nsbl

Language Tutorial

Professor Alfred V. Aho

Team 19

Project Manager	Chantal Galvez	cg2486@columbia.edu
System integrator	Kunal Mishra	ksm2135@columbia.edu
System Architect	Lixing Pan	lp2441@columbia.edu
Language Guru	Jing Zhang	jz2300@columbia.edu

1. A tutorial Introduction

This is meant to be a simple introduction to NSBL Language. It tries to concentrate on the basics of the NSBL language: how to declare graphs, vertices, edges and lists; show control flow, the use of arithmetic variables and expressions and graph operations implemented natively in the language. It also tries to give examples of the most common uses of the language; while this highlights the features of NSBL, it is not comprehensive of all the situations and the full use of NSBL.

We will start by showing a simple Hello World example, and explain how the program structure works, plus how the types and declarations work specifically for NSBL. It has an extensively detailed explanation to show how to correctly think about graphs for future construction. We have also isolated snippets for easy reference. We later rework the example with more specific NSBL syntax. We will then issue a simple program to show arithmetic operators on variables.

The Graph operation section shows snippets on how to work with the Graph grammar of NSBL, and then applies this ideas into an example on how to build a corporate hierarchy graph.

The IO section shows different example on how to build and write a graph file and how to read from it.

1.1 Simple Hello World

The following Hello World example is meant to illustrate the basics syntax for graph creation, variable declaration and the structure of NSBL programs. Four nodes represent friends (numbered 1 to 4) that are connected through edges to a node World. The edges represent the action of the friends; they all “say”.

Example 1.1: simple Hello World

```
//Primitive Variables declaration
int x=0;
int y=0;

//Graph Creation
//Declaration of graph elements
vertex v1,v2,v3,v4,v5;
edge e1,e2,e3,e4;
list lv = [v1, v2, v3, v4];
list le = [e1, e2, e3];
graph g1;
//Element’s attributes
string v1::project = “World”;
string v2::name = “Friend one”;
string v3::name = “Friend two”;
string v4::name = “Friend three”;
string v5::name = “Friend four”;
string e1::relationship = “says”;
string e2::relationship = e1.relationship;
string e2::relationship;
string e4::relationship;
e2.relationship = e1.relationship;
e4.relationship= "says";
// Graph composition
lv <: v5;
le <: e4;
e1: v2 -> v1;
e2: v3 -> v1;
e3: v4 -> v1;
e4: v5 -> v1;
g1 <: lv;
g1 <: le;
//Start of program
void hello(){
    vertex vtemp;
    edge etemp;
    print(“Welcome to NSBL! ”);
    for (i=1; i<lv.length, i++)
    {
        vtemp= lv[i];
        etemp= le[i-1];
        print(vtemp.name);
        print(etemp.relationship):
        print("Hello");
        print(lv[0].project);
    }
}
```

Output

```
Welcome to NSBL!
Friend one says Hello World
Friend two says Hello World
Friend three says Hello World
Friend four says Hello World
```

As can be shown form the example, NSBL mostly follows C style, to make the language familiar to those adapting from another programming language. Expressions must be terminated with a semicolon ; and the statements of a function are enclosed in braces { }. What the structure differs from C is that there is no main function to be called. All functions are called when they are declared or when they are called afterwards. This means that functions can be nested.

In lines **1** through **3** we have primitive variables declaration. In order to work with the derived types of a graph, there are four primitive data types that can be used as attributes for derived types or as standalone variables. The scope of a variable is that of the block it is declared in. If the variable is declared outside of a function, like these lines; it has a global scope. Variables x and y are integers. Integers represents all signed integer numbers. In this Hello World example we use the integers to create counters. The declaration and assignment is like C. Examples for the different ways of declarations and assignments of integers are shown below.

Snippet 1: Integer declaration

`int a;`	*integer declaration*
`a=1;`	*integer assignment*
`int b=2, c=3;`	*integer declaration and assignment*

The other number type is float, which supports decimal and exponential numbers. Integers and floats can be explicitly cast into each other, with floats losing accuracy.

