Hippograph

The language for High Performance Parsing of Graphs

Benjamin Lewinter
Manager
bsl2121

Irina Mateescu
Language Guru
im2441

Harry Smith
System Architect
hs3061

Yasunari Watanabe
Test Expert
yw3239

December 19, 2018

Contents

1 Introduction

Hippograph aims to serve as a playground for graph theorists and experimenters. It has its roots in *giraph*, a project from PLT Fall 2017, bringing improvements in graph creation and graph query capabilities. The language supports graphs with node values and edge weights of multiple types, specified during declaration and instantiation.

Hippograph is ideally suited for a wide range of graph problems that demand flexible representation of data and use of built-in methods and functions. Its graph syntax is visual and intuitive, which can help the hippographer create and experiment with new problems quickly and easily.

2 Tutorial

2.1 Basic Syntax

Hippograph uses C-like syntax. Programs enter the main function on execution. Semi-colons are needed to separate statements, as well as arguments in function/method declarations. Main can neglect to return as below:

```
int gcd(int a; int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return a;
}

int main() {
    print_int(gcd(2; 12));
}
```

2.2 Graphs, Nodes and Edges

Hippograph represents all graphs as directed graphs. The user can choose to make use of weights or not. The following line of code declares a directed graph, with string node names, int node data and no edge weights:

```
graph<string:int> g = ["hello":1 -()> "world":2 <()> "!":3];
```

An example with int edge weights declared:

```
graph<string:int, int> h = ["A":1 <(2)> "B":1 -(3)> "C":0 -(5)> "A"];
```

Nodes can also be declared independently, which can be handy for adding nodes to graphs (among other use scenarios):

```
node<string:int> x = "D":4;
h.set_node(n); (*adds node n to graph h*)
```

2.3 Graph Iterators

Hippograph offers two iterators for cycling through graph elements: `for_node` iterates through a graph node list, while `for_edge` iterates through its edge list. `for_node` stores the current node in the first argument. `for_edge` provides the source node, destination node, and weight of the current edge.

```
for_node( m : graph ) { }
for_edge( src_node, dst_node, w : graph) { }
```

2.4 Hello, world!

With just these components, plus a built-in function for `get_data()` which retrieves data from a given node, one can build a graph-based implementation of Hello, world!

```
int main() {
  graph<int:string> hello = [0:"H" -()> 1:"E" -()> 2:"L" -()> 3:"L" -()> 4:"O" -()> 5:" " -()> 6:"World!"];
  for_node (i : hello) {
    print(i.get_data());
  }
  return 0;
}
```

Calling `hello.print()`, instead of iterating through nodes, and just printing the data, would print the entire graph, including nodes and edges.

3 Language Manual

3.1 Lexical Conventions

Tokens can be identifiers, keywords, literals and operators. Any amount of whitespace can be used to separate the tokens. Comments follow the format (* .... *), with nested comments not permitted.

- Identifiers are sequences of alphanumeric characters and underscores, starting with a lower- or upper-case alphabetic character.

- Reserved keywords cannot be used as identifiers in Hippograph. The reserved keywords are `bool, int, string, fun, true, false, graph, node, if, else, while, for, for_node, for_edge, return, void, NULL`.

3.2 Data Types

Hippograph is a statically scoped language. Scoping is determined through the use of curly braces {}, like in C and Java. In particular, these define the scope of function and iteration bodies in addition to graph instantiations. The following data types are supported:

3.2.1 Primitives

- `bool` - One byte `TRUE/FALSE` value.

- `int` - Signed 32-bit integer.

- `fun` - A function, with input types, return type, and name.

- `NULL` - A null value.

3.2.2 Reference Types

- `string` - A sequence of `char` instances enclosed in double quotes (e.g. `string s = "PLT is cool!"`).

- `node` - Consists of a name and data pair. The name must be unique within a graph, and be of type int, bool or string. The data does not have to be unique, and can be NULL. Non-existence of data is represented by either omitting it or by giving it a `NULL` value of `Void` type. Under the hood, however, the data will be given default values, and the node's `has_value` flag will be set to `false`.

```
graph<int, int> g = [1 -(3)- 4; 6; 8];
graph<int:void, int> g = [1:NULL -(3)- 4:NULL; 6:NULL; 8:NULL];
```

  The Hippograph node methods are:

  - `node.get_name()` - Returns name of `node`.
  - `node.get_data()` - Returns data stored in `node`. Does not apply to nodes with `Void` data type.
  - `node.set_data(data)` - Sets data in specified node, overwriting any previous data.
  - `node.print()` - Pretty prints a node, with name and data.

- graph - Consists of a set of nodes, as well as a set of edges connecting those nodes. Every item in the node set must have the same name and data types. Every item in the edge set must have the same weight type. The type signatures of the nodes and edges may differ.

  The Hippograph graph methods are:

  - graph.set_node(node) - Adds a copy of node to graph. If the graph already contains a node with that name, its previous data gets overwritten. Returns 0 if successful, -1 otherwise.
  - graph.set_edge(from_name, to_name, edge_weight) - Adds an edge in graph from node with from_name to node with to_name, where these nodes are already in the graph. If an edge between these nodes already exists, the weight is updated. Returns 0 if successful, -1 otherwise.
  - graph.remove_node(node_name) - Removes node with node_name from graph. Edges pointing to/from node are also removed. Returns 0 if graph has been modified as a result and -1 otherwise.
  - graph.remove_edge(from_name, to_name) - Removes an edge from node with from_name to node with to_name from graph, if it exists. Returns 0 if graph has been modified as a result and -1 otherwise.
  - graph.has_node(node_name) - Returns 0 if node with node_name is in graph, or -1 otherwise.
  - graph.get_node(node_name) - Returns the node with node_name if the node is in the graph and a node with null name and data otherwise.
  - graph.get_weight(src_name, dst_name) - Returns the weight of the edge with source src_name and destination dst_name and null for the weight type otherwise.
  - graph.are_neighbors(from_name, to_name) - Returns true if there exists an edge from node with from_name to node with to_name, or false otherwise.
  - graph.neighbors(node_name) - Returns a graph containing all immediate neighbors of node with node_name, and their adjoining edges. The initial node is not included in the returned graph. The returned graph is a copy of the original, as are its nodes and edges.
  - graph.neighbors(node_name, level, include_current) - Returns a graph containing all neighbors of node with node_name up to a depth of level, and their adjoining edges. If include_current is true, the graph includes the initial node. The returned graph is a copy of the original, as are its nodes and edges.
  - graph.find(data) - Returns a graph containing all nodes that have data.
  - graph.print() - Pretty prints graph nodes, edges, and the data they contain.
  - graph.is_empty() - Returns true if the graph has no nodes, and false otherwise.

3.3 Variable Declaration and Assignment

A variable is declared by specifying its type and name. Variables are assigned a value using the = operator. The left hand side argument must be an identifier, and the right hand side can be a value or identifier of the same type as the left. Type conversions are not supported. The global scope only allows for variable declaration, and not value assignment.

3.3.1 Primitive Types

Type declaration and value assignment for primitive types follow C-style syntax.

```
(* type declaration only *)
bool val1;
int val2;
string val4;

(* combined type declaration and value assignment *)
bool val1 = true;
int val2 = 42;
string val4 = "hi";
```

Each primitive type has a nullary value. A variable of a primitive type can be assigned its nullary value with the expression NULL.

type	nullary value
bool	false
int	0
string	""

This is only relevant the context of nodes and graphs, where a nullary value specified with NULL indicates the absence of a value in an optional field. (See sections 3.4.1 and 3.4.3.)

3.4 Graph Construction

3.4.1 Nodes

The node type declaration requires parameters that indicate the types of the node's name and data. These are specified in angle brackets.
Nodes are declared with the type signature:

```
node<name type : data type>
```

Examples of declarations are given below.

```
node<int:int> n1;     (* node with name of int type and data of int type *)
node<int:string> n2;  (* node with name of int type and data of string type *)
```

Variable declarations can omit the type of the optional node data. If not present, this defaults to type bool. In the examples below, nodes n3 and n4 have equivalent types.

```
(* both nodes with name of int type and data of bool type *)
node<int:bool> n3;
node<int> n4;
```

A stand-alone node instance independent of a graph is created by writing the node name and data, separated by the : operator. This operator has higher precedence than other graph operators.
A shorthand notation allows the omission of the : operator and node data. If not present, the node data defaults to the nullary value of its type. (See section 3.3.1.) For example, the following operations behave identically.

```
node<int:string> n5 = 123:NULL; (* explicit null string *)
node<int:string> n6 = 123;      (* inferred null string *)
node<int> n7 = 123:NULL;        (* explicit null bool *)
node<int> n8 = 123;             (* inferred null bool *)
```

3.4.2 Edges

Edges have the type signature:

```
edge<weight type>
```

However, edges are graph-dependent types that may not be declared explicitly.

3.4.3 Graphs

The `graph` type declaration requires parameters that indicate the types of the node's name and data. These are specified in angle brackets.
Graphs are declared with the type signature:

```
graph<node name type :  node data type , edge data type>
```

Variable declarations can omit the type of the optional node data and edge weight. If not present, each defaults to type `bool`. In the examples below, graphs `g1`, `g2`, and `g3` have equivalent types.

```
(* all graphs with node name of int type, node data/edge weight of bool type *)
graph<int:bool, bool> g1;
graph<int, bool> g2;
graph<int> g3;
```

A graph expression instance is created by enclosing a sequence of expressions that describe the graph's node and edge layout in a pair of square brackets. The same rules apply for the shorthand notation of creating data-less nodes. (See section 3.4.1.)
In a graph expression, an edge can have the following formats. If an edge has no weight, it may be omitted. If not present, the edge weight defaults to the nullary value of its type. (See section 3.3.1.)

- `-(weight)>`: A right-singly-directed edge.
- `<(weight)-`: A left-singly-directed edge.
- `-(weight)-` or `<(weight)>`: An undirected edge or doubly-directed pair of edges.

An undirected edge between nodes `A` and `B` is represented internally as two edges from `A` to `B` and from `B` to `A`, with identical weights. This allows Hippograph to represent both directed and undirected graphs with the same generic `graph` type. There can be at most one edge in each direction between nodes. In graphs with duplicate edge declarations, only the left-most declaration is considered. In the examples below, graphs `g4` and `g7` are equivalent.

```
(* all graphs with a directed edge from 1 to 2 and another from 2 to 1 *)
graph<int, string> g4 = [1-("a")-2];
graph<int, string> g5 = [1<("a")>2];
graph<int, string> g6 = [1<("a")-2; 1-("a")>2];
graph<int, string> g7 = [1-("a")-2; 1<("a")-2];
```

Graph declarations are read from left to right. Once a node that has a name is created, the same node is referenced the next time the name occurs in the graph declaration.
The following example declares a graph, whose nodes have a name type of `string` and data type of `int`, and whose edges have a weight type of `int`. Note that the second usage of the name `"A"` refers to the first usage, where it was associated with the integer `2`.

```
graph<string:int, int> g = ["A":2 -(3)> "B":4 <(2)> "C":8 <(2)- "A"]
```

3.5 Function Expressions

In Hippograph, functions are can be defined as expressions and have the results stored in a variable, which provides the name for an otherwise anonymous function. These are defined using the following syntax:

```
fun<type1:type2:...  , ret_type> var = ret_type (type1 arg1, type2 arg2, ...)  (expr);
```

where `var` is a new variable of type `fun<...>` that refers to the newly defined function. Such a function can be called wherever `var` is in scope as in the following example, with a declaration proceeding the function call itself:

```
fun<int:int, bool> gt = bool (int x, int y) (x > y)

bool b = gt(4, 3)
```

These function expressions are only permitted to take in a single expression for their bodies. Variables defined in the scope of the function declaration but outside of the body of the function itself are accessible in calls to these functions.
No functions are first class in Hippograph, and thus functions cannot be passed as arguments to other functions.

