GOBLAN

A Graph OBject LANguage

by

John Kim (yk2553)
Jee Hyun Wang (jw3046)
Sean Garvey (sjg2174)
Sameer Lal (sl3368)

1. Introduction

GOBLAN (Graph OBject LANguage) is a domain-specific, simple language designed for constructing and manipulating complex structured graphs. Its design is similar to the conventional imperative programming language for the convenience of the user, and yet it introduces a brand new paradigm of graph programming - namely, that of message passing.

The intuition behind many graph related algorithms can be characterized by the communication between nodes, which consists two major components: the ability to (1) exchange message packets between nodes, and to (2) use those messages by processing them and updating the data or state of each node. In fact, message passing is the intuition behind many graph algorithms including shortest path search or binary tree search (as we further describe in our language reference manual), and other crucial statistical inference algorithms such as the sum-product algorithm. Yet, implementing these node-oriented algorithms using conventional programming languages can be tedious, time consuming, and very unintuitive, since most of the existing programming languages implement these algorithms through loops or list/arrays that may appear cryptic and difficult to grasp.

In this context, GOBLAN is a graph programming language that focuses, not on manipulating the graph as a whole (as conventional programming languages would do using lists and arrays), but rather on enabling the communication between individual nodes and using the communicated 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 much more ease by providing syntactic and structural simplicity and intuitive code structures. GOBLAN is compiled down to LLVM.

2. Language Tutorial - How to Use GOBLAN

Inside the GOBLAN folder, run “make.” This creates the GOBLAN to LLVM compiler, “goblan.native”. Once the compiler is made, run “./goblan.sh file_name” to compile a GOBLAN program to LLVM. This script compiles the backend data structure modules in the “src” directory from C to LLVM bitcode, links the bitcode with the output of “goblan.native,” produces an LLVM IR, and compiles it into an executable using CLANG. The script removes the IRs and executable after execution.

The following brief GOBLAN code demonstrates how to create the mandatory main function, calls the built-in “print” function using a string literal.

```
int main(){
	print("Hello World!\n");
}
```

```
>> ./goblan.sh hello_world.gb
Hello World!
```

The following section describes how to write a program in GOBLAN and its syntactic rules.

3. Language Reference Manual

3.1Types and Literals

GOBLAN has a set of types and literals which are similar to those of many existing programming languages. Its specifications resemble most of the compiled languages. Types in GOBLAN can be largely classified into primitive types and non-primitive (user-defined) types. List, named tuples (including edges and message packets), and nodes fall into the latter. While primitive data types are passed by value, non-primitive data types are always passed by reference. All user defined types are heap-allocated using the “new” keyword and can thus be reached at any point in the program.

3.1.1 Primitive Types

3.1.1.1 Integers

Integer variables are indicated with the type keyword *int.* In GOBLAN, all integers are 4 bytes in size and thus can represent any signed integer in the range [-2147483648, +2147483647].

```
int d  = 30;
```

3.1.1.2 Floating Point Numbers

To store rational numbers and values outside the range provided by the integer constant, the floating point variable can be declared using the keyword *float*. 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
```

3.1.1.3 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;
```

3.1.2 Lists

GOBLAN supports lists of any types, including both primitive and user defined types (even lists of lists). A list variable can be declared using the “list” keyword followed by the type keyword of the values stored in the list.

```
list int l;       // l refers to a list whose elements are integers
list list int ll; // ll will store the reference to a list of list of integers
```

Since all user defined type variables including lists contain references, one must initialize the list before it can be used. The expression “`new` *`type`* `[| arg1, arg2,... |]`” creates a list of type *type* from the arguments that are provided.

```
l = new int [| 1,2,3 |] // creates a list of 3 integers
ll = new list int [| |] // creates an empty list of lists of ints and store its reference in
ll
```

Users can modify lists of user defined types by adding or removing to and from lists. The operator “+=” is used to append an element to a list, and “-=” is used to remove the first instance of the element from the list. They both return the reference to the list. For lists of primitive types, one can add to the list but are not allowed to remove from it.

```
list tuple:A l; tuple:A a; tuple:A b;
a = new tuple:A(1); b = new tuple:A(2);
l = new tuple:A [| a |]   // l contains a only
l += b;                   // l contains a and b
l -= a;                   // now l contains b only
```

3.1.3 Named Tuple

Named tuples are flexible data structures that store multiple variables 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,...};
```

Names of tuple types must start with "`tuple:`" followed by a user-defined type name. Named tuples are instantiated as "`new` *`tuple:type_name`* `[| arg1, arg2,... |]`" . This statement returns the reference(address) to the tuple instance, which can be stored in a reference variable. An example of declaring a tuple type and instantiating it:

```
tuple:new_type{int cats; bool hat;} // declare tuple type "tuple:new_type"
tuple:new_type a;                   // declare a reference variable of type "tuple:new_type"
a = new tuple:new_type(0, true);    // a stores the address of the tuple instance
```

Tuple members can be accessed by from its reference. To access the value for a given member in a tuple, the member name is followed by a dot:

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

3.2 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, edge and packet types, 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. Undirected graphs can also be implemented from directed graphs by connecting nodes with two directed edges of opposite direction with the same edge attributes. A graph in GOBLAN is conceptually a group of multiple, interconnected nodes.

3.2.1 Declaring a Node Type

Users can declare different types of a node by specifying its data attributes, edge attributes, packet attributes, asynchronous functions, and synchronous functions. The node types must begin with “`node:`” followed by the user-defined node type name. The declaration of a node type must be preceded by the node type name and braces that contain 5 blocks: the “data” block for defining node members, the “edge” block for declaring the edge attributes, the “pack” block for declaring packet members, the “do” block for defining the asynchronous function, and the “catch” block for defining the synchronous function. (Note the semantic difference of the “catch” block from the conventional C-style exception handler “try-catch.”)

```
node:NodeType {
        data {
        /* data specification */
        }
        edge {
                /* edge attribute specification */
        }
        pack {
                /* message attribute specification */
        }
        type do (type arg1, type arg2, ...) {
        /* asynchronous function definition */
        }
        catch {
        /* synchronous function definition */
        }
}
```

Declaring the members in the data{}, edge{}, and pack{} blocks 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 the messages that are passed, and they do not require return values. Following is the declaration of the node:Dijkstra, which will be referred to repeatedly throughout this section.

```
node:Dijkstra{
        data{
                int id;
                int dist;
                node:d prev;
                bool visited;
        }
        edge{
                int w;
        }
        pack{
                node:d sender;
                int dist;
        }
        void do (list node:d l){
                pack node:d p;

                self.visited = true;
                p = new pack(self, self.dist);
                pass p->chld;
                l-=self;
        }
        catch{
                if (self.visited)
                        return;
                if (msg.sender.dist + msg.sender[self].w < self.dist){
                        self.dist = msg.sender.dist + msg.sender[self].w;
                        self.prev = msg.sender;
                }
        }
}
```

Following are the keywords and statements that are used within the scope of graph object function declarations.

3.2.1.1 The “pack” Type Keyword

Used within the context of the graph object functions, the “pack” keyword is type keyword for the message packet. For example, the following highlighted portion of the “do” function in the Dijkstra example declares a message packet that will be passed to neighboring children at the call of the “do” block and processed in the “catch” block.

```
node:Dijkstra{
        ...
        void do (list node:d l){
                pack p;
                self.visited = true;
                p = new pack(self, self.dist);
                pass p->chld;
                l-=self;
        ...}
...}
```

3.2.1.2 The “self” Keyword

Used also within the context of the graph object functions, the “self” keyword is the reference to the node itself. For example, the following highlighted part of the “do” function sets the “visited” attribute of the current node to “true” and creates a new creates a new packet that contains the reference to the node itself and the “dist” value it holds.

```
node:Dijkstra{
        ...
        void do (list node:d l){
                pack p;
                self.visited = true;
                p = new pack(self, self.dist);
                pass p->chld;
                l-=self;
        ...}
...}
```

3.2.1.3 The “pass … -> ...” Statement

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

```
pass pkt -> {node|parents|children|all}
```

, where *pkt* has to be of the “pack” type defined in the “pack” block, transfers *pkt* from the current node to a particular neighbor or a list of neighbors, which is processed by the catch function of the receiving node. The user can also use the keywords “prnt,” “chld,” “chld_prnt,” or “prnt_chld” to refer to specific groups of neighbors and in which order of the group *pkt* is to be sent. The pass statement may only be used in object functions.

The highlighted portion of the do function passes the packet “p” to and invokes the catch function of each child.

```
node:Dijkstra{
        ...
        void do (list node:d l){
                pack p;
                self.visited = true;
                p = new pack(self, self.dist);
                pass p->chld;
                l-=self;
        ...}
...}
```

3.2.1.4 The “msg” Keyword

The “msg” keyword is valid only within the “catch” block to refer to the message packet that was received by the node. Messages are treated like regular named tuples, whose attributes are defined in the “pack” block. The following snippet of the catch block for the Dijkstra example compares the current view of the node's distance to the origin with the distance of the path via the sender of the node and updates the shortest path info.

```
node:Dijkstra{
        ...
        catch{
                if (self.visited)
                        return;
                if (msg.sender.dist + msg.sender[self].w < self.dist){
                        self.dist = msg.sender.dist + msg.sender[self].w;
                        self.prev = msg.sender;
                }
        }
...}
```

3.2.2 Constructing and Using Nodes

Once a node type is declared, users can create instances of each node type by calling constructors. Node constructors are syntactically same with the tuple constructor except that the tuple type is replaced by the node type. The arguments to a constructor must appear in the order in which members are declared in the "data" block of the node type declaration. A constructor returns a reference to the node instance. The following statement declares a reference to a node:Dijkstra called n, initializes a node:Dijkstra instance, and store the reference to that instance in n. All node instances must be accessed through its reference.

```
node:Dijkstra n;
n = new node:Dijkstra(0, 100000, null(node:Dijkstra), false);
```

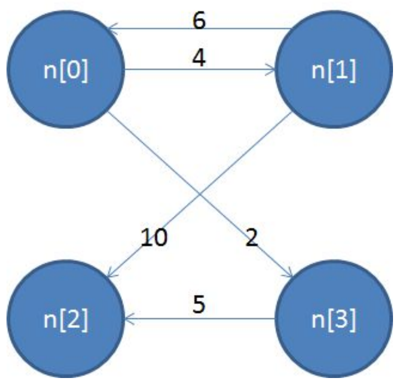
3.2.3 Constructing Graphs

A graph is constructed by taking nodes of the same type and connecting them with edges that can hold data attributes. A graph constructor takes the form:

```
new graph(node:somenode)[| edge[src_node->dst_node](arg1,arg2,...), ... |];
```

, where the arguments are used to initialize the attributes of the edge (declared in the "edge" block of the node declaration) that connects *src_node* to *dst_node*. A graph must contain one and only one type of nodes.

Consider, for example, the following directed graph of node:A.



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

```
list node:A n;
int i;

n = new node:A[| |];
for (i = 0; i < 4; i = i + 1)
        n += new node:A(...);

new graph(node:A)[|
        edge[n[0]->n[1]](4);
        edge[n[0]->n[3]](2);
        edge[n[1]->n[0]](6);
        edge[n[1]->n[2]](10);
        edge[n[3]->n[2]](5);
|];
```

3.2.3 Using Nodes and Graph

Once nodes are constructed using the node constructor and interconnected using the graph constructor, its members, edges, and object functions can be utilized. Following keywords and operators are used to manipulate graphs and nodes.

3.2.3.1 The "." operator

The members of a node can be accessed by "." operator as in "*node_name.member*" just as in tuple member access.

3.2.3.2 The "[","]" operator

The subscript operator is used to access the edge of two connected nodes.

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].w))) // this prints 4
```

3.2.3.3 The *run* keyword

The statement "*`run{node}(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 the graph algorithm and the exchange of message packets that are involved.

For example, the following simple 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

```
tuple:msg{int k;}

node:SimpleNode{
    data{
        int num;
    }

    void do(int k){
        self.num = self.num + k;
        pass (new tuple:msg(k)) -> children;
    }
    catch{
        print("adding %d to %d" % (msg.add, self.num));
        self.num = self.num + msg.add;
    }
}

list node:SimpleNode nodes;
list tuple:edge edges
graph node:SimpleNode g;

add (new node:SimpleNode(1)) to nodes;
add (new node:SimpleNode(2)) to nodes;
edges = [new edge(nodes[0], nodes[1])];
g = new node: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"
```

3.3 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.

3.3.1 Primary expressions

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

3.3.1.1 *identifier*

The identifier refers to a previously declared and initialized identifier.

3.3.1.2 *constant, null(type)*

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(type)* represents a constant null value of type.

3.3.1.3 (*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.

3.3.1.4 *expression* [*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 also be a node, and the primary expression is the edge from left expression to the right expression.

3.3.1.5 *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.

3.3.2 Unary operators

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

3.3.2.1 -*expression*

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

3.3.2.2 !*expression*

The ! operator denotes boolean negation for boolean types. When the expression is a node, this this operator returns true if the node is a leaf node (has no children)

3.3.2.3 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.

3.3.2.4 *expression* * *expression*, *expression* *. *expression*

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

3.3.2.5 *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.

3.3.2.6 *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.

3.3.3 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.

3.3.3.1 *expression* + *expression*, *expression* - *expression*

Above are arithmetic add/subtract operators. The types of expressions must be both int or both float.

3.3.4 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.

3.3.4.1 *expression* < *expression*, *expression* > *expression*

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

3.3.4.2 *expression* <= *expression*, *expression* >= *expression*

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

3.3.5 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.

3.3.5.1 *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. Reference types (user defined types) can be compared as well.

3.3.6 Boolean AND

3.3.6.1 *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.

3.3.7 Boolean OR

3.3.7.1 *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.

3.3.8 List Add and List Remove

3.3.8.1 *expression* += *expression*

The += operator adds the RHS to the LHS, which must be a list holding variables of the same type as the RHS. This operator returns the reference to the list. List addition can be done on lists of any type.

3.3.8.2 *expression* -= *expression*

The -= operator removes the RHS to the LHS, which must be a list holding variables of the same type as the RHS, or does nothing if LSH doesn’t contain RHS. This operator returns the reference to the list. List remove can only be done on lists of user defined type references.

3.3.9 Assignment

3.3.9.1 *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.

3.4 Statements

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

3.4.1 Conditional

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

3.4.1.1 single conditional statement

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

3.4.1.2 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.)

3.4.2 while

3.4.2.1 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.

3.4.3 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.

3.4.4 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.

3.4.5 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.

3.5 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");
}
```

3.6 Standard Library

GOBLAN allows for a very limited access to standard libraries. It is designed to be a standalone language without much reliance on external libraries. The only pre-defined functionality is the "print" function, which prints to standard out using a c-style format string.

3.7. Program Structure

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

The *main* function 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 `node:d`. After adding constructed node objects to the list, it then builds a graph by connecting these nodes with edges. 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 the length of the list. This function is specifically used in the while loop statement for computing the Dijkstra's algorithm. `min_dist_node` takes a list of `node:d`s and returns a `node:d` that has the smallest dist value.

The block `node:d` is a node declarator. Each constructed will have data `dist` of type int and `prev` of type `node:d`. They represent current estimates of the shortest distance from the source and the previous node. Every edge between two `node:d`s would contains weight "w." Instances of `node:d` communicate with each other their current view of the optimal distance from the origin through message packets, and these packets contain the sender info and the distance. 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.

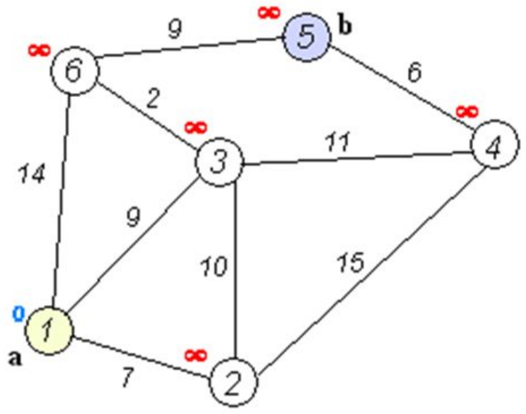
The outermost scoping is the whole program. Functions, graphs, and main create their own scope, and they are enclosed in curly brackets. For loops, conditional statement, and while loop have their own local scope. When an inner scope update its outer scope's declared variable, that variable is overridden.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

3.8 Sample Program

There are a few examples of graph programs in GOBLAN/example_code.

3.8.1 Dijkstra's Algorithm

The following program creates the following undirected graph, searches for the shortest path from node[0] to node[4], and prints it in reverse order. (The indices in the picture is greater than our code by 1)



```
node:d{
       data{
              int id;
              int dist;
              node:d prev;
              bool visited;
       }
       edge{
              int w;
       }
       pack{
              node:d sender;
              int dist;
       }
       void do (list node:d l){
              pack p;

              self.visited = true;
              p = new pack(self, self.dist);
              pass p->chld;
              l-=self;
       }
       catch{
              if (self.visited)
                     return;
              if (msg.sender.dist + msg.sender[self].w < self.dist){
                     self.dist = msg.sender.dist + msg.sender[self].w;
```

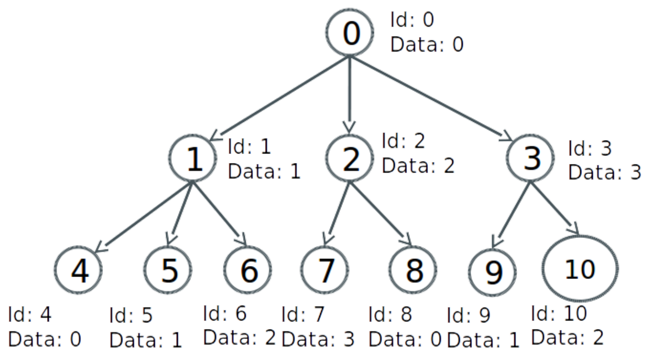
```
                      self.prev = msg.sender;
               }
       }
}

int len(list node:d l){
       node:d n;
    int i;
    i = 0;

    for (n in l)
       i = i + 1;
    return i;
}

node:d min_dist_node(list node:d l){
       node:d n;
       node:d min;

       if (len(l) == 0)
               return null(node:d);
       min = l[0];
       for (n in l){
               if (n.dist < min.dist)
                      min = n;
       }
       return min;
}

int main(){
       list node:d nodes;
       list node:d l;
       node:d n;
       node:d dst;
       node:d next;
       int i;

       nodes = new node:d [|new node:d(0,0,null(node:d),false)|];
       l = new node:d [|nodes[0]|];
       for (i = 1; i < 6; i = i + 1){
               n = new node:d(i,100000,null(node:d),false);
               nodes += n;
               l += n;
       }

       new graph(node:d)[|
               edge[nodes[0]->nodes[1]](7), edge[nodes[1]->nodes[0]](7),
               edge[nodes[0]->nodes[2]](9), edge[nodes[2]->nodes[0]](9),
               edge[nodes[0]->nodes[5]](14),       edge[nodes[5]->nodes[0]](14),
               edge[nodes[1]->nodes[2]](10),       edge[nodes[2]->nodes[1]](10),
               edge[nodes[1]->nodes[3]](15),       edge[nodes[3]->nodes[1]](15),
               edge[nodes[2]->nodes[3]](11),       edge[nodes[3]->nodes[2]](11),
               edge[nodes[2]->nodes[5]](2), edge[nodes[5]->nodes[2]](2),
               edge[nodes[3]->nodes[5]](6), edge[nodes[5]->nodes[3]](6),
               edge[nodes[4]->nodes[5]](9), edge[nodes[5]->nodes[4]](9)
       |];

       dst = nodes[4];
       while (len(l) > 0){
               next = min_dist_node(l);
               run{next}(l);
       }
       while (dst != null(node:d)){
               print("node[%d]\n",dst.id);
               dst = dst.prev;
```

```
        }
}
```

```
>> ./goblan.sh dijkstra
node[4]
node[5]
node[2]
node[0]
```

3.8.2 Tree Search Algorithm

The following program creates the following undirected tree and searches for the paths to the nodes containing data=2. (The indices in the picture is greater than our code by 1)



```
node:bin{
        data{
                int id;
                int d;
                list list node:bin l;
        }
        edge{}
        pack{
                node:bin sender;
                bool is_query;
                int k;
                list node:bin l;
        }
        list list node:bin do(int k){
            pack p;

            p = new pack(self,true,k,null(list node:bin));
            pass p->chld;
            return self.l;
        }
        catch{
                list node:bin l;
                pack p;

                if (msg.is_query){
                   if (self.d == msg.k){
                      p = new pack(self, false, msg.k, new node:bin [|self|]);

                      pass p->prnt;
                   }
                   if (!!self)
                      run{self}(msg.k);
                }
                else{
                        msg.l += self;
                        self.l += msg.l;
                        p = new pack(self,false,0,msg.l);
                        pass p->prnt;
                }
        }
}
node:bin sender;

int main()
{
        list node:bin l;
        list int ll;
        node:bin n;
        int i;

        l = new node:bin [||];
        for (i = 0; i < 20; i = i + 1)
            l += new node:bin(i, i%4, new node:bin [||]);

        new graph(node:bin)[|
            edge[l[0]->l[1]](),
            edge[l[0]->l[2]](),
            edge[l[0]->l[3]](),
            edge[l[1]->l[4]](),
            edge[l[1]->l[5]](),
            edge[l[1]->l[6]](),
            edge[l[2]->l[7]](),
            edge[l[2]->l[8]](),
            edge[l[3]->l[9]](),
            edge[l[3]->l[10]](),
            edge[l[3]->l[4]]()
        |];
        for (l in run{l[0]}(2)){
            print("----\n");
            for (n in l)
                    print("%d\n",n.id);
        }
}
```