Snippet 2: Type casting

`int icast = (int)3.45;`	*integer cast*
`float fcast = (float)10;`	*float cast*

Lines **6** to **29** show how a graph is constructed in NSBL. All of graph construction deals with NSBL's derived data types. To be able to construct the graph, let's first look at the elements. A logical order lo look at the elements is think that the vertices hold information about an object, and that two vertices are connected through a relationship represented by an edge. However, the edges and vertices are elements independent of each other, collected each in homogeneous lists. A graph will be a list of edges and a list of vertices. Now let's look at each element and its implementation in NSBL.

The vertices are the nodes of a graph. Each vertex has a unique identifier set by the compiler (id). It can be assigned when declared. Line **6** shows the declaration of the vertices,

Snippet 3: Vertex declaration

```
vertex v1,v2,v3,v4,v5;            Declaration of vertices
```

Lines **11** to **15** show the declaration and assignment of vertices attributes;
A vertex can hold any number of named attributes, which can be of any primitive type. An attribute is declared by indicating the attribute's type the vertex it belongs to followed by the belong operator (::) and the name of the attribute.

Snippet 4: Declaration of Vertices' attributes

```
string v1::project = “World”;        Declaration and assignment  of vertex attribute.
```

After an attribute is assigned or declared without a value, the value can be changed by accessing the attribute as a member of the vertex.

Lines **7** deals with the declaration of edges, which is in the same format as the declaration of vertices. An edge is meant to represent the relationship between two vertices. Like a vertex, it has a unique identifier set by the compiler (id), and can hold any number of named attributes of any primitive type.

Lines **16** to **21** show the different ways of declaration and assignment of vertices attributes. Attributes work the same way for vertices and edges.

Snippet 5: Declaration of Edges' vertices

```
string e1::relationship = “says”;    Declaration and assignment  of vertex attribute.

string e2::relationship =            Declaration and assignment  of vertex attribute.
e1.relationship;

string e2::relationship;             Declaration of vertex attribute.
```

As in line **21 ,** after an attribute is assigned or declared the value can be accessed as a member of the element (it works the same way for edges and vertices)

Snippet 6: Reassignment of Vertices

```
e4.relationship= "says";          Reassignment of vertex attribute
```

Since managing element by element is not efficient for the user, we have implemented lists. Lists are just a type that holds a list of homogenous primitives, edges or vertices. That is, a list can only hold one type at a time. Lists' declarations are on lines **8** to **9.** Elements within a list are declared in square brackets [ ] and separated by commas. A list can be declared empty.

Snippet 7: List declaration and assignment

```
list empty = [];                    Declaration and assignment  of a list.
list lv = [v1, v2, v3];             Declaration and assignment  of a list.
list le = [e1, e2, e3];             Declaration and assignment  of a list.
```

Assignment can also be done by using index of the element in the list. An alternate version of line **8** , using indices:

Snippet 8: Assignment on a list by index

```
lv [0] = v1;                        Assignment  on  a list. by index
lv [1] = v2;
lv [2] = v3;
lv [3] = v4;
```

All lists have a length property.

A graph in NSBL holds lists of edges and vertices. A graph is declared in line **10**.

Snippet 9: Graph declaration

```
graph g1;                           Declaration for a graph
```

Graphs are not assigned attributes.

From line **22** to line **29** its the construction of a graph, which is concerned in grouping the elements in the graph.

Line **22 to 23** show appending elements to a list. When a list is declared, it can have elements assigned to it, or assigned by choosing an index on them. Another way to construct them is by appending elements of the end of the list. To append an element to the list, we can use the append operand (<:). Since the list is homogeneous, we can only append items of the same type of those already on the list.

Snippet 10: Appending element to list

```
lv <: v5;                           Appending an element to the list
```

Lines **24** to **27** connect two edges through a declared node. Since the edge represent the edge between nodes, it needs to have an incoming vertex and an outgoing vertex assigned. To do this assignment the edge name is followed by a colon (:), the incoming vertex, the arrow operator (->) and the outgoing vertex. This helps create isolation between the positioning of the edge in a list relative to the vertices it connects.