3.6 Arithmetic Operators

The precedence follows the standard mathematical "order of operations", with, in decreasing order of precedence:

1. Parentheses, non-associative
2. Multiplicative operators `*`, `/`, left-associative
3. Additive operators `+`, `-`, left-associative

3.7 Boolean Operators

In descending order of precedence:

1. `==`, `!=`, `>=`, `<=`, `>`, `<`, non-associative
2. `not`, right-associative
3. `and` and `or`, left-associative

3.8 Comments

Comments will be formatted as `(* ... *)` and will not allow nested comments.

3.9 Control Flow

3.9.1 Conditionals

Conditional expressions follow C-style syntax.

```
if (condition) {statements}
if (condition) {statements} else {statements}
if (condition1) {statements} else if (condition2) {statements} else {statements}
```

A `if` block may optionally be followed by any number of `else if` blocks. It may also be followed by a `else` block.

3.9.2 Loops

Loops follow C-style syntax.

```
while (condition) {statement}
for (initialization; condition; update) {statement}
```

Hippograph also supports iteration over nodes and edges.

- `for_node(node : graph) {statements}` - Iterates through the nodes in a graph with an arbitrary ordering. For each iteration, a local copy of the current node is created.

- `for_edge(src, dst, weight : graph) {statements}` - Iterates through the edges in a graph with an arbitrary ordering, where `src` and `dst` are nodes and `weight` is the weight value of the edge. For each iteration, a local copy of the current edge `src`, `dst`, and `weight` are created.

3.10 Program Structure

Programs consist of sequences of functions including a `main` function. `main` will be the entry point for the program's executable. Functions have the following syntax, where names are mandatory.

```
return_type name(type arg1, type arg2, ...) {body}
```

4 Project Plan - Ben

4.1 Workflow

We met on average twice per week, typically with one midweek extended group coding and planning session, and one 30 minute check-in meeting after class on Mondays. We found that this was very effective because it balanced efficient, independent work with collaboration, while keeping us each accountable for our tasks. Our group coding sessions in particular were very helpful at helping us solve some of the most challenging parts of the project. We would find space that offered an extra screen, so that we could discuss lines of code together as a group. We also had some especially helpful meetings with our TA, Jennifer. We were fortunate that she had worked on giraph, which was the inspiration for our project, so she was able to share some important lessons learned.

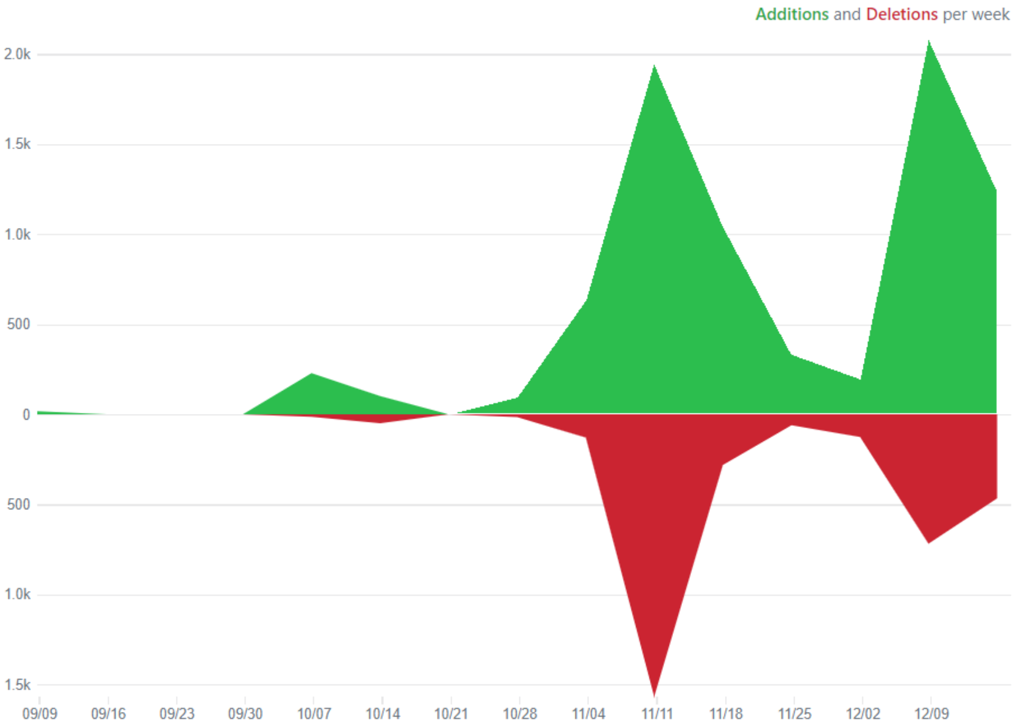
Outside of in-person meetings, we were constantly communicating online. Our team fostered a highly collaborative attitude, and we eagerly helped each other with our assigned tasks. As such, we were frequently posting questions and discussing solutions in our group chat, which helped us stay productive, even when not meeting in-person.

Ultimately, our team was highly effective at planning and hitting our deadlines. To be sure, as deadlines approached the stress level increased. But we (almost) never needed late-night or overnight coding sessions to get things done in time. We set frequent deadlines for ourselves, and always kept our eyes on the next goal. This resulted in a moderate and steady pace of work from start to finish, which made the project much more relaxed and enjoyable to work on.

4.2 Tools and Software

Languages: OCaml and C
Version Control: Git and Github
Report Creation: Overleaf
Testing: Bash scripts

4.3 Code frequency graph



5 Language Evolution - Irina

Hippograph is a programming language whose main purpose is to make graph creation and graph manipulation intuitive to the user. It draws most of its syntax from C, with edge declaration inspired from Cypher and functions from functional programming. The language went through several iterations, initially meaning to make extensive use of the Cypher querying structure, later planning to focus more on a unified graph type, and finally aiming to implement anonymous functions. We ran into several design decisions such as settling how NULL is resolved in our language or having to choose between passing nodes by reference or by value.

6 Translator Architecture - Harry

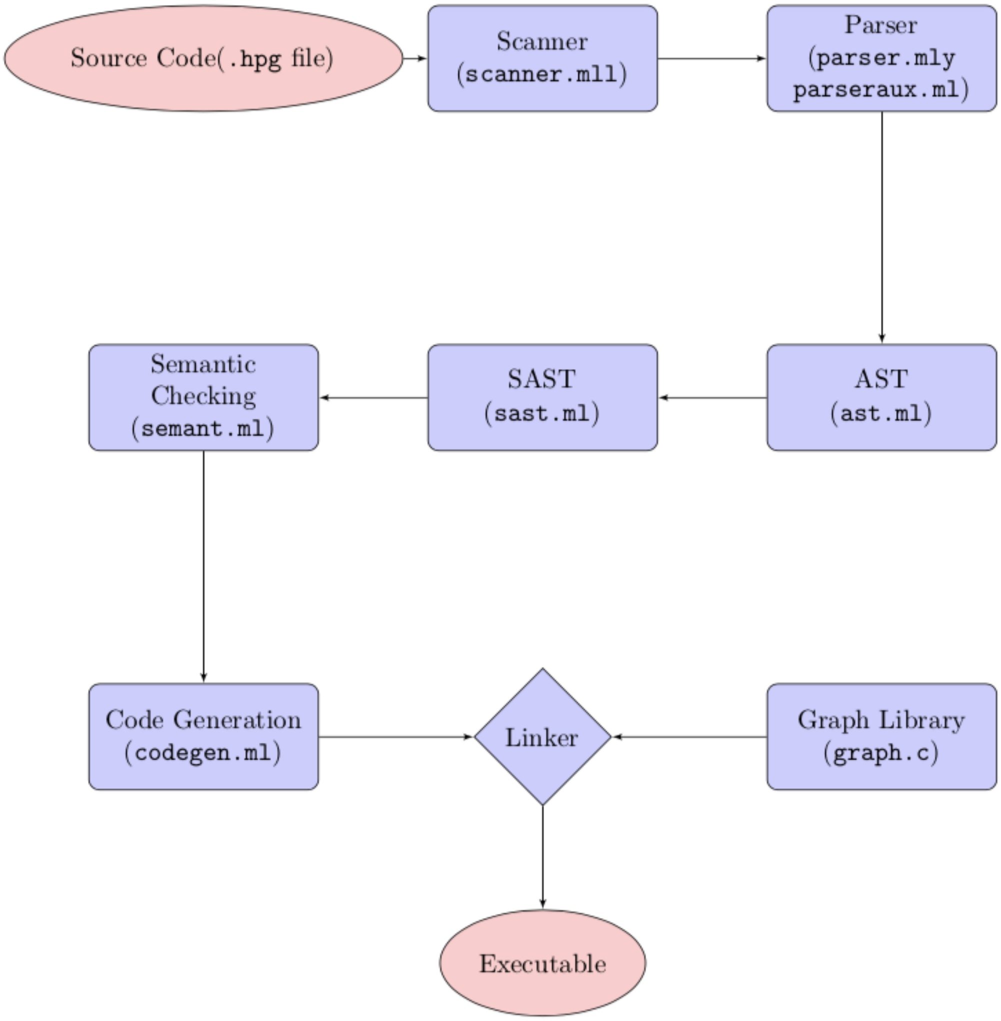


Figure 1: Compiler Architecture

The Complete Pipeline

1. The `.hpg` file is read from the file system as a string into the `Scanner`, which tokenizes the text into tokens for the parser.

2. The atomic tokens are fed to the `Parser`, which interprets the position and ordering of the tokens into an `Abstract Syntax Tree`, or `AST`. If a well-formed `AST` cannot be created from the tokens as parsed, then the compiler rejects the source code on the basis of improper syntax.
   A signature feature of our language, the Graph Expression, is implemented at this stage. Our `Parser` interprets sequences of node specifications — names with optional data values — along with optionally weighted edges to allow for quick instantiations of graphs which express relational structure and contain data.

The Parser performs no semantic checking. In particular, expressions and operations of improper types may persist at this stage. Graphs may have mixed node name types, node data types, and edge weight types.

3. The AST is analyzed by our semantic checker to produce a semantically valid SAST. In the case that the program has semantic errors which cannot be resolved through coercion or inference, the semantic checker will reject the input AST and display a message explaining the semantic error.
   Most type mismatches (e.g. attempting to compare a string with an integer) will result in rejected programs. The notable exception is the case of parsing nodes and edges for which the user has chosen not to provide data/weights. These omissions are interpreted as null values in the AST, but null is not available to the users of the language. To that end, missing data is interpreted by the semantic checker as the nullary value of the data type or weight type of the graph.

4. Code Generation is the next step in the pipeline, wherein semantically checked statement blocks are converted into LLVM. As the statement blocks are consumed, changes to local state (of both variable and function definitions) are passed along throughout. This allows for declaration of new variables within functions and for implementation of first-class, locally-scoped functions. One-line functions can be defined in the bodies of the classic function declaration, and the LLVM for these one-liners is generated to the output file as well. It is the passing of state throughout the semantic checking and code generation processes which allow these functions, which remain in the LLVM output even when they fall out of scope, to be used only in contexts for which they are in scope.

5. Our Graph Library, which was written by personally by the group to meet our specific needs for the project, is linked in along with the LLVM output to produce the final executable. This library defines all of the graph operations which are permitted except for filter, which is implemented in Hippograph itself. Our library was written with our emphasis on querying in mind. In particular, our internal traversals of the graph structures were performed on the lists of edges and nodes, rather than on the relational structure of the graph itself.
   In order to support the limited polymorphism of the graph type, each built-in function had a separate internal implementation. This resulted in significant repetition of very similar code in some cases up to four times in a row. We attempted to ameliorate the issue of repetition through use of a Union primitive type.

7 Test Plan - Yasu

Regression tests are used in two ways. Positive test cases check that valid Hippograph programs produce the expected output. Negative test cases verify that syntactically valid but semantically invalid programs fail for the expected reason.

7.1 Test Structure and Automation

All test cases are stored in the `test/` directory. Each positive test case consists of a Hippograph file that follows the form `test-<FILENAME>.hpg` and the corresponding output file that follows the form `test-<FILENAME>.hpg`. Each negative test case consists of a Hippograph file that follows the form `fail-<FILENAME>.hpg` and the corresponding error file that follows the form `fail-<FILENAME>.hpg`.

