hold
int size(bintree b) //returns an int representing the total number of
    non-null nodes in tree t
removeNode(bintree b, int index) //will set the indicated node and all
    of its children equal to null in the tree's nodeset.
list<int>preorder (bintree b) //returns a list of nodes in tree t, with
    the root node first followed by the nodes in the left subtree and
    the right subtree (recursively generated)
list<int> inorder(bintree b) //returns a list of nodes in tree t.
    Traverses the left subtree, the root node, then the right subtree.
list<int> postorder(bintree b) //returns a list of nodes in tree t by
    recursively traversing the left and right subtrees followed by the
    root node.
int height(bintree b) //returns an integer representing the height of
```

```
    the tree
boolean isEmpty(bintree b) //returns whether or not a tree is empty
    (i.e. it's nodeset only contains null values)
int branchingFactor(bintree b) //returns an int representing the average
    branching factor of a tree (i.e. the average number of children)
bintree leftRotate(bintree b) //returns a left rotated version of the
    tree, but does not change the tree it is acting on, this function
    treats the tree as a binary search tree
bintree rightRotate(bintree b) //returns a right rotated version of the
    tree, but does not change the tree it is acting on, this function
    treats the tree like a binary search tree
bintree balance(bintree b) //returns a balanced version of the tree, but
    does not change the tree it is acting on, this fuction treats the
    tree like a binary search tree
int numberOfChildren(int nodeIndex, bintree b) //returns the number of
    children of the node with the inputted index in the graph
\newpage
```

11 Examples

11.1 Hello world

```
print("Hello world!");
```

11.2 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
```

11.3 Creating a graph, nodes and edges

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

Graph cityGraph = Graph<City>(3); //creates a graph with room
                                   //for three nodes
cityGraph[0] = ("nyc", 8000, "NY");
cityGraph[1] = ("boston", 500, "MA");
cityGraph [2]= ("philadelphia", 800, "PA");

cityGraph.eM = [[0, 8, 6],
            [8, 0, null],
              [6, null, 0]];

cityGraph.eM[1][0] = 4; //changes the distance from boston to nyc to 4
```

11.4 Finding neighboring nodes in a graph

```
//cityGraph created in last example

//find the neighbors of node cityGraph[0], and return them in a list of
    ints representing their indices in the list cityGraph

list<int> neighbors = [];
int numNodes = size(cityGraph);

for (int i = 0; i < numNodes; i = i + 1) {
    if (cityGraph.eM[0][i] != null) {
        neighbors.append(i);
    }
}
print(i);
//expected result: [1, 2]
```