GOBLAN

Language Reference Manual

UNI	Name	Role
yk2553	Yunsung Kim	Project Manager
sl3368	Sameer Lal	Verification & Validation
jw3046	Jee Hyun Wang	Language Guru
dl2997	Da Liu	System Architect
sjg2174	Sean Garvey	System Architect

Table of Contents

1. Introduction

Graphs appear naturally in problems of various domains such as physics, chemistry, sociology, linguistics etc. for the way they intuitively express entities and the complex relationships among them. Especially in the fields of computer science and statistics, graphs are used to model information diffusion or as a statistical formalism for describing the relationship between events. Graphs are used in this context to represent distributions and formulas which generate data that is observed.

A method for exact inference in these statistical graphical models is the Junction Tree Algorithm (JTA). JTA is an algorithm that is based on the node connectivity in a graphical model. Each node passes to its neighboring nodes the "messages" that encode the partial information about the posterior distribution, and the neighbors collect these messages, run a function to create its own message, and pass it to their neighbors down the stream. The messages received by the last node in the stream is then used to reconstruct the desired answer.

The intuition behind this algorithm falls into the paradigm of “message passing” among nodes, which is characterized by two major components: the ability to (1) exchange messages between nodes, and to (2) update the data or state of each node using those messages by computing certain functions. In fact, message passing is the intuition behind many graphs algorithms including shortest path search or binary tree search. Yet, implementing these node-oriented algorithms can be tedious and time consuming.

In this context, GOBLAN is a language that focuses, not on manipulating the graph as a whole (with the use of adjacency matrices), but rather on exchanging data (messages) among individual nodes and using these data to update the data at each node. This domain specific language is expected to enable developers to work with graph based structures and algorithms with more ease by thinking in terms of the operations of each node.

2. Types and Literals

GOBLAN has a set of types and literals which are similar to those of most programming languages. Since GOBLAN compiles into C, most of these specifications match those of the compiled language. Types in GOBLAN can be largely classified into primitive types and non-primitive (user-defined) types. List, named tuples and graph nodes fall into the latter. While primitive data types are passed by value, non-primitive data types are always passed by reference.

2.1 Primitive Types

A number of primitive data types are defined as follows:

2.1.1 Integers

Integer constants (e.g. 14, 2131, 335112, etc.) are indicated with the keyword *int.* In GOBLAN, all integers are 4 bytes in size and thus can represent any signed integer in the range [-2147483647, +2147483647].

```
int d  = 30;
```

2.1.2 Characters

Character constants, signified by a single character, surrounded by single quotes: ‘a’. Only ASCII characters are permitted. For variable instantiation, the *char* keyword is used to state the this type.

```
char first = ‘a’;
```

2.1.3 Strings

String constants are defined as a list of characters greater than size 1. They are identified as being surrounded by double quotes: “hello world”. For variable instantiation, the *String* keyword is used to state type. These may only be defined using GOBLAN defined characters. In order for double quote (“) or single quote (‘) characters to be a part of the string, they must be preceded with a backslash (e.g. “he said \“Hi\” ”).

```
String greeting = “hello world”;
```

2.1.4 Floating Point Numbers

To represent rational numbers and values outside the range provided by the integer constant, the double precision floating point data type is available using the keyword *float*. This is signified by the double keyword for variables. As mentioned in the C Language manual, these are defined as follows: “A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.”

```
float number = 3.213
```

2.1.5 Boolean

This data type in GOBLAN represents the most basic unit of logic. It takes either the value of *true* or *false*. Variables are typed using the keyword *bool*.

```
bool win = false;
```

2.2 Lists

The list data type is signified using the left and right braces “[ ]”. List types are like arrays in that they allow for manipulation using indexing, but are also variable in size. They automatically resize to the amount of data in the list. Lists can be mutated to include more or less values using keywords: *add* and *delete.* The add keyword appends the value to the end of the list. The delete keyword deletes the object at the index specified:

```
delete 3 from I
```