Test execution is automated using a script adapted from the provided Micro-C compiler. The `testall.sh` script scans the `test/` directory for all `test-*.hpg` and `fail-*.hpg` files. For each test case, it links the C library dependency to produce a Hippograph executable, which is then immediately run. All program outputs and errors are piped to stdout, and is compared with the content of the relevant `.out` or `.err` file, respectively. The command `make test` compiles the C library, runs the test script, and cleans residual files.

7.2 Examples

7.2.1 Positive Case: `test-graph-neighbors5.hpg`

Source

```
int main() {
  graph<int> g = [1 -()- 2 -()- 3 -()- 4 -()- 2];
  graph<int> g_sub1 = g.neighbors(1; 4; false);
  for_node (n : g_sub1) {
    n.print();
    print("");
  }
  print("");
  graph<int> g_sub2 = g.neighbors(1; 4; true);
  for_node (n : g_sub2) {
    n.print();
    print("");
  }
}
```

LLVM IR

```
; ModuleID = 'Hippograph'
source_filename = "Hippograph"

;; C library function declarations are omitted for brevity

define i32 @main() {
entry:
  %g = alloca i8*
  %create_graph = call i8* (...) @create_graph()
  %create_node = call i8* (...) @create_node()
  call void (i8*, i32, ...) @set_node_label_int(i8* %create_node, i32 1)
  call void (i8*, i1, i1, ...) @set_node_data_bool(i8* %create_node, i1 false, i1
      false)
  %add_node = call i8* (i8*, i8*, ...) @add_node(i8* %create_graph, i8* %create_node
      )
  %create_node1 = call i8* (...) @create_node()
  call void (i8*, i32, ...) @set_node_label_int(i8* %create_node1, i32 2)
  call void (i8*, i1, i1, ...) @set_node_data_bool(i8* %create_node1, i1 false, i1 false)
  %add_node2 = call i8* (i8*, i8*, ...) @add_node(i8* %create_graph, i8* %create_node1)
  %create_node3 = call i8* (...) @create_node()
  call void (i8*, i32, ...) @set_node_label_int(i8* %create_node3, i32 3)
  call void (i8*, i1, i1, ...) @set_node_data_bool(i8* %create_node3, i1 false, i1 false)
  %add_node4 = call i8* (i8*, i8*, ...) @add_node(i8* %create_graph, i8* %create_node3)
  %create_node5 = call i8* (...) @create_node()
  call void (i8*, i32, ...) @set_node_label_int(i8* %create_node5, i32 4)
  call void (i8*, i1, i1, ...) @set_node_data_bool(i8* %create_node5, i1 false, i1 false)
  %add_node6 = call i8* (i8*, i8*, ...) @add_node(i8* %create_graph, i8* %create_node5)
  %edge = call i8* (...) @create_edge()
  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge, i1 false, i1 false)
  %add_edge = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph, i8* %edge, i32 1, i32 2)
  %edge7 = call i8* (...) @create_edge()
  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge7, i1 false, i1 false)
  %add_edge8 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph, i8* %edge7, i32 2, i32 1)
  %edge9 = call i8* (...) @create_edge()
  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge9, i1 false, i1 false)
  %add_edge10 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph, i8* %edge9, i32 2, i32 3)
  %edge11 = call i8* (...) @create_edge()
  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge11, i1 false, i1 false)
  %add_edge12 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph, i8* %edge11, i32 3, i32 2)
  %edge13 = call i8* (...) @create_edge()
  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge13, i1 false, i1 false)
  %add_edge14 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph, i8* %edge13, i32 3, i32 4)
  %edge15 = call i8* (...) @create_edge()
  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge15, i1 false, i1 false)
  %add_edge16 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph, i8* %edge15, i32 4, i32 3)
  %edge17 = call i8* (...) @create_edge()
  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge17, i1 false, i1 false)
  %add_edge18 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph, i8* %edge17, i32 4, i32 2)
  %edge19 = call i8* (...) @create_edge()
  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge19, i1 false, i1 false)
  %add_edge20 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph, i8* %edge19, i32 2, i32 4)
  store i8* %create_graph, i8** %g
  %g_sub1 = alloca i8*
  %g21 = load i8*, i8** %g
  %get_node_by_label_int = call i8* (i8*, i32, ...) @get_node_by_label_int(i8* %g21, i32 1)
  %neighbors = call i8* (i8*, i32, i1, ...) @neighbors(i8* %get_node_by_label_int, i32 4, i1 false)
  store i8* %neighbors, i8** %g_sub1
  %g_sub122 = load i8*, i8** %g_sub1
  %n = alloca i8*
  %hd_node = call i8* (i8*, ...) @graph_to_node_iterable(i8* %g_sub122)
  store i8* %hd_node, i8** %n
  br label %while

while:                                            ; preds = %while_body, %entry
  %node_tmp = load i8*, i8** %n
  %bool_val = icmp ne i8* %node_tmp, null
  br i1 %bool_val, label %while_body, label %merge

while_body:                                       ; preds = %while
  %n23 = load i8*, i8** %n
  call void (i8*, ...) @print_node(i8* %n23)
  call void (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8], [1 x i8]* @str, i32 0, i32 0))
  %curr_node = load i8*, i8** %n
  %next_node = call i8* (i8*, ...) @get_graph_next_node(i8* %curr_node)
  store i8* %next_node, i8** %n
  br label %while

merge:                                            ; preds = %while
  call void (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8], [1 x i8]* @str.2, i32 0, i32 0))
  %g_sub2 = alloca i8*
  %g24 = load i8*, i8** %g
  %get_node_by_label_int25 = call i8* (i8*, i32, ...) @get_node_by_label_int(i8* %g24, i32 1)
  %neighbors26 = call i8* (i8*, i32, i1, ...) @neighbors(i8* %get_node_by_label_int25, i32 4, i1 true)
  store i8* %neighbors26, i8** %g_sub2
  %g_sub227 = load i8*, i8** %g_sub2
  %n28 = alloca i8*
  %hd_node29 = call i8* (i8*, ...) @graph_to_node_iterable(i8* %g_sub227)
  store i8* %hd_node29, i8** %n28
  br label %while30

while30:                                          ; preds = %while_body31, %merge
  %node_tmp35 = load i8*, i8** %n28
  %bool_val36 = icmp ne i8* %node_tmp35, null
  br i1 %bool_val36, label %while_body31, label %merge37

while_body31:                                     ; preds = %while30
  %n32 = load i8*, i8** %n28
  call void (i8*, ...) @print_node(i8* %n32)
  call void (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8], [1 x i8]* @str.3, i32 0, i32 0))
  %curr_node33 = load i8*, i8** %n28
  %next_node34 = call i8* (i8*, ...) @get_graph_next_node(i8* %curr_node33)
  store i8* %next_node34, i8** %n28
  br label %while30

merge37:                                          ; preds = %while30
  ret i32 0
}
```

Output

```
2:null
3:null
4:null

1:null
```

```
2:null
3:null
4:null
```

7.2.2 Negative Case: `fail-for-node1.hpg`

Source

```
int main() {
  int g = 0;
  for_node(n : g) {
    print_int(n);
  }
}
```

Error

Here, `for_node` expects a graph to iterate on, but is instead provided an integer `g`.

```
Fatal error: exception Failure("illegal argument found node<int, int> expected int in n")
```

8 Conclusions

Ben
This project taught me a lot, both about programming languages and compilers, and about technology project management and collaboration. The programming required was some of the most in-depth and challenging coding I have ever worked on, and it boosted my skills tremendously. But for me, the lessons learned about working in a team to build a complex system were more important. One key take away was that treating teammates with respect, and fostering a positive spirit in the group are essential for keeping everyone motivated and accountable. One can use as many task management tools and programs as they like, but ultimately the culture in the group will play an essential role in determining it's productivity. I feel I was very lucky to have the teammates I had, as everyone stayed motivated and contributed greatly throughout the project.

Irina
Working on this project was both scary and empowering. In the first few weeks we met and worked as a team. But as schoolwork picked up, we decided to work independently as well as during team meetings. And I panicked. My teammates are absolutely brilliant. They code with such confidence and they came to the class with a stronger understanding of language structures than I did. But I tried to learn from them, initially in silence, and bit by bit I started contributing more, until I got to write more code and feel more confident. By the end I very much enjoyed my midnight coding sessions and I got to appreciate the importance of constructing the pipeline end-to-end. I realized how easy it is to create a regression testing suite if built incrementally and I finally get why "adding a new layer of indirection" is the answer to all my problems.

Harry
Working on Hippograph helped me learn about the dangers of optimism when engaging with novel programming projects. During our initial project meetings, I was often overwhelmed with excitement at the possibilities of our language—lazy graph generation using functional paradigms! implementations of sets and lists and maps using our graphs!—but time constraints throughout the semester kept my aspirations more limited in scope. Then, even after settling on features which I found more interesting than the basic functionality, revisiting OCaml after four years and deploying it in a new context made every task take orders of magnitude longer than I anticipated. Only by the very end did I begin to appreciate that even the most banal of tasks—implementing just one more built-in—could take half an hour. Now, at the conclusion of the project, I am more pessimistic than ever while also being more encouraged than ever. My abilities in reading documentation, understanding the work of others, and interpreting functional languages are significantly stronger than before. What's more, I have a spiffy new graph language as a reward for my efforts. It's certainly not the best language, and probably not even a *good* one, but it represents a finished product and inspires me to try again soon.

Yasu
Throughout this project, I learned about the challenge of reconciling expressiveness and feasibility of language features. On the one hand, the more potential utility and expressiveness we gave a particular feature, the more "holes" it would create in the system. As the tester, my primary role was to account for such edge cases in designing the regression tests. On numerous occasions, this would reveal some inconsistency in the language which we would then have to discuss. On the other hand, limiting a feature's capacity serves to avoid such headaches, but invariably reduces the number of programs that can be written in the language. I think one way to strike the right balance is to be very clear at the beginning about what programs you are seeking to write, and start by designing a small, selective set of expressive components to make it possible. It's good to go through this painful process early and

scale up.

9 Full Code

9.1 Scanner

9.1.1 scanner.mll

```
(* Authors:
  Benjamin Lewinter bsl2121
  Irina Mateescu    im2441
  Harry Smith       hs3061
  Yasunari Watanabe yw3239
*)

{ open Parser }

rule token = parse
  [’ ’ ’\t’ ’\r’ ’\n’] { token lexbuf }
| ’+’  { PLUS }
| ’-’  { MINUS }
| ’*’  { TIMES }
| ’/’  { DIVIDE }
| ’;’  { SEQUENCE }
| ’=’  { ASSIGN }
| ’{’  { LBRACE }
| ’}’  { RBRACE }
| ’.’  { DOT }
| ’,’  { COMMA }
| ’(’  { LPAREN }
| ’)’  { RPAREN }
| ’<’  { LANGLE }
| ’>’  { RANGLE }
| ’[’  { LBRAK }
| ’]’  { RBRAK }
| ’\’’ { SQUOTE }
| ’\"’ { DQUOTE }
| ’:’  { COLON }
| "=="     { EQ }
| "!="     { NEQ }
| "<="     { LEQ }
| ">="     { GEQ }
| "and"    { AND }
| "or"     { OR }
| "not"    { NOT }
| "int"    { INTTYPE }
| "bool"   { BOOLTYPE }
| "string" { STRINGTYPE }
| "fun"    { FUNTYPE }
| "void"   { VOIDTYPE }
| "graph"  { GRAPHTYPE }
| "node"   { NODETYPE }
| "-("     { LUEDGE }
| ")-"     { RUEDGE }
| "<("     { LDEDGE }
| ")>"     { RDEDGE }
| "if"     { IF }
| "else"   { ELSE }
| "while"  { WHILE }
| "for"    { FOR }
| "for_node"    { FORNODE }
| "for_edge"    { FOREDGE }
| "in"     { IN }
| "NULL"   { NULL }
| "return" { RETURN }
| [’0’-’9’]+ as int_lit                 { INTLIT(int_of_string int_lit) }
| ’\"’ ([^’\"’]* as string_lit) ’\"’    { STRINGLIT(string_lit) }
| ("true" | "false") as bool_lit        { BOOLLIT(bool_of_string bool_lit) }
| [’a’-’z’ ’A’-’Z’][’0’-’9’ ’a’-’z’ ’A’-’Z’ ’_’]* as id { VARIABLE(id) }
| eof { EOF }
| "(*" { comment lexbuf }
and comment = parse
   "*)" { token lexbuf }
  | _ { comment lexbuf }
```