```
>> ./goblan.sh search.gb
----
6
1
0
----
2
0
----
10
3
0
```

3.9 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
`null`	null generator
`data`	Data block declarator
`edge`	Edge type declarator / edge constructor
`pack`	Packet type declarator
`do`	Asynchronous function keyword
`catch`	Synchronous function keyword
`self`	Self-referential variable
`msg`	Message packet
`new`	Heap allocation
`run`	Asynchronous function caller
`Prnt, chld, prnt_chld, chld_prnt`	Message target specifiers
`for`	For loop
`while`	While loop
`if`	Conditional statement
`else`	Conditional else
`in`	In keyword

4. Project Plan

GOBLAN has a set of types and literals which are similar to those of many existing programming languages. Its specifications resemble most of the compiled languages. Types in GOBLAN can be largely classified into primitive types and non-primitive (user-defined) types. List, named tuples and nodes fall into the latter. While primitive data types are passed by value, non-primitive data types are always passed by reference. All user defined types are heap-allocated and thus can be reached at any point in the program.

4.1 Planning and Scheduling

The GOBLAN team held meetings twice a week. One meeting usually involved recapping the work team members did, the progress that was made, and setting an agenda for the next several days. The second meeting was usually to discuss design and implementation details and resolve any issues that may have come up while working.

4.2 Design and Specifications

Given that the idea came out of the message passing algorithm in statistical inference, the first specification which we decided on was the sending and catching of packets. This then influenced our blueprint for node types, which contain data, packets, do function, and catch function. As we begun writing the grammar, scanner, parser, and scanner, other issues which needed design decisions came up. Defining the scope and limitations of user defined types along with lists was a big hurdle we overcame in the process. Obviously, as we further implemented our compiler and semantic checker, specifications were adapted in order to better fit the overall flow of the language and enable easier coding.

4.3 Development And Testing

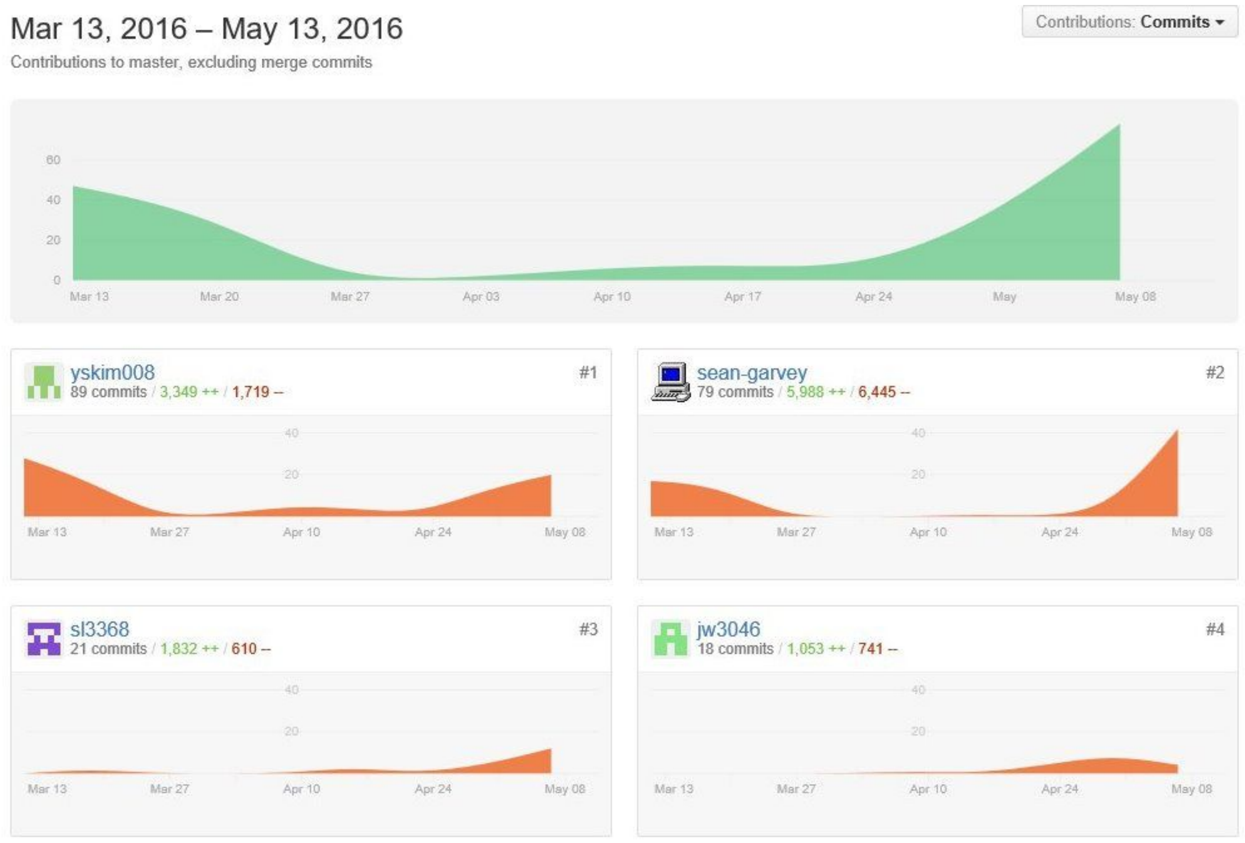
Developing and Testing our codegen and semantic checker was an iterative process. With each step of implementation or development, new tests, however simple they may be, were written. Yunsung first focused on building the fundamental blocks of codegen. With this basic milestone complete, the others in the group were able to work on their respective parts. Most of the functionality was broken up into discrete tasks, such as: checking for member access or compiling function calls within a node do block. As these tasks were iteratively completed, tests were written in the testing suite, and upon success, pushed to the master repository. This was beneficial in many instances, since the work of another team member would usually expose a bug in some other part of the code. Ultimately we wanted clear and unique examples of GOBLAN programs, some of the final tests, were writing up more complex algorithms (such as Dijkstra’s or sum-product) in GOBLAN and testing whether many different features in conjunction would succeed.

4.4 Team Roles and Responsibilities

Once the compiler front end (lexer and the parser) was complete, the roles of each member were divided roughly into two: architecture design & code generation (Yunsung Kim, Jee Hyun Wang) and semantic checking (Sean Garvey, Sameer Lal). Nonetheless, the roles were quite flexible and each member contributed to multiple parts of the project towards the end.

Member	Responsibility
Yunsung Kim	Team management, Language design, system architecture design, compiler front end (parser), code generation (LLVM IR), test case creation
Sean Garvey	Compiler front end (scanner, utility functions), semantic checking, test case creation
Sameer Lal	Semantic checking, test case creation
Jee Hyun Wang	Code generation (internal data structure support), module linking

Following is the graph indicating the contribution statistics.



4.4 Project Timeline

Following is the timeline of our project.

Date	Milestone

February 10	Language proposal
March 17	Language reference manual c
March 27	Compiler front end (lexer and parser) complete
April 6	Hello World runs
May 4	Code generation complete
May 10	Semantic checking complete. Test cases written
May 11	Final report complete

4.5 Software Environment

- Programming language for the compiler:
  - Codegen: Ocaml version 4.00.1.
  - Parsing & Lexing: Ocamlyacc, Ocamllex extensions
  - Internal data structures: C
- Software used for module generation:
  - Lli: main LLVM interpreter
  - Llvm-link: linking bitcode of the internal data structure with the LLVM IR
  - Clang: generating LLVM module for internal data structure & compiling executable
- Miscellaneous:
  - Operating systems: Ubuntu 10.3, Mac.
  - Team management utilities: Slack, email

4.6 Programming Style Guide

Our group aimed to follow some general guidelines regarding programming style. We aimed to comment as much of code as possible throughout the development process. In Ocaml this can be tricky sometimes, we tried to write comments wherever possible in line, and subsequently write some quick points for blocks of entire code. The second main guideline was using spaces instead of tabs, so that there would be no issues across editors.

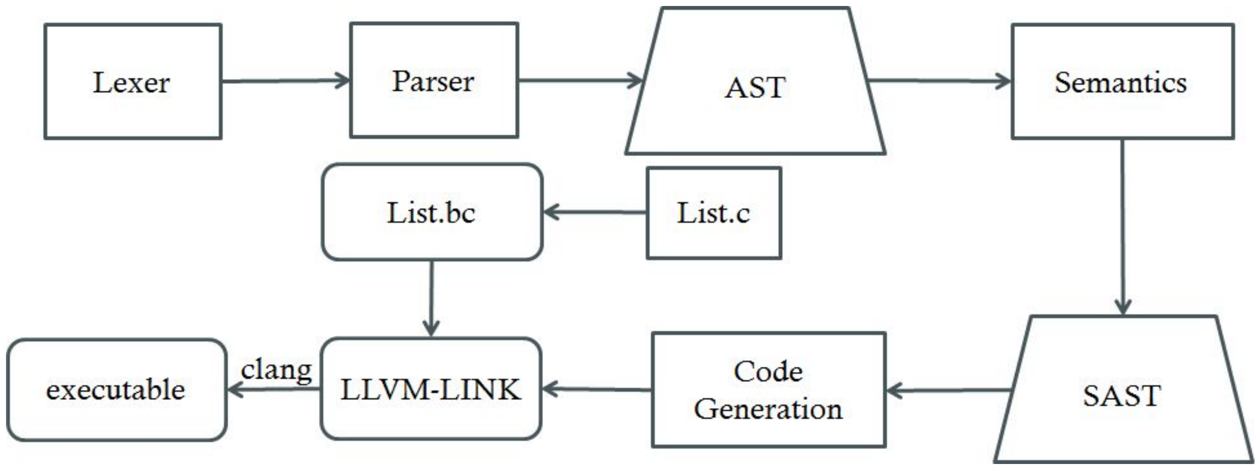
4.7 Project Log

We followed the intended project timeline described in section 4.4. For a more detailed sense of our project progress over time, our github repository is located at: https://github.com/sean-garvey/GOBLAN.

5. Architectural Design

5.1 Compiler

The GOBLAN compiler has the standard structure learned in class. The front end contained the lexer, scanner, and parser, while the back end had the semantic checker and the code generation. In addition to these components, we constructed some C modules for list data structures and linked these to the LLVM IR that was output from our codegen. Refer to the diagram below for a sense of the compiler structure.



5.1.1 Lexer (Yungsung, Sean)

The lexer takes in the GOBLAN source code as standard input and parses it into specific tokens of comments, identifiers, keywords, operators, nodes, functions, and tuples. The rules for this are parsing are described in the language manual.

5.1.2 Parser (Yunsung, Sean)

The parser component of the compiler took the tokens generated by the lexer, and then constructed the AST based on the GOBLAN grammar.

5.1.3 Semantic Checker (Sameer, Sean)

The semantic checker takes in the AST constructed by the parser and checks for semantic errors in the source code. This involves checking that there are no type mismatches, proper node declarations, valid function calls, and proper scopes for various keywords. On a higher level, the checker cycles through the various sections of the AST (functions, node declarations, tuples, and globals) and recursively evaluates each statement of code. The checker ensures that the input AST is semantic correct, and raises a type specific exception if any issues exists.

5.1.4 Code Generator (Yunsung)

The codegen takes in the semantically correct AST and produces the necessary LLVM IR. It utilizes the ocaml-LLVM binding and builds basic blocks. The ocaml-LLVM binding allows the construction of basic blocks line by line.

5.2 External Module Linking - Lists (Jee)

GOBLAN internally uses lists as its core data structure. A linked list customized for GOBLAN was implemented in C and compiled into LLVM bitcode (.bc) using “clang.” The LLVM bitcode is linked with the output file through llvm-link, which outputs an LLVM IR that is run through the LLVM interpreter.

6. Test Plan

6.1 Automation and Testing Suite

For our testing process, we built one suite for all features. All tests are placed in a single folder. There is a script (testall.sh) which automates the compilation, execution, and output comparison for each test.

6.2 Tests Types

Our test suite consists of over 200 units tests to ensure that our compiler is working as intended. We choose to implement our test suite using unit tests as we intend to test only a single aspect of the compiler at a time, so that when a test fails, we can easily identify the circumstances surrounding the failure and resolve the issue by either rolling back the changes that caused the failure, or updating the code to prevent the error.

In our test suite, there are two types of unit tests: tests which test for failure and tests which test for success. Testing for failure is performed to ensure that the semantic checker component of the compiler is indeed identifying and throwing exceptions for semantic errors as outlined by our language reference manual. Tests which test for success are to ensure that the code generation component of the compiler is properly producing functional code. These two types of tests in combination can be used to make sure that our compiler produces valid programs.

6.3 Algorithm Tests

In order to do end to end testing of many different combinations of GOBLAN functionality, large tests were written mimicking some more applied examples. There is a tree graph search, which constructs a graph of a tree, and a query is sent to the tree to find the nodes with the specific data. Another test is implemented for finding the shortest path in a graph using Dijkstra's algorithm (Yunsung). One that leans on the original inspiration of the language is applying the sum product algorithm to a basic tree graphical model with defined cliques (Sameer). Running these tests would bring minor, yet important, bugs to light.

7. Lessons Learned

7.1 Yunsung

At first, writing in LLVM - let alone compiling a new language to it - seemed very cryptic, but I began to realize the fun and excitement of creating a new language and seeing how this enables algorithms to be implemented in different programming paradigms. Building a compiler seems to be less about the technical details of translating every tiny bit of code, but more about creating the a grander arena in which users would be allowed to think and program differently and more creatively. It's all about thinking outside the box of “conventional” programming languages, and how to reconcile its shortcomings.

7.2 Jee

I realized that compilers are quite intricate structures. Not only did we have to build a new language down to LLVM, but we also had to build our own modules to implement internal data structures that were used in code generation. It was the first time that I was involved in a back-end oriented project as big as this one, and I realized how important it is to communicate with the people in the group to get things done efficiently and effectively.

7.3 Sean

Lessons Learned

Murphy's law applies. This project is further testament to the reality of all programming projects, especially group projects: that they always fall behind. Unpredictable and unforseen obstables inevitably popped up, delaying the project schedule. Starting early and allocating more than enough time to complete each task, is better than being in the opposite situation.

Advice

Read the MicroC compiler. Then read it again. Until you understand each component inside and out. This example was an incredibly useful learning aid for my entire team. As for OCaml, if it's your first functional programming language, it'll be a challenge, but keep at it. You'll learn the syntax and paradigm slowly, but eventually, much like how your code in OCaml suddenly works, the paradigm 'clicks' and you'll begin programming, and thinking, just as naturally as you do in your favorite imperative language. When you're stuck, refer to Edwards OCaml slides, Google it, and/or play around with the language in interactive mode.

7.4 Sameer

As I began to get an understanding of how compilers work, I learned that creating languages are less about syntax and functionality and more about creating a way to think about problems. With GOBLAN I was forced to think about algorithms in a different context, and in many ways the implementation was completely different than what I had done traditionally. Obviously the framework works better for some types of problems more than others, but like Professor Edwards mentioned many times in class, “you can’t fit a square peg in a round hole.” In many ways the language makes it easier to think about a problem and thus implement it. From a project standpoint, I learned that you can really only get a deep knowledge of how things work by starting to hack. I attempted to read through the code and learn the structure, but only until I actually faced the task of building things, did I learn how different components were acting.

Scanner

```
(* Ocamllex scanner for GOBLAN *)

{ open Parser }

let exp = ('e'|'E')('+'|'-')?['0'-'9']+

rule token = parse
  [' ' '\t' '\r' '\n']              { token lexbuf }     (* Whitespace *)
| "/*"                              { comment lexbuf }   (* Comments *)
| "//"                              { slcomment lexbuf } (* Comments *)
| '#'                               { slcomment lexbuf } (* Comments *)
| '"'                                { read_string (Buffer.create 16) lexbuf }
| '('                               { LPAREN }
| ')'                               { RPAREN }
| '{'                               { LBRACE }
| '}'                               { RBRACE }
| "[|"                              { LLIST }
| "|]"                              { RLIST }
| '['                               { LBRACKET }
| ']'                               { RBRACKET }
| ';'                               { SEMI }
| ','                               { COMMA }
| '.'                               { PERIOD }
| '+'                               { PLUS }
| '-'                               { MINUS }
| '*'                               { TIMES }
| '/'                               { DIVIDE }
| '%'                               { MODULO }
| '='                               { ASSIGN }
| "+="                              { LSTADD }
| "-="                              { LSTRMV }
| "=="                              { EQ }
| "!="                              { NEQ }
| '<'                               { LT }
| "<="                              { LEQ }
| ">"                               { GT }
| ">="                              { GEQ }
| "&&"                              { AND }
| "||"                              { OR }
| "!"                               { NOT }
| "->"                              { ARROW }
| "if"                              { IF }
| "else"                            { ELSE }
| "for"                             { FOR }
| "in"                                { IN }
| "while"                             { WHILE }
| "return"                            { RETURN }
| "bool"                              { BOOL }
| "int"                               { INT }
| "float"                             { FLOAT }
| "string"                            { STRING }
| "list"                              { LIST }
| "new"                               { NEW }
| "void"                              { VOID }
| "graph"                             { GRAPH }
| "true"                              { TRUE }
| "false"                             { FALSE }
| "data"                              { DATA }
| "edge"                              { EDGE }
| "pack"                              { PACK }
| "do"                                { DO }
| "catch"                             { CATCH }
| "self"                              { SELF }
| "prnt"                              { PARENT }
| "chld"                              { CHILD }
| "prnt_chld"                         { PRNTCHLD }
| "chld_prnt"                         { CHLDPRNT }
| "msg"                               { MESSAGE }
| "pass"                              { PASS }
| "run"                               { RUN }
| "null"                              { NULL }
| "print"                             { PRINT }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm
                                      { ID(lxm) }
| ('.'['0'-'9']+ exp?|['0'-'9']+('.'['0'-'9']* exp? | exp)) as lxm
                                      { FLT_LIT(float_of_string lxm) }
| ['0'-'9']+ as lxm                   { INT_LIT(int_of_string lxm) }
| "node:"['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm
                                      { NODE_TYP(lxm) }
| "tuple:"['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm
                                      { TUPLE_TYP(lxm) }
| eof                                 { EOF }
| _ as char                           { raise (Failure("illegal character " ^
                                        Char.escaped char))
                                      }

and read_string buf = parse
  | '"'       { STR_LIT(Buffer.contents buf) }
  | '\\' '/'  { Buffer.add_char buf '/'; read_string buf lexbuf }
  | '\\' '\\' { Buffer.add_char buf '\\'; read_string buf lexbuf }
  | '\\' 'b'  { Buffer.add_char buf '\b'; read_string buf lexbuf }
  | '\\' 'f'  { Buffer.add_char buf '\012'; read_string buf lexbuf }
  | '\\' 'n'  { Buffer.add_char buf '\n'; read_string buf lexbuf }
  | '\\' 'r'  { Buffer.add_char buf '\r'; read_string buf lexbuf }
  | '\\' 't'  { Buffer.add_char buf '\t'; read_string buf lexbuf }
  | '\\' '"'  { Buffer.add_char buf '\"'; read_string buf lexbuf }
  | [^ '"' '\\']+
    { Buffer.add_string buf (Lexing.lexeme lexbuf);
      read_string buf lexbuf
    }
  | _ { raise (Failure("Illegal string character: " ^ Lexing.lexeme lexbuf)) }
  | eof { raise (Failure("String is not terminated")) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }

and slcomment = parse
  '\n' { token lexbuf }
| _    { slcomment lexbuf }
```

Parser

```
/* Ocamlyacc parser for GOBLAN */

%{
open Ast;;
let parse_error s = (* Called by the parser function on error *)
  print_endline s;
  flush stdout;;
let get1 (a,_,_,_) = a;;
let get2 (_,a,_,_) = a;;
let get3 (_,_,a,_) = a;;
let get4 (_,_,_,a) = a;;

let add_self n_typ fdecl =
  { f_typ     = fdecl.f_typ;
    f_name    = n_typ ^ "." ^ fdecl.f_name;
    f_formals = (NodeTyp(n_typ), "self")::fdecl.f_formals;
    f_locals  = fdecl.f_locals;
    f_body    = fdecl.f_body }

let add_self_pack n_typ fdecl =
  { f_typ     = fdecl.f_typ;
    f_name    = n_typ ^ "." ^ fdecl.f_name;
    f_formals = (NodeTyp(n_typ), "self")::(PackTyp(n_typ), "msg")::fdecl.f_formals;
    f_locals  = fdecl.f_locals;
    f_body    = fdecl.f_body }
%}