A list is declared by a type specifier followed by the keyword “list” and an identifier. If the list stores named tuples, the type specifier is replaced by the keyword “tuple”.

```
int list l; add 1 to l;           // declares a list l of integers and appends 1 to it
edge list l; add {int i = 0, int val = 0} to l;
```

2.3 Named Tuple

Named tuples are flexible data structures intended for holding the necessary data for graph computations. These are essentially a list of key,value pairs which can be of different types. Each type of a named tuple should be first declared by using the keyword *tuple* as in:

```
tuple type_name{type1 member1, type2 member2,...};
```

Named tuples are instantiated by specifying the tuple type and writing out the members separated by commas in brackets (“{}”). This statement adds returns the reference(address) to the tuple instance, which can be stored as a separate variable. An example of declaring a tuple type and instantiating it:

```
tuple new_type{int cats, bool hat};   // declare tuple type “new_type”
new_type a = new_type{0, true};  // a stores the address of the tuple instance
```

Tuples must be accessed by its reference. To access the value for a given key in a tuple, the key is followed by a dot:

```
a.cats == 0 // true
```

3. Graphs and Nodes

GOBLAN is a language for graph-based programming, suitable for implementing algorithms within the “message passing” framework.

In the framework of message passing, a graph is a group of nodes that possess data attributes and object functions that manipulate these data or communicate them to other nodes in the form of “messages”. There are two types of object functions. First are the “synchronous object functions,” which are functions invoked by a node upon receiving a message to process the message. These functions are used to update the data of the node based on the message communicated from other nodes. The second type of object functions are “asynchronous object functions,” which are functions that manipulate the data of a node or initiate message passing at any point in the program.

Many major graph algorithms can be constructed within this framework. In a shortest path search algorithm (Dijkstra’s algorithm), for instance, each node maintains current estimates of the shortest distance from the source (data). Going in the increasing order of distance from the source, each node sends to its neighbors its current estimate of the distance from source (asynchronous function). Upon receiving the distance estimates from each node, the receiving node updates its own distance estimate and saves the neighbor that connects the source and itself in shortest distance (synchronous function). When the root node is reached, the algorithm terminates.

Binary tree search also falls into the paradigm of message passing. Each node in the tree holds data, and the algorithm starts from the root by recursively running binary search on its subtrees. Each descendent of the root collects the search results of its subtrees, searches its own data, and passes the search result to its parent (synchronous functions). The root initiates this search and collects the final search results (asynchronous functions).

The data attributes, a synchronous function sequence, and an asynchronous function sequence thus collectively define a node type. Graphs in GOBLAN are homogeneous directed graphs, which are directed graphs whose nodes are of identical types. A graph in GOBLAN is conceptually a group of multiple nodes, and GOBLAN internally represents a graph as a list of nodes.

3.1 Node Types

Users can declare different types of a node by specifying their data members, asynchronous functions, and synchronous functions. The declaration of a node type must contain 3 blocks: the “data” block for defining node members, the “do” block for defining the asynchronous function, and the “catch” block for defining the synchronous function.

```
node NodeType {
        data {
        /* data specification */
         type_1 member_1;
         ...
        }
        type do (type arg1, type arg2, ...) {
        /* asynchronous function definition */
        }
        catch {
        /* synchronous function definition */
        }
}
```