9.2 Parser

9.2.1 parser.mly

```
/* Authors:
  Benjamin Lewinter bsl2121
  Irina Mateescu    im2441
  Harry Smith       hs3061
  Yasunari Watanabe yw3239
*/

%{
  open Ast
  open Parseraux
%}

%token PLUS MINUS TIMES DIVIDE SEQUENCE ASSIGN EOF
%token LBRACE RBRACE DOT COMMA LPAREN RPAREN LANGLE RANGLE LBRAK RBRAK SQUOTE DQUOTE COLON
%token EQ NEQ LEQ GEQ AND OR NOT
%token INTTYPE BOOLTYPE STRINGTYPE FUNTYPE GRAPHTYPE NODETYPE VOIDTYPE
%token LUEDGE RUEDGE LDEDGE RDEDGE
%token IF ELSE NOELSE WHILE FOR FORNODE FOREDGE IN NULL RETURN
%token <int> INTLIT
%token <string> STRINGLIT
%token <bool> BOOLLIT
%token <string> VARIABLE

%left SEQUENCE
%left DOT
%right ASSIGN
%left AND OR
%right COLON
%nonassoc EQ NEQ
%nonassoc LEQ GEQ LANGLE RANGLE
%left PLUS MINUS
%left TIMES DIVIDE
%right NEG NOT
%nonassoc LPAREN RPAREN
%nonassoc NOELSE
%nonassoc ELSE

%start program
%type <Ast.program> program

%%

program: decls EOF { $1 }
```

```

decls:
  { [], [] }
| decls vdecl { ($2 :: fst $1), snd $1 }
| decls fdecl { fst $1, ($2 :: snd $1) }

vdecl:
  typ VARIABLE SEQUENCE { ($1, $2) }

fdecl:
  typ VARIABLE LPAREN args_opt RPAREN LBRACE stmt_list RBRACE
    { { typ = $1; fname = $2; args = $4; body = List.rev $7 } }

args_opt:
  { [] }
| args_list { List.rev $1 }

args_list:
  typ VARIABLE                    { [($1, $2)] }
| args_list SEQUENCE typ VARIABLE { ($3, $4) :: $1 }



actuals_opt:
  { [] }
| actuals_list { List.rev $1 }

actuals_list:
  expr                       { [$1] }
| actuals_list SEQUENCE expr { $3 :: $1 }

typ:
  VOIDTYPE   { Void }
| INTTYPE    { Int }
| BOOLTYPE   { Bool }
| STRINGTYPE { String }
| FUNTYPE LANGLE typ COLON typ_list_opt RANGLE     { Fun($3, $5) }
| GRAPHTYPE LANGLE typ COLON typ COMMA typ RANGLE  { Graph($3, $5, $7) }
| GRAPHTYPE LANGLE typ COMMA typ RANGLE            { Graph($3, Bool, $5) }
| GRAPHTYPE LANGLE typ COLON typ RANGLE            { Graph($3, $5, Bool) }
| GRAPHTYPE LANGLE typ RANGLE                      { Graph($3, Bool, Bool) }
| NODETYPE  LANGLE typ COLON typ RANGLE            { Node($3, $5) }
| NODETYPE  LANGLE typ RANGLE                      { Node($3, Bool) }

typ_list_opt:
  { [] }
| typ_list { List.rev $1 }

typ_list:
  typ                       { [$1] }
| typ_list COMMA typ { $3 :: $1 }

stmt_list:
  { [] }
| stmt_list stmt { $2 :: $1 }

stmt:
  expr SEQUENCE                                            { Expr $1 }
| FOR LPAREN expr SEQUENCE expr SEQUENCE expr RPAREN stmt  { For($3, $5, $7, $9) }
| FORNODE LPAREN VARIABLE COLON VARIABLE RPAREN stmt       { ForNode($3, Var($5), $7
    ) }
```

```
| FOREDGE LPAREN VARIABLE COMMA VARIABLE COMMA VARIABLE COLON VARIABLE RPAREN stmt { ForEdge($3, $5, $7, Var($9), $11) }
| WHILE LPAREN expr RPAREN stmt                            { While($3, $5) }
| IF LPAREN expr RPAREN stmt %prec NOELSE                  { If($3, $5, Block([])) }
| IF LPAREN expr RPAREN stmt ELSE stmt                     { If($3, $5, $7) }
| LBRACE stmt_list RBRACE                                  { Block(List.rev $2) }
| typ VARIABLE SEQUENCE                                    { Vdecl($1, $2, Noexpr) }
| typ VARIABLE ASSIGN expr SEQUENCE                        { Vdecl($1, $2, Asn($2, $4)) }
| RETURN SEQUENCE                                          { Return Null }
| RETURN expr SEQUENCE                                     { Return $2 }

expr:
  INTLIT                { Intlit($1) }
| STRINGLIT             { Stringlit($1) }
| BOOLLIT               { Boollit($1) }
| typ LPAREN args_opt RPAREN LPAREN expr RPAREN { Funsig($1, $3, $6) }
| NULL                  { Null }
| VARIABLE              { Var($1) }
| LPAREN expr RPAREN    { $2 }
| expr PLUS   expr      { Binop($1, Add, $3) }
| expr MINUS  expr      { Binop($1, Sub, $3) }
| expr TIMES  expr      { Binop($1, Mul, $3) }
| expr DIVIDE expr      { Binop($1, Div, $3) }
| expr AND expr         { Binop($1, And, $3) }
| expr OR expr          { Binop($1, Or, $3) }
| expr EQ expr          { Binop($1, Eq, $3) }
| expr NEQ expr         { Binop($1, Neq, $3) }
| expr LEQ expr         { Binop($1, Leq, $3) }
| expr GEQ expr         { Binop($1, Geq, $3) }
| expr LANGLE expr          { Binop($1, Lt, $3) }
| expr RANGLE expr          { Binop($1, Gt, $3) }
| MINUS expr %prec NEG  { Unop(Neg, $2) }
| NOT expr              { Unop(Not, $2) }
| VARIABLE ASSIGN expr  { Asn($1, $3) }
| VARIABLE LPAREN actuals_opt RPAREN { FCall($1, $3) }
| expr DOT VARIABLE LPAREN actuals_opt RPAREN { MCall($1, $3, $5) }
| expr COLON  expr      { NodeExpr($1, $3) }
| LBRAK graph_item_opt RBRAK                  { match $2 with (node_list, edge_list) ->
                                                  GraphExpr(node_list, edge_list) }

graph_item_opt:
  { [], [] }
| graph_item_list { match $1 with (node_list, edge_list) ->
                      List.rev node_list, List.rev edge_list }

graph_item_list:
  node_edge_list                          { $1 }
| graph_item_list SEQUENCE                { $1 }
| graph_item_list SEQUENCE node_edge_list { merge_node_edge_lists $1 $3 }

node_edge_list:
  expr                                   { [construct_node_expr $1], [] }
| node_edge_list LUEDGE expr RUEDGE expr { update_node_edge_list_with_edge $1 $3 (construct_node_expr $5)  }
| node_edge_list LDEDGE expr RDEDGE expr { update_node_edge_list_with_edge $1 $3 (construct_node_expr $5)  }
| node_edge_list LUEDGE expr RDEDGE expr { update_node_edge_list_with_redge $1 $3 (construct_node_expr $5)  }
| node_edge_list LDEDGE expr RUEDGE expr { update_node_edge_list_with_ledge $1 $3 (
    construct_node_expr $5)  }
| node_edge_list LUEDGE RUEDGE expr      { update_node_edge_list_with_edge $1 Null (construct_node_expr $4)  }
| node_edge_list LDEDGE RDEDGE expr      { update_node_edge_list_with_edge $1 Null (construct_node_expr $4)  }
| node_edge_list LUEDGE RDEDGE expr      { update_node_edge_list_with_redge $1 Null (construct_node_expr $4)  }
| node_edge_list LDEDGE RUEDGE expr      { update_node_edge_list_with_ledge $1 Null (construct_node_expr $4)  }
```

9.2.2 parseraux.ml

```
open Ast;;

let construct_node_expr expr =
  (* if already a NodeExpr, keep it that way; otherwise create a NodeExpr with Null data *)
  match expr with
  | NodeExpr(_, _) -> expr
  | _ -> NodeExpr(expr, Null)
;;

let node_list_append_opt n_expr n_list =
  (* add to list if key doesn't exist; ignore otherwise *)
  let label, _ = unwrap_node_expr n_expr in
  if List.exists (fun n -> let l, _ = unwrap_node_expr n in l = label) n_list
  then n_list
  else n_expr :: n_list
;;

let edge_list_append_opt e_expr e_list =
  (* add to list if src-dst pair doesn't exist; ignore otherwise regardless of weight *)
  let src, dst, _ = unwrap_edge_expr e_expr in
  if List.exists (fun e -> let s, d, _ = unwrap_edge_expr e in s = src && d = dst) e_list
  then e_list
  else e_expr :: e_list
;;

let update_node_edge_list_with_edge (n_list, e_list) weight n_expr =
  let prev_n_label, _ = unwrap_node_expr (List.hd n_list) in
  let n_label, _ = unwrap_node_expr n_expr in
  let n_list' = node_list_append_opt n_expr n_list in
  let e_list' = edge_list_append_opt (EdgeExpr(n_label, prev_n_label, weight))
                                     (edge_list_append_opt (EdgeExpr(prev_n_label, n_label, weight))
                                                           e_list)
  in
  (n_list', e_list')
;;

let update_node_edge_list_with_redge (n_list, e_list) weight n_expr =
  let prev_n_label, _ = unwrap_node_expr (List.hd n_list) in
  let n_label, _ = unwrap_node_expr n_expr in
  let n_list' = node_list_append_opt n_expr n_list in
  let e_list' = edge_list_append_opt (EdgeExpr(prev_n_label, n_label, weight)) e_list in
  (n_list', e_list')
;;

```

```
let update_node_edge_list_with_ledge (n_list, e_list) weight n_expr =
  let prev_n_label, _ = unwrap_node_expr (List.hd n_list) in
  let n_label, _ = unwrap_node_expr n_expr in
  let n_list' = node_list_append_opt n_expr n_list in
  let e_list' = edge_list_append_opt (EdgeExpr(n_label, prev_n_label, weight))
      e_list in
  (n_list', e_list')
;;

let merge_node_edge_lists (n_list1, e_list1) (n_list2, e_list2) =
  let n_list' = List.fold_right node_list_append_opt n_list2 n_list1 in
  let e_list' = List.fold_right edge_list_append_opt e_list2 e_list1 in
  (n_list', e_list')
;;
```

9.3 AST

9.3.1 ast.ml

```
(* Authors:
  Benjamin Lewinter bsl2121
  Irina Mateescu    im2441
  Harry Smith       hs3061
  Yasunari Watanabe yw3239
*)

type typ =
    Int
  | Fun of typ * (typ list)
  | String
  | Bool
  | Void
  | Graph of typ * typ * typ
  | Node of typ * typ
  | Edge of typ

type binding = typ * string

type binop =
  | Add
  | Sub
  | Mul
  | Div
  | And
  | Or
  | Eq
  | Neq
  | Leq
  | Geq
  | Lt
  | Gt

type unop =
  | Not
  | Neg

type expr =
  | Intlit of int
  | Stringlit of string
  | Boollit of bool
  | Funsig of typ * binding list * expr
  | Null
  | Var of string
  | Binop of expr * binop * expr
  | Unop of unop * expr
  | Asn of string * expr
  | FCall of string * expr list
  | MCall of expr * string * expr list
  | NodeExpr of expr * expr
  | EdgeExpr of expr * expr * expr
  | GraphExpr of node_list * edge_list
  | Noexpr
and node_list = expr list
and edge_list = expr list

type stmt =
  | Expr of expr
  | For of expr * expr * expr * stmt
  | ForNode of string * expr * stmt
  | ForEdge of string * string * string * expr * stmt
  | While of expr * stmt
  | If of expr * stmt * stmt
  | Block of stmt list
  | Vdecl of typ * string * expr
  | Return of expr

type fdecl = {typ: typ; fname:string; args:binding list; body:stmt list}

type program = binding list * fdecl list

exception Unsupported_constructor;;

let unwrap_node_expr n_expr =
  match n_expr with
  | NodeExpr(label, data) -> (label, data)
  | _ -> raise Unsupported_constructor
;;

let unwrap_edge_expr e_expr =
  match e_expr with
  | EdgeExpr(src, dst, w) -> (src, dst, w)
  | _ -> raise Unsupported_constructor
;;

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mul -> "*"
  | Div -> "/"
  | Eq -> "=="
  | Neq -> "!="
  | Lt -> "<"
  | Leq -> "<="
  | Gt -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

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