%token LPAREN RPAREN LBRACE RBRACE LLIST RLIST LBRACKET RBRACKET
%token SEMI COMMA PERIOD PLUS MINUS TIMES DIVIDE MODULO
%token ASSIGN EQ NEQ LT LEQ GT GEQ AND OR NOT ARROW LSTADD LSTRMV
%token IF ELSE FOR IN WHILE RETURN PRINT
%token BOOL INT FLOAT STRING LIST NEW VOID EDGE PACK GRAPH
%token TRUE FALSE DATA DO CATCH SELF PARENT CHILD PRNTCHLD CHLDPRNT MESSAGE
%token PASS RUN NULL LSTADD LSTRMV

%token <string> ID
%token <float>  FLT_LIT
%token <int>    INT_LIT
%token <string> STR_LIT
%token <string> NODE_TYP TUPLE_TYP
%token <string> STR
%token EOF

%nonassoc NOELSE
```

```
%nonassoc ELSE
%right ASSIGN
%left LSTADD LSTRMV
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%nonassoc LBRACKET RBRACKET
%left NOT NEG
%left PERIOD

%start program
%type <Ast.program> program

%%

program:
  decls EOF                           { $1 }

decls:
   /* nothing */                      { [], [], [], [] }
 | decls ndecl                        { ($2 :: get1 $1), get2 $1, get3 $1, get4 $1 }
 | decls tdecl                        { get1 $1, ($2 :: get2 $1), get3 $1, get4 $1 }
 | decls vdecl                        { get1 $1, get2 $1, ($2 :: get3 $1), get4 $1 }
 | decls fdecl                        { get1 $1, get2 $1, get3 $1, ($2 :: get4 $1) }

vdecl:
    typ ID SEMI { ($1, $2) }

vdecl_list:
    /* nothing */                     { [] }
  | vdecl_list vdecl                  { $2 :: $1 }

typ:
    INT                               { Int }
  | BOOL                              { Bool }
  | VOID                              { Void }
  | STRING                            { Str }
  | FLOAT                             { Float }
  | NODE_TYP                          { NodeTyp($1) }
  | PACK                              { Packet }
  | LIST typ                          { ListTyp($2) }
  | TUPLE_TYP                         { TupleTyp($1) }

ndecl:
   NODE_TYP LBRACE n_data n_edge n_pack n_do n_catch RBRACE
```

```
                                         { { n_typ = $1;
                                             n_data = $3;
                                             n_edge = $4;
                                             n_pack = $5;
                                             n_do = add_self $1 $6;
                                             n_catch = add_self_pack $1 $7 } }

n_data:
   DATA LBRACE vdecl_list RBRACE       { List.rev $3 }

n_edge:
   EDGE LBRACE vdecl_list RBRACE       { List.rev $3 }

n_pack:
   PACK LBRACE vdecl_list RBRACE       { List.rev $3 }

n_do:
   typ DO LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
                                         { { f_typ = $1;
                                             f_name = "do";
                                             f_formals = $4;
                                             f_locals = (List.rev $7);
                                             f_body = List.rev $8 } }

n_catch:
   CATCH LBRACE vdecl_list stmt_list RBRACE
                                         { { f_typ = Void;
                                             f_name = "catch";
                                             f_formals = [];
                                             f_locals = List.rev $3;
                                             f_body = List.rev $4 } }

tdecl:
    TUPLE_TYP LBRACE vdecl_list RBRACE { { t_typ = $1;
                                             t_attributes = List.rev $3 } }

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
                                        { { f_typ = $1;
                                            f_name = $2;
                                            f_formals = $4;
                                            f_locals = List.rev $7;
                                            f_body = List.rev $8 } }
formals_opt:
    /* nothing */                        { [] }
  | formal_list                          { List.rev $1 }

formal_list:
    typ ID                              { [($1,$2)] }
  | formal_list COMMA typ ID            { ($3,$4) :: $1 }

stmt_list:
    /* nothing */                       { [] }
  | stmt_list stmt                      { $2 :: $1 }

pass_typ:
    PARENT    { Prnt }
  | CHILD     { Chld }
  | PRNTCHLD  { PrntChld }
  | CHLDPRNT  { ChldPrnt }
  | expr      { Target($1) }

stmt:
    expr SEMI                           { Expr $1 }
  | LBRACE stmt_list RBRACE             { Block(List.rev $2) }
  | RETURN SEMI                         { Return Noexpr }
  | RETURN expr SEMI                    { Return $2 }
  | IF LPAREN expr RPAREN stmt %prec NOELSE
                                        { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt
                                        { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                        { For($3, $5, $7, $9) }
  | FOR LPAREN expr IN expr RPAREN stmt
                                        { ForEach($3, $5, $7)}
  | WHILE LPAREN expr RPAREN stmt
                                        { While($3, $5) }
  | PASS expr ARROW pass_typ SEMI       { Pass($2, $4) }

expr_opt:
    /* nothing */                       { Noexpr }
  | expr                                { $1 }

expr:
    expr ASSIGN  expr                   { Assign($1, $3) }
  | expr PLUS    expr                   { Binop($1, Add, $3) }
  | expr MINUS   expr                   { Binop($1, Sub, $3) }
  | expr TIMES   expr                   { Binop($1, Mult, $3) }
  | expr DIVIDE  expr                   { Binop($1, Div, $3) }
  | expr MODULO  expr                   { Binop($1, Mod, $3) }
  | expr EQ      expr                   { Binop($1, Equal, $3) }
  | expr NEQ     expr                   { Binop($1, NEqual, $3) }
  | expr LT      expr                   { Binop($1, Less, $3) }
  | expr LEQ     expr                   { Binop($1, Leq, $3) }
  | expr GT      expr                   { Binop($1, Greater, $3) }
  | expr GEQ     expr                   { Binop($1, Geq, $3) }
  | expr AND     expr                   { Binop($1, And, $3) }
  | expr OR      expr                   { Binop($1, Or, $3) }
  | expr LSTADD  expr                   { ListAdd($1, $3) }
  | expr LSTRMV  expr                   { ListRmv($1, $3) }
  | TRUE                                { BoolLit(true) }
  | FALSE                               { BoolLit(false) }
  | ID LPAREN actuals_opt RPAREN        { Call($1, $3) }
  | FLT_LIT                             { FloatLit($1) }
  | ID                                  { Id($1) }
  | INT_LIT                             { IntLit($1) }
  | NEW typ LLIST actuals_opt RLIST     { Lst($2, $4) }
  | expr PERIOD  ID                     { Member($1, $3) }
  | MESSAGE                             { Message }
  | NEW NODE_TYP LPAREN actuals_opt RPAREN
                                        { Node($2, $4) }
  | NEW GRAPH LPAREN NODE_TYP RPAREN LLIST edge_opt RLIST
                                        { Graph(NodeTyp($4), $7) }
  | NEW PACK LPAREN actuals_opt RPAREN { Pack($4) }
  | NULL LPAREN typ RPAREN              { Null($3) }
  | PRINT LPAREN STR_LIT print_actuals RPAREN
                                        { Print($3, List.rev $4) }
  | RUN LBRACE expr RBRACE LPAREN actuals_opt RPAREN
                                        { Run ($3,$6) }
  | SELF                                { Self }
  | STR_LIT                             { StrLit($1) }
  | expr LBRACKET expr RBRACKET         { Subscript($1, $3) }
  | NEW TUPLE_TYP LPAREN actuals_opt RPAREN
                                        { Tuple($2, $4) }
  | MINUS expr %prec NEG                { Unop(Neg, $2) }
  | NOT          expr                   { Unop(Not, $2) }
  | LPAREN expr RPAREN                  { $2 }

edge_opt:
    /* nothing */                       { [] }
  | edge_list                           { List.rev $1 }

edge_list:
    edge_decl                           { [$1] }
  | edge_list COMMA edge_decl           { $3 :: $1 }

edge_decl:
  EDGE LBRACKET expr ARROW expr RBRACKET LPAREN actuals_opt RPAREN
                                        { ($3,$5,$8) }

print_actuals:
    /* nothing */                       { [] }
  | COMMA actuals_list                  { $2 }
```

```
actuals_opt:
    /* nothing */                     { [] }
  | actuals_list                      { List.rev $1 }

actuals_list:
    expr                              { [$1] }
  | actuals_list COMMA expr           { $3 :: $1 }
```

Abstract Syntax Tree

```
(* Abstract Syntax Tree and functions for printing it *)

type op =
    Add
  | And
  | Div
  | Equal
  | Geq
  | Greater
  | Leq
  | Less
  | Mod
  | Mult
  | NEqual
  | Or
  | Sub

type uop =
    Neg
  | Not

type typ =
    Bool
  | EdgeTyp of string
  | Float
  | Int
  | ListTyp of typ
  | NodeTyp of string
  | PackTyp of string
  | Packet
  | Str
  | TupleTyp of string
  | Void

type bind = typ * string

type expr =
    Assign of expr * expr
  | Binop of expr * op * expr
  | BoolLit of bool
  | Call of string * expr list
  | FloatLit of float
  | Id of string
  | IntLit of int
  | Lst of typ * expr list
  | ListAdd of expr * expr
  | ListRmv of expr * expr
  | Member of expr * string
  | Message
  | Node of string * expr list
  | Graph of typ * (expr * expr * expr list) list
  | Noexpr
  | Null of typ
  | Pack of expr list
  | Print of string * expr list
  | Run of expr * expr list
  | Self
  | StrLit of string
  | Subscript of expr * expr
  | Tuple of string * expr list
  | Unop of uop * expr

type pass_typ =
    Chld
  | ChldPrnt
  | Prnt
  | PrntChld
  | Target of expr

type stmt =
    Block of stmt list
  | Expr of expr
  | For of expr * expr * expr * stmt
  | ForEach of expr * expr * stmt
  | If of expr * stmt * stmt
  | Pass of expr * pass_typ
  | Return of expr
  | While of expr * stmt

type func_decl = {
    f_typ : typ;
    f_name : string;
    f_formals : bind list;
    f_locals : bind list;
    f_body : stmt list;
}

type node_decl = {
    n_typ  : string;
    n_data : bind list;
    n_edge : bind list;
    n_pack : bind list;
    n_do : func_decl;
    n_catch : func_decl;
}

type tuple_decl = {
    t_typ : string;
    t_attributes : bind list;
}

type program = node_decl list * tuple_decl list * bind list * func_decl list

(* Pretty-printing functions *)
let rec string_of_typ = function
    Int            -> "int"
  | Bool           -> "bool"
  | Void           -> "void"
  | Str            -> "string"
  | Float          -> "float"
  | NodeTyp(name)  -> name
  | Packet         -> "Packet"
  | EdgeTyp(name)  -> "edge//"^name
  | PackTyp(name)  -> "pack//"^name
  | ListTyp(typ)   -> "list " ^ string_of_typ typ
  | TupleTyp(name) -> name

module Typ =
  struct
    type t = typ
    let compare t1 t2 =
      String.compare (string_of_typ t1) (string_of_typ t2)
    let equal t1 t2 = compare t1 t2 = 0
  end

let string_of_pass_typ = function
    Chld     -> "Child"
  | ChldPrnt -> "ChildParent"
  | Prnt     -> "Parent"
  | PrntChld -> "ParentChild"
  | Target(_)-> "Target"

let string_of_op = function
    Add       -> "+"
  | Sub       -> "-"
  | Mult      -> "*"
  | Div       -> "/"
  | Equal     -> "=="
  | NEqual    -> "!="
  | Less      -> "<"
  | Leq       -> "<="
  | Greater   -> ">"
  | Geq       -> ">="
  | And       -> "&&"
  | Or        -> "||"
  | Mod       -> "%"

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"

let rec string_of_expr = function
  | Assign(e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | FloatLit(f) -> string_of_float f
  | Graph(_, _) -> "graph"
  | Id(s) -> s
  | IntLit(i) -> string_of_int i
  | Lst(typ, el) -> "new " ^ string_of_typ typ ^ " [|" ^
      String.concat ", " (List.map string_of_expr el) ^ "|]"
  | ListAdd(e1, e2) -> string_of_expr e1 ^ " += " ^ string_of_expr e2
  | ListRmv(e1, e2) -> string_of_expr e1 ^ " -= " ^ string_of_expr e2
  | Member(e, id) -> string_of_expr e ^ "." ^ id
  | Message -> "message"
  | Node(typ, el) -> typ ^ "(" ^
      String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  | Null(typ) -> "null(" ^ string_of_typ typ ^ ")"
  | Pack(el) -> String.concat "\n" (List.map string_of_expr el)
  | Print(s, al) -> "print(\"" ^ String.escaped s ^ "\"" ^ if List.length al > 0
      then ", " ^ String.concat ", " (List.map string_of_expr al) ^ ")" else ")"
  | Run(e, bl) -> "run{" ^ string_of_expr e ^ "}(" ^
      String.concat ", " (List.map string_of_expr bl) ^ ")"
  | Self -> "self"
  | StrLit(s) -> s
  | Subscript(e1, e2) -> string_of_expr e1 ^ "[" ^ string_of_expr e2 ^ "]"
  | Tuple(typ, el) -> typ ^ "(" ^
      String.concat ", " (List.map string_of_expr el) ^ ")"
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e

let rec string_of_stmt = function
    Expr(e)
      -> string_of_expr e ^ ";"
  | Block(stmts)
      -> "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "\n}"
  | Return(expr)
      -> "return " ^ string_of_expr expr ^ ";"
  | If(e, s1, s2)
      -> "if (\n" ^ string_of_expr e ^ "\n)\n" ^ string_of_stmt s1 ^
         "\nelse\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s)
      -> "for(" ^ string_of_expr e1  ^ "; " ^ string_of_expr e2 ^ "; " ^
         string_of_expr e3  ^ ") " ^ string_of_stmt s
  | ForEach(e1, e2, s)
      -> "for(" ^ string_of_expr e1 ^ " in " ^ string_of_expr e2 ^ ") " ^
         string_of_stmt s
  | While(e, s)
      -> "while(" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | Pass (e, ptyp)
      -> "Pass(" ^ string_of_expr e ^ ")\n" ^ string_of_pass_typ ptyp

let string_of_bind (typ, id) =
  string_of_typ typ ^ " " ^ id

let string_of_formals formals =
  String.concat ", " (List.map string_of_bind formals)

let string_of_bind_list blist indent =
  if List.length blist > 0
    then indent ^ String.concat (";\n" ^ indent) (List.map string_of_bind blist)
    ^ ";\n"
  else ""

let string_of_body body indent =
  if List.length body > 0
    then indent ^ String.concat ("\n" ^ indent) (List.map string_of_stmt body)
  else ""

let string_of_func_decl func_decl =
  string_of_typ func_decl.f_typ ^ " " ^ func_decl.f_name ^ "(" ^
  string_of_formals func_decl.f_formals ^ ") {\n" ^
  string_of_bind_list func_decl.f_locals "  " ^
  string_of_body func_decl.f_body "  " ^ "\n}"

let string_of_n_do n_do =
  "do(" ^
  string_of_formals n_do.f_formals ^ ") {\n" ^
  string_of_bind_list n_do.f_locals "    " ^
  string_of_body n_do.f_body "    " ^ "\n  }"

let string_of_n_catch n_catch =
  string_of_bind_list n_catch.f_locals "    " ^
  string_of_body n_catch.f_body "    "

let string_of_node_decl node_decl =
  node_decl.n_typ ^ " {\n  data {\n" ^
  string_of_bind_list node_decl.n_data "    " ^ "  }\n  edge {\n" ^
  string_of_bind_list node_decl.n_edge "    " ^ "  }\n  pack {\n" ^
  string_of_bind_list node_decl.n_pack "    " ^ "  }\n  " ^
  string_of_n_do node_decl.n_do ^ "\n  catch {\n" ^
  string_of_n_catch node_decl.n_catch ^ "\n  }\n}"

let string_of_tuple_decl tuple_decl =
  tuple_decl.t_typ ^ " {\n" ^
  string_of_bind_list tuple_decl.t_attributes "  " ^ "}"

let string_of_program (nodes, tuples, vars, funcs) =
  (if List.length vars > 0 then
  String.concat ";\n" (List.map string_of_bind (List.rev vars)) ^ ";\n\n"
  else "") ^ (if List.length tuples > 0 then
  String.concat "\n\n" (List.map string_of_tuple_decl (List.rev tuples)) ^ "\n\n"
  else "") ^ (if List.length nodes > 0 then
  String.concat "\n\n" (List.map string_of_node_decl (List.rev nodes)) ^ "\n\n"
  else "") ^ (if List.length funcs > 0 then
  String.concat "\n\n" (List.map string_of_func_decl (List.rev funcs))
  else "") ^ "\n"
```

Semantic Checker

```
(* GOBLAN Semantic Checker *)

open Ast
module StringMap = Map.Make(String)
module TypMap = Map.Make(Typ)

let check (nodes, tuples, globals, functions) =

(* Declaring typedef  *******************************************************)

  let add_to_attr attr_dict (t,n) =
    StringMap.add n t attr_dict in

  let typedefs =
    let tuple_typedefs =
      let add_tuple_attr typedef tdecl =
        let typ_name = TupleTyp tdecl.t_typ in
        let attr_dict =
          List.fold_left add_to_attr StringMap.empty tdecl.t_attributes in
        TypMap.add typ_name attr_dict typedef in
      List.fold_left add_tuple_attr TypMap.empty (List.rev tuples) in

    let node_typedefs =
      let add_node_attr typedef ndecl =
        let n_typ_name = NodeTyp ndecl.n_typ in
        let p_typ_name = PackTyp ndecl.n_typ in
        let e_typ_name = EdgeTyp ndecl.n_typ in
        let (n_data,n_edge, n_pack) =
          (ndecl.n_data, ndecl.n_edge, ndecl.n_pack) in
        let n_attr = List.fold_left add_to_attr StringMap.empty n_data in
        let e_attr = List.fold_left add_to_attr StringMap.empty n_edge in
        let p_attr = List.fold_left add_to_attr StringMap.empty n_pack in

        let typedef = TypMap.add n_typ_name n_attr typedef in
        let typedef = TypMap.add e_typ_name e_attr typedef in
        TypMap.add p_typ_name p_attr typedef in
      List.fold_left add_node_attr tuple_typedefs (List.rev nodes)
    in node_typedefs
  in

(*****************************************************************************)

  (* Raise an exception if the given list has a duplicate *)
  let report_duplicate exceptf list =
    let rec helper = function
          n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
        | _ :: t -> helper t
        | [] -> ()
    in helper (List.sort compare list)
  in

  (* Raise an exception if a given binding is to a void type *)
  let check_not_void exceptf = function
      (Void, n) -> raise (Failure (exceptf n))
    | _ -> ()
  in

  (* Raise an exception if the binding is to a PackTyp *)
  let check_not_packet exceptf = function
      (Packet, n) -> raise (Failure (exceptf n))
    | _ -> ()
  in

  (* Raise an exception of the given rvalue type cannot be assigned to
     the given lvalue type *)
  let check_assign lvaluet rvaluet err =
     if lvaluet = rvaluet then lvaluet else raise err
  in

  (* Raies an exception if the node or tuple type is not in typedefs *)
  let check_defined_nt typ instr =
    (match typ with
        NodeTyp(_) | TupleTyp(_) ->
          if not (TypMap.mem typ typedefs) then
            raise (Failure ("In " ^ instr ^ ", undefined " ^
            "reference to " ^ string_of_typ typ))
          else
            ()
      | _ -> ()
    )
  in

(* Checking Global Variables **************************************************)

  (* Checking globals for undefined references to tuples or nodes *)
  List.iter (fun (typ, _) -> check_defined_nt typ "globals"; ()) globals;

  (* Check global variables for void types *)
  List.iter (check_not_void (fun n -> "Illegal void global " ^ n)) globals;

  (* Check global variables for pack types *)
  List.iter (check_not_packet (fun n -> "Illegal pack global " ^ n)) globals;
  (* Check global variables for duplicate names *)
  report_duplicate (fun n -> "Duplicate global " ^ n) (List.map snd globals);

(* Checking Tuples ***********************************************************)

  (* Check that there are no duplicate tuple types *)
  report_duplicate (fun n -> "Duplicate tuple " ^ n) (List.map (fun tuple ->
    tuple.t_typ) tuples);

  (* A function that is used to check each tuple (tuple) in the tuples_decl
     list *)
  let check_tuple tuple =

    (* Checks for undefined references to tuples or nodes *)
    List.iter (fun (typ, _) -> check_defined_nt typ tuple.t_typ; ())
    tuple.t_attributes;

    (* Checks for void types *)
    List.iter (check_not_void (fun n -> "Illegal void attribute " ^ n ^
      " in " ^ tuple.t_typ)) tuple.t_attributes;

    (* Checks for pack types *)
    List.iter (check_not_packet (fun n -> "Illegal pack attribute " ^ n ^
      " in " ^ tuple.t_typ)) tuple.t_attributes;