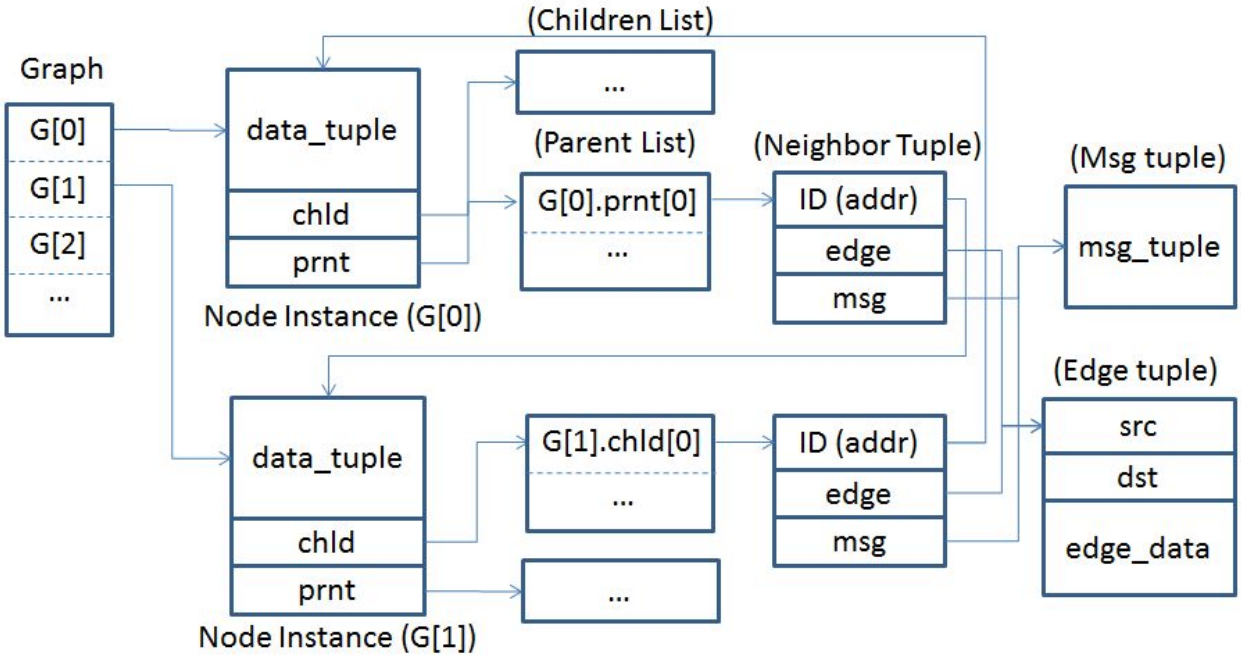
Declaring the members in the data{} block requires declaring the types and names of each member (as in declaring tuple types). Asynchronous functions are defined in the do(){} block in the way typical functions are defined. Asynchronous functions can take arguments or return a value. Synchronous functions are defined in the catch{} block, and they do not take arguments nor return values; conceptually, their arguments are passed via messages, and they do not require return values. Following is the declaration of the DijkstraNode. (It is explained in greater detail in the last section.)

```
node DijkstraNode{
    data{
         int dist = infinity;
         DijkstraNode prev;
    }
    void do (list l){
         msg = {DijkstraNode sender = self,
                        int dist = self.dist};
         pass msg to all;
         delete self from l;
    }
    catch{
         if (pack.dist + self[pack.sender].dist < self.dist){
                self.dist = pack.dist + self[pack.sender].dist;
                self.prev = pack.senderl
         }
    }
}
```

In addition to the members specified in the data block, each node has two default members called “chld” and “prnt.” Both of these members are lists of neighbor tuples. A neighbor tuple

contains the address of the neighbor node instance (ID), the address to the edge tuple for the edge that connects the neighbor and the node, and the address to the message tuple that is shared between the two ends of the edge. Note that the neighbor tuple is ***not*** a node instance but instead a metadata of the relationship between the two connected nodes.

The following figure visualizes the complete data structure of graphs and nodes.



In the graph in this figure, there exists an edge that goes from node G[1] to G[0]. Arrows indicate addresses and the structures to which the addresses refer to. Note that two nodes on each side of an edge share the same edge and message tuples. When a message is passed and the synchronous function is invoked, the shared message tuple is populated by the sender and its contents are retrieved by the receiver.

Following are the keywords that are used in graph declaration.