```

```

let rec string_of_typ = function
    Int    -> "int"
  | Bool   -> "bool"
  | Void   -> "void"
  | Fun(_)     -> "fun"
  | String -> "string"
  | Node(nl, nd)  -> "node<" ^ string_of_typ nl ^ ", " ^ string_of_typ nd ^ ">"
  | Edge(vl)      -> "edge<" ^ string_of_typ vl ^ ">"
  | Graph(nl, nd, ew) -> "graph<" ^ string_of_typ nl ^ ", " ^ string_of_typ nd ^ ", " ^ string_of_typ ew ^ ">"

let string_of_vdecl (t, var) = string_of_typ t ^ " " ^ var ^ "; "

let rec string_of_expr = function
    Intlit(l) -> string_of_int l
  | Boollit(true) -> "true"
  | Boollit(false) -> "false"
  | Var(s) -> s
  | Stringlit(l) -> l
  | Funsig(typ, bl, e) -> " fun: " ^ (string_of_typ typ) ^ " (" ^ (String.concat "" (List.map string_of_vdecl bl)) ^ ") { " ^ string_of_expr e ^ " }"
  | Null -> "null"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Asn(v, e) -> v ^ " = " ^ string_of_expr e
  | FCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | MCall(caller, f, el) ->
      string_of_expr caller ^ "." ^ f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | NodeExpr(e1, e2) -> string_of_expr e1 ^ ": " ^ string_of_expr e2
  | EdgeExpr(src, dst, w) -> "(" ^ (string_of_expr src) ^ ", " ^ (string_of_expr dst) ^ ", " ^ (string_of_expr w) ^ ")"
  | GraphExpr(node_list, edge_list) ->
      "[nodes: [" ^ String.concat ", " (List.map string_of_expr node_list) ^
      "], edges: [" ^ String.concat ", " (List.map string_of_expr edge_list) ^ "]]"
  | Noexpr -> ""


let string_of_vdecl (t, var) = string_of_typ t ^ " " ^ var ^ ";\n"

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\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
  | ForNode(n, g, body) ->
      "for (" ^  n  ^ " : " ^ string_of_expr g ^ ") " ^ string_of_stmt body
  | ForEdge(src, dst, w, g, body) ->
      "for (" ^ src  ^ ", " ^ dst  ^ ", " ^ w ^ " : " ^ string_of_expr g ^ ") " ^ string_of_stmt body
  | While(e, body) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt body
  | Vdecl(t, var, expr) ->
```

```
      match expr with
      | Noexpr ->
          string_of_vdecl (t, var)
      | _ ->
          string_of_typ t ^ " " ^ string_of_expr expr ^ ";\n"


let string_of_vdecl (t, var) = string_of_typ t ^ " " ^ var ^ ";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.args) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

9.4 SAST

9.4.1 sast.ml

```
(* Authors:
  Benjamin Lewinter bsl2121
  Irina Mateescu    im2441
  Harry Smith       hs3061
  Yasunari Watanabe yw3239
*)
(*Semantically checked abstract syntax tree*)

open Ast

type sexpr = typ * sx
and sx =
    SIntlit of int
  | SStringlit of string
  | SBoollit of bool
  | SFunsig of typ * binding list * sexpr
  | SNull
  | SVar of string
  | SBinop of sexpr * binop * sexpr
  | SUnop of unop * sexpr
  | SAsn of string * sexpr
  | SFCall of string * sexpr list
  | SMCall of sexpr * string * sexpr list
  | SNodeExpr of sexpr * sexpr
  | SEdgeExpr of sexpr * sexpr * sexpr
  | SGraphExpr of node_list * edge_list
  | SNoexpr
and node_list = sexpr list
and edge_list = sexpr list

type sstmt =
    SExpr of sexpr
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SForNode of string * sexpr * sstmt
  | SForEdge of string * string * string * sexpr * sstmt
  | SWhile of sexpr * sstmt
  | SIf of sexpr * sstmt * sstmt
  | SBlock of sstmt list
  | SVdecl of typ * string * sexpr
  | SReturn of sexpr

type sfdecl = {
  styp: typ;
  sfname: string;
  sargs: binding list;
  sbody: sstmt list;
}

type sprogram = binding list * sfdecl list

let string_of_svdecl (t, var) = string_of_typ t ^ " " ^ var ^ "; "

let rec string_of_sexpr (t, e) =
  match e with
    SIntlit(l) -> string_of_int l
  | SBoollit(true) -> "true"
  | SBoollit(false) -> "false"
  | SVar(s) -> s
  | SStringlit(l) -> l
  | SFunsig(typ, bl, e) -> " fun: " ^ (string_of_typ typ) ^ " (" ^ (String.concat "" (List.map string_of_svdecl bl)) ^ ") { " ^ string_of_sexpr e ^ " }"
  | SNull -> string_of_typ t ^ " null"
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
  | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
  | SAsn(v, e) -> v ^ " = " ^ string_of_sexpr e
  | SFCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
  | SMCall(caller, f, el) ->
      string_of_sexpr caller ^ "." ^ f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
  | SNodeExpr(e1, e2) -> string_of_sexpr e1 ^ ": " ^ string_of_sexpr e2
  | SEdgeExpr(src, dst, w) -> "(" ^ (string_of_sexpr src) ^ ", " ^ (string_of_sexpr dst) ^ ", " ^ (string_of_sexpr w) ^ ")"
  | SGraphExpr(node_list, edge_list) ->
      "[nodes: [" ^ String.concat ", " (List.map string_of_sexpr node_list) ^
      "], edges: [" ^ String.concat ", " (List.map string_of_sexpr edge_list) ^ "]]"
  | SNoexpr -> ""

let rec string_of_sstmt = function
    SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s, SBlock([])) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
      "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
      string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
  | SForNode(n, g, body) ->
      "for (" ^  n  ^ " : " ^ string_of_sexpr g ^ ") " ^ string_of_sstmt body
  | SForEdge(src, dst, w, g, body) ->
      "for (" ^ src  ^ ", " ^ dst  ^ ", " ^ w ^ " : " ^ string_of_sexpr g ^ ") " ^ string_of_sstmt body
  | SWhile(e, body) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt body
  | SVdecl(t, var, expr) ->
```

```
      match expr with
      | (_, SNoexpr) ->
          string_of_svdecl (t, var)
      | _ ->
          string_of_typ t ^ " " ^ string_of_sexpr expr ^ ";\n"

let string_of_sfdecl fdecl = string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sargs) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"


let string_of_sprogram (vars, funcs) =
  (match vars with
   | [] -> ""
   | _ -> String.concat "" (List.map string_of_svdecl vars) ^ "\n") ^
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