Snippet 11: Edge assignment to vertices

```
e1: v1 -> v2;                    Edge assignment to vertices
```

Finally, for finishing the construction of a graph, a graph needs to have a list of vertices and a list of edges appended to it, as in lines **28** to **29.**

Snippet 12: Graph construction from a list

```
g1 <: lv;                        Graph construction from list
g1 <: le;                        Graph construction from list
```

An alternative to construct the graph is to append edges or vertices one by one.

Lines **31** to **44** define and implement the function hello. Function's structure is the same as C. First, the return type is stated, then the function name with the variables in parenthesis **().** The execution statements are enclosed in **{ }** .

After local variables are declared in lines **32** and **33**. The programs prints to the console by means of the function **print()**. It can take a string or a variable. Strings in NSBL are of primitive type and they are also is used to represent characters. They are enclosed in quotation marks. Since the string is only used by the print function, and not used elsewhere it does not need to be declared. The following snippet shows string declaration.

Snippet 13: String declaration and assignment

```
string s1="hello world";         string declaration and assignment
```

Control flow statements for, if/else and while in NSBL work like their C counter parts.

From lines **37** to **42** the current vertex and edge (selected from the index increment in the loop) is saved to a local variable. Then the attributes name and relationship are printed for the vertex and the edge respectively. Each iteration also prints "Hello" (line **41**) and World, which is the name of node 0 (line **42**).

1.2 Hello World foreach

While the previous Hello world example works for the given graph, we can tell from the for loop that we know each of the friends has only one edge with only one one attribute. To make the example use the features implemented in NSBL we have the following modified version.

Example 1.2: Hello World foreach

```
//Variables declaration
int x=0;
int y=0;
//Graph Creation
vertex v1,v2,v3,v4,v5;
edge e1,e2,e3,e4;
list lv = [v1, v2, v3, v4];
list le = [e1, e2, e3];
graph g1;
string v1::project = “World”;
string v2::name = “Friend one”;
string v3::name = “Friend two”;
string v4::name = “Friend three”;
string v5::name = “Friend four”;
string e1::relationship = “says”;
string e2::relationship;
e2.relationship = e1.relationship;
string e3::relationship = e2.relationship;
string e4::relationship;
e4.relationship= "says";
lv <: v5;
le <: e4;
e1: v2 -> v1;
e2: v3 -> v1;
e3: v4 -> v1;
e4: v5 -> v1;
g1 <: lv;
g1 <: le;
//Start of program
void hello(){
     vertex vtemp;
     edge etemp;
     print(“Welcome to NSBL! ”);
     foreach (vtemp: lv)
     {
          print(vtemp.name);
          list l= vtemp|outE;
          foreach(etemp:l){
               print(etemp.relationship);
          }
          print("Hello");
          print(lv[0].project);
     }
}
```

Output

```
Welcome to NSBL!
Friend one says Hello World
Friend two says Hello World
Friend three says Hello World
Friend four says Hello World
```

The variables declaration, graph creation and Output are the same as in the previous Hello world program. The hello function, however, implements a control flow function not present in C; **for each(item:collection)**. The foreach function traverses through all the items of a collection. It is particularly useful, as lines **37** and **38** show, as you are able to get all of the outgoing edges of a node saved in a list, and be able to traverse each edge in the list.
The pipe and outE on line **37** are explained in the examples on graph operations.

1.3 Arithmetic Operations

The arithmetic operations of addition, difference, multiplication and division are implemented for integers and floats.

The third primitive is a **boolean** that can take the value true or false. It cannot be cast to or form another data type. It can be declared as follows.

Snippet 14: Boolean declaration

```
boolean b= true;                        boolean declaration and assignment
```

Booleans are used in NSBL's conditional expressions, as they return booleans.Booleans can also use logical AND and logical OR operators.