    (* Checks for duplicates *)
    report_duplicate (fun n -> "Duplicate tuple attribute " ^ n) (List.map (fun
      tuple_attr -> snd tuple_attr) tuple.t_attributes);

  in

  (* Call check_tuple on each tuple in the tuple list *)
  List.iter check_tuple tuples;

(* Checking Functions ********************************************************)

  (* Check that a function named main is defined *)
  if not (List.mem "main" (List.map (fun fd -> fd.f_name) functions))
  then raise (Failure ("Main function not found")) else ();

  (* Check that a function named print is not defined *)
  if List.mem "print" (List.map (fun fd -> fd.f_name) functions)
  then raise (Failure ("Function print may not be defined")) else ();

  (* Check that a function named node is not defined *)
  if List.mem "node" (List.map (fun fd -> fd.f_name) functions)
  then raise (Failure ("Function node may not be defined")) else ();
```

```
(* Check that a function named tuple is not defined *)
if List.mem "tuple" (List.map (fun fd -> fd.f_name) functions)
then raise (Failure ("Function tuple may not be defined")) else ();

(* Check that there are no duplicate function names *)
report_duplicate (fun n -> "Duplicate function " ^ n)
  (List.map (fun fd -> fd.f_name) functions);

(* Builds a string map (f_name --> func_decl) that contains all of the
   functions built in to our language *)
let built_in_decls = StringMap.singleton "print"
  { f_typ = Void; f_name = "print"; f_formals = [(Int, "x")];
    f_locals = []; f_body = [] }
in

(* Builds a string map (f_name --> func_decl) that contains all of the
   functions in the program, including built in functions *)
let function_decls = List.fold_left (fun m fd -> StringMap.add fd.f_name fd m)
  built_in_decls functions
in

(* A function that takes f_name, looks up the f_name in the
   f_name --> func_decl string map, and returns the func_decl, or an exception
   if the f_name is not found in the string map *)
let function_decl s = try StringMap.find s function_decls
  with Not_found -> raise (Failure ("Unrecognized function " ^ s))
in

(* A function that is used to check each function (func) in the functions
   func_decl list *)
let check_function func =

  (* Checks formals for undefined references to tuples or nodes *)
  List.iter (fun (typ, _) -> check_defined_nt typ func.f_name; ())
  func.f_formals;

  (* Checks locals for undefined references to tuples or nodes *)
  List.iter (fun (typ, _) -> check_defined_nt typ func.f_name; ())
  func.f_locals;

  (* Checks formals for void types *)
  List.iter (check_not_void (fun n -> "In " ^ func.f_name ^ ", illegal void" ^
    " formal " ^ n)) func.f_formals;

  (* Checks locals for void types *)
  List.iter (check_not_void (fun n -> "In " ^ func.f_name ^ ", illegal void" ^
    " local " ^ n)) func.f_locals;
```

```
(* Checks formals for type pack *)
List.iter (check_not_packet (fun n -> "In " ^ func.f_name ^ ", illegal pack"
  ^ " formal " ^ n)) func.f_formals;

(* Checks locals for type pack *)
List.iter (check_not_packet (fun n -> "In " ^ func.f_name ^ ", illegal pack"
  ^ " local " ^ n)) func.f_locals;

(* Checks for duplicates in formals *)
report_duplicate (fun n -> "In " ^ func.f_name ^ ", duplicate formal " ^ n)
  (List.map snd func.f_formals);

(* Checks for duplicates in locals *)
report_duplicate (fun n -> "In " ^ func.f_name ^ ", duplicate local " ^ n)
  (List.map snd func.f_locals);

(* Builds a string map (s_name --> s_typ) *)
let symbols = List.fold_left (fun strmap (var_type, var_name) ->
  StringMap.add var_name var_type strmap) StringMap.empty (globals @
    func.f_formals @ func.f_locals)
in

(* A function that takes a s_name, looks up the s_name in the
   s_name --> s_typ string map, and returns the s_typ, or an exception
   if the s_name is not found in the string map *)
let type_of_identifier s =
  try StringMap.find s symbols
  with Not_found -> raise (Failure ("In " ^ func.f_name ^ ", undeclared " ^
  "identifier " ^ s))
in

(* A function that returns the type of the member *)
let type_of_member e s =
  let e_typ = e in
  let attr_dict = try TypMap.find e_typ typedefs
  with Not_found -> raise (Failure ("In " ^ func.f_name ^ ", " ^
  string_of_typ e_typ ^ " type not found")) in
  try StringMap.find s attr_dict
  with Not_found -> raise (Failure ("In " ^ func.f_name ^ ", " ^
  s ^ " member not found"))
in

(* A recursive function that validates returns the type of an expression, or
   throws an exception if the expression is invalid *)
let rec expr = function
    Assign(e1, e2) as exp -> let lt =
      expr e1 and rt = expr e2 in
      check_assign lt rt (Failure ("In " ^ func.f_name ^ ", illegal " ^
      "assignment " ^ string_of_typ lt ^ " = " ^ string_of_typ rt ^ " in "
      ^ string_of_expr exp))
| Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
  (match op with
      Add       when t1 = Float && t2 = Float -> Float
    | Add       when t1 = Int && t2 = Int -> Int
    | And       when t1 = Bool && t2 = Bool -> Bool
    | Div       when t1 = Float && t2 = Float -> Float
    | Div       when t1 = Int && t2 = Int -> Int
    | Equal     when t1 = t2 -> Bool
    | Geq       when t1 = Float && t2 = Float -> Bool
    | Geq       when t1 = Int && t2 = Int -> Bool
    | Greater   when t1 = Float && t2 = Float -> Bool
    | Greater   when t1 = Int && t2 = Int -> Bool
    | Leq       when t1 = Float && t2 = Float -> Bool
    | Leq       when t1 = Int && t2 = Int -> Bool
    | Less      when t1 = Float && t2 = Float -> Bool
    | Less      when t1 = Int && t2 = Int -> Bool
    | Mod       when t1 = Int && t2 = Int -> Int
    | Mult      when t1 = Float && t2 = Float -> Float
    | Mult      when t1 = Int && t2 = Int -> Int
    | NEqual    when t1 = t2 -> Bool
    | Or        when t1 = Bool && t2 = Bool -> Bool
    | Sub       when t1 = Float && t2 = Float -> Float
    | Sub       when t1 = Int && t2 = Int -> Int
    | _ -> raise (Failure ("In " ^ func.f_name ^ ", illegal binary " ^
      "operator " ^ string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
      string_of_typ t2 ^ " in " ^ string_of_expr e)))
| BoolLit(_) -> Bool
| Call(s, el) as call -> let func_decl = function_decl s in
  if List.length el <> List.length func_decl.f_formals then
    raise (Failure ("In " ^ func.f_name ^ ", expecting " ^ string_of_int
    (List.length func_decl.f_formals) ^ " arguments in call to " ^
    string_of_expr call))
  else
    List.iter2 (fun (formal, _) exp -> let actual = expr exp in
      ignore (check_assign formal actual (Failure ("In " ^ func.f_name ^
      ", illegal actual argument " ^ string_of_typ actual ^ ", expected " ^
      string_of_typ formal ^ " in parameter " ^ string_of_expr exp ^
      " in call to " ^ string_of_expr call))))
      func_decl.f_formals el; func_decl.f_typ
| FloatLit(_) -> Float
| Graph(_, _) -> Void
| Id(s) -> type_of_identifier s
| IntLit(_) -> Int
| Lst (t, _) -> ListTyp(t)
| ListAdd(e1, e2) -> let lst = expr e1 in
```

```
  (match lst with
      ListTyp(t) -> let ele = expr e2 in
        if t <> ele then
          raise (Failure ("In " ^ func.f_name ^ ", illegal list addition, "
          ^ string_of_typ lst ^ " += " ^ string_of_typ ele ^ " in " ^
          string_of_expr e1 ^ " += " ^ string_of_expr e2))
        else
          ListTyp(t)
    | _ -> raise (Failure ("In " ^ func.f_name ^ ", expected left side " ^
      "of operator to be of type list, in " ^ string_of_expr e1 ^ " += " ^
      string_of_expr e2)))
| ListRmv(e1, e2) -> let lst = expr e1 in
  (match lst with
      ListTyp(t) -> let ele = expr e2 in (match t with
          NodeTyp(_) | TupleTyp(_) ->
            if t <> ele then
              raise (Failure ("In " ^ func.f_name ^ ", type mismatch in "
              ^ "list removal, " ^ string_of_typ lst ^ " -= " ^
              string_of_typ ele ^ " in " ^ string_of_expr e1 ^ " -= " ^
              string_of_expr e2))
            else
              ListTyp(t)
        | _ -> raise (Failure ("In " ^ func.f_name ^ ", illegal list " ^
          "removal of non-mutable list type, " ^ string_of_typ lst ^ " -= "
          ^ string_of_typ ele ^ " in " ^ string_of_expr e1 ^ " -= " ^
          string_of_expr e2)))
    | _ -> raise (Failure ("In " ^ func.f_name ^ ", expected left side of"
      ^ " operator to be of type list, in " ^ string_of_expr e1 ^ " -= " ^
      string_of_expr e2)))
| Member(e, s) -> type_of_member (expr e) s
| Message -> raise (Failure ("In " ^ func.f_name ^ ", illegal use of " ^
    "msg"))
| Node(s, _) -> NodeTyp(s)
| Noexpr -> Void
| Null(t) -> t
| Pack(_) -> raise (Failure ("In " ^ func.f_name ^ ", illegal use of " ^
    "pack"))
| Print(_, _) -> Void
| Run(n, _) -> let n_decl = try (List.find (fun node_decl ->
  node_decl.n_typ = string_of_typ (expr n)) nodes) with Not_found ->
  raise (Failure ("In " ^ func.f_name ^ ", did not find " ^
  string_of_expr n)) in n_decl.n_do.f_typ
| Self -> raise (Failure ("In " ^ func.f_name ^ ", illegal use of self"))
| StrLit(_) -> Str
| Subscript(e1, e2) -> let ltyp = expr e1 in
  (match ltyp with
      ListTyp(t) -> let rtyp = expr e2 in
        (match rtyp with
              Int -> t
            | _ -> raise (Failure ("In " ^ func.f_name ^ ", expected " ^
              "integer index type for list subscript access, in " ^
              string_of_expr e1 ^ "[ " ^
              string_of_expr e2 ^ " ]")))
      | NodeTyp(t1) -> let rtyp = expr e2 in
          (match rtyp with
              NodeTyp(t2) when t1 = t2 -> EdgeTyp(t2)
            | _ -> raise (Failure ("In " ^ func.f_name ^ ", expected node" ^
              " index type for node subscript access, in " ^
              string_of_expr e1 ^ "[ " ^ string_of_expr e2 ^ " ]")))
      | _ -> raise (Failure ("In " ^ func.f_name ^ ", expected left side " ^
        "of operator to be of type list or node, in " ^ string_of_expr e1 ^
        "[ " ^ string_of_expr e2 ^ " ] ")))
  | Tuple(s, _) -> TupleTyp(s)
  | Unop(uop, e) as ex -> let t = expr e in
    (match uop with
        Neg when t = Int -> Int
      | Not when t = Bool -> Bool
      | _ -> raise (Failure ("In " ^ func.f_name ^ ", illegal unary " ^
        "operator " ^ string_of_typ t ^ " " ^ string_of_uop uop ^ " in " ^
        string_of_expr ex)))
in

(* A function that checks if the expression is a boolean type, and throws
   an exception if it isn't *)
let check_bool_expr e =
  if expr e <> Bool then
    raise (Failure ("In " ^ func.f_name ^ ", expected a boolean " ^
    "expression, in " ^ string_of_expr e))
  else ()
in

(* A recursive function that validates a statement, and throws an
   exception if the statement is invalid *)
let rec stmt = function
    Block sl -> let rec check_block = function
        [Return _ as s] -> stmt s
      | Return _ :: _ -> raise (Failure ("In " ^ func.f_name ^ ", illegal" ^
          " code following return"))
      | Block sl :: ss -> check_block (sl @ ss)
      | s :: ss -> stmt s ; check_block ss
      | [] -> ()
        in check_block sl
  | Expr e -> ignore (expr e)
  | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
    ignore (expr e3); stmt st
  | ForEach(e1, e2, st) -> let ltyp = expr e1 and rtyp = expr e2 in
```

```
        (match (ltyp, rtyp) with
            (t1, ListTyp(t2)) ->
            if t1 <> t2 then
              raise (Failure ("In " ^ func.f_name ^ ", type mismatch " ^
              string_of_typ t1 ^ " != " ^ string_of_typ t2 ^ " in " ^
              string_of_expr e1 ^ " in " ^ string_of_expr e2))
            else
              stmt st
          | (_, _) -> raise (Failure ("In " ^ func.f_name ^ ", expected " ^
              "identifier and list in, " ^ string_of_expr e1 ^ " in " ^
              string_of_expr e2)))
      | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2
      | Pass(_, _) -> raise (Failure ("In " ^ func.f_name ^ ", illegal use of" ^
          " pass"))
      | Return e -> let t = expr e in
        if t = func.f_typ
          then ()
        else
          raise (Failure ("In " ^ func.f_name ^ ", return type is " ^
          string_of_typ t ^ " but expecting " ^
          string_of_typ func.f_typ ^ " in " ^ string_of_expr e))
      | While(p, s) -> check_bool_expr p; stmt s

    (* Call stmt on the func body *)
    in
    stmt (Block func.f_body)

  (* Call check_function on each function in the function list *)
  in
  List.iter check_function functions;

(* Checking Nodes *************************************************************)

  (* Check that there are no duplicate node types *)
  report_duplicate (fun n -> "Duplicate node " ^ n) (List.map (fun node ->
    node.n_typ) nodes);

  (* A function that is used to check each node (node) in the nodes
     node_decl list *)
  let check_node node =

    (* Checking data block for undefined references to tuples or nodes *)
    List.iter (fun (typ, _) -> check_defined_nt typ (node.n_typ ^ " do block");
    ()) node.n_data;

    (* Checking edge block for undefined references to tuples or nodes *)
    List.iter (fun (typ, _) -> check_defined_nt typ (node.n_typ ^ " do block");
    ()) node.n_edge;
```

```
(* Checking pack block for undefined references to tuples or nodes *)
List.iter (fun (typ, _) -> check_defined_nt typ (node.n_typ ^ " do block");
()) node.n_pack;

(* Checking do block formals for undefined references to tuples or nodes *)
List.iter (fun (typ, _) -> check_defined_nt typ (node.n_typ ^ " do block");
()) node.n_do.f_formals;

(* Checking do block locals for undefined references to tuples or nodes *)
List.iter (fun (typ, _) -> check_defined_nt typ (node.n_typ ^ " do block");
()) node.n_do.f_locals;

(* Checking catch block for undefined references to tuples or nodes *)
List.iter (fun (typ, _) -> check_defined_nt typ (node.n_typ ^ " do block");
()) node.n_catch.f_locals;

(* Checks data block for void types *)
List.iter (check_not_void (fun n -> "In " ^ node.n_typ ^ ", illegal void " ^
  "type " ^ n ^ " in data block")) node.n_data;

(* Checks edge block for void types *)
List.iter (check_not_void (fun n -> "In " ^ node.n_typ ^ ", illegal void " ^
  "type " ^ n ^ " in edge block")) node.n_edge;

(* Checks pack block for void types *)
List.iter (check_not_void (fun n -> "In " ^ node.n_typ ^ ", illegal void " ^
  "type " ^ n ^ " in pack block")) node.n_pack;

(* Checks do block formals for void types *)
List.iter (check_not_void (fun n -> "In " ^ node.n_typ ^ ", illegal void " ^
  "type " ^ n ^ " in do formals")) node.n_do.f_formals;

(* Checks do block locals for void types *)
List.iter (check_not_void (fun n -> "In " ^ node.n_typ ^ ", illegal void " ^
  "type " ^ n ^ " in do locals")) node.n_do.f_locals;

(* Checks catch block locals for void types *)
List.iter (check_not_void (fun n -> "In " ^ node.n_typ ^ ", illegal void " ^
  "type " ^ n ^ " in catch locals")) node.n_catch.f_locals;

(* Checks data block for pack types *)
List.iter (check_not_packet (fun n -> "In " ^ node.n_typ ^ ", illegal " ^
  "pack type " ^ n ^ " in data block")) node.n_data;

(* Checks edge block for pack types *)
List.iter (check_not_packet (fun n -> "In " ^ node.n_typ ^ ", illegal " ^
  "pack type " ^ n ^ " in edge block")) node.n_edge;
  (* Checks pack block for pack types *)
  List.iter (check_not_packet (fun n -> "In " ^ node.n_typ ^ ", illegal " ^
    "pack type " ^ n ^ " in pack block")) node.n_pack;

  (* Checks do block formals for pack types *)
  List.iter (check_not_packet (fun n -> "In " ^ node.n_typ ^ ", illegal " ^
    "pack type " ^ n ^ " in do formals")) node.n_do.f_formals;

  (* Checks data block for duplicates *)
  report_duplicate (fun n -> "In " ^ node.n_typ ^ ", duplicate variable " ^
    n ^ " in data block") (List.map snd node.n_data);

  (* Checks edge block for duplicates *)
  report_duplicate (fun n -> "In " ^ node.n_typ ^ ", duplicate variable " ^
    n ^ " in edge block") (List.map snd node.n_edge);

  (* Checks pack block for duplicates *)
  report_duplicate (fun n -> "In " ^ node.n_typ ^ ", duplicate variable " ^
    n ^ " in pack block") (List.map snd node.n_pack);

  (* Checks do block formals for duplicates *)
  report_duplicate (fun n -> "In " ^ node.n_typ ^ ", duplicate variable " ^
    n ^ " in do formals") (List.map snd node.n_do.f_formals);

  (* Checks do block locals for duplicates *)
  report_duplicate (fun n -> "In " ^ node.n_typ ^ ", duplicate variable " ^
    n ^ " in do locals") (List.map snd node.n_do.f_locals);

  (* Checks catch block locals for duplicates *)
  report_duplicate (fun n -> "In " ^ node.n_typ ^ ", duplicate variable " ^
    n ^ " in catch locals") (List.map snd node.n_catch.f_locals);

(* Call check_node on each node in the node list *)
in
List.iter check_node nodes;

(* A function that is used to check the do block in each node in the nodes
   list *)
let check_node_do_block node =

  (* Builds a string map (s_name --> s_typ) *)
  let symbols = List.fold_left (fun strmap (var_type, var_name) ->
    let res_var_type = match var_type with
      Packet ->
        let node_typ = StringMap.find "self" strmap in
        (match node_typ with
          NodeTyp n_str -> PackTyp (n_str)
        | _ -> raise (Failure ("pack type not found")))
    | typ -> typ
    in
    StringMap.add var_name res_var_type strmap) StringMap.empty (globals @
      node.n_do.f_formals @ node.n_do.f_locals)
  in

  (* A function that takes a s_name, looks up the s_name in the
     s_name --> s_typ string map, and returns the s_typ, or an exception
     if the s_name is not found in the string map *)
  let type_of_identifier s =
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("In " ^ node.n_typ ^ " do block, " ^
    "undeclared identifier " ^ s))
  in

  (* A function that returns the type of the member *)
  let type_of_member e s =
    let e_typ = e in
    let attr_dict = try TypMap.find e_typ typedefs
    with Not_found -> raise (Failure ("In " ^ node.n_typ ^ " do block, " ^
    string_of_typ e_typ ^ " type not found")) in
    try StringMap.find s attr_dict
    with Not_found -> raise (Failure ("In " ^ node.n_typ ^ " do block, " ^
    s ^ " member not found"))
  in