3.1.1 The “self” Keyword

Used within the context of the graph definition, the “self” keyword is the reference to the node itself. For example, the following statement construct a named tuple containing the reference of the sender node and the distance value.

```
node SomeGraph{
        ...
        int do{...
                tuple msg = tuple{SomeGraph sender = self, int dist = 0};
        ...}
...}
```

3.1.2 The “message” Keyword

The “message” keyword is valid only within the “catch” block to refer to the message that was received by the node. Messages are treated like regular named tuples. In the context of the sample code in the previous subsection, the following statement declares a synchronous function which prints the sender of the message and the dist field in the message.

```
node SomeGraph{
        ...
        catch{...
                print(“%s, dist: %d” % (to_string(message.sender), message.dist) );
        ...}
...}
```

3.1.3 The “pass … to ...” Keyword

The “pass … to ...” statement is used to initiate the exchange of a packet. The statement

```
pass msg to {node|[node1, node2,...]|parents|children|all}
```

, where msg has to be a tuple type, transfers the msg tuple from the owner node to a particular node or a list of nodes. The user can also use the keywords “parents,” “children,” or “all” to refer to a specific group of nodes. The pass keyword can be used both inside and outside of a node definition.

```
node SomeGraph{
        ...
        int do{
                tuple msg = tuple{SomeGraph sender = self, int dist = 0};
                pass msg to children;
        }
...}
```

3.2 Constructing and Using Nodes

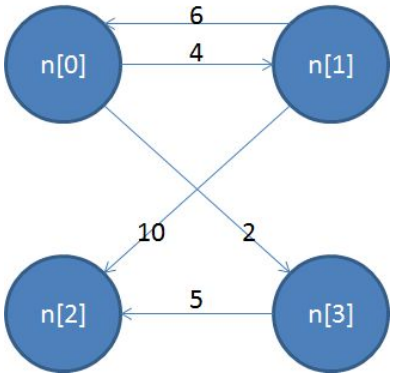
Once a node type is declared, users can create instances of each node type by calling constructors. Constructors are used just like regular functions, except that they use the names of the node types. The arguments to a constructor have to follow the order in which members are declared in the node type declaration. A constructor returns a reference to the instance of the created node. The following statement declares a reference to a DijkstraNode called n, creates a DijkstraNode instance, and store the reference to that instance in n. All node instances must be accessed through its reference.

```
DijkstraNode n = DijkstraNode(0, null); // n is the address to a new Dijkstra node
```

3.3 Constructing Graphs

A graph is declared by the node type followed by the keyword “graph” and an identifier. Graph instances are initialized by the graph constructor, which is also denoted by the keyword “graph,” but instead called like a function with two arguments: a list of nodes and a list of edge tuples. An edge tuple must contain a source node, a destination node, and optionally some data attributes, in order. The graph constructor automatically populates the chld and prnt list of each node in the graph.

Consider, for example, the following directed graph.



The following code creates the graph displayed in the previous figure. The numbers on the edges indicate weights.

```
list DijkstraNode nodes;
for (int i = 0; i < 4; i = i + 1){
       add DijkstraNode(inf, null) to nodes; // add 4 default DijskstraNode’s to the list
node
}
list tuple edges = [
{DijkstraNode src = nodes[0], DijkstraNode dst = n[1], int weight = 4},
{DijkstraNode src = nodes[0], DijkstraNode dst = n[3], int weight = 2},
{DijkstraNode src = nodes[1], DijkstraNode dst = n[0], int weight = 6},
{DijkstraNode src = nodes[1], DijkstraNode dst = n[2], int weight = 10},
{DijkstraNode src = nodes[3], DijkstraNode dst = n[2], int weight = 5}]; // creating a list
of edges

DijkstraNode graph g = DijkstraNode graph(nodes, edges); // creating graph
```

3.4 Using Nodes and Graph

Once a node type or a graph is declared, its members and functions can be accessed. Following keywords and operators are used to manipulate graphs and nodes.

3.4.1 The "." operator

The members of a node can be accessed by "." operator as in "*node_name.member*". Continuing on the graph constructor example in section 3.3, the statement

```
for (nb in nodes[0].chld){
    print("%s, %s" % (to_string(nb.id), to_string(nb.edge.dist)));
}
```

prints the ID of and distance to the children of nodes[0].