One way to use NSBL is to create programs to calculate total weights of edges or differences between them. Following is a very simple example of three nodes and two edges, and operations on integers and booleans

Example 1.3: Arithmetic operation example:

```
graph g;
vertex v1,v2,v3;
edge e1,e2,temp;
string v1::name = "First House";
string v2::name = ”Second House”;
string v3::name = ”Third House”;
boolean v1::isBought = true;
boolean v2::isBought = false;
boolean v3::isBought = true;
int e1::weight = 10;
int e2::weight = 15;
int total=0;
int difference=0;
boolean result=true;
e1:v1->v2;
e2:v2->v3;
g<:v1;
g<:v2;
g<:v3;
g<:e1;
g<:e2;
```

```
22.         total = e1.weight + e2.weight;
23.         difference = e2.weight - e1.weight;
24.         print("The distance from v1 to v2 is ");
25.         print(total);
26.         print("The distance difference between 2 to 3 and 2 to 1 is");
27.         print(difference);
28.         result= v1.isBought || v2.isBought;
29.         print(“House 1 or House 2 are bought”);
30.         print(result);
31.         result= v1.isBought && v2.isBought;
32.         print(“House 1 and House 3 are bought”);
33.         print(result);
```

Output

```
The distance from v1 to v2 is 25
The distance difference between 2 to 3 and 2 to 1 is 5
House 1 or House 2 are bought TRUE
House 1 and House 2 are bought FALSE
```

Lines **22** and **23** show addition and difference on the attribute weight, of type integer, of the edges between vertices, respectively

Lines **28** and **31** show the OR and AND operation on boolean attribute of the vertices, respectively.

2. Graph Operations

With NSBL, we have introduced several graph operations that can aid a programmer to think about the bigger problem and not worry about how to retrieve information from the graph. The main pillars of graph information retrieval are the graphs/vertices/edges properties ***allV, allE, outE, inE, strtV and endV***. When combined with the ***pipe operator ('|')*** and ***function literals***, these properties would aid the programmer to find all possible information that the graph contains.

2.1 allV

The allV property is associated with a graph and returns all the vertices that a graph contains. The property returns a list.

Snippet 14: Using allV

```
graph g;
vertex v1,v2;
edge e1;
string v1::name = "First Vertex";
string v2::name=”Second Vertex”;
int e1::weight = 11;
string e1::relation = "friends";
e1:v1->v2;
g <: v1;
g <:v2;
g <:e1;
list vertices=g.allV ;//vertices now contains {v1,v2,v3}
```

In the above example, vertices is of list type and contains all the vertices contained in graph g.

2.2 allE

The allE property is associated with a graph and returns all the edges present in a graph. The property returns a list as well.

Snippet 15: Using allE

```
graph g;
vertex v1,v2,v3;
edge e1,e2;
string v1::name = "First Vertex";
string v2::name=”Second Vertex”;
string v3::name=”Third Vertex”;
int e1::weight = 11;
string e1::relation = "friends";
int e2::weight = 12;
string e2::relation = "friends";
e1:v1->v2;
e2:v2->v3;
g<:v1;
g<:v2;
```

```
g<:v3;
g<:e1;
g<:e2;
list edges=g.allE;//edges now contains {e1,e2}
```

In the above example, edges is of list type and contains all the edges contained in graph g;

2.3 outE

The outE property is associated with a vertex. The property returns a list of all the outgoing edges from the given vertex.

Snippet 16: Using outE

```
graph g;
vertex v1,v2,v3;
edge e1,e2;
string v1::name = "First Vertex";
string v2::name=”Second Vertex”;
string v3::name=”Third Vertex”;
int e1::weight = 11;
string e1::relation = "friends";
int e2::weight = 12;
string e2::relation = "friends";
e1:v1->v2;
e2:v2->v3;
g<:v1;
g<:v2;
g<:v3;
g<:e1;
g<:e2;
list vertices=g.allV;
list outedges_of_first_vertex=vertices[0]|outE;//
outedges_of_first_vertex={e1}
```

In the above example, outedges_of_first_vertex is a list that contains all the outgoing edges from the first vertex in list vertices. You must have noticed a special operator placed between vertices[0] and outE. We call it pipe (‘|’). The pipe operator allows multiple properties/function literals to work in tandem and Output what the programmer desires. It should be noted that outE can only work with the pipe operator. We shall take a closer look at pipe later in the tutuorial.

2.4 inE

The inE property is associated with a vertex. The property returns a list of all incoming edges to a given vertex.