9.5 Semantic Checking

9.5.1 semant.ml

```
(* Authors:
  Benjamin Lewinter bsl2121
  Irina Mateescu    im2441
  Harry Smith       hs3061
  Yasunari Watanabe yw3239
*)

open Ast
open Sast

module StringMap = Map.Make(String)

let check (globals, funcs) =
  (* Verify a list of bindings has no void types or duplicate names *)
  let check_binds (kind : string) (binds : binding list) =
    List.iter (function
  (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
      | _ -> ()) binds;
    let rec dups = function
        [] -> ()
      | ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
    raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
      | _ :: t -> dups t
    in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
  in

  (**** Check global variables ****)
  check_binds "global" globals;

  (**** Check functions and methods ****)

  let built_in_fdecls =
    let add_bind map (name, (ty, args)) =
      StringMap.add name
                    { typ = ty; fname = name; args = args; body = [] }
                    map in
    let mappings = [
      "print_int", (Void, [(Int, "x")]);
      "print_bool", (Void, [(Bool, "x")]);
      "print", (Void, [(String, "x")])
    ] in
    List.fold_left add_bind StringMap.empty mappings
  in

  (* Add function name to symbol table *)
  let add_func map fd =
    let built_in_err = "function " ^ fd.fname ^ " may not be defined"
    and dup_err = "duplicate function " ^ fd.fname
    and make_err er = raise (Failure er)
    and n = fd.fname (* Name of the function *)
    in match fd with (* No duplicate functions or redefinitions of built-ins *)
         _ when StringMap.mem n built_in_fdecls -> make_err built_in_err
       | _ when StringMap.mem n map -> make_err dup_err
       | _ ->  StringMap.add n fd map
  in

  (* Collect all function names into one symbol table *)
  let fdecls = List.fold_left add_func built_in_fdecls funcs
  in

  (* Return a function from our symbol table *)
  let find_func local_fdecls s =
    try StringMap.find s fdecls
    with Not_found ->
      try StringMap.find s local_fdecls
      with Not_found -> raise (Failure ("unrecognized function " ^ s))
  in

  (* Return a method from our symbol table *)
  let find_method libtyp s margs=
    try match libtyp with
        | Node(lt, dt) ->
           (match s with
            | "get_data" ->
               { typ = dt; fname = s; args = []; body = [] }
            | "set_data" ->
               { typ = Void; fname = s; args = [(dt, "d")]; body = [] }
            | "get_name" ->
               { typ = lt; fname = s; args = []; body = [] }
            | "print" ->
               { typ = Void; fname = s; args = []; body = [] }
            | _ ->
               raise Not_found)
        | Graph(lt, dt, wt) ->
          (match s with
          | "set_node" ->
              { typ = Int; fname = s; args = [(Node(lt, dt), "x")]; body = [] }
          | "set_edge" ->
              if List.length margs = 2 then
              { typ = Int; fname = s; args = [(lt, "src"); (lt, "dst")]; body = [] }
              else
              { typ = Int; fname = s; args = [(lt, "src"); (lt, "dst"); (wt, "w")]; body = [] }
          | "remove_node" ->
              { typ = Int; fname = s; args = [(lt, "x")]; body = [] }
          | "remove_edge" ->
              { typ = Int; fname = s; args = [(lt, "src"); (lt, "dst")]; body = [] }
          | "get_node" ->
              { typ = Node(lt, dt); fname = s; args = [(lt, "l")]; body = [] }
          | "get_weight" ->
              { typ = wt; fname = s; args = [(lt, "src"); (lt, "dst")]; body = [] }
          | "print" ->
             { typ = Void; fname = s; args = []; body = [] }
          | "has_node" ->
             { typ = Int; fname = s; args = [(lt, "src")]; body = [] }
          | "are_neighbors" ->
             { typ = Bool; fname = s; args = [(lt, "src"); (lt, "dst")]; body = [] }
          | "is_empty" ->
             { typ = Bool; fname = s; args = []; body = [] }
          | "neighbors" ->
             if List.length margs = 1 then
             { typ = Graph(lt, dt, wt); fname = s; args = [(lt, "label")]; body = [] }
             else
             { typ = Graph(lt, dt, wt); fname = s; args = [(lt, "label"); (Int, "level"); (Bool, "include_current")]; body = [] }
          | "find" ->
             { typ = Graph(lt, dt, wt); fname = s; args = [(dt, "data")]; body = [] }
          | "dfs" ->
             { typ = Graph(lt, dt, wt); fname = s; args = [(lt, "label")]; body = [] }
          | "bfs" ->
             { typ = Graph(lt, dt, wt); fname = s; args = [(lt, "label")]; body = [] }
          | _ -> raise Not_found)
        | _ -> raise Not_found
    with Not_found -> raise (Failure ("unrecognized method " ^ string_of_typ libtyp ^ "." ^ s))
  in

  let _ = find_func fdecls "main" in (*Ensure "main" is defined*)

  let check_function func =
    (* Make sure no args are void or duplicates *)
    check_binds "args" func.args;

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

    (* Build global and local symbol table of variables for this function *)

    let global_vars = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
                                    StringMap.empty (globals)
    in

    let local_vars = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
                                    StringMap.empty (func.args)
    in

    let funcs' = List.map (fun (ty, name) ->
                            match ty with
                            | Fun(ret_t, args_t) ->
                               {typ = ret_t; fname = name; args = List.map (fun t -> (t, "x")) args_t; body = []}
                            | _ -> raise Unsupported_constructor)
                         (List.filter (fun (ty, _) -> match ty with Fun(_) -> true |
                             _ -> false) func.args) in


    let local_fdecls = List.fold_left add_func StringMap.empty funcs’ in

    (* Return a variable from our local symbol table *)
    let type_of_variable vars s =
      try StringMap.find s vars
      with Not_found ->
        try StringMap.find s global_vars
        with Not_found ->
          raise (Failure ("undeclared variable " ^ s))
    in

    (* Return a semantically-checked expression, i.e., with a type *)
    let rec expr fdecls vars = function
        Intlit l  -> (Int, SIntlit l)
      | Boollit l   -> (Bool, SBoollit l)
      | Stringlit l -> (String, SStringlit l)
      | Null -> (Bool, SNull)
      | Funsig (t, bl, e) ->
          check_anon_func_expr fdecls vars t bl e
      | Noexpr      -> (Void, SNoexpr)
      | Var s       -> (type_of_variable vars s, SVar s)
      | NodeExpr (l, d)  ->
        let (lt, _) as l’ = expr fdecls vars l in
        let (dt, _) as d’ = expr fdecls vars d in
        (Node(lt, dt), SNodeExpr(l’, d’))
      | EdgeExpr (src, dst, w)  ->
        let (wt, _) as w’ = expr fdecls vars w in
        (Edge(wt), SEdgeExpr(expr fdecls vars src, expr fdecls vars dst, w’))
      | Asn(var, e) ->
          check_asn_expr fdecls vars var e
      | Unop(op, e) ->
          check_unop_expr fdecls vars op e
      | Binop(e1, op, e2) ->
          check_binop_expr fdecls vars e1 op e2
      | FCall(fname, args) ->
         check_fcall_expr fdecls vars fname args
      | MCall(instance, mname, args) ->
         check_mcall_expr fdecls vars instance mname args
      | GraphExpr(node_list, edge_list) ->
         check_graph_expr fdecls vars node_list edge_list

    and coerce_null_to_typ new_typ e =
      match e with
      | (Bool, SNull) -> (new_typ, SNull)
      | _ -> e

    and check_asn_expr fdecls vars var e =
      let lvt = type_of_variable vars var in
      let (rvt, e’) = coerce_null_to_typ lvt (expr fdecls vars e) in
      let err = "illegal assignment " ^ string_of_typ lvt ^ " = " ^
        string_of_typ rvt ^ " in " ^ string_of_expr (Asn(var, e))
      in

      match lvt, rvt, e’ with
         (* If left expression is a node with bool type data, wrap right expression
            in a SNodeExpr *)
      | Node(llt, ldt), _, SNodeExpr((lt, _), d) ->
         let (dt, _) = coerce_null_to_typ ldt d in
         let lt = check_asn llt lt err in
         let dt = check_asn ldt dt err in
         (Node(lt, dt), SAsn(var, (lvt, e')))
      | Node(_), Node(_), _ ->
         let lt = check_asn lvt rvt err in
         (lt, SAsn(var, (lt, e')))
      | Node(llt, ldt), _, _ ->
         let lt = check_asn llt rvt err in
         (Node(lt, ldt), SAsn(var, (lvt, SNodeExpr((llt, e'), (ldt, SNull)))))
      | Graph(llt, ldt, lwt), Graph(_, _, _), SGraphExpr(nl, el) ->
         (* Coerce (Bool, SNull) to correct type then check type equality *)
         let nl' = List.map (fun (_, e) -> match e with
                                            | SNodeExpr((lt, le), d) ->
                                               let (dt, de) = coerce_null_to_typ ldt d in
                                               let lt = check_asn llt lt err in
                                               let dt = check_asn ldt dt err in
                                               (Node(lt, dt), SNodeExpr((lt, le), (dt, de)))
                                            | _ -> raise Unsupported_constructor) nl in
         let el' = List.map (fun (_, e) -> match e with
                                            | SEdgeExpr(src, dst, w) ->
                                               let (wt, we) = coerce_null_to_typ lwt w in
                                               let wt = check_asn lwt wt err in
                                               (Edge(lwt), SEdgeExpr(src, dst, (wt, we)))
                                            | _ -> raise Unsupported_constructor) el in

         let t = Graph(llt, ldt, lwt) in
         (t, SAsn(var, (t, SGraphExpr(nl', el'))))
      | Fun(_), Fun(_), SFunsig(_) ->
          let (_, new_expr) = expr fdecls vars e in
              (check_asn lvt rvt err, SAsn(var, (rvt, new_expr)))

      | _ ->
         (check_asn lvt rvt err, SAsn(var, (rvt, e')))

    and check_unop_expr fdecls vars op e =
      let (t, e') = expr fdecls vars e in
      let ty =
        match op with
        | Not when t = Bool -> Bool
        | Neg when t = Int -> Int
        | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op ^
                               string_of_typ t ^ " in " ^ string_of_expr (Unop(op, e))))
      in (ty, SUnop(op, (t, e')))

    and check_binop_expr fdecls vars e1 op e2 =
      let (t1, e1') = expr fdecls vars e1
      and (t2, e2') = expr fdecls vars e2 in
      (* All binary operators require operands of the same type *)
      let same = t1 = t2 in
      (* Determine expression type based on operator and operand types *)
      let ty = match op with
               | Add | Sub | Mul | Div when same && t1 = Int -> Int
               | Eq  | Neq when same -> Bool
               | Lt | Leq | Gt | Geq when same && (t1 = Int || t1 = String) -> Bool
               | And | Or when same && t1 = Bool -> Bool
               | _ ->
                  raise (Failure ("illegal binary operator " ^
                                    string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                                    string_of_typ t2 ^ " in " ^ string_of_expr (Binop (e1, op, e2))))
      in (ty, SBinop((t1, e1'), op, (t2, e2')))

    and check_fcall_expr fdecls vars fname args =
      let fd = find_func fdecls fname in
      let param_length = List.length fd.args in
      if List.length args != param_length
      then raise (Failure ("expecting " ^ string_of_int param_length ^
                           " arguments in " ^ string_of_expr (FCall(fname, args))))
      else let check_call (ft, _) e =
             let (et, e') = expr fdecls vars e in
             let err = "illegal argument found " ^ string_of_typ et ^
                       " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e in
             (check_asn ft et err, e')
           in
           let args' = List.map2 check_call fd.args args in
           (fd.typ, SFCall(fname, args'))

    and check_mcall_expr fdecls vars instance mname args =
      let (instance_typ, _) as instance' = expr fdecls vars instance in
      let md = find_method instance_typ mname args in
      let param_length = List.length md.args in
      if List.length args != param_length
      then raise (Failure ("expecting " ^ string_of_int param_length ^
                           " arguments in method " ^ string_of_expr (MCall(instance, mname, args))))
      else let check_call (ft, _) e =
             let (et, e') = expr fdecls vars e in
             let err = "illegal argument found " ^ string_of_typ et ^
                       " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e in
             (check_asn ft et err, e')
           in
           let args' = List.map2 check_call md.args args in
           (md.typ, SMCall(instance', mname, args'))

    and check_graph_expr fdecls vars node_list edge_list =
     (* infer node label/data types from first nodes in list if any,
        and check that all items have the same type *)
      let node_label_typ, node_data_typ, s_node_list =
        if node_list = []
        then (Bool, Bool, []) (* bool type, for now *)
        else let err = "type mismatch in graph nodes" in
          let check_node_typ (lt_opt, dt_opt) n =
            match n with
            | (Node(lt, dt), SNodeExpr(_, d)) ->
               (* check matching node label *)
               let lt_opt = (match lt_opt with
               | None -> Some(lt)
               | Some(lt') -> if lt = lt'
                               then lt_opt
                               else raise (Failure err)) in
               (* check matching node data *)
               let dt_opt = (match d with
               | (Bool, SNull) -> dt_opt
               | _ -> match dt_opt with
                      | None -> Some(dt)
                      | Some(dt') -> if dt = dt'
                                      then dt_opt
                                      else raise (Failure err))
               in (lt_opt, dt_opt)
            | _ -> raise Unsupported_constructor
          in
          let node_list' = List.map (expr fdecls vars) node_list in
          match List.fold_left check_node_typ (None, None) node_list' with
          | None, _ -> raise (Failure "graph node names are required")
          | Some(lt), None -> (lt, Bool, node_list')
          | Some(lt), Some(dt) -> (lt, dt, node_list')
      in
      (* infer edge weight types from first edge in list if any,
        and check that all items have the same type *)
      let edge_typ, s_edge_list =
        if edge_list = []
        then (Bool, []) (* bool type, for now *)
        else let err = "type mismatch in graph edges" in
          let check_edge_typ wt_opt e =
            match e with
            | (Edge(wt), SEdgeExpr(_, _, w)) ->
              (match w with
               | (Bool, SNull) -> wt_opt
               | _ -> (match wt_opt with
                       | None -> Some(wt)
                       | Some(wt') -> if wt = wt'
                                       then wt_opt
                                       else raise (Failure err)))
            | _ -> raise Unsupported_constructor
          in
          let edge_list' = List.map (expr fdecls vars) edge_list in
          match List.fold_left check_edge_typ None edge_list' with
          | None -> (Bool, edge_list')
          | Some(wt) -> (wt, edge_list')
     in
     (Graph(node_label_typ, node_data_typ, edge_typ), SGraphExpr(s_node_list, s_edge_list))


    and check_anon_func_expr fdecls vars typ b_list ex =
      let vars' = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
                                     vars (b_list) in
      let (ty, sx) = expr fdecls vars' ex in
      let err = "type mismatch in result of anonymous function" in
      let checked_type = check_asn typ ty err in
      let typ_list = List.map (fun (ty, _) -> ty) b_list in
      (Fun(typ, typ_list), SFunsig(checked_type, b_list, (ty, sx)))
    in

    let check_bool_expr fdecls vars e =
      let (t', e') = expr fdecls vars e
      and err = "expected Boolean expression in " ^ string_of_expr e
      in if t' != Bool then raise (Failure err) else (t', e')
    in

    (* Return a semantically-checked statement i.e. containing sexprs *)
    let rec check_stmt fdecls vars = function
        Expr e -> (*
          (match e with
              Asn(s, Funsig(ty, bl, body)) ->
                    let new_fdecl = {typ = ty; fname = s; args = bl; body = [Expr(body)]} in
                    let fdecls' = add_func fdecls new_fdecl in
                    (fdecls', vars, SExpr (expr fdecls' vars e))
            | _ ->
               *)(fdecls, vars, SExpr (expr fdecls vars e))
      | For (e1, e2, e3, st) ->
          let (_, _, st') = check_stmt fdecls vars st in
          (fdecls, vars, SFor (expr fdecls vars e1, check_bool_expr fdecls vars e2, expr fdecls vars e3, st'))
      | ForNode (n, g, st) ->
          (match expr fdecls vars g with
           | (Graph(lt, dt, _), _) as ge ->
              let vars' = StringMap.add n (Node (lt, dt)) vars in
              let (_, _, st') = check_stmt fdecls vars' st in
              (fdecls, vars', SForNode (n, ge, st'))
           | (ty, _) -> raise (Failure ("illegal argument found: expected graph, got " ^ string_of_typ ty)))
      | ForEdge (src, dst, w, g, st) ->
          (match expr fdecls vars g with
           | (Graph(lt, dt, wt), _) as ge ->
              let nt = Node(lt, dt) in
              let vars' = StringMap.add src nt (StringMap.add dst nt (StringMap.add w wt vars)) in
              let (_, _, st') = check_stmt fdecls vars' st in
              (fdecls, vars', SForEdge (src, dst, w, ge, st'))
           | (ty, _) -> raise (Failure ("illegal argument found: expected graph, got " ^ string_of_typ ty)))
      | While (p, st) ->
          let (_, _, st') = check_stmt fdecls vars st in
          (fdecls, vars, SWhile (check_bool_expr fdecls  vars p, st'))
      | If (p, b1, b2) ->
          let (_, _, b1') = check_stmt fdecls vars b1 in
          let (_, _, b2') = check_stmt fdecls vars b2 in
          (fdecls, vars, SIf (check_bool_expr fdecls vars p, b1', b2'))
      | Vdecl (ty, s, e) ->
        if ty = Void
        then raise (Failure ("variable '" ^ s ^ "' declared void"))
        else let vars' = StringMap.add s ty vars in
          (match ty, e with
          | Fun(_), Asn(_, Var(var_name)) ->
              (print_string "here\n");
              let new_fun = {(find_func fdecls var_name) with fname = s} in
              let fdecls' = add_func fdecls new_fun in
              (fdecls', vars', SVdecl (ty, s, expr fdecls vars' e))
          | _ ->
              (fdecls, vars', SVdecl (ty, s, expr fdecls vars' e)))
      | Return e ->
         let (t, e') = coerce_null_to_typ func.typ (expr fdecls vars e) in
         if t = func.typ then (fdecls, vars, SReturn (t, e'))
         else raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                              string_of_typ func.typ ^ " in " ^ string_of_expr e))
      (* A block is correct if each statement is correct and nothing
         follows any Return statement.  Nested blocks are flattened. *)
      | Block sl ->
          let rec check_stmt_list fdecls vars = function
             [Return _ as s] ->
                let (_, _, s') = check_stmt fdecls vars s in [s']
            | Return _ :: _   -> raise (Failure "nothing may follow a return")
            | Block sl :: ss  -> check_stmt_list fdecls vars (sl @ ss) (* Flatten
               blocks *)
            | s :: ss         ->
                (match s with
                  | Vdecl(_, var_name, Asn(_, Funsig(ty, bl, body))) ->
                    let new_fdecl = {typ = ty; fname = var_name; args = bl; body = [Expr(body)]} in
                    let fdecls' = add_func fdecls new_fdecl in
                    let (fdecls'', vars', s') = check_stmt fdecls' vars s in
                    s' :: check_stmt_list fdecls'' vars' ss
                  | _ ->
                    let (fdecls', vars', s') = check_stmt fdecls vars s in
                    s' :: check_stmt_list fdecls' vars' ss)
            | []              -> []
          in (fdecls, vars, SBlock(check_stmt_list fdecls vars sl))

    in (* body of check_function *)
    { styp = func.typ;
      sfname = func.fname;
      sargs = func.args;
      sbody = let (_, _, st) = check_stmt local_fdecls local_vars (Block func.body)
              in match st with
               SBlock(sl) -> sl
              | _ -> raise (Failure ("internal error: block didn't become a block?"))
    }
  in (globals, List.map (check_function) funcs)
```