3.4.2 The "[","]" operator

This operator can be used with both nodes and graphs in different contexts. When used as "*node_name_1[node_name_2]*", evaluating this statement returns the neighbor tuple of the edge going from *node_1* to *node_2*. Continuing on the graph constructor example in section 3.3, the following statement is valid:

```
node_1 = nodes[0];
node_2 = nodes[1];
print("%d" % (to_string(node_1[node_2].edge[weight]))) // this prints 4
```

When used as "*graph_name[integer_expression]*", this statement returns a node in the graph indexed by the value of *integer_expression*. (Recall that graphs are internally represented as lists of node addresses. The [] operator is used in the same context.)

3.4.2 The *run* keyword

The statement "*run node_name(arg1, arg2,...)*" invokes the asynchronous function of *node_name* and passes *arg1, arg2,...* as the arguments to it. Usually asynchronous functions are the functions that initiate recursion of the algorithm.

For example, the following code creates a graph of two nodes whose values are initialized to 1 and 2, connects node_0 to node_1, and invokes the asynchronous function that adds a value k to the values of the node and its children

```
node SimpleNode{
    data{
        int num;
    }
    void do(int k){
        num = num + k;
        pass {int add = k} to children;
    }
```

```
    catch{
        print("adding %d to %d" % (msg.add, self.num));
        self.num = self.num + msg.add;
    }
}

SimpleNode list nodes;
add (SimpleNode(1)) to nodes;
add (SimpleNode(2)) to nodes;
edges = [{SimpleNode src = nodes[0], SimpleNode dst = nodes[1]}];
SimpleNode graph g = SimpleNode graph(nodes, edges)

run g[0](1); // now node[0].num is 1, node[1].num is 2; output: "adding 1 to 1"
run g[0](2); // now node[0].num is 3, node[1].num is 4; output: "adding 2 to 2"
```

4. Expressions

In this section, the order of the subsections denotes the precedence; expressions in section 3.1 are of the highest precedence and expressions in section 3.9 are of the lowest. Within a subsection, the precedence of the expressions are equivalent. Each subsection outlines the associativity of the expressions in its respective subsection.

4.1 Primary expressions

The operators in this section have left-to-right associativity.

4.1.1 *identifier*

The identifier refers to a previously declared and initialized identifier.

4.1.2 *constant, null, inf*

A constant refers to a data type literal. The integer, floating point, and boolean constants have a range of possible values defined in the previous section. The keyword *null* represents a non-existing reference. The keyword *inf* represents the mathematical concept of infinity.

4.1.3 (*expression*)

The set of opening and closing parenthesis can be used to group an expression into a higher precedence. The type and value of this primary expression are that of the contained expression.

4.1.4 *identifier* [*expression*]

An identifier followed by expression surrounded by opening and closing brackets, denotes element index access. If the identifier is a list, the expression must be a non-negative integer, and the type and value of the primary expression are that of the element at index. If the identifier is a node, the expression must be a node, and the primary expression is a tuple which contains

edge information between the nodes (from the identifier node, to the expression node). If the identifier is a graph, the expression must be a non-negative integer, and the primary expression is a node. Type others than the list, node, and graph are not supported.

4.1.5 *identifier.member*

An identifier followed by a period and a member name denotes member access. The identifier must be either a node or a named tuple. Must member be one the attributed of the identifier. The type and value of the primary expression are that of the member.

4.2 Unary operators

The operators in this section have right-to-left associativity.

4.2.1 -*expression*

The - operator denotes arithmetic negation for integer and floating point types.

4.2.2 !*expression*

The ! operator denotes boolean negation for boolean types.

4.3 Multiplicative operators

The operators in this section have left-to-right associativity. Multiplicative operands must be of the same type; type conversions, promotions, or demotions are not supported.

4.3.1 *expression * expression*, *expression *. expression*

The * and *. operators denote integer and floating point multiplication.

4.3.2 *expression* / *expression*, *expression* /. *expression*

The / and /. operators denote integer and floating point division. In integer division, if the number produced by the evaluation of the operator is not an integer, the fractional portion of the number will be truncated.

4.3.3 *expression* % *expression*

The % operator denotes the integer modulo operator. The number produced by the evaluation of the operator will have the same sign as the dividend (left) operand.