Snippet 17: Using inE

```
graph g;
vertex v1,v2,v3;
edge e1,e2;
string v1::name = "First Vertex";
string v2::name="Second Vertex";
string v3::name="Third Vertex";
int e1::weight = 11;
string e1::relation = "friends";
int e2::weight = 12;
string e2::relation = "friends";
e1:v1->v2;
e2:v2->v3;
g<:v1;
g<:v2;
g<:v3;
g<:e1;
g<:e2;
list vertices=g.allV;
list inedges_of_first_vertex=vertices[2]|inE;
// inedges_of_first_vertex={e2}
```

In the above example, inedges_of_first_vertex is a list that contains all the incoming edges to the third vertex in list vertices. Again, inE property will only function with the pipe operator.

2.5 Pipe Operator (|)

The pipe operator is very important part of NSBL's query processing unit. The pipe operator works as a redirection operator which reads the Output of one command and feeds it as the input of another command. The pipe operator in NSBL always returns a list. Snippets 16 and 17 demonstrate how pipe can be used in NSBL. Let us have a closer look at those examples:

Snippet 18: Pipe on snippet 16

```
list outedges_of_first_vertex=vertices[0]|outE;
```

In here, the vertex stored in vertices[0] (v1) is the Output that is fed into the keyword outE. outE finds out all the outgoing edges of that vertex(which is e1) and returns the Output in list format.

Snippet 19: Pipe on snippet 17

```
list inedges_of_first_vertex=vertices[2]|inE;
```

inedges_of_first_vertex contains all the edges incoming (in this case e2) into the third vertex (v3) in the list vertices.

As we proceed further in the tutorial, you will find more uses of pipe.

2.6 strtV and endV

strtV keyword finds the start vertex of an edge. Similarly endV finds the end vertex of an edge. These two properties will only work with the pipe operator. The following example demonstrates the use of strtV and endV and is an extension of the previous examples.

Snippet 20: strtV and endV

```
list start_edges=vertices[0]|outE|strtV;
```

Let’s break down the right hand side of the assignment. vertices[0]|outE, as discussed above, returns a list of edges going out of the first vertex of the list vertices. This list is now feed into strtV which finds the start vertex of each edge contained in that list.
For e.g., if e1:v1->v2 then.
list l=e1|startV;
would contain v1 which is the start vertex of the list. Similarly,
list l=e1|endV;
would contain v2 which is the end vertex of the list.

2.7 @ operator

The @ (pronounced as ‘at’) operator is used to retrieve the information form an attribute that belongs to a vertex or an edge. The @ operator when followed by the attribute’s name can retrieve the attribute’s value.

Snippet 21: AT operator

```
vertex v1,v2;
string v1::name=”Tom”;
string v2::name=”Bob”;
list l=[v1,v2];
list at_Output=l[0]|outE[@name==”Tom”]//list contains v1
```

The code above shows how @ operator is used to retrieve attribute information for the vertices contained in the list l. You must have noticed that @ operator is used inside a square bracket. We call this Match. What is Match? The later section explains this. The @ operator is to be used inside the Match square brackets. The attribute name that follows @ retrieves the information contained in that attribute and return type is the same as that of the attribute.

2.8 Match([ ])

Match is a concept in NSBL that is implemented with the help of square brackets ([ ]), @ operator and/or function literals. The value inside the brackets is of type boolean. This means that any conditional expression that is relevant with any property or attribute of the graph can be put inside Match. The concept of matching helps to build queries that retrieve information from the graph. The Match concept needs to be used with any of the above mentioned graph operations like allV, allE, outE, inE, strtV and ednV in order to retrieve graph information.   The following example illustrates how queries are built in NSBL using Match.

Snippet 22: MATCH operator

```
graph g;
vertex v1,v2,v3;
edge e1,e2;
string v1::name = "First Vertex";
string v2::name=”Second Vertex”;
string v3::name=”Third Vertex”;
int e1::weight = 11;
string e1::relation = "friends";
```

```
int e2::weight = 12;
string e2::relation = "friends";
e1:v1->v2;
e2:v2->v3;
g<:v1;
g<:v2;
g<:v3;
g<:e1;
g<:e2;
list edges=vertices[0]|outE[@weight>10];//edges={e1,e2}
```

The list edges contains all the edges that are outgoing from the first vertex of list vertices and have a weight greater than 10. Here, the edge attribute ‘weight’ is used to build the query and retrieve information from the graph.