  (* A recursive function that validates returns the type of an expression, or
     throws an exception if the expression is invalid *)
  let rec expr = function
      Assign(e1, e2) as exp -> let lt =
        expr e1 and rt = expr e2 in
          check_assign lt rt (Failure ("In " ^ node.n_typ ^ " do block, " ^
          "illegal assignment " ^ string_of_typ lt ^ " = " ^ string_of_typ
          rt ^ " in " ^ string_of_expr exp))
    | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
      (match op with
          Add       when t1 = Float && t2 = Float -> Float
        | Add       when t1 = Int && t2 = Int -> Int
        | And       when t1 = Bool && t2 = Bool -> Bool
        | Div       when t1 = Float && t2 = Float -> Float
        | Div       when t1 = Int && t2 = Int -> Int
        | Equal     when t1 = t2 -> Bool
        | Geq       when t1 = Float && t2 = Float -> Bool
        | Geq       when t1 = Int && t2 = Int -> Bool
        | Greater   when t1 = Float && t2 = Float -> Bool
        | Greater   when t1 = Int && t2 = Int -> Bool
        | Leq       when t1 = Float && t2 = Float -> Bool
        | Leq       when t1 = Int && t2 = Int -> Bool
        | Less      when t1 = Float && t2 = Float -> Bool
        | Less      when t1 = Int && t2 = Int -> Bool
        | Mod       when t1 = Int && t2 = Int -> Int
        | Mult      when t1 = Float && t2 = Float -> Float
        | Mult      when t1 = Int && t2 = Int -> Int
        | NEqual    when t1 = t2 -> Bool
        | Or        when t1 = Bool && t2 = Bool -> Bool
        | Sub       when t1 = Float && t2 = Float -> Float
        | Sub       when t1 = Int && t2 = Int -> Int
        | _ -> raise (Failure ("In " ^ node.n_typ ^ " do block, illegal " ^
          "binary operator " ^ string_of_typ t1 ^ " " ^ string_of_op op ^ " "
          ^ string_of_typ t2 ^ " in " ^ string_of_expr e)))
    | BoolLit(_) -> Bool
    | Call(s, el) as call -> let func_decl = function_decl s in
      if List.length el <> List.length func_decl.f_formals then
        raise (Failure ("In " ^ node.n_typ ^ "do block, expecting " ^
        string_of_int (List.length func_decl.f_formals) ^ " arguments in " ^
        "call to " ^ string_of_expr call))
      else
        List.iter2 (fun (formal, _) exp -> let actual = expr exp in
          ignore (check_assign formal actual (Failure ("In " ^ node.n_typ ^
          " do block, illegal actual argument " ^ string_of_typ actual ^
          ", expected " ^ string_of_typ formal ^ " in parameter " ^
          string_of_expr exp ^ " in call to " ^ string_of_expr call))))
          func_decl.f_formals el; func_decl.f_typ
    | FloatLit(_) -> Float
    | Graph(_, _) -> Void
    | Id(s) -> type_of_identifier s
    | IntLit(_) -> Int
    | Lst (t, _) -> ListTyp(t)
    | ListAdd(e1, e2) -> let lst = expr e1 in
      (match lst with
          ListTyp(t) -> let ele = expr e2 in
            if t <> ele then
              raise (Failure ("In " ^ node.n_typ ^ " do block, illegal list "
              ^ "addition, " ^ string_of_typ lst ^ " += " ^ string_of_typ ele
              ^ " in " ^ string_of_expr e1 ^ " += " ^ string_of_expr e2))
            else
              ListTyp(t)
        | _ -> raise (Failure ("In " ^ node.n_typ ^ " do block, expected left"
          ^ " side of operator to be of type list, in " ^ string_of_expr e1 ^
          " += " ^ string_of_expr e2)))
    | ListRmv(e1, e2) -> let lst = expr e1 in
      (match lst with
          ListTyp(t) -> let ele = expr e2 in (match t with
              NodeTyp(_) | TupleTyp(_) ->
                if t <> ele then
```

```
              raise (Failure ("In " ^ node.n_typ ^ " do block, type " ^
              "mismatch in list removal, " ^ string_of_typ lst ^ " -= " ^
              string_of_typ ele ^ " in " ^ string_of_expr e1 ^ " -= " ^
              string_of_expr e2))
            else
              ListTyp(t)
        | _ -> raise (Failure ("In " ^ node.n_typ ^ " do block, illegal" ^
          " list removal of non-mutable list type, " ^ string_of_typ lst ^
          " -= " ^ string_of_typ ele ^ " in " ^ string_of_expr e1 ^ " -= "
          ^ string_of_expr e2)))
    | _ -> raise (Failure ("In " ^ node.n_typ ^ " do block, expected left"
      ^ " side of operator to be of type list, in " ^ string_of_expr e1 ^
      " -= " ^ string_of_expr e2)))
| Member(e, s) -> type_of_member (expr e) s
| Message -> raise (Failure ("In " ^ node.n_typ ^ " do block, illegal " ^
    "use of msg"))
| Node(s, _) -> NodeTyp(s)
| Noexpr -> Void
| Null(t) -> t
| Pack(_) -> PackTyp(node.n_typ)
| Print(_, _) -> Void
| Run(n, _) -> let n_decl = try (List.find (fun node_decl ->
  node_decl.n_typ = string_of_typ (expr n)) nodes) with Not_found ->
  raise (Failure ("In " ^ node.n_typ ^ " do block, did not find " ^
  string_of_expr n)) in n_decl.n_do.f_typ
| Self -> expr (Id("self"))
| StrLit(_) -> Str
| Subscript(e1, e2) -> let ltyp = expr e1 in
  (match ltyp with
      ListTyp(t) -> let rtyp = expr e2 in
        (match rtyp with
            Int -> t
          | _ -> raise (Failure ("In " ^ node.n_typ ^ " do block, " ^
            " expected integer index type for list subscript access, in "
            ^ string_of_expr e1 ^ "[ " ^ string_of_expr e2 ^ " ]")))
    | NodeTyp(t1) -> let rtyp = expr e2 in
        (match rtyp with
            NodeTyp(t2) when t1 = t2 -> EdgeTyp(t2)
          | _ -> raise (Failure ("In " ^ node.n_typ ^ " do block, " ^
            " expected node index type for node subscript access, in " ^
            string_of_expr e1 ^ "[ " ^ string_of_expr e2 ^ " ]")))
    | _ -> raise (Failure ("In " ^ node.n_typ ^ " do block, expected left"
      ^ " side of operator to be of type list or node, in " ^
      string_of_expr e1 ^ "[ " ^ string_of_expr e2 ^ " ] ")))
| Tuple(s, _) -> TupleTyp(s)
| Unop(uop, e) as ex -> let t = expr e in
  (match uop with
      Neg when t = Int -> Int
      | Not when t = Bool -> Bool
      | _ -> raise (Failure ("In " ^ node.n_typ ^ " do block, illegal unary"
        ^ " operator " ^ string_of_typ t ^ " " ^ string_of_uop uop ^ " in "
        ^ string_of_expr ex)))
in

(* A function that checks if the expression is a boolean type, and throws
   an exception if it isn't *)
let check_bool_expr e =
  if expr e <> Bool then
    raise (Failure ("In " ^ node.n_typ ^ " do block, expected a boolean " ^
    "expression, in " ^ string_of_expr e))
  else ()
in

(* A recursive function that validates a statement, and throws an
   exception if the statement is invalid *)
let rec stmt = function
    Block sl -> let rec check_block = function
        [Return _ as s ] -> stmt s
      | Return _ :: _ -> raise (Failure ("In " ^ node.n_typ ^ " do block," ^
          " code following return"))
      | Block sl :: ss -> check_block (sl @ ss)
      | s :: ss -> stmt s ; check_block ss
      | [] -> ()
        in check_block sl
  | Expr e -> ignore (expr e)
  | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
    ignore (expr e3); stmt st
  | ForEach(e1, e2, st) -> let ltyp = expr e1 and rtyp = expr e2 in
    (match (ltyp, rtyp) with
        (t1, ListTyp(t2)) ->
        if t1 <> t2 then
          raise (Failure ("In " ^ node.n_typ ^ " do block, type mismatch " ^
          string_of_typ t1 ^ " != " ^ string_of_typ t2 ^ " in " ^
          string_of_expr e1 ^ " in " ^ string_of_expr e2))
        else
          stmt st
      | (_, _) -> raise (Failure ("In " ^ node.n_typ ^ " do block, expected"
          ^ " identifier and list in, " ^ string_of_expr e1 ^ " in " ^
          string_of_expr e2)))
  | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2
  | Pass(_, _) -> ()
  | Return e -> let t = expr e in
    if t = NodeTyp(node.n_typ)
      then ()
    else
      raise (Failure ("In " ^ node.n_typ ^ " do block, return type is " ^
        string_of_typ t ^ " but expecting " ^ node.n_typ ^ " in " ^
        string_of_expr e))
      | While(p, s) -> check_bool_expr p; stmt s

    (* Call stmt on the n_do body *)
    in
    stmt (Block node.n_do.f_body)

  (* Call check_node_do_block on each node in the node list *)
  in
  List.iter check_node_do_block nodes;

  (* A function that is used to check the catch block in each node in the nodes
     list *)
  let check_node_catch_block node =

    (* Builds a string map (s_name --> s_typ) *)
    let symbols = List.fold_left (fun strmap (var_type, var_name) ->
      let res_var_type = match var_type with
        Packet ->
          let node_typ = StringMap.find "self" strmap in
          (match node_typ with
            NodeTyp n_str -> PackTyp (n_str)
          | _ -> raise (Failure ("pack type not found")))
      | typ -> typ
      in
      StringMap.add var_name res_var_type strmap) StringMap.empty (globals @
        node.n_catch.f_formals @ node.n_catch.f_locals)
    in

    (* A function that takes a s_name, looks up the s_name in the
       s_name --> s_typ string map, and returns the s_typ, or an exception
       if the s_name is not found in the string map *)
    let type_of_identifier s =
      try StringMap.find s symbols
      with Not_found -> raise (Failure ("In " ^ node.n_typ ^ " catch block, " ^
      "undeclared identifier " ^ s))
    in

    (* A function that returns the type of the member *)
    let type_of_member e s =
      let e_typ = e in
      let attr_dict = try TypMap.find e_typ typedefs
      with Not_found -> raise (Failure ("In " ^ node.n_typ ^ " catch block, " ^
      string_of_typ e_typ ^ " type not found")) in
      try StringMap.find s attr_dict
      with Not_found -> raise (Failure ("In " ^ node.n_typ ^ " catch block, " ^
      s ^ " member not found"))
```

```
in

(* A recursive function that validates returns the type of an expression, or
   throws an exception if the expression is invalid *)
let rec expr = function
    Assign(e1, e2) as exp -> let lt =
      expr e1 and rt = expr e2 in
        check_assign lt rt (Failure ("In " ^ node.n_typ ^ " catch block, " ^
        "illegal assignment " ^ string_of_typ lt ^ " = " ^ string_of_typ
        rt ^ " in " ^ string_of_expr exp))
  | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
    (match op with
        Add       when t1 = Float && t2 = Float -> Float
      | Add       when t1 = Int && t2 = Int -> Int
      | And       when t1 = Bool && t2 = Bool -> Bool
      | Div       when t1 = Float && t2 = Float -> Float
      | Div       when t1 = Int && t2 = Int -> Int
      | Equal     when t1 = t2 -> Bool
      | Geq       when t1 = Float && t2 = Float -> Bool
      | Geq       when t1 = Int && t2 = Int -> Bool
      | Greater   when t1 = Float && t2 = Float -> Bool
      | Greater   when t1 = Int && t2 = Int -> Bool
      | Leq       when t1 = Float && t2 = Float -> Bool
      | Leq       when t1 = Int && t2 = Int -> Bool
      | Less      when t1 = Float && t2 = Float -> Bool
      | Less      when t1 = Int && t2 = Int -> Bool
      | Mod       when t1 = Int && t2 = Int -> Int
      | Mult      when t1 = Float && t2 = Float -> Float
      | Mult      when t1 = Int && t2 = Int -> Int
      | NEqual    when t1 = t2 -> Bool
      | Or        when t1 = Bool && t2 = Bool -> Bool
      | Sub       when t1 = Float && t2 = Float -> Float
      | Sub       when t1 = Int && t2 = Int -> Int
      | _ -> raise (Failure ("In " ^ node.n_typ ^ " catch block, illegal " ^
        "binary operator " ^ string_of_typ t1 ^ " " ^ string_of_op op ^ " "
        ^ string_of_typ t2 ^ " in " ^ string_of_expr e)))
  | BoolLit(_) -> Bool
  | Call(s, el) as call -> let func_decl = function_decl s in
    if List.length el <> List.length func_decl.f_formals then
      raise (Failure ("In " ^ node.n_typ ^ "catch block, expecting " ^
      string_of_int (List.length func_decl.f_formals) ^ " arguments in " ^
      "call to " ^ string_of_expr call))
    else
      List.iter2 (fun (formal, _) exp -> let actual = expr exp in
        ignore (check_assign formal actual (Failure ("In " ^ node.n_typ ^
        " catch block, illegal actual argument " ^ string_of_typ actual ^
        ", expected " ^ string_of_typ formal ^ " in parameter " ^
        string_of_expr exp ^ " in call to " ^ string_of_expr call))))
        func_decl.f_formals el; func_decl.f_typ
  | FloatLit(_) -> Float
  | Graph(_, _) -> Void
  | Id(s) -> type_of_identifier s
  | IntLit(_) -> Int
  | Lst (t, _) -> ListTyp(t)
  | ListAdd(e1, e2) -> let lst = expr e1 in
    (match lst with
        ListTyp(t) -> let ele = expr e2 in
          if t <> ele then
            raise (Failure ("In " ^ node.n_typ ^ " catch block, illegal " ^
            "list addition, " ^ string_of_typ lst ^ " += " ^ string_of_typ
            ele ^ " in " ^ string_of_expr e1 ^ " += " ^ string_of_expr e2))
          else
            ListTyp(t)
      | _ -> raise (Failure ("In " ^ node.n_typ ^ " catch block, expected" ^
        " left side of operator to be of type list, in " ^ string_of_expr e1
        ^ " += " ^ string_of_expr e2)))
  | ListRmv(e1, e2) -> let lst = expr e1 in
    (match lst with
        ListTyp(t) -> let ele = expr e2 in (match t with
            NodeTyp(_) | TupleTyp(_) ->
              if t <> ele then
                raise (Failure ("In " ^ node.n_typ ^ " catch block, type " ^
                "mismatch in list removal, " ^ string_of_typ lst ^ " -= " ^
                string_of_typ ele ^ " in " ^ string_of_expr e1 ^ " -= " ^
                string_of_expr e2))
              else
                ListTyp(t)
          | _ -> raise (Failure ("In " ^ node.n_typ ^ " catch block, " ^
            "illegal list removal of non-mutable list type, " ^
            string_of_typ lst ^ " -= " ^ string_of_typ ele ^ " in " ^
            string_of_expr e1 ^ " -= " ^ string_of_expr e2)))
      | _ -> raise (Failure ("In " ^ node.n_typ ^ " catch block, expected "
        ^ "left side of operator to be of type list, in " ^ string_of_expr
        e1 ^ " -= " ^ string_of_expr e2)))
  | Member(e, s) -> type_of_member (expr e) s
  | Message -> PackTyp(node.n_typ)
  | Node(s, _) -> NodeTyp(s)
  | Noexpr -> Void
  | Null(t) -> t
  | Pack(_) -> PackTyp(node.n_typ)
  | Print(_, _) -> Void
  | Run(n, _) -> let n_decl = try (List.find (fun node_decl ->
    node_decl.n_typ = string_of_typ (expr n)) nodes) with Not_found ->
    raise (Failure ("In " ^ node.n_typ ^ " do catch, did not find " ^
    string_of_expr n)) in n_decl.n_do.f_typ
  | Self -> expr (Id("self"))
```

```
  | StrLit(_) -> Str
  | Subscript(e1, e2) -> let ltyp = expr e1 in
    (match ltyp with
        ListTyp(t) -> let rtyp = expr e2 in
          (match rtyp with
              Int -> t
            | _ -> raise (Failure ("In " ^ node.n_typ ^ " catch block, " ^
              " expected integer index type for list subscript access, in "
              ^ string_of_expr e1 ^ "[ " ^ string_of_expr e2 ^ " ]")))
      | NodeTyp(t1) -> let rtyp = expr e2 in
          (match rtyp with
              NodeTyp(t2) when t1 = t2 -> EdgeTyp(t2)
            | _ -> raise (Failure ("In " ^ node.n_typ ^ " catch block, " ^
              " expected node index type for node subscript access, in " ^
              string_of_expr e1 ^ "[ " ^ string_of_expr e2 ^ " ]")))
      | _ -> raise (Failure ("In " ^ node.n_typ ^ " catch block, expected" ^
        " left side of operator to be of type list or node, in " ^
        string_of_expr e1 ^ "[ " ^ string_of_expr e2 ^ " ] ")))
  | Tuple(s, _) -> TupleTyp(s)
  | Unop(uop, e) as ex -> let t = expr e in
    (match uop with
        Neg when t = Int -> Int
      | Not when t = Bool -> Bool
      | _ -> raise (Failure ("In " ^ node.n_typ ^ " catch block, illegal "
        ^ "unary operator " ^ string_of_typ t ^ " " ^ string_of_uop uop ^
        " in " ^ string_of_expr ex)))
in

(* A function that checks if the expression is a boolean type, and throws
   an exception if it isn't *)
let check_bool_expr e =
  if expr e <> Bool then
    raise (Failure ("In " ^ node.n_typ ^ " catch block, expected a boolean "
    ^ "expression, in " ^ string_of_expr e))
  else ()
in

(* A recursive function that validates a statement, and throws an
   exception if the statment is invalid *)
let rec stmt = function
    Block sl -> let rec check_block = function
        [Return _ as s ] -> stmt s
      | Return _ :: _ -> raise (Failure ("In " ^ node.n_typ ^ " catch block"
          ^ ", code following return"))
      | Block sl :: ss -> check_block (sl @ ss)
      | s :: ss -> stmt s ; check_block ss
      | [] -> ()
        in check_block sl
    | Expr e -> ignore (expr e)
    | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
      ignore (expr e3); stmt st
    | ForEach(e1, e2, st) -> let ltyp = expr e1 and rtyp = expr e2 in
      (match (ltyp, rtyp) with
          (t1, ListTyp(t2)) ->
          if t1 <> t2 then
            raise (Failure ("In " ^ node.n_typ ^ " catch block, type " ^
            "mismatch" ^ string_of_typ t1 ^ " != " ^ string_of_typ t2 ^ " in "
            ^ string_of_expr e1 ^ " in " ^ string_of_expr e2))
          else
            stmt st
        | (_, _) -> raise (Failure ("In " ^ node.n_typ ^ " catch block, " ^
            "expected identifier and list in, " ^ string_of_expr e1 ^ " in " ^
            string_of_expr e2)))
    | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2
    | Pass(_, _) -> ()
    | Return e -> let t = expr e in
      if t = Void
        then ()
      else
        raise (Failure ("In " ^ node.n_typ ^ " catch block, return type is " ^
        string_of_typ t ^ " but expecting " ^ node.n_typ ^ " in " ^
        string_of_expr e))
    | While(p, s) -> check_bool_expr p; stmt s

  (* Call stmt on the n_catch body *)
  in
  stmt (Block node.n_catch.f_body)