4.4 Additive operators

The operators in this section have left-to-right associativity. Additive operands must be of the same type; type conversions, promotions, or demotions are not supported.

4.4.1 *expression* + *expression*, *expression* +. *expression*

The + and +. operators denote integer and floating point addition.

4.4.2 *expression* - *expression*, *expression* -. *expression*

The - and -. operators denote integer and floating point subtraction.

4.5 Relational operators

The operators in this section have left-to-right associativity. The result of relational operator evaluation is a boolean type, the value of which corresponds to the truth value of the expression. Relational operators support integer or floating point operands, but not a combination of both.

4.5.1 *expression* < *expression*, *expression* > *expression*

The < and > operators denote the less than and greater than comparison.

4.5.2 *expression* <= *expression*, *expression* >= *expression*

The <= and >= operators denote the less than or equal to and greater than or equal to comparison.

4.6 Equality operators

The operators in this section have left-to-right associativity. The result of equality operator evaluation is a boolean type, the value of which corresponds to the truth value of the expression.

4.6.1 *expression* == *expression*, *expression* != *expression*

The == and != operators denote the equal to and not equal to comparison, which support integer and floating point operands. If an integer is compared to a floating point, the integer is automatically promoted to a floating point.

4.6.2 *identifier* === *identifier*, *identifier* <> *identifier*

The === and <> operators denote the reference equal to and the not equal to comparison, which support reference operands. These are defined as Nodes, Graphs, and Strings.

4.7 Boolean AND

4.7.1 *expression* && *expression*

The && operator denotes the boolean AND operation. It has left-to-right associativity and only supports boolean operators; type conversions or demotions are not supported.

4.8 Boolean OR

4.8.1 *expression* | | *expression*

The | | operator denotes the boolean OR operation. It has left-to-right associativity and only supports boolean operators; type conversions or demotions are not supported.

4.9 Assignment

4.9.1 *identifier* = *expression*

The = operator denotes assignment. It has right-to-left associativity. The identifier must have the same type as the expression; type conversions, promotions, or demotions, are not supported.

5. Statements

Except otherwise noted, statements are executed in sequence providing instructions to the computer at the language level.

5.1 Conditional

Conditionals are generally boolean value expressions, which take the forms as the following:

5.1.1 single conditional statement

```
if (condition expression) {
        // statement
}
```

5.1.2 multiple conditional statement

```
if (condition expression) {
        // statement
} elif (condition) {
        // statement
} else {
        // statement
}
```

where our conditional statement is either a true or false boolean value for determining whether or not to execute the statement inside the curly brace block.

The keyword elif is used after the first condition unless it reaches to the last statement. (Curly braces are not optional in case of single line of conditional expression.)

5.2 while

5.2.1 while loop

```
while(condition expression) {
        // statement
        // statement
}
```

if the conditional is true, the expression or statements inside the curly braces are executed, otherwise, it is not.

5.2.2 do-while loop

```
do {
        // statement
        // statement
} while (condition expression)
```

in this do-while loop, the expression in the block is executed once no matter the conditional being true or false, if it is true, then the expression(s) will be executed again, until it becomes false. will be executed again, until it is false.

5.3 for-each loop

In for-each loop, it takes the following form:

```
for (variable in list){
        // statement
}
```

This is a python-style for loop where variable is any primitive data type of graph object, and list is a list higher-order data structure. The expression is being executed while the variable traversing the entire list elements.

5.4 for loop

```
for(expression-1; conditional expression; expression-2) {
        // statement
}
```

This is a c-style for loop, where the init expression defines the initialization of the loop, conditional expression takes value of boolean, true of false and will be evaluated before any execution of the statement. Post expression can be any value expression. The statement inside the block will be executed if the conditional expression is true; it will stop the execution once the conditional becomes false.