9.6 Code Generation

9.6.1 codegen.ml

```
(* Authors:
  Benjamin Lewinter bsl2121
  Irina Mateescu    im2441
  Harry Smith       hs3061
  Yasunari Watanabe yw3239
*)

module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)

let translate (globals, functions) =
  let context = L.global_context () in
  let the_module = L.create_module context "Hippograph" in

  let void_t  =   L.void_type context
  and i32_t  = L.i32_type  context
  and i32_ptr_t  = L.pointer_type (L.i32_type context)
  and i8_t  =   L.i8_type context
  and i1_t   = L.i1_type   context
  and str_t = L.pointer_type (L.i8_type context)
  and void_ptr_t = L.pointer_type (L.i8_type context)
  in

  let rec ltype_of_typ = function
    | A.Void    -> void_t
    | A.Int     -> i32_t
    | A.Bool    -> i1_t
    | A.String  -> str_t
    | A.Fun(ret_t, args) ->
      let formal_types = Array.of_list (List.map ltype_of_typ args) in
      L.pointer_type (L.function_type (ltype_of_typ ret_t) formal_types)
    | A.Node(_, _) -> void_ptr_t
    | A.Edge(_) -> void_ptr_t
    | A.Graph(_,_,_) -> void_ptr_t
  in

  (* Declare each global variable; remember its value in a map *)
  let global_vars =
    let global_var m (t, n) =
      let init =
        match t with
        | A.String -> L.const_bitcast (L.const_stringz context "") str_t
        | _ -> L.const_int (ltype_of_typ t) 0
      in StringMap.add n (L.define_global n init the_module) m in
    List.fold_left global_var StringMap.empty globals in

  (* C Functions *)

  let print_t : L.lltype = L.var_arg_function_type void_t [| L.pointer_type i8_t |] in
  let print_func : L.llvalue = L.declare_function "printf" print_t the_module in

  let strcmp_t : L.lltype = L.var_arg_function_type i32_t [| str_t; str_t |] in
  let strcmp_func : L.llvalue = L.declare_function "strcmp" strcmp_t the_module in

  let create_graph_t : L.lltype = L.var_arg_function_type void_ptr_t [| |] in
  let create_graph_func : L.llvalue = L.declare_function "create_graph" create_graph_t the_module in

  let add_node_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; void_ptr_t |] in
  let add_node_func : L.llvalue = L.declare_function "add_node" add_node_t the_module in

  let create_node_t : L.lltype = L.var_arg_function_type void_ptr_t [| |] in
  let create_node_func : L.llvalue = L.declare_function "create_node" create_node_t the_module in

  let clone_node_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
  let clone_node_func : L.llvalue = L.declare_function "clone_node" clone_node_t the_module in

  let create_edge_t : L.lltype = L.var_arg_function_type void_ptr_t [| |] in
  let create_edge_func : L.llvalue = L.declare_function "create_edge" create_edge_t the_module in

  let add_edge_int_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; void_ptr_t; i32_t; i32_t |] in
  let add_edge_int_func : L.llvalue = L.declare_function "add_edge_int" add_edge_int_t the_module in

  let add_edge_bool_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; void_ptr_t; i1_t; i1_t |] in
  let add_edge_bool_func : L.llvalue = L.declare_function "add_edge_bool" add_edge_bool_t the_module in

  let add_edge_str_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t;
```

```
      void_ptr_t; str_t; str_t |] in
  let add_edge_str_func : L.llvalue = L.declare_function "add_edge_str"
      add_edge_str_t the_module in

  let set_node_label_int_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t
      ; i32_t |] in
  let set_node_label_int_func : L.llvalue = L.declare_function "set_node_label_int"
      set_node_label_int_t the_module in

  let set_node_label_bool_t : L.lltype = L.var_arg_function_type void_t [|
      void_ptr_t; i1_t |] in
  let set_node_label_bool_func : L.llvalue = L.declare_function "set_node_label_bool
      " set_node_label_bool_t the_module in

  let set_node_label_str_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t
      ; str_t |] in
  let set_node_label_str_func : L.llvalue = L.declare_function "set_node_label_str"
      set_node_label_str_t the_module in

  let set_node_data_int_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t;
       i32_t; i1_t |] in
  let set_node_data_int_func : L.llvalue = L.declare_function "set_node_data_int"
      set_node_data_int_t the_module in

  let set_node_data_bool_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t
      ; i1_t; i1_t |] in
  let set_node_data_bool_func : L.llvalue = L.declare_function "set_node_data_bool"
      set_node_data_bool_t the_module in

  let set_node_data_str_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t;
       str_t; i1_t |] in
  let set_node_data_str_func : L.llvalue = L.declare_function "set_node_data_str"
      set_node_data_str_t the_module in

  let get_node_label_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t
       |] in
  let get_node_label_func : L.llvalue = L.declare_function "get_node_label"
      get_node_label_t the_module in

  let get_node_data_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t
      |] in
  let get_node_data_func : L.llvalue = L.declare_function "get_node_data"
      get_node_data_t the_module in

  let graph_has_node_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t;
       i32_t |] in
  let graph_has_node_int_func : L.llvalue = L.declare_function "graph_has_node_int"
      graph_has_node_int_t the_module in

  let graph_has_node_str_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t;
       str_t |] in
  let graph_has_node_str_func : L.llvalue = L.declare_function "graph_has_node_str"
      graph_has_node_str_t the_module in

  let graph_has_node_bool_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t
      ; i1_t |] in
  let graph_has_node_bool_func : L.llvalue = L.declare_function "graph_has_node_int"
       graph_has_node_bool_t the_module in

  let graph_set_edge_int_int_t : L.lltype = L.var_arg_function_type i32_t [|
      void_ptr_t; i32_t; i32_t; i32_t |] in
```

```
    let graph_set_edge_int_int_func : L.llvalue = L.declare_function "graph_set_edge_int_int" graph_set_edge_int_int_t the_module in

    let graph_set_edge_bool_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i1_t; i1_t; i32_t |] in
    let graph_set_edge_bool_int_func : L.llvalue = L.declare_function "graph_set_edge_int_int" graph_set_edge_bool_int_t the_module in

    let graph_set_edge_str_bool_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; str_t; str_t; i1_t |] in
    let graph_set_edge_str_bool_func : L.llvalue = L.declare_function "graph_set_edge_str_int" graph_set_edge_str_bool_t the_module in

    let graph_set_edge_bool_str_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i1_t; i1_t; str_t |] in
    let graph_set_edge_bool_str_func : L.llvalue = L.declare_function "graph_set_edge_int_str" graph_set_edge_bool_str_t the_module in

    let graph_set_edge_int_bool_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i32_t; i32_t; i1_t |] in
    let graph_set_edge_int_bool_func : L.llvalue = L.declare_function "graph_set_edge_int_int" graph_set_edge_int_bool_t the_module in

    let graph_set_edge_bool_bool_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i1_t; i1_t; i1_t |] in
    let graph_set_edge_bool_bool_func : L.llvalue = L.declare_function "graph_set_edge_int_int" graph_set_edge_bool_bool_t the_module in

    let graph_set_edge_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i32_t; i32_t |] in
    let graph_set_edge_int_func : L.llvalue = L.declare_function "graph_set_edge_int" graph_set_edge_int_t the_module in

    let graph_set_edge_str_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; str_t; str_t |] in
    let graph_set_edge_str_func : L.llvalue = L.declare_function "graph_set_edge_str" graph_set_edge_str_t the_module in

    let graph_set_edge_bool_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i1_t; i1_t |] in
    let graph_set_edge_bool_func : L.llvalue = L.declare_function "graph_set_edge_int" graph_set_edge_bool_t the_module in

    let graph_set_edge_str_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; str_t; str_t; i32_t |] in
    let graph_set_edge_str_int_func : L.llvalue = L.declare_function "graph_set_edge_str_int" graph_set_edge_str_int_t the_module in

    let graph_set_edge_int_str_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i32_t; i32_t; str_t |] in
    let graph_set_edge_int_str_func : L.llvalue = L.declare_function "graph_set_edge_int_str" graph_set_edge_int_str_t the_module in

    let graph_set_edge_str_str_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; str_t; str_t; str_t |] in
    let graph_set_edge_str_str_func : L.llvalue = L.declare_function "graph_set_edge_str_str" graph_set_edge_str_str_t the_module in

    let get_node_by_label_int_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; i32_t |] in
    let get_node_by_label_int_func : L.llvalue = L.declare_function "
    get_node_by_label_int" get_node_by_label_int_t the_module in
  let get_node_by_label_int_opt_func : L.llvalue = L.declare_function "get_node_by_label_int_opt" get_node_by_label_int_t the_module in

  let get_node_by_label_bool_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; i1_t |] in
  let get_node_by_label_bool_opt_func : L.llvalue = L.declare_function "get_node_by_label_bool_opt" get_node_by_label_bool_t the_module in

  let get_node_by_label_str_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; str_t |] in
  let get_node_by_label_str_func : L.llvalue = L.declare_function "get_node_by_label_str" get_node_by_label_str_t the_module in
  let get_node_by_label_str_opt_func : L.llvalue = L.declare_function "get_node_by_label_str_opt" get_node_by_label_str_t the_module in

  let print_node_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t |] in
  let print_node_func : L.llvalue = L.declare_function "print_node" print_node_t the_module in

  let print_graph_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t |] in
  let print_graph_func : L.llvalue = L.declare_function "print_graph" print_graph_t the_module in

  let set_edge_w_int_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t; i32_t; i1_t |] in
  let set_edge_w_int_func : L.llvalue = L.declare_function "set_edge_w_int" set_edge_w_int_t the_module in

  let set_edge_w_bool_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t; i1_t; i1_t |] in
  let set_edge_w_bool_func : L.llvalue = L.declare_function "set_edge_w_bool" set_edge_w_bool_t the_module in

  let set_edge_w_str_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t; str_t; i1_t |] in
  let set_edge_w_str_func : L.llvalue = L.declare_function "set_edge_w_str" set_edge_w_str_t the_module in

  let get_edge_src_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