(* Call check_node_catch_block on each node in the node list *)
in
List.iter check_node_catch_block nodes;
```

Code Generation

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

*)
module L = Llvm
module A = Ast

module StringMap = Map.Make(String)
module TypMap = Map.Make(A.Typ)

type f_cat = Func|Do|Catch
type bind = {typ : A.typ; idx : int}
type tuple_info = {ltyp : L.lltype; attrs : bind StringMap.t}

let translate (nodes, tuples, globals, functions) =
  let context = L.global_context () in
  let llctx = L.global_context () in
  let the_module = L.create_module context "MicroC" in
  let listm = L.MemoryBuffer.of_file "list.bc" in
  let llm = Llvm_bitreader.parse_bitcode llctx listm in
  let i32_t  = L.i32_type  context
  and i8_t   = L.i8_type   context
  and i1_t   = L.i1_type   context
  and flt_t  = L.float_type context
  and void_t = L.void_type context
  and idlist_t = L.pointer_type (match L.type_by_name llm "struct.IdList" with
    None -> raise (Invalid_argument "Option.get idlist")
  | Some x -> x)
      and node_t = L.pointer_type (match L.type_by_name llm "struct.ListNode" with
            None -> raise (Invalid_argument "Option.get")
      | Some x -> x)  in

      let update_typedef typ typedef =
            match typ with
              A.ListTyp _              -> typedef
            | A.NodeTyp _ | A.TupleTyp _ | A.PackTyp _ | A.EdgeTyp _ ->
                  if not (TypMap.mem typ typedef)
                         then
                         let name = A.string_of_typ typ in
                         let ltyp = L.named_struct_type context name in
                         let typ_info = {ltyp = ltyp; attrs = StringMap.empty} in
                         TypMap.add typ typ_info typedef
                  else typedef
  | _ -> typedef in

     let ltype_of_typ typedef = function
    A.Int   -> i32_t
  | A.Bool  -> i1_t
  | A.Float -> flt_t
  | A.Void  -> void_t
  | A.Str   -> L.pointer_type i8_t
  | A.ListTyp _  -> idlist_t
  | typ -> L.pointer_type (TypMap.find typ typedef).ltyp in

let (typedefs, obj_funs, do_decls, catch_decls) =
            let make_attr (lst,attrs,typedef,i) (t,n) =
    let new_typedef =
                         match t with
                                A.Int | A.Float | A.Bool | A.Void -> typedef
                         | _ -> update_typedef t typedef in
                  let new_attrs = StringMap.add n {typ=t; idx=i} attrs in
                  let new_l = (ltype_of_typ new_typedef t)::lst in
                  (new_l, new_attrs, new_typedef, i+1) in

  let make_tuple_info typedef ltyp attrs =
    let (new_l,new_attrs,_,_) =
      List.fold_left make_attr ([],StringMap.empty,typedef,0) attrs in
    ignore(L.struct_set_body ltyp (Array.of_list (List.rev new_l)) false);
    {ltyp = ltyp; attrs = new_attrs} in

     let tuple_typedefs =
            let add_tuple_info typedef tdecl =
      let typ_name = A.TupleTyp tdecl.A.t_typ in
      let typedef = if TypMap.mem typ_name typedef
        then typedef else update_typedef typ_name typedef in
      let ltyp = (TypMap.find typ_name typedef).ltyp in
                  (* Adding attribute *)
      TypMap.add typ_name (make_tuple_info typedef ltyp tdecl.A.t_attributes) typedef
            in List.fold_left add_tuple_info TypMap.empty (List.rev tuples) in

       let (node_typedefs, obj_funs, do_decls, catch_decls) =
    let add_node_info (typedef, obj_funs, do_decls, catch_decls) ndecl =
      let n_typ_name = A.NodeTyp ndecl.A.n_typ in
```

```
        let p_typ_name = A.PackTyp ndecl.A.n_typ in
        let e_typ_name = A.EdgeTyp ndecl.A.n_typ in

        let (n_edge,n_pack,n_do,n_catch) =
          (ndecl.A.n_edge,ndecl.A.n_pack,ndecl.A.n_do,ndecl.A.n_catch) in
                    (* Has this been forward declared? *)
        let typedef = if TypMap.mem n_typ_name typedef
          then typedef else update_typedef n_typ_name typedef in
        let ltyp = (TypMap.find n_typ_name typedef).ltyp in

        let edge_ltyp = L.named_struct_type context ("edge//"^ndecl.A.n_typ) in
        let edge_info = make_tuple_info typedef edge_ltyp n_edge in

        let pack_ltyp = L.named_struct_type context ("pack//"^ndecl.A.n_typ) in
        let pack_info = make_tuple_info typedef pack_ltyp n_pack in

        let typedef = TypMap.add e_typ_name edge_info typedef in
        let typedef = TypMap.add p_typ_name pack_info typedef in

        (* Declaring the DO & CATCH functions *)
        let obj_fun_decls fdecl =
          let name = fdecl.A.f_name
          and formal_types =
            Array.of_list (List.map (fun (t,_) -> ltype_of_typ typedef t)
fdecl.A.f_formals) in
          let ftyp = L.function_type (ltype_of_typ typedef fdecl.A.f_typ) formal_types
in
          let fdef = L.define_function name ftyp the_module in
        (ftyp, fdef) in

                    let (do_typ,do_fun) = obj_fun_decls n_do in
                    let (catch_typ,catch_fun) = obj_fun_decls n_catch in

                    (* Adding attributes *)
        let (new_l,new_attrs,new_typedef,_) =
          List.fold_left make_attr
          ([L.pointer_type catch_typ;
            L.pointer_type do_typ;
            idlist_t;
            idlist_t],StringMap.empty,typedef,4) ndecl.A.n_data in

                           ignore(L.struct_set_body ltyp (Array.of_list (List.rev
new_l)) false);

                           (TypMap.add n_typ_name {ltyp = ltyp; attrs = new_attrs}
new_typedef,
          TypMap.add n_typ_name ((n_do,do_fun), (n_catch,catch_fun)) obj_funs,
          n_do::do_decls,
          n_catch::catch_decls)
      in List.fold_left add_node_info
        (tuple_typedefs, TypMap.empty, [], []) (List.rev nodes)
         in
    (node_typedefs, obj_funs, do_decls, catch_decls)  in

  (* Declare each global variable; remember its value in a map *)
  let global_vars =
    let global_var m (t, n) =
      let init = L.const_null (ltype_of_typ typedefs t)
      in StringMap.add n ((L.define_global n init the_module),t) m in
    List.fold_left global_var StringMap.empty globals in

  (* Declare printf(), which the print built-in function will call *)
  let printf_t    = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func = L.declare_function "printf" printf_t the_module in

  let initIdList_t  = L.function_type idlist_t [| |] in
  let initIdList_f  = L.declare_function "initIdList" initIdList_t the_module in
  let appendId_t    = L.function_type idlist_t [| idlist_t; L.pointer_type i8_t;
L.pointer_type i8_t |] in
  let appendId_f    = L.declare_function "appendId" appendId_t the_module in
  let indexIdList_t = L.function_type node_t [| idlist_t; i32_t |] in
  let indexIdList_f = L.declare_function "indexIdList" indexIdList_t the_module in
  let removeIdList_t = L.function_type idlist_t [| idlist_t; L.pointer_type i8_t |] in
  let removeIdList_f = L.declare_function "removeIdList" removeIdList_t the_module in
  let findNodeId_t = L.function_type node_t [| idlist_t; L.pointer_type i8_t |] in
  let findNodeId_f = L.declare_function "findNodeId" findNodeId_t the_module in
  let isEmptyIdList_t = L.function_type i8_t [| idlist_t |] in
  let isEmptyIdList_f = L.declare_function "isEmptyList" isEmptyIdList_t the_module in


  (* Define each function (arguments and return type) so we can call it *)
  let function_decls =
    let function_decl m fdecl =
      let name = fdecl.A.f_name
      and formal_types =
                         Array.of_list (List.map (fun (t,_) -> ltype_of_typ typedefs
t) fdecl.A.f_formals)      in
                    let ftype = L.function_type (ltype_of_typ typedefs fdecl.A.f_typ)
formal_types
          in StringMap.add name (L.define_function name ftype the_module, fdecl) m
         in List.fold_left function_decl StringMap.empty functions in

  (* Fill in the body of the given function *)
  let build_function_body fcat fdecl =
    let (the_function, _) = match fcat with
      Func -> (fst (StringMap.find fdecl.A.f_name function_decls),A.Void)
    | Do -> let ((_,do_fun),_) = TypMap.find (fst (List.hd fdecl.A.f_formals)) obj_funs
            in (do_fun, fst (List.hd fdecl.A.f_formals))
    | Catch -> let (_,(_,catch_fun)) = TypMap.find (fst (List.hd fdecl.A.f_formals))
obj_funs
                in (catch_fun, fst (List.hd fdecl.A.f_formals))
    in
    let builder = L.builder_at_end context (L.entry_block the_function) in

    let local_vars =
      let add_formal m (t, n) p = L.set_value_name n p;
        let local = L.build_alloca (ltype_of_typ typedefs t) n builder in
       ignore (L.build_store p local builder);
      StringMap.add n (local,t) m in

      let add_local m (t, n) =
        let (ltyp,typ) = match t with
          A.Packet ->
            let (_,node_typ) = StringMap.find "self" m in
            (match node_typ with
              A.NodeTyp n_str ->
                let p_typ = A.PackTyp n_str in
                (ltype_of_typ typedefs p_typ,p_typ)
            | _ -> raise (Invalid_argument (A.string_of_typ node_typ)))
        | typ -> (ltype_of_typ typedefs typ,typ) in
       let local_var = L.build_alloca ltyp n builder
      in StringMap.add n (local_var, typ) m in

      let formals = List.fold_left2 add_formal StringMap.empty fdecl.A.f_formals
          (Array.to_list (L.params the_function)) in
      List.fold_left add_local formals fdecl.A.f_locals in

    (* Return the value for a variable or formal argument *)
    let lookup n = try StringMap.find n local_vars
                   with Not_found -> StringMap.find n global_vars in

    let get_nstr typ = match typ with
        A.NodeTyp nstr -> nstr | _ -> raise (Invalid_argument "Pack(nodetype)") in

    let build_tuple_ptr ltyp name act expr builder =
      let actuals = List.map (fun e -> fst (expr builder e)) act in
      let tup_ptr = L.build_malloc ltyp name builder in
      let add_attr i actual =
        let a_ptr = L.build_struct_gep tup_ptr i "act" builder in
        ignore(L.build_store actual a_ptr builder); i+1 in
      ignore(List.fold_left add_attr 0 actuals);
      tup_ptr in

    let get_dest_binding expr builder e_d =
      match e_d with
        A.Id s -> lookup s
      | A.Member (e, s) ->
          let (e_val, e_typ) = expr builder e in
          let attr_binding = StringMap.find s (TypMap.find e_typ typedefs).attrs in
          let idx = attr_binding.idx in
          (L.build_struct_gep e_val idx "ptr" builder, attr_binding.typ)
      | _ -> raise (Invalid_argument "dest_bining") in

    (* Construct code for an expression; return its value *)
    let rec expr builder = function
       A.IntLit i          -> (L.const_int i32_t i, A.Int)
    | A.FloatLit f  -> (L.const_float flt_t f, A.Float)
    | A.BoolLit b   -> (L.const_int i1_t (if b then 1 else 0), A.Bool)
    | A.StrLit s -> (* TODO *)
      let str_ptr = L.build_alloca (L.array_type i8_t (String.length s)) "str" builder
in
      (str_ptr, A.Str)
    | (* TODO *) A.Noexpr -> (L.const_int i32_t 0, A.Void)
    | A.Id s                      ->
       let binding = lookup s
       in (L.build_load (fst binding) s builder, snd binding)
    | A.Binop (e1, op, e2) ->
         let (e1',t1) = expr builder e1
         and (e2',_) = expr builder e2 in
      let (oper,typ) = (match op with
          A.Mod     -> (L.build_urem, A.Int)
        | A.And     -> (L.build_and, A.Bool)
        | A.Or      -> (L.build_or, A.Bool)
        | A.Add     ->
          if t1 = A.Float
          then (L.build_fadd, A.Float)
          else (L.build_add, A.Int)
        | A.Sub     ->
          if t1 = A.Float
          then (L.build_fsub, A.Float)
          else (L.build_sub, A.Int)
        | A.Mult    ->
          if t1 = A.Float
          then (L.build_fmul, A.Float)
          else (L.build_mul, A.Int)
        | A.Div     ->
          if t1 = A.Float
          then (L.build_fdiv, A.Float)
          else (L.build_sdiv, A.Int)
        | A.Equal   ->
          if t1 = A.Float
          then (L.build_fcmp L.Fcmp.Oeq, A.Bool)
          else (L.build_icmp L.Icmp.Eq, A.Bool)
        | A.NEqual   ->
          if t1 = A.Float
          then (L.build_fcmp L.Fcmp.One, A.Bool)
          else (L.build_icmp L.Icmp.Ne, A.Bool)
        | A.Less   ->
          if t1 = A.Float
          then (L.build_fcmp L.Fcmp.Olt, A.Bool)
          else (L.build_icmp L.Icmp.Slt, A.Bool)
        | A.Leq   ->
          if t1 = A.Float
          then (L.build_fcmp L.Fcmp.Ole, A.Bool)
          else (L.build_icmp L.Icmp.Sle, A.Bool)
        | A.Greater   ->
          if t1 = A.Float
          then (L.build_fcmp L.Fcmp.Ogt, A.Bool)
          else (L.build_icmp L.Icmp.Sgt, A.Bool)
        | A.Geq   ->
          if t1 = A.Float
          then (L.build_fcmp L.Fcmp.Oge, A.Bool)
          else (L.build_icmp L.Icmp.Sge, A.Bool)) in
      (* ignore (print_endline (A.string_of_typ typ)); *)
         (oper e1' e2' "tmp" builder, typ)

    | A.ListAdd(l, d) ->
      let (l_val, l_typ) = expr builder l in
      let (d_val, d_typ) = expr builder d in
      let d_ptr = match d_typ with
        A.NodeTyp _ | A.TupleTyp _ | A.PackTyp _ | A.EdgeTyp _-> d_val
      | A.ListTyp _ -> d_val
      | _ ->
        let d_ltyp = ltype_of_typ typedefs d_typ in
        let d_ptr = L.build_malloc d_ltyp "tmp" builder in
        ignore (L.build_store d_val d_ptr builder);
        d_ptr in

      let void_d_ptr = L.build_bitcast d_ptr (L.pointer_type i8_t) "ptr" builder in
      ignore(L.build_call appendId_f [| l_val; void_d_ptr; void_d_ptr |] "tmp"
builder);
      (l_val, l_typ)

    | A.ListRmv(l, d) ->
      let (l_val, l_typ) = expr builder l in
      let (d_val, _) = expr builder d in
      let void_d_ptr = L.build_bitcast d_val (L.pointer_type i8_t) "ptr" builder in
      ignore (L.build_call removeIdList_f [| l_val; void_d_ptr |] "" builder);
      (l_val, l_typ)
    | A.Unop(op, e) ->
      let (e_val,e_typ) = expr builder e in
      (match e_typ with
        A.NodeTyp _ ->
          let chld_lst_ptr = L.build_struct_gep e_val 0 "chld_ptr" builder in
          let chld_lst = L.build_load chld_lst_ptr "chld" builder in
          let is_empty = L.build_call isEmptyIdList_f [| chld_lst |] "is_empty" builder
in
          (L.build_icmp L.Icmp.Ne is_empty (L.const_null i8_t) "is_empty_bool" builder,
A.Bool)
      | _ ->
        ((match op with
                    A.Neg     -> L.build_neg
                  | A.Not     -> L.build_not) e_val "tmp" builder, A.Bool))
    | A.Member (e, s) ->
      let (e_val,e_typ) = expr builder e in
      (* ignore(print_endline (L.string_of_lltype (L.type_of e_val))); *)
      let attr_binding = StringMap.find s (TypMap.find e_typ typedefs).attrs in
      let idx = attr_binding.idx in
      let ptr = L.build_struct_gep e_val idx "ptr" builder in
      (L.build_load ptr "tmp" builder, attr_binding.typ)

  (* FIX *)
    | A.Assign (e_d, e_s) ->
      let (lval,_) = expr builder e_s in
       let (dst_val,dst_typ) = get_dest_binding expr builder e_d in
      ignore (L.build_store lval dst_val builder); (lval, dst_typ)

  (* FIX *)
    | A.Print (fmt, act) ->
      let build_print_arg e =
        let (e_val, e_typ) = expr builder e in
          match e_typ with
            A.Float -> L.build_fpext e_val (L.double_type context) "dbl_cast" builder
          | _ -> e_val in
      let expr_lst = List.map build_print_arg act in
      let format_str = L.build_global_stringptr fmt "fmt" builder in
      (L.build_call printf_func (Array.of_list (format_str::expr_lst)) "printf"
builder, A.Void)

    | A.Call (f, act) ->
      let (fdef, fdecl) = StringMap.find f function_decls in
      let actuals = List.rev (List.map (fun e -> fst (expr builder e)) (List.rev act))
in
      let result = (match fdecl.A.f_typ with A.Void -> ""
                                          | _ -> f ^ "_result") in
      (L.build_call fdef (Array.of_list actuals) result builder, fdecl.A.f_typ)
    | A.Run (e, act) ->
      let (n_val, n_typ) = expr builder e in
      let ((do_typ,do_fun),_) = TypMap.find n_typ obj_funs in
      let actuals = n_val::(List.rev (List.map (fun e -> fst (expr builder e))
(List.rev act))) in
      let result = (match do_typ.A.f_typ with A.Void -> ""
                                | _ -> (A.string_of_typ n_typ)^"do_result" ) in
      (L.build_call do_fun (Array.of_list actuals) result builder, do_typ.A.f_typ)

    | A.Self -> expr builder (A.Id "self")

    | A.Message -> expr builder (A.Id "msg")

    | A.Null typ ->
      let ltyp = ltype_of_typ typedefs typ in
      (L.const_null ltyp,typ)


    | A.Tuple (typ, act) ->
       let ltyp = (TypMap.find (A.TupleTyp typ) typedefs).ltyp in
      (build_tuple_ptr ltyp "tuple" act expr builder, A.NodeTyp typ)

    | A.Node (typ, act) ->
      let ltyp = (TypMap.find (A.NodeTyp typ) typedefs).ltyp in
      let ((_,do_fun), (_,catch_fun)) = TypMap.find (A.NodeTyp typ) obj_funs in
      let actuals = List.append [L.build_call initIdList_f [||] "init" builder;
                              L.build_call initIdList_f [||] "init" builder;
                              do_fun;
                              catch_fun;]
                  (List.map (fun e -> fst (expr builder e)) act) in
      let node_ptr = L.build_malloc ltyp "tmp" builder in
      let add_attr i actual =
        let a_ptr = L.build_struct_gep node_ptr i "act" builder in
        ignore(L.build_store actual a_ptr builder); i+1 in
      ignore(List.fold_left add_attr 0 actuals);
      (node_ptr, A.NodeTyp typ)

    | A.Lst (typ, act) -> (* (L.const_null i32_t, A.Void) *)
       let d_ltyp = ltype_of_typ typedefs typ in
       let lst_ptr = L.build_call initIdList_f [||] "init" builder in
       let add_act dat =
         let d_ptr = match typ with
           A.NodeTyp _ | A.TupleTyp _ | A.PackTyp _ | A.EdgeTyp _->
             fst (expr builder dat)
         | A.ListTyp _ -> fst (expr builder dat)
         | _ ->
               let d_val = fst (expr builder dat) in
           let d_ptr = L.build_malloc d_ltyp "tmp" builder in
```

```
          ignore (L.build_store d_val d_ptr builder);
                d_ptr in

              let void_d_ptr = L.build_bitcast d_ptr (L.pointer_type i8_t) "ptr"
builder in
              ignore (L.build_call appendId_f [| lst_ptr; void_d_ptr; void_d_ptr |]
"tmp" builder)
       in ignore (List.map add_act act);
       (lst_ptr, A.ListTyp typ)

    | A.Pack (act) ->
      let (_,n_typ) = lookup "self" in
      let n_str = get_nstr n_typ in
      let pack_info = TypMap.find (A.PackTyp n_str) typedefs in
      let pack_ltyp = pack_info.ltyp in
      (build_tuple_ptr pack_ltyp "msg" act expr builder, A.PackTyp(n_str))

    | A.Graph (n_typ, edge_lst) ->
      let n_str = get_nstr n_typ in
      let edge_info = TypMap.find (A.EdgeTyp n_str) typedefs in
      let edge_ltyp = edge_info.ltyp in
      let build_edges edge_decl =
        let (src,dst,act_l) = edge_decl in
        let (src_val,_) = expr builder src in
        let (dst_val,_) = expr builder dst in

        let edge = build_tuple_ptr edge_ltyp "edge" act_l expr builder in
        let void_edge = L.build_bitcast edge (L.pointer_type i8_t) "edge_ptr" builder
in

        let chld_lst_ptr = L.build_struct_gep src_val 0 "chld_lst_ptr" builder in
        let chld_lst = L.build_load chld_lst_ptr "chld_lst" builder in
        let void_dst_val = L.build_bitcast dst_val (L.pointer_type i8_t) "dst_ptr"
builder in
        ignore (L.build_call appendId_f [| chld_lst; void_edge; void_dst_val |]
"append" builder);

        let prnt_lst_ptr = L.build_struct_gep dst_val 1 "prnt_lst_ptr" builder in
        let prnt_lst = L.build_load prnt_lst_ptr "prnt_lst" builder in
        let void_src_val = L.build_bitcast src_val (L.pointer_type i8_t) "src_ptr"
builder in
        ignore (L.build_call appendId_f [| prnt_lst; void_edge; void_src_val |]
"append" builder);

      in ignore(List.iter build_edges edge_lst);
      (L.const_null i8_t, A.Void)

              | A.Subscript(e, sub) ->
```

```
                  let (sub_val,_) = expr builder sub in
      let (e_val,e_typ) = expr builder e in
      match e_typ with
        A.ListTyp typ ->
          let l_node_ptr = L.build_call indexIdList_f [| e_val; sub_val |] "node"
builder in
          let l_node_data_ptr = L.build_struct_gep l_node_ptr 0 "l_node_data_ptr"
builder in
          (match typ with
            A.NodeTyp _ | A.TupleTyp _ | A.PackTyp _ | A.EdgeTyp _->
              let l_dtyp = ltype_of_typ typedefs typ in
              let void_d_ptr = L.build_load l_node_data_ptr "void_d_ptr" builder in
              (L.build_bitcast void_d_ptr l_dtyp "data" builder, typ)
          | A.ListTyp _ ->
              let l_dtyp = ltype_of_typ typedefs typ in
              let void_d_ptr = L.build_load l_node_data_ptr "void_d_ptr" builder in
              (L.build_bitcast void_d_ptr l_dtyp "data" builder, typ)
          | _ ->
            let l_dtyp = ltype_of_typ typedefs typ in
            let void_d_ptr = L.build_load l_node_data_ptr "void_d_ptr" builder in
            let d_ptr = L.build_bitcast void_d_ptr (L.pointer_type l_dtyp) "d_ptr"
builder in
            (L.build_load d_ptr "d_ptr" builder, typ))

      | A.NodeTyp n_str ->
          let e_ltyp = ltype_of_typ typedefs (A.EdgeTyp n_str) in
          (* let edge_info = TypMap.find (A.EdgeTyp n_str) typedefs in *)
          let chld_lst_ptr = L.build_struct_gep e_val 0 "chdl_lst_ptr" builder in
          let chld_lst = L.build_load chld_lst_ptr "chld_lst" builder in

          let id_val = L.build_bitcast sub_val (L.pointer_type i8_t) "void_id_val"
builder in
          let lst_node = L.build_call findNodeId_f [| chld_lst; id_val |] "lst_node"
builder in
          let void_d_ptr = L.build_struct_gep lst_node 0 "lst_node" builder in
          (* let void_d_ptr = L.build_load data_ptr "void_d_ptr" builder in *)
          let data_ptr = L.build_bitcast void_d_ptr (L.pointer_type e_ltyp) "d_ptr"
builder in
          (L.build_load data_ptr "data" builder, A.EdgeTyp(n_str))
      | typ -> raise (Invalid_argument (A.string_of_typ typ))
         in

    let add_terminal builder f =
      match L.block_terminator (L.insertion_block builder) with
                     Some _ -> ()
      | None -> ignore (f builder) in

    (* Build the code for the given statement; return the builder for
       the statement's successor *)
    let rec stmt builder = function
                    A.Block sl -> List.fold_left stmt builder sl
    | A.Expr e -> ignore (expr builder e); builder
    | A.Return e -> ignore (match fdecl.A.f_typ with
                    A.Void -> L.build_ret_void builder
                    | _ -> L.build_ret (fst (expr builder e)) builder); builder

    | A.If (predicate, then_stmt, else_stmt) ->
      let bool_val = fst (expr builder predicate) in
                    let merge_bb = L.append_block context "merge" the_function in

                    let then_bb = L.append_block context "then" the_function in
                    add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
                     (L.build_br merge_bb);

                    let else_bb = L.append_block context "else" the_function in
                    add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
                     (L.build_br merge_bb);

                    ignore (L.build_cond_br bool_val then_bb else_bb builder);
                    L.builder_at_end context merge_bb

    | A.While (predicate, body) ->
               let pred_bb = L.append_block context "while" the_function in
               ignore (L.build_br pred_bb builder);

               let body_bb = L.append_block context "while_body" the_function in
               add_terminal (stmt (L.builder_at_end context body_bb) body)
                 (L.build_br pred_bb);

               let pred_builder = L.builder_at_end context pred_bb in
               let bool_val = fst (expr pred_builder predicate) in

               let merge_bb = L.append_block context "merge" the_function in
               ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
               L.builder_at_end context merge_bb

    | A.For (e1, e2, e3, body) -> stmt builder
          ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr e3]) ] )

    | A.ForEach (s, l, body) ->
      let lst_node = L.build_alloca node_t "lst_node" builder in
      let (lst,typ) = expr builder l in
      let head_ptr_ptr = L.build_struct_gep lst 0 "head_ptr" builder in
      let head_ptr = L.build_load head_ptr_ptr "head" builder in
      ignore (L.build_store head_ptr lst_node builder);
```

```
      let pred_bb = L.append_block context "while" the_function in
      ignore (L.build_br pred_bb builder);

      let body_bb = L.append_block context "while_body" the_function in
      let body_builder = L.builder_at_end context body_bb in
      let (l_dtyp,dtyp) = match typ with
        A.ListTyp dtyp -> (ltype_of_typ typedefs dtyp, dtyp)
      | _ -> (void_t, A.Void) in

      let curr = L.build_load lst_node "curr" body_builder in
      let void_data_ptr_ptr = L.build_struct_gep curr 0 "void_data_ptr_ptr"
body_builder in
      let void_data_ptr = L.build_load void_data_ptr_ptr "void_data_ptr" body_builder
in
      let data_val = match dtyp with
        A.NodeTyp _ | A.TupleTyp _ | A.EdgeTyp _
          -> L.build_bitcast void_data_ptr l_dtyp "data" body_builder
      | A.ListTyp _
          -> L.build_bitcast void_data_ptr l_dtyp "data" body_builder
      | _ ->
          let data_ptr =
          L.build_bitcast void_data_ptr (L.pointer_type l_dtyp) "data_ptr" body_builder
in
          L.build_load data_ptr "data" body_builder in
      let (dst_ptr,_) = get_dest_binding expr body_builder s in
      ignore(L.build_store data_val dst_ptr body_builder);

      let body_builder = stmt body_builder body in

      let next_ptr = L.build_struct_gep curr 2 "next_ptr" body_builder in
      let next = L.build_load next_ptr "next" body_builder in
      ignore(L.build_store next lst_node body_builder);
      add_terminal body_builder (L.build_br pred_bb);

      let pred_builder = L.builder_at_end context pred_bb in
      let bool_val =
        let curr = L.build_load lst_node "curr" pred_builder in
        L.build_icmp L.Icmp.Ne curr (L.const_null node_t) "tmp" pred_builder in

      let merge_bb = L.append_block context "merge" the_function in
      ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
      L.builder_at_end context merge_bb

    | A.Pass (msg, p_typ) ->
      let (msg_val,_) = expr builder msg in

      let (self_ptr,n_typ) = (lookup "self") in
      let self = L.build_load self_ptr "self" builder in
      let (_,(_,catch_fun)) = TypMap.find n_typ obj_funs in
      let lst_node = L.build_alloca node_t "curr_node" builder in

      let chld_lst_ptr = L.build_struct_gep self 0 "lst_ptr" builder in
      let prnt_lst_ptr = L.build_struct_gep self 1 "lst_ptr" builder in

      let lst_iter_catch builder lst_ptr =
        let lst = L.build_load lst_ptr "lst" builder in
        let head_ptr = L.build_struct_gep lst 0 "head_ptr" builder in
        let head = L.build_load head_ptr "head" builder in
        ignore(L.build_store head lst_node builder);
        let pred_bb = L.append_block context "while" the_function in
        ignore (L.build_br pred_bb builder);

        let body_bb = L.append_block context "while_body" the_function in
        let body_builder = L.builder_at_end context body_bb in

        let curr = L.build_load lst_node "curr" body_builder in
        let void_dst_ptr = L.build_struct_gep curr 1 "void_dst_ptr" body_builder in
        let void_dst = L.build_load void_dst_ptr "void_dst" body_builder in
        let dst = L.build_bitcast void_dst (ltype_of_typ typedefs n_typ) "dst"
body_builder in
        ignore (L.build_call catch_fun [| dst; msg_val |] "" body_builder);

        let next_ptr = L.build_struct_gep curr 2 "next_ptr" body_builder in
        let next = L.build_load next_ptr "next" body_builder in
        ignore(L.build_store next lst_node body_builder);
        add_terminal body_builder (L.build_br pred_bb);

        let pred_builder = L.builder_at_end context pred_bb in
        let bool_val =
          let curr = L.build_load lst_node "curr" pred_builder in
          L.build_icmp L.Icmp.Ne curr (L.const_null node_t) "tmp" pred_builder in

        let merge_bb = L.append_block context "merge" the_function in
        ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
        L.builder_at_end context merge_bb
      in

      match p_typ with
        A.Prnt      -> List.fold_left lst_iter_catch builder [prnt_lst_ptr]
      | A.Chld      -> List.fold_left lst_iter_catch builder [chld_lst_ptr]
      | A.PrntChld  -> List.fold_left lst_iter_catch builder [prnt_lst_ptr;
chld_lst_ptr]
      | A.ChldPrnt  -> List.fold_left lst_iter_catch builder [chld_lst_ptr;
prnt_lst_ptr]
      | A.Target dst  ->
          let (dst_val,_) = expr builder dst in
          ignore(L.build_call catch_fun [| dst_val; msg_val |] "" builder); builder

    in


    (* Build the code for each statement in the function *)
    let builder = stmt builder (A.Block fdecl.A.f_body) in

    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.A.f_typ with
        A.Void -> L.build_ret_void
      | t -> L.build_ret (L.const_int (ltype_of_typ typedefs t) 0))
  in
  List.iter (build_function_body Do) do_decls;
  List.iter (build_function_body Catch) catch_decls;
  List.iter (build_function_body Func) functions;
the_module
```