The same can be achieved with a function literal.

Snippet 23: Function literal

```
//defining the function literal
func cond():boolean = { return (@weight>11); };
graph g;
vertex v1,v2,v3;
edge e1,e2;
string v1::name = "First Vertex";
string v2::name=”Second Vertex”;
string v3::name=”Third Vertex”;
int e1::weight = 11;
string e1::relation = "friends";
int e2::weight = 12;
string e2::relation = "friends";
e1:v1->v2;
e2:v2->v3;
g<:v1;
g<:v2;
g<:v3;
g<:e1;
g<:e2;
list edges=vertices[0]|outE[cond()];//edges contains {e2}
```

In the above example, the attribute is replaced by a function literal which checks for weight >11. Thus, the list edges would now contain only e2. Match concept can be implemented in similar fashion to build many complex queries and can aid in information retrieval.
To sum up the section, let’s look at the corporate hierarchy example. The example contains 5 persons (vertices) who are related to each other professionally. Consider this as a graph where each person is a vertex and the professional relation that each one shares with his/her co-worker is an edge. The following example demonstrates the same:

Example 2.1: Corporate hierarchy

```
graph social;
vertex person1,person2,person3,person4,person5,tempV;
edge connect1, connect2, connect3, connect4;
```

```
string person1::name = "Jim";
string person2::name = "Martha";
string person3::name = "Kevin";
string person4::name = "Alicia";
string person5::name = "Silvia";

string person1::position = "Project Manager";
string person2::position = "Project Lead";
string person3::position = "Associate";
string person4::position = "Associate";
string person5::position = "Associate";

int person1::salary = 2000;
int person2::salary = 1500;
int person3::salary = 1200;
int person4::salary = 1150;
int person5::salary = 1000;

string connect1::relation = "Boss of";
string connect2::relation = "Boss of";
string connect3::relation = "Boss of";
string connect4::relation = "Boss of";

connect1:person1->person2;
connect2:person2->person3;
connect3:person2->person4;
connect4:person2->person5;

list vertices=[person1, person2, person3, person4, person5];
list edges=[connect1, connect2, connect3, connect4];

social<:vertices;
social<:edges;

//function literal
func cond(int a):boolean = { return (@salary>a); };

//Match operations
list query1=social.allV[@salary>1100];
list query2=social.allE|endV[@postion=="Associate"];


print("The employees who are in the position of associate are:");
foreach (tempV:l2){
     print(tempV.name);
}

print("Determining the names of the employees who have a salary greater
than 1100 through match and pipe query:");
foreach (tempV:l1){
     print(tempV.name);
}

print("Determining the names of the employees who have a salary greater
than 1100 through function literal:");
foreach(tempV:social.allV[cond(1100)]){
    print(tempV.name);
}
```

Output:

```
The employees who are in the position of associate are:
Kevin
Alicia
Silvia
Determining the names of the employees who have a salary greater than 1100 through match
and pipe query:
Jim
Martha
Kevin
Alicia
Determining the names of the employees who have a salary greater than 1100 through function
literal:
Jim
Martha
Kevin
Alicia
```

Let’s look at the query statements.

```
list query2=social.allE|endV[@postion=="Associate"];
```

This query returns a list of all the employees with position of Associate that are the end vertex for an edge. The @ operator is used to retrieve information contained in the position attribute of vertices. The pipe operator is used to find the end vertices of the all the edges in the graph social.

```
list query1=social.allV[@salary>1100];
```

contains a list of all the employees who have a salary greater than 1100. The Match operation with the help of @ operator finds all the persons who have a salary greater than 1100 and the list query1 stores the vertices returned by the query.
Similar objective is achieved using a function literal.

```
func cond(int a):boolean = { return (@salary>a); };
```

The function literal has a condition that returns true if the salary of the person (vertex) is greater than 1100. The value 1100 is passed as argument in the loop

```
foreach(tempV:social.allV[cond(1100)]).
```

This example should help you invent your own queries on the graphs that you create with NSBL!