  let get_edge_src_func : L.llvalue = L.declare_function "get_edge_src" get_edge_src_t the_module in

  let get_edge_dst_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
  let get_edge_dst_func : L.llvalue = L.declare_function "get_edge_dst" get_edge_dst_t the_module in

  let get_edge_w_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t |] in
  let get_edge_w_int_func : L.llvalue = L.declare_function "get_edge_w_int" get_edge_w_int_t the_module in

  let get_edge_w_bool_t : L.lltype = L.var_arg_function_type i1_t [| void_ptr_t |] in
  let get_edge_w_bool_func : L.llvalue = L.declare_function "get_edge_w_int" get_edge_w_bool_t the_module in

  let get_edge_w_str_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
  let get_edge_w_str_func : L.llvalue = L.declare_function "get_edge_w_str"
      get_edge_w_str_t the_module in

  let graph_to_node_iterable_t : L.lltype = L.var_arg_function_type void_ptr_t [|
      void_ptr_t |] in
  let graph_to_node_iterable_func : L.llvalue = L.declare_function "
      graph_to_node_iterable" graph_to_node_iterable_t the_module in

  let graph_to_edge_iterable_t : L.lltype = L.var_arg_function_type void_ptr_t [|
      void_ptr_t |] in
  let graph_to_edge_iterable_func : L.llvalue = L.declare_function "
      graph_to_edge_iterable" graph_to_edge_iterable_t the_module in

  let get_graph_next_node_t : L.lltype = L.var_arg_function_type void_ptr_t [|
      void_ptr_t |] in
  let get_graph_next_node_func : L.llvalue = L.declare_function "get_graph_next_node
      " get_graph_next_node_t the_module in

  let get_graph_next_edge_t : L.lltype = L.var_arg_function_type void_ptr_t [|
      void_ptr_t |] in
  let get_graph_next_edge_func : L.llvalue = L.declare_function "get_graph_next_edge
      " get_graph_next_edge_t the_module in

  let graph_set_node_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t;
      void_ptr_t |] in
  let graph_set_node_func : L.llvalue = L.declare_function "graph_set_node"
      graph_set_node_t the_module in

  let remove_edge_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t;
      void_ptr_t |] in
  let remove_edge_func : L.llvalue = L.declare_function "remove_edge" remove_edge_t
      the_module in

  let remove_node_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t;
      i32_t |] in
  let remove_node_int_func : L.llvalue = L.declare_function "remove_node_int"
      remove_node_int_t the_module in

  let remove_node_str_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t;
      str_t |] in
  let remove_node_str_func : L.llvalue = L.declare_function "remove_node_str"
      remove_node_str_t the_module in

  let remove_node_bool_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t;
      i1_t |] in
  let remove_node_bool_func : L.llvalue = L.declare_function "remove_node_int"
      remove_node_bool_t the_module in

  let get_edge_by_src_and_dst_int_t : L.lltype = L.var_arg_function_type void_ptr_t
      [| void_ptr_t; i32_t; i32_t |] in
  let get_edge_by_src_and_dst_int_func : L.llvalue = L.declare_function "
      get_edge_by_src_and_dst_int" get_edge_by_src_and_dst_int_t the_module in

  let get_edge_by_src_and_dst_bool_t : L.lltype = L.var_arg_function_type void_ptr_t
       [| void_ptr_t; i1_t; i1_t |] in
  let get_edge_by_src_and_dst_bool_func : L.llvalue = L.declare_function "
      get_edge_by_src_and_dst_int" get_edge_by_src_and_dst_bool_t the_module in

  let get_edge_by_src_and_dst_str_t : L.lltype = L.var_arg_function_type void_ptr_t
      [| void_ptr_t; void_ptr_t; void_ptr_t |] in
  let get_edge_by_src_and_dst_str_func : L.llvalue = L.declare_function "
```

```
get_edge_by_src_and_dst_str" get_edge_by_src_and_dst_str_t the_module in

  let neighbors_one_arg_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
  let neighbors_one_arg_func : L.llvalue = L.declare_function "neighbors_one_arg" neighbors_one_arg_t the_module in

  let neighbors_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; i32_t; i1_t |] in
  let neighbors_func : L.llvalue = L.declare_function "neighbors" neighbors_t the_module in

  let find_data_int_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; i32_t |] in
  let find_data_int_func : L.llvalue = L.declare_function "find_data_int" find_data_int_t the_module in

  let find_data_bool_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; i1_t |] in
  let find_data_bool_func : L.llvalue = L.declare_function "find_data_int" find_data_bool_t the_module in

  let find_data_str_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; str_t |] in
  let find_data_str_func : L.llvalue = L.declare_function "find_data_str" find_data_str_t the_module in

  let are_neighbors_int_t : L.lltype = L.var_arg_function_type i1_t [| void_ptr_t; i32_t; i32_t |] in
  let are_neighbors_int_func : L.llvalue = L.declare_function "are_neighbors_int" are_neighbors_int_t the_module in

  let are_neighbors_bool_t : L.lltype = L.var_arg_function_type i1_t [| void_ptr_t; i1_t; i1_t |] in
  let are_neighbors_bool_func : L.llvalue = L.declare_function "are_neighbors_int" are_neighbors_bool_t the_module in

  let are_neighbors_str_t : L.lltype = L.var_arg_function_type i1_t [| void_ptr_t; str_t; str_t |] in
  let are_neighbors_str_func : L.llvalue = L.declare_function "are_neighbors_str" are_neighbors_str_t the_module in

  let is_empty_t : L.lltype = L.var_arg_function_type i1_t [| void_ptr_t |] in
  let is_empty_func : L.llvalue = L.declare_function "is_empty" is_empty_t the_module in

  let dfs_int_t : L.lltype = L.function_type void_ptr_t [| void_ptr_t; i32_t |] in
  let dfs_int_func : L.llvalue = L.declare_function "dfs_int" dfs_int_t the_module in

  let dfs_str_t : L.lltype = L.function_type void_ptr_t [| void_ptr_t; str_t |] in
  let dfs_str_func : L.llvalue = L.declare_function "dfs_str" dfs_str_t the_module in

  let bfs_int_t : L.lltype = L.function_type void_ptr_t [| void_ptr_t; i32_t |] in
  let bfs_int_func : L.llvalue = L.declare_function "bfs_int" bfs_int_t the_module in

  let bfs_str_t : L.lltype = L.function_type void_ptr_t [| void_ptr_t; str_t |] in
  let bfs_str_func : L.llvalue = L.declare_function "bfs_str" bfs_str_t the_module in

  let global_fdecls : (L.llvalue * sfdecl) StringMap.t =
    let function_decl m (sfdecl : sfdecl) =
      let name = sfdecl.sfname
      and formal_types =
        Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) sfdecl.sargs)
      in let ftype =
        L.function_type (ltype_of_typ sfdecl.styp) formal_types in
      StringMap.add name (L.define_function name ftype the_module, sfdecl) m in
    List.fold_left function_decl StringMap.empty functions in

  let build_function_body local_fdecls sfdecl =
    let (the_function, _) = try StringMap.find sfdecl.sfname local_fdecls
                            with Not_found -> StringMap.find sfdecl.sfname global_fdecls
                            in
    let builder = L.builder_at_end context (L.entry_block the_function) in

    let str_format_str = L.build_global_stringptr "%s\n" "fmt" builder in
    let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder in

    (* Construct the function's "locals": formal arguments and locally
       declared variables.  Allocate each on the stack, initialize their
       value, if appropriate, and remember their values in the "locals" map *)
    let add_arg builder m (t, n) p = L.set_value_name n p;
      let local = L.build_alloca (ltype_of_typ t) n builder in
      ignore (L.build_store p local builder);
      StringMap.add n local m in

    let add_local_var builder m (t, n) =
      let local_var = L.build_alloca (ltype_of_typ t) n builder
      in StringMap.add n local_var m in

    let local_vars =
      List.fold_left2 (add_arg builder) StringMap.empty sfdecl.sargs
          (Array.to_list (L.params the_function)) in

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

    let funcs' = List.map (fun (ty, name) ->
                            match ty with
                            | A.Fun(ret_t, args_t) ->
                               (ty, {styp = ret_t; sfname = name; sargs = List.map (fun t -> (t, "x")) args_t; sbody = []})
                            | _ -> raise A.Unsupported_constructor)
                         (List.filter (fun (ty, _) -> match ty with A.Fun(_) -> true | _ -> false) sfdecl.sargs) in

    let add_local_fdecl vars fdecls (t, n) =
      match t with
      | A.Fun(ret_t, args_t) ->
        StringMap.add n (lookup vars n, {styp = ret_t; sfname = n; sargs = List.map (fun t -> (t, "x")) args_t; sbody = []}) fdecls
      | _ -> raise A.Unsupported_constructor in

    let local_fdecls =
      List.fold_left (fun m (_, sfdecl) ->
                        StringMap.add sfdecl.sfname (lookup local_vars sfdecl.sfname, sfdecl) m) StringMap.empty funcs' in
    let lookup_func fdecls n = try StringMap.find n fdecls
                              with Not_found -> StringMap.find n global_fdecls
    in

    let rec expr fdecls vars builder ((ty,e) : sexpr) = match e with
      | SStringlit s -> L.build_global_stringptr s "str" builder
      | SIntlit i -> L.const_int i32_t i
      | SBoollit b -> L.const_int i1_t (if b then 1 else 0)
      | SVar s -> L.build_load (lookup vars s) s builder
      | SUnop(op, e) ->
          let e' = expr fdecls vars builder e in
          (match op with
            A.Neg     -> L.build_neg
          | A.Not     -> L.build_not) e' "tmp" builder
      | SBinop ((e1_t, e1), op, e2) ->
          let e1' = expr fdecls vars builder (e1_t, e1)
          and e2' = expr fdecls vars builder e2 in
          (match e1_t with
          | A.String ->
            let e1' = L.build_call strcmp_func [| e1'; e2' |] "strcmp" builder in
            let e2' = L.const_int i32_t 0 in
            (match op with
            | A.Eq      -> L.build_icmp L.Icmp.Eq
            | A.Neq     -> L.build_icmp L.Icmp.Ne
            | A.Lt      -> L.build_icmp L.Icmp.Slt
            | A.Leq     -> L.build_icmp L.Icmp.Sle
            | A.Gt      -> L.build_icmp L.Icmp.Sgt
            | A.Geq     -> L.build_icmp L.Icmp.Sge
            | _         -> raise A.Unsupported_constructor)
            e1' e2' "tmp" builder
          | _ ->
            (match op with
              A.Add     -> L.build_add
            | A.Sub     -> L.build_sub
            | A.Mul     -> L.build_mul
            | A.Div     -> L.build_sdiv
            | A.And     -> L.build_and
            | A.Or      -> L.build_or
            | A.Eq      -> L.build_icmp L.Icmp.Eq
            | A.Neq     -> L.build_icmp L.Icmp.Ne
            | A.Lt      -> L.build_icmp L.Icmp.Slt
            | A.Leq     -> L.build_icmp L.Icmp.Sle
            | A.Gt      -> L.build_icmp L.Icmp.Sgt
            | A.Geq     