5.5 break

```
for(expression-1; conditional expression; expression-2) {
        // statement
        if (condition expression) {
                break; // exit the current loop
        }
        // statement-after-break
}
```

break is a keyword that can be used in a for loop or while loop or any procedural control manipulation. If the conditional inside the block is true, the runtime will exit the current loop without executing the expression-after-break statement and runtime will continue with the following code after the loop block, if there is any.

5.6 continue

```
for(expression-1; conditional expression; expression-2) {
        // statement
        if (condition expression) {
                continue; // skip the following statement in this loop block
        }
        // statement-after-continue
}
```

continue is a keyword that can be used in a for loop or while loop, or any code procedural control. If the conditional inside the block is true, the runtime will skip executing the expression-after-conditional statement and continue starting from the loop block again if the loop conditional expression is still being true.

5.7 return

Return keyword returns a value to the caller of the function or expression with the corresponding predefined type of the function, being the type of the expected value if non-void type. If it a void type, nothing is returned.

6. Functions

Functions are defined with the *fun* keyword.

fun *Function-name* (*arg1-type arg1, arg2-type arg2,...*) { *expression* }

They should be defined before being used. Arguments are passed by reference for graphs, and are passed by value for other data types, and passing arguments requires the specification of their types. Types include primitives, graph, node, list, and named tuples.

There can be multiple arguments given to a function. Each argument's type should be defined prior to its name when defining a function. The type of the argument should not be specified when calling a function. The program checks the types of arguments as the function is being called.

The *return* keyword returns value. The function will terminate when it sees the *return* keyword, and the program will return to where the function was called. Return type can be any types that are supported in GOBLAN, but it needs to match the predefined type of the function. If the return value is type void, the function returns nothing.

```
fun some_function(int a, String b){

    String newString= to_string(a);
    newString = a + b;
    return newString;

}

main {
    String returned = some_function(1, “example”);
}
```

7. Standard Library

7.1 Print

Prints a statement. Formatted in c convention.

7.2 Type conversions

7.2.1 *float_of_int(identifier)*

Takes an identifier of type integer as an argument and returns the floating point representation.

7.2.2 *int_of_float(identifier)*

If the floating point number has a fractional portion, this will be truncated. If the floating point number is too large to represent as an integer

7.2.3 *to_string(identifier)*

Takes an identifier of any type, including nodes, graphs, tuples, and lists, and converts it to a string representation.

8. Program Structure

Every statement in GOBLAN must belong to one of the three blocks: function declaration, node type declaration, and the main block. Each program may contain multiple function and node declarations, but should have one and only one main block.

The *main* block is the designated start of the program. It has several unique features. First, the *main* block cannot be called or used anywhere in the program. Second, it cannot be overloaded. The name *main* in the global name space is reserved. Third, it does not need to have the *return* statement. Also, it can call any objects or functions that are defined in global scope. Functions and nodes should be defined before being used.

In the sample code, the *main* block first initializes the list of nodes, which are predefined as `DijkstraNode`. After adding constructed node objects to the list, it then builds edges. With edges and nodes, graph G of `DijkstraNode` is created. The remaining lines in the *main* block uses predefined functions in the script to do computations for Dijkstra's algorithm then prints the results out.

The `min_dist_node` and `len` are functions. `len` takes type list as an argument and returns an int. This function is specifically used in the while loop statement for computing the Dijkstra's algorithm. `min_dist_node` takes a list of `DijkstraNode`s and returns a `DijkstraNode`.

The block `DijkstraNode` is a node constructor. Each constructed will have data `dist` of type int and `prev` of type `DijkstraNode`. They represent current estimates of the shortest distance from the source and the previous node. The asynchronous function sends to its neighbors its current estimate of the distance from source. The synchronous function for this object gets triggered when receiving the distance estimates from each node. The node updates its own distance estimate and saves the neighbor that connects the source and itself in shortest distance.

The outermost scoping is the whole program. Functions, graphs, and main create their own scope, and they are enclosed in curly brackets. For loops, conditional statement, and while loop have their own local scope. When an inner scope update its outer scope's declared variable, that variable is overridden. However, vise versa in not true.