Makefile

```
OBJS = ast.cmx parser.cmx scanner.cmx semant.cmx goblan.cmx

goblan.native :
	ocamlbuild -use-ocamlfind -pkgs
llvm,llvm.analysis,llvm.linker,llvm.bitreader,llvm.irreader -cflags -w,+a-4
goblan.native

goblan : $(OBJS)
	ocamlfind ocamlopt -linkpkg -package llvm.linker -package llvm.bitreader
-package llvm.irreader -package llvm -package llvm.analysis $(OBJS) -o goblan

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -c $<

%.cmi : %.mli
	ocamlc -c $<

%.cmx : %.ml
	ocamlfind ocamlopt -c -package llvm $<

.PHONY : clean
clean :
	ocamlbuild -clean
	rm -rf testall.log *.diff goblan scanner.ml parser.ml parser.mli
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o

# Generated by ocamldep *.ml *.mli
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo list.bc
codegen.cmx : ast.cmx list.bc
goblan.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
goblan.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
goblan.cmo : scanner.cmo parser.cmi ast.cmo
goblan.cmx : scanner.cmx parser.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmx
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo
```

list.h

```
#ifndef _LIST_H_
#define _LIST_H_

struct ListNode {
    void *data;
  void *Id;
    struct ListNode *next;
};

struct IdList {
  struct ListNode *head;
};

struct IdList* initIdList();

struct ListNode *addFrontId(struct IdList *list, void *data, void *Id);

struct ListNode *appendId(struct IdList *list, void *data, void *Id);

struct ListNode *indexIdList(struct IdList *list, int idx);

void removeIdList(struct IdList *list, void *Id);

struct ListNode *findNodeId(struct IdList *list, void *Id);

int isEmptyList(struct IdList *list);

#endif /* #ifndef _LIST_H_ */
```

list.c

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"


struct IdList* initIdList()
{
  struct IdList* new = (struct IdList*) malloc(sizeof(struct IdList));
  new->head = 0;
  return new;
}

struct ListNode *addFrontId(struct IdList *list, void *data, void *Id){
  struct ListNode *node = (struct ListNode*) malloc(sizeof(struct ListNode));
  if (node == NULL){
    perror("malloc returned a NULL");
    exit(1);
  }
  node->data = data;
  node->Id = Id;
  node->next = list->head;
  list->head = node;
  return node;
};

struct ListNode *appendId(struct IdList *list, void *data, void *Id){
  struct ListNode* new;
  if (list->head == NULL){
    list->head = (struct ListNode*) malloc(sizeof(struct ListNode));
    new = list->head;
  }
  else{
    struct ListNode* node = list->head;
    while (node->next != NULL)
      node = node->next;
    node->next = (struct ListNode*) malloc(sizeof(struct ListNode));
    new = node->next;
  }
  new->Id = Id;
  new->next = NULL;
  new->data = data;
  return new;
};
```

```
struct ListNode *indexIdList(struct IdList *list, int idx){
  struct ListNode* node = list->head;
  while (idx > 0){
    node = node->next;
    idx--;
  }
  return node;
};

void removeIdList(struct IdList *list, void *Id){
  if (isEmptyList(list))
    return;

  if (list->head->Id == Id){
    struct ListNode *tmp = list->head;
    list->head = list->head->next;
    free(tmp);
    return;
  }

  struct ListNode* fst = list->head;
  struct ListNode* snd = list->head->next;
  while(snd){
    if (snd->Id != Id){
      snd = snd->next;
      continue;
    }
    fst->next = snd->next;
    free(snd);
    return;
  }
};

struct ListNode *findNodeId(struct IdList *list, void *Id){
    struct ListNode* node = list->head;
    while (node != 0){
        if (node->Id == Id)
            return node;
        node = node->next;
    }
    return NULL;
}

int isEmptyList(struct IdList *list)
{
  return (list->head == 0);
}
```

goblan.sh

```
#!/bin/bash

clang -emit-llvm -o list.bc -c src/list.c
if [ $# -eq 1 ]
then
    ./goblan.native <$1 >a.ll
else
    ./goblan.native $1 <$2 >a.ll
fi
llvm-link list.bc a.ll -S > run.ll
clang run.ll
./a.out
rm a.ll
rm run.ll
rm ./a.out
```

Testall.sh

```
#!/bin/sh

# Regression testing script for GOBLAN programming language
# Step through a list of files
# Compile, run, and check the output of each expected-to-work test
# Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the GOBLAN compiler.
GOBLAN="./goblan.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.gb files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
      echo "FAILED"
      error=1
    fi
    echo "  $1"
}
```

```
# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to
difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
      SignalError "$1 differs"
      echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
      SignalError "$1 failed on $*"
      return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
      SignalError "failed: $* did not report an error"
      return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.gb//'`
    reffile=`echo $1 | sed 's/.gb$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
    Run "$GOBLAN" "<" $1 ">" "${basename}.ll" &&
    Run "$LLI" "${basename}.ll" ">" "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
      if [ $keep -eq 0 ] ; then
          rm -f $generatedfiles
      fi
      echo "OK"
      echo "###### SUCCESS" 1>&2
    else
      echo "###### FAILED" 1>&2
      globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.gb//'`
    reffile=`echo $1 | sed 's/.gb$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$GOBLAN" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files
    if [ $error -eq 0 ] ; then
      if [ $keep -eq 0 ] ; then
          rm -f $generatedfiles
      fi
      echo "OK"
      echo "###### SUCCESS" 1>&2
    else
      echo "###### FAILED" 1>&2
      globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
      k) # Keep intermediate files
          keep=1
          ;;
      h) # Help
          Usage
          ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in
testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail


if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.gb tests/fail-*.gb"
fi

for file in $files
```

```
do
    case $file in
      *test-*)
          Check $file 2>> $globallog
          ;;
      *fail-*)
          CheckFail $file 2>> $globallog
          ;;
      *)
          echo "unknown file type $file"
          globalerror=1
          ;;
    esac
done

exit $globalerror
```

Unit Tests

```
--------------------------------------------------------------------------------
fail-function-assign-tuple-member.gb / fail-function-assign-tuple-member.err
--------------------------------------------------------------------------------

tuple:A {
  int a;
}

int main() {
    tuple:A b;
    b = new tuple:A(1);
    b.a = 3.6;
    return 0;
}

Fatal error: exception Failure("In main, illegal assignment int = float in b.a = 3.6")

--------------------------------------------------------------------------------
fail-function-binop-add.gb / fail-function-binop-add.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i + f;
  return 0;
}

Fatal error: exception Failure("In main, illegal binary operator int + float in i + f")

--------------------------------------------------------------------------------
fail-function-binop-and.gb / fail-function-binop-and.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i && f;
  return 0;
}
```

```
Fatal error: exception Failure("In main, illegal binary operator int && float in i &&
f")

--------------------------------------------------------------------------------
fail-function-binop-assign.gb / fail-function-binop-assign.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i = f;
  return 0;
}

Fatal error: exception Failure("In main, illegal assignment int = float in i = f")

--------------------------------------------------------------------------------
fail-function-binop-divide.gb / fail-function-binop-divide.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i / f;
  return 0;
}

Fatal error: exception Failure("In main, illegal binary operator int / float in i / f")

--------------------------------------------------------------------------------
fail-function-binop-equal.gb / fail-function-binop-equal.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i == f;
  return 0;
}
```

```
Fatal error: exception Failure("In main, illegal binary operator int == float in i ==
f")

--------------------------------------------------------------------------------
fail-function-binop-geq.gb / fail-function-binop-geq.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i >= f;
  return 0;
}

Fatal error: exception Failure("In main, illegal binary operator int >= float in i >=
f")

--------------------------------------------------------------------------------
fail-function-binop-greater.gb / fail-function-binop-greater.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i > f;
  return 0;
}

Fatal error: exception Failure("In main, illegal binary operator int > float in i > f")

--------------------------------------------------------------------------------
fail-function-binop-leq.gb / fail-function-binop-leq.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i <= f;
  return 0;
}
```

```
Fatal error: exception Failure("In main, illegal binary operator int <= float in i <=
f")

--------------------------------------------------------------------------------
fail-function-binop-less.gb / fail-function-binop-less.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i < f;
  return 0;
}

Fatal error: exception Failure("In main, illegal binary operator int < float in i < f")

--------------------------------------------------------------------------------
fail-function-binop-mod.gb / fail-function-binop-mod.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i % f;
  return 0;
}

Fatal error: exception Failure("In main, illegal binary operator int % float in i % f")

--------------------------------------------------------------------------------
fail-function-binop-mult.gb / fail-function-binop-mult.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i * f;
  return 0;
}

Fatal error: exception Failure("In main, illegal binary operator int * float in i * f")
```

```
--------------------------------------------------------------------------------
fail-function-binop-nequal.gb / fail-function-binop-nequal.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i != f;
  return 0;
}

Fatal error: exception Failure("In main, illegal binary operator int != float in i !=
f")

--------------------------------------------------------------------------------
fail-function-binop-or.gb / fail-function-binop-or.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i || f;
  return 0;
}

Fatal error: exception Failure("In main, illegal binary operator int || float in i ||
f")

--------------------------------------------------------------------------------
fail-function-binop-subtract.gb / fail-function-binop-subtract.err
--------------------------------------------------------------------------------

int main() {
  int i;
  float f;
  i = 0;
  f = 1.0;
  i - f;
  return 0;
}

Fatal error: exception Failure("In main, illegal binary operator int - float in i - f")

--------------------------------------------------------------------------------
```

```
fail-function-call-arg-length.gb / fail-function-call-arg-length.err
--------------------------------------------------------------------------------

void test(int a) {

}

int main() {
  test();
  return 0;
}

Fatal error: exception Failure("In main, expecting 1 arguments in call to test()")

--------------------------------------------------------------------------------
fail-function-call-arg-type.gb / fail-function-call-arg-type.err
--------------------------------------------------------------------------------

void test(int a) {

}

int main() {
  float f;
  f = 1.2;
  test(f);
  return 0;
}

Fatal error: exception Failure("In main, illegal actual argument float, expected int in
parameter f in call to test(f)")

--------------------------------------------------------------------------------
fail-function-duplicate.gb / fail-function-duplicate.err
--------------------------------------------------------------------------------

void a() {
}

void a() {
}

int main() {
  return 0;
}

Fatal error: exception Failure("Duplicate function a")
```

```
--------------------------------------------------------------------------------
fail-function-foreach-type-mismatch.gb / fail-function-foreach-type-mismatch.err
--------------------------------------------------------------------------------

int main() {
  list int li;
  float f;
  li = new int [|1, 2|];
  for(f in li) {}
  return 0;
}

Fatal error: exception Failure("In main, type mismatch float != int in f in li")

--------------------------------------------------------------------------------
fail-function-foreach-wrong-types.gb / fail-function-foreach-wrong-types.err
--------------------------------------------------------------------------------

int main() {
  float a;
  float b;
  for(a in b) {}
  return 0;
}

Fatal error: exception Failure("In main, expected identifier and list in, a in b")

--------------------------------------------------------------------------------
fail-function-formal-duplicate.gb / fail-function-formal-duplicate.err
--------------------------------------------------------------------------------

void test(int a, int a) {
}

int main() {
  return 0;
}

Fatal error: exception Failure("In test, duplicate formal a")

--------------------------------------------------------------------------------
fail-function-formal-pack.gb / fail-function-formal-pack.err
--------------------------------------------------------------------------------

void test(pack p) {
}

int main() {
  return 0;
}

Fatal error: exception Failure("In test, illegal pack formal p")

--------------------------------------------------------------------------------
fail-function-formal-undefined-node.gb / fail-function-formal-undefined-node.err
--------------------------------------------------------------------------------

void test(node:A a) {
}

int main() {
  return 0;
}

Fatal error: exception Failure("In test, undefined reference to node:A")

--------------------------------------------------------------------------------
fail-function-formal-undefined-tuple.gb / fail-function-formal-undefined-tuple.err
--------------------------------------------------------------------------------

void test(tuple:A a) {
}

int main() {
  return 0;
}

Fatal error: exception Failure("In test, undefined reference to tuple:A")

--------------------------------------------------------------------------------
fail-function-formal-void.gb / fail-function-formal-void.err
--------------------------------------------------------------------------------

void test(void a) {
}

int main() {
  return 0;
}

Fatal error: exception Failure("In test, illegal void formal a")

--------------------------------------------------------------------------------
fail-function-if-not-boolean.gb / fail-function-if-not-boolean.err
--------------------------------------------------------------------------------
```

```
int main() {
  if(1) {}
  return 0;
}

Fatal error: exception Failure("In main, expected a boolean expression, in 1")

--------------------------------------------------------------------------------
fail-function-list-add-not-a-list.gb / fail-function-list-add-not-a-list.err
--------------------------------------------------------------------------------

int main() {
  list int li;
  int i;
  li = new int [|1, 2|];
  i += 3;
  return 0;
}

Fatal error: exception Failure("In main, expected left side of operator to be of type
list, in i += 3")

--------------------------------------------------------------------------------
fail-function-list-add-wrong-type.gb / fail-function-list-add-wrong-type.err
--------------------------------------------------------------------------------

int main() {
  list int li;
  li = new int [|1, 2|];
  li += 3.4;
  return 0;
}

Fatal error: exception Failure("In main, illegal list addition, list int += float in li
+= 3.4")

--------------------------------------------------------------------------------
fail-function-list-rmv-non-mutable.gb / fail-function-list-rmv-non-mutable.err
--------------------------------------------------------------------------------

int main() {
  list int li;
  li = new int [|1, 2|];
  li -= 2;
  return 0;
}
```

```
Fatal error: exception Failure("In main, illegal list removal of non-mutable list type,
list int -= int in li -= 2")