3 Graph I/O

It is at times cumbersome to create graphs in NSBL; look at the example in the earlier section. So, how do we decrease the manual labor? Like any other programming language, NSBL has the capability to read files as well as write files. NSBL , in fact, can read graph files that were previously created in NSBL and do operations on those graphs. Further it can write graph files as well. This gives the user the freedom create graphs once, write them to the disk and then create functions to operate on the graph later and fetch the graph from the disk multiple times and run these operations on it. So, the user does not have to create a graph again and again but just has to worry about the functions or queries that are to be implemented.

3.1 Reading Graphs

openGraph(string file_loc) is the inbuilt function that can be used to read a file in NSBL. The openGraph function takes one single argument of type string that specifies the file location and returns a graph.

Snippet 24: Reading Graphs

```
graph g;
g=openGraph("C:\graphDemo.g");
```

The code snippet shows how openGraph is to be used. The argument is the string that specifies the file location and graphDemo.nsbl is retrieved and now stored in graph g. The extension of graph files is .g.

3.2 Writing Graphs

writeGraph(string loc, graph g) is the inbuilt function that is to be used to write graph created in NSBL.

Snippet 25: Writing Graphs

```
graph  g;
vertex v1,v2;
edge e1;
e1:v1->v2;
g<:v1;
g<:v2;
g<:e1;
writeGraph("C:\",g);
```

The above code snippet would create a file g.nsbl in C drive. The function has two arguments. The first argument is of type string that specifies the file location and the second argument is of type graph that is to be written to the disk.

The following example shows how to create a graph and write it to disk in NSBL.

Example 3.1: Write and save graph

```
graph g;
vertex v1,v2,v3,v4,v5;
edge e1,e2,e3,e4;
string v1::name = "level 1";
string v2::name=”level 2.1”;
string v3::name=”level 2.2”;
string v3::name=”level 2.3”;
string v3::name=”level 2.4”;
string e1::relation = "level 1 to level 2.1";
string e2::relation = "level 1 to level 2.2";
string e3::relation = "level 2.1 to level 3.1";
string e4::relation = "level 2.2 to level 3.2";
e1:v1->v2;
e2:v1->v3;
e3:v2->v4;
e4:v3->v5;
g<:v1;
g<:v2;
g<:v3;
g<:v4;
g<:v5;
g<:e1;
g<:e2;
g<:e3;
g<:e4;
writeGraph(“C:\”,g);
print(“Graph g written to disk”);
```

Output:

```
Graph g written to disk
```

As mentioned earlier the writeGraph function writes the graph g to C drive on the disk. Now graph g can be retrieved at a later time from the disk so as to run functions or queries on it.

In the next example we use the written graph g to perform BFS. For this we need to read the graph. The example also demonstrates the use of functions in NSBL.

Example 3.2: Use of Graph files

```
graph x;
x=readGraph(“C:\g.g”);
print(“reading operation successful”);
void BFS(vertex v, graph g){
      list lv=[v];
              vertex temp,y;
              temp=v;
 int no_of_vertices=0;
              int i=0;
while(i<l.len()){
     temp=v[i];
     print(“ Traversed “);
     print(temp.name);
     foreach(y: temp|outE|endV){
          lv<:y;
     }
     i = i + 1;
}
}
list vertices=x.allV
BFS(vertices[0],x);
```

Output:

```
reading operation successful
Traversed level 1  Traversed level 2.1  Traversed level 2.2  Traversed level 3.1  Traversed level
3.2
```

You must have noticed how BFS is performed on the graph x which contains graph g that is read from the disk using inbuilt function readGraph. Let’s look at BFS function. The function is a user declared function of return type void. It has two arguments of type vertex and graph. The vertex is the point of reference from where the BFS is supposed to start. The function has two loops. The main loop is while loop which prints the name of the vertex that is currently being traversed. The while loop contains a foreach loop that adds the vertices connected to the vertex which is currently being traversed to the list lv. List lv stores all the vertices that are to be traversed and the while loop runs till all the vertices are covered. The foreach loop uses the pipe operation and graph properties outE and endV to find all the vertices that are connected to the current vertex. The foreach loop runs till all such vertices are added to list lv. As you have noticed, the readGraph function makes it so easy for the user to just concentrate on BFS function and not worry about creating the graph again.