9. Sample Program

```
node DijkstraNode{
    data{
        // distance froms source, initially inf and updated at each iteration
        int dist = infinity;
        DijkstraNode prev;  // the coparent that connects to source in shortest path
    }
    void do (list l){
        msg = {DijkstraNode sender = self,
                       int dist = self.dist};    // msg has sender address and distance
        pass msg to all;    // pass message to all neighbors I'm connected to
        delete self from l; // Remove myself from l
    }
    catch{
        // When receiving distance estimate from one of its neighbors,
        // Update my current distance estimate
        // if the sender can connect me to the source in shorter distance
        // than my current view of the shortest path
        if (pack.dist + self[pack.sender].dist < self.dist){
                // Save and that distance and neighbor
                self.dist = pack.dist + self[pack.sender].dist;
                self.prev = pack.senderl
        }
    }
}

// Finds the length of a DijkstraNode list
```

```
fun len(DijkstraNode list l){
    int i = 0;
    for (x in l) {i = i + 1;}
    return i;
}

// Finds the node in a DijkstraNode list that has smallest distance
fun min_dist_node(Dijkstra list n){
    DijkstraNode min = l[0];
    for (n in G) {
        if (n.dist < min.dist){
               min = n;
        }
    }
    return min;
}

main{
    // Initializing nodes
    list DijkstraNode nodes;
    for (int i = 0; i < 10; i++){
        add DijkstraNode(inf,null) to nodes; // initialize dist to infinity
    }

    // Initializing edges
    edges =
    [
        {DijkstraNode src = n[0],
        DijkstraNode dst = n[1],
        data = 10},... // list is truncated for brevity. "..." is not reserved
    ];

    // Constructing a graph from nodes and edges
    graph DijkstraNode G = Graph(nodes,edges);

    // Add all but the first(source) node to a separate list
    DijkstraNode list l;
    for (n in G){
        if (n != G[0]){
               add n to l
        }
    };
       // Setting the destination to the last nodes of G
    DijkstraNode dest = G[len(G) - 1];

    // Running the Dijkstra's algorithm until the destination is reached
    DijkstraNode next;
    while (len(l) > 0){
        next = min_dist_node(l);
        run next(l);
    }

       // Printing the shortest path(in reverse) from destination to source
    while (dest != G[0]){
        print dest;
        dest = dest.prev;
    }
}
```

10. Complete Table of Keywords

Keywords	*Description*
`int`	signed integer value
`char`	ASCII character
`String`	array of characters
`double`	double precision floating point number
`bool`	boolean value
`{data1: val1, data2: val2, ...}`	named tuple
`Null`	null pointer
`inf`	infinity
`fun` *`fun_name`* `(type arg1, type arg2, …)`	defines an external function
`graph` *`graph_name`*	declares a graph definition
`main(argv,argc)`	the main script for execution (with command line arguments)
`remove` *`x`* `from` *`l`*	removes the element in the array
`add` *`x`* `to` *`l`*	adds element to the end of array
`[`*`fun(n)`* `for` *`x`* `in` *`l`*`]`	iterative definition of an array
`run` *`node`*`(`*`arg`*`)`	executes the asynchronous object function
`return` *`val`*	returns value
`data`	begins the definition of graph attributes
`do (`*`type arg1`*`,` *`type arg2`*`, …)`	begins the definition of a node operation and the arguments used for the function
`pass` *`packet`* `to {node|all}`	triggers the node to forward the specified packet to a neighbor / all neighbors.
`catch`	definition of an asynchronous function
`self`	self-referring keyword for a node

`message`	keyword in the *catch* statement that refers to the packet being passed
`prnt`	The parents of a node in a directed graph
`chld`	The set of nodes which are children of the current node in a directed graph
`neighbors`	All connecting nodes in an undirected graph
`for(`*`type name; conditional; post-loop`*`) {}`	c-style for loop
`for` *`variable`* `in` *`list`* `{}`	python-style for loop
`while(`*`conditional`*`) {}`	while loop
`if(condition){}elif(condition){}else{}`	conditional statement
`tuple`	Tuple type declaration

References:
The C LRM, The Music Language LRM