--------------------------------------------------------------------------------
fail-function-list-rmv-not-a-list.gb / fail-function-list-rmv-not-a-list.err
--------------------------------------------------------------------------------

int main() {
  list int li;
  int i;
  li = new int [|1, 2|];
  i -= 3;
  return 0;
}

Fatal error: exception Failure("In main, expected left side of operator to be of type
list, in i -= 3")

--------------------------------------------------------------------------------
fail-function-list-rmv-type-mismatch.gb / fail-function-list-rmv-type-mismatch.err
--------------------------------------------------------------------------------

tuple:A {}
tuple:B {}

int main() {
  list tuple:A lt;
  tuple:B b;
  lt = new tuple:A [|new tuple:A()|];
  lt -= b;
  return 0;
}

Fatal error: exception Failure("In main, type mismatch in list removal, list tuple:A -=
tuple:B in lt -= b")

--------------------------------------------------------------------------------
fail-function-list-rmv-wrong-type.gb / fail-function-list-rmv-wrong-type.err
--------------------------------------------------------------------------------

int main() {
  list int li;
  li = new int [|1, 2|];
  li -= 3.4;
  return 0;
}
```

```
Fatal error: exception Failure("In main, illegal list removal of non-mutable list type,
list int -= float in li -= 3.4")

--------------------------------------------------------------------------------
fail-function-local-duplicate.gb / fail-function-local-duplicate.err
--------------------------------------------------------------------------------

int main () {
  int a;
  int a;
  return 0;
}

Fatal error: exception Failure("In main, duplicate local a")

--------------------------------------------------------------------------------
fail-function-local-pack.gb / fail-function-local-pack.err
--------------------------------------------------------------------------------

int main() {
  pack p;
  return 0;
}

Fatal error: exception Failure("In main, illegal pack local p")

--------------------------------------------------------------------------------
fail-function-local-undefined-node.gb / fail-function-local-undefined-node.err
--------------------------------------------------------------------------------

int main() {
  node:A a;
  return 0;
}

Fatal error: exception Failure("In main, undefined reference to node:A")

--------------------------------------------------------------------------------
fail-function-local-undefined-tuple.gb / fail-function-local-undefined-tuple.err
--------------------------------------------------------------------------------

int main() {
  tuple:A a;
  return 0;
}

Fatal error: exception Failure("In main, undefined reference to tuple:A")
```

```
--------------------------------------------------------------------------------
fail-function-local-void.gb / fail-function-local-void.err
--------------------------------------------------------------------------------

int main() {
  void a;
  return 0;
}

Fatal error: exception Failure("In main, illegal void local a")

--------------------------------------------------------------------------------
fail-function-msg.gb / fail-function-msg.err
--------------------------------------------------------------------------------

int main() {
  msg;
  return 0;
}

Fatal error: exception Failure("In main, illegal use of msg")

--------------------------------------------------------------------------------
fail-function-name-node.gb / fail-function-name-node.err
--------------------------------------------------------------------------------

void node() {

}

int main() {
  return 0;
}

Fatal error: exception Failure("Function node may not be defined")

--------------------------------------------------------------------------------
fail-function-name-tuple.gb / fail-function-name-tuple.err
--------------------------------------------------------------------------------

void tuple() {

}

int main() {
  return 0;
}
```

```
Fatal error: exception Failure("Function tuple may not be defined")

--------------------------------------------------------------------------------
fail-function-no-main.gb / fail-function-no-main.err
--------------------------------------------------------------------------------

void test() {

}

Fatal error: exception Failure("Main function not found")


--------------------------------------------------------------------------------
fail-function-not-found.gb / fail-function-not-found.err
--------------------------------------------------------------------------------

int main() {
  test();
  return 0;
}

Fatal error: exception Failure("Unrecognized function test")

--------------------------------------------------------------------------------
fail-function-pass.gb / fail-function-pass.err
--------------------------------------------------------------------------------

int main() {
  pass p->chld;
  return 0;
}

Fatal error: exception Failure("In main, illegal use of pass")

--------------------------------------------------------------------------------
fail-function-return-code-after.gb / fail-function-return-code-after.err
--------------------------------------------------------------------------------

int main() {
  int i;
  return 0;
  i = 1;
}

Fatal error: exception Failure("In main, illegal code following return")

--------------------------------------------------------------------------------
```

```
fail-function-return-type-mismatch.gb / fail-function-return-type-mismatch.err
--------------------------------------------------------------------------------

int main() {
  return 1.2;
}

Fatal error: exception Failure("In main, return type is float but expecting int in
1.2")

--------------------------------------------------------------------------------
fail-function-self.gb / fail-function-self.err
--------------------------------------------------------------------------------

int main() {
  self;
  return 0;
}

Fatal error: exception Failure("In main, illegal use of self")

--------------------------------------------------------------------------------
fail-function-subscript.gb / fail-function-subscript.err
--------------------------------------------------------------------------------

int main() {
  int i;
  int j;
  i[j];
  return 0;
}

Fatal error: exception Failure("In main, expected left side of operator to be of type
list or node, in i[ j ] ")

--------------------------------------------------------------------------------
fail-function-subscript-list.gb / fail-function-subscript-list.err
--------------------------------------------------------------------------------

int main() {
  list int li;
  li = new int [|1, 2|];
  li[2.1];
  return 0;
}

Fatal error: exception Failure("In main, expected integer index type for list subscript
access, in li[ 2.1 ]")
```

```
--------------------------------------------------------------------------------
fail-function-subscript-node.gb / fail-function-subscript-node.err
--------------------------------------------------------------------------------

node:A {
  data{}
  edge{}
  pack{}
  void do(){}
  catch{}
}

int main() {
  node:A n;
  n = new node:A();
  n[0];
  return 0;
}

Fatal error: exception Failure("In main, expected node index type for node subscript
access, in n[ 0 ]")

--------------------------------------------------------------------------------
fail-function-undeclared-identifier.gb / fail-function-undeclared-identifier.err
--------------------------------------------------------------------------------

int main() {
  i;
  return 0;
}

Fatal error: exception Failure("In main, undeclared identifier i")

--------------------------------------------------------------------------------
fail-function-using-pass.gb / fail-function-using-pass.err
--------------------------------------------------------------------------------

void test() {
  pass somepackage->chld;
}

int main() {
  return 0;
}

Fatal error: exception Failure("In test, illegal use of pass")
```

```
--------------------------------------------------------------------------------
fail-function-while-not-boolean.gb / fail-function-while-not-boolean.err
--------------------------------------------------------------------------------

int main() {
  while(1.2) {}
  return 0;
}

Fatal error: exception Failure("In main, expected a boolean expression, in 1.2")

--------------------------------------------------------------------------------
fail-global-duplicate.gb / fail-global-duplicate.err
--------------------------------------------------------------------------------

int i;
int i;

int main() {
  return 0;
}

Fatal error: exception Failure("Duplicate global i")

--------------------------------------------------------------------------------
fail-global-undefined-node.gb / fail-global-undefined-node.err
--------------------------------------------------------------------------------

node:A a;

int main() {
  return 0;
}

Fatal error: exception Failure("In globals, undefined reference to node:A")

--------------------------------------------------------------------------------
fail-global-undefined-tuple.gb / fail-global-undefined-tuple.err
--------------------------------------------------------------------------------

tuple:A a;

int main() {
  return 0;
}

Fatal error: exception Failure("In globals, undefined reference to tuple:A")
```

```
--------------------------------------------------------------------------------
fail-global-void.gb / fail-global-void.err
--------------------------------------------------------------------------------

void v;

int main() {
  return 0;
}

Fatal error: exception Failure("Illegal void global v")

--------------------------------------------------------------------------------
fail-node-catch-local-duplicate.gb / fail-node-catch-local-duplicate.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do() {}
  catch {
    int a;
    int a;
  }
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, duplicate variable a in catch locals")

--------------------------------------------------------------------------------
fail-node-catch-local-undeclared-identifier.gb /
fail-node-catch-local-undeclared-identifier.err
--------------------------------------------------------------------------------

node:A {
  data{}
  edge{}
  pack{}
  void do(){}
  catch{
    i;
  }
}
```

```
int main() {
  return 0;
}

Fatal error: exception Failure("In node:A catch block, undeclared identifier i")

--------------------------------------------------------------------------------
fail-node-catch-local-undefined-node.gb / fail-node-catch-local-undefined-node.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do() {}
  catch {
    node:B b;
  }
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undefined reference to node:B")

--------------------------------------------------------------------------------
fail-node-catch-local-undefined-tuple.gb / fail-node-catch-local-undefined-tuple.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do() {}
  catch {
    tuple:B b;
  }
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undefined reference to tuple:B")

--------------------------------------------------------------------------------
fail-node-catch-local-void.gb / fail-node-catch-local-void.err
```

```
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do() {}
  catch{
    void v;
  }
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, illegal void type v in catch locals")

--------------------------------------------------------------------------------
fail-node-data-duplicate.gb / fail-node-data-duplicate.err
--------------------------------------------------------------------------------

node:A {
  data {
    int a;
    int a;
  }
  edge {}
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, duplicate variable a in data block")

--------------------------------------------------------------------------------
fail-node-data-pack.gb / fail-node-data-pack.err
--------------------------------------------------------------------------------

node:A {
  data {
    pack pa;
  }
  edge {}
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, illegal pack type pa in data block")

--------------------------------------------------------------------------------
fail-node-data-undefined-node.gb / fail-node-data-undefined-node.err
--------------------------------------------------------------------------------

node:A {
  data {
    node:B b;
  }
  edge {}
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undefined reference to node:B")

--------------------------------------------------------------------------------
fail-node-data-undefined-tuple.gb / fail-node-data-undefined-tuple.err
--------------------------------------------------------------------------------

node:A {
  data {
    tuple:A a;
  }
  edge {}
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}
```

```
Fatal error: exception Failure("In node:A do block, undefined reference to tuple:A")

--------------------------------------------------------------------------------
fail-node-data-void.gb / fail-node-data-void.err
--------------------------------------------------------------------------------

node:A {
  data {
    void a;
  }
  edge {}
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, illegal void type a in data block")


--------------------------------------------------------------------------------
fail-node-do-formal-duplicate.gb / fail-node-do-formal-duplicate.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do(int a, int a) {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, duplicate variable a in do formals")

--------------------------------------------------------------------------------
fail-node-do-formal-pack.gb / fail-node-do-formal-pack.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do(pack pa) {
  }
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, illegal pack type pa in do formals")

--------------------------------------------------------------------------------
fail-node-do-formal-undefined-node.gb / fail-node-do-formal-undefined-node.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do(node:B b) {
  }
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undefined reference to node:B")

--------------------------------------------------------------------------------
fail-node-do-formal-undefined-tuple.gb / fail-node-do-formal-undefined-tuple.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do(tuple:B b) {
  }
  catch {}
}

int main() {
  return 0;
}
```

```
Fatal error: exception Failure("In node:A do block, undefined reference to tuple:B")

--------------------------------------------------------------------------------
fail-node-do-formal-void.gb / fail-node-do-formal-void.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do(void v) {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, illegal void type v in do formals")

--------------------------------------------------------------------------------
fail-node-do-local-duplicate.gb / fail-node-do-local-duplicate.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do() {
    int a;
    int a;
  }
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, duplicate variable a in do locals")

--------------------------------------------------------------------------------
fail-node-do-local-undeclared-identifier.gb /
fail-node-do-local-undeclared-identifier.err
--------------------------------------------------------------------------------

node:A {
  data{}
  edge{}
  pack{}
  void do(){
    i;
  }
  catch{}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undeclared identifier i")

--------------------------------------------------------------------------------
fail-node-do-local-undefined-node.gb / fail-node-do-local-undefined-node.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do() {
    node:B b;
  }
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undefined reference to node:B")

--------------------------------------------------------------------------------
fail-node-do-local-undefined-tuple.gb / fail-node-do-local-undefined-tuple.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do() {
    tuple:B b;
  }
  catch {}
}
```

```
int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undefined reference to tuple:B")

--------------------------------------------------------------------------------
fail-node-do-local-void.gb / fail-node-do-local-void.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do() {
    void v;
  }
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, illegal void type v in do locals")

--------------------------------------------------------------------------------
fail-node-duplicate.gb / fail-node-duplicate.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {}
  void do() {}
  catch {}
}

node:A {
  data {}
  edge {}
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("Duplicate node node:A")

--------------------------------------------------------------------------------
fail-node-edge-duplicate.gb / fail-node-edge-duplicate.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge{
    int a;
    int a;
  }
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, duplicate variable a in edge block")

--------------------------------------------------------------------------------
fail-node-edge-pack.gb / fail-node-edge-pack.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {
    pack pa;
  }
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, illegal pack type pa in edge block")

--------------------------------------------------------------------------------
fail-node-edge-undefined-node.gb / fail-node-edge-undefined-node.err
```

```
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {
    node:B b;
  }
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undefined reference to node:B")

--------------------------------------------------------------------------------
fail-node-edge-undefined-tuple.gb / fail-node-edge-undefined-tuple.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {
    tuple:B b;
  }
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undefined reference to tuple:B")

--------------------------------------------------------------------------------
fail-node-edge-void.gb / fail-node-edge-void.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {
    void a;
  }
  pack {}
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, illegal void type a in edge block")

--------------------------------------------------------------------------------
fail-node-pack-duplicate.gb / fail-node-pack-duplicate.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {
    int a;
    int a;
  }
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, duplicate variable a in pack block")

--------------------------------------------------------------------------------
fail-node-pack-pack.gb / fail-node-pack-pack.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {
    pack pa;
  }
  void do() {}
  catch {}
}

int main() {
  return 0;
}
```

```
Fatal error: exception Failure("In node:A, illegal pack type pa in pack block")

--------------------------------------------------------------------------------
fail-node-pack-undefined-node.gb / fail-node-pack-undefined-node.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {
    node:B b;
  }
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undefined reference to node:B")

--------------------------------------------------------------------------------
fail-node-pack-undefined-tuple.gb / fail-node-pack-undefined-tuple.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {
    tuple:A a;
  }
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A do block, undefined reference to tuple:A")

--------------------------------------------------------------------------------
fail-node-pack-void.gb / fail-node-pack-void.err
--------------------------------------------------------------------------------

node:A {
  data {}
  edge {}
  pack {
    void a;
  }
  void do() {}
  catch {}
}

int main() {
  return 0;
}

Fatal error: exception Failure("In node:A, illegal void type a in pack block")

--------------------------------------------------------------------------------
fail-tuple-attribute-duplicate.gb / fail-tuple-attribute-duplicate.err
--------------------------------------------------------------------------------

tuple:A {
  int a;
  int a;
}

int main() {
  return 0;
}

Fatal error: exception Failure("Duplicate tuple attribute a")

--------------------------------------------------------------------------------
fail-tuple-attribute-pack.gb / fail-tuple-attribute-pack.err
--------------------------------------------------------------------------------

tuple:A {
  pack pa;
}

int main() {
  return 0;
}

Fatal error: exception Failure("Illegal pack attribute pa in tuple:A")

--------------------------------------------------------------------------------
fail-tuple-attributes-undefined-node.gb / fail-tuple-attributes-undefined-node.err
--------------------------------------------------------------------------------
```

```
tuple:A {
  node:A a;
}

int main() {
  return 0;
}

Fatal error: exception Failure("In tuple:A, undefined reference to node:A")

--------------------------------------------------------------------------------
fail-tuple-attributes-undefined-tuple.gb / fail-tuple-attributes-undefined-tuple.err
--------------------------------------------------------------------------------

tuple:A {
  tuple:B b;
}

int main() {
  return 0;
}

Fatal error: exception Failure("In tuple:A, undefined reference to tuple:B")

--------------------------------------------------------------------------------
fail-tuple-attribute-void.gb / fail-tuple-attribute-void.err
--------------------------------------------------------------------------------

tuple:A {
  void a;
}

int main() {
  return 0;
}

Fatal error: exception Failure("Illegal void attribute a in tuple:A")

--------------------------------------------------------------------------------
fail-tuple-duplicate.gb / fail-tuple-duplicate.err
--------------------------------------------------------------------------------

tuple:A {}

tuple:A {}

int main() {
  return 0;
}

Fatal error: exception Failure("Duplicate tuple tuple:A")

--------------------------------------------------------------------------------
test-binop-add.out / test-binop-add.gb
--------------------------------------------------------------------------------

14

int main() {
  int a;
  int b;
  int c;
  a = 9;
  b = 5;
  c = a + b;
  print("%d\n", c);
  return 0;
}

--------------------------------------------------------------------------------
test-binop-and.out / test-binop-and.gb
--------------------------------------------------------------------------------

working

int main() {
  bool t;
  bool f;
  bool r;
  t = true;
  f = false;
  r = t && f;
  if(r) {
    print("NOT working\n");
  } else {
    print("working\n");
  }
  return 0;
}

--------------------------------------------------------------------------------
test-binop-divide.out / test-binop-divide.gb
--------------------------------------------------------------------------------

1
```

```
int main() {
  int a;
  int b;
  int c;
  a = 9;
  b = 5;
  c = a / b;
  print("%d\n", c);
  return 0;
}

--------------------------------------------------------------------------------
test-binop-equal.out / test-binop-equal.gb
--------------------------------------------------------------------------------

working

int main() {
  int i;
  int j;
  i = 0;
  j = 0;
  if(i == j) {
    print("working\n");
  } else {
    print("NOT working\n");
  }
  return 0;
}

--------------------------------------------------------------------------------
test-binop-geq.out / test-binop-geq.gb
--------------------------------------------------------------------------------

working

int main() {
  int i;
  int j;
  i = 2;
  j = 0;
  if(i >= j) {
    print("working");
  } else {
    print("NOT working");
  }
  return 0;
}
```

```
--------------------------------------------------------------------------------
test-binop-greater.out / test-binop-greater.gb
--------------------------------------------------------------------------------

working

int main() {
  int i;
  int j;
  i = 2;
  j = 0;
  if(i > j) {
    print("working\n");
  } else {
    print("NOT working\n");
  }
  return 0;
}

--------------------------------------------------------------------------------
test-binop-leq.out / test-binop-leq.gb
--------------------------------------------------------------------------------

working

int main() {
  int i;
  int j;
  i = 0;
  j = 2;
  if(i <= j) {
    print("working\n");
  } else {
    print("NOT working\n");
  }
  return 0;
}

--------------------------------------------------------------------------------
test-binop-less.out / test-binop-less.gb
--------------------------------------------------------------------------------

working

int main() {
  int i;
  int j;
  i = 0;
  j = 2;
  if(i < j) {
    print("working\n");
  } else {
    print("NOT working\n");
  }
  return 0;
}

--------------------------------------------------------------------------------
test-binop-mod.out / test-binop-mod.gb
--------------------------------------------------------------------------------

4

int main() {
  int a;
  int b;
  int c;
  a = 9;
  b = 5;
  c = a % b;
  print("%d\n", c);
  return 0;
}

--------------------------------------------------------------------------------
test-binop-mult.out / test-binop-mult.gb
--------------------------------------------------------------------------------

45

int main() {
  int a;
  int b;
  int c;
  a = 9;
  b = 5;
  c = a * b;
  print("%d\n", c);
  return 0;
}

--------------------------------------------------------------------------------
test-binop-nequal.out / test-binop-nequal.gb
--------------------------------------------------------------------------------
```

```
working

int main() {
  int i;
  int j;
  i = 1;
  j = 0;
  if(i != j) {
    print("working\n");
  } else {
    print("NOT working\n");
  }
  return 0;
}

--------------------------------------------------------------------------------
test-binop-or.out / test-binop-or.gb
--------------------------------------------------------------------------------

working

int main() {
  bool t;
  bool f;
  bool r;
  t = true;
  f = false;
  r = t || f;
  if(r) {
    print("working\n");
  } else {
    print("NOT working\n");
  }
  return 0;
}

--------------------------------------------------------------------------------
test-binop-sub.out / test-binop-sub.gb
--------------------------------------------------------------------------------

4

int main() {
  int a;
  int b;
  int c;
  a = 9;
  b = 5;
  c = a - b;
  print("%d\n", c);
  return 0;
}

--------------------------------------------------------------------------------
test-nested-tuple-member.out / test-nested-tuple-member.gb
--------------------------------------------------------------------------------

2

tuple:A{int a; int b;}
tuple:B{tuple:A a; int b;}

int main(){
	tuple:A tpl;
	tuple:B tmp;

	tpl = new tuple:A(1,2);
	tmp = new tuple:B(tpl, 3);

	print("%d\n",tmp.a.b);
}

--------------------------------------------------------------------------------
test-tuple-assign-member.out / test-tuple-assign-member.gb
--------------------------------------------------------------------------------

45

tuple:A{int a; int b;}

int main(){
    tuple:A b;
    int result;

    b = new tuple:A(10,20);
    b.a = b.a + b.b;
    result = 15 + b.a;
    print("%d\n",result);
}

--------------------------------------------------------------------------------
test-tuple-member.out / test-tuple-member.gb
--------------------------------------------------------------------------------

1
```

```
tuple:A {
  int i;
}

int main(){
    tuple:A a;
    a = new tuple:A(1);
    print("%d\n", a.i);
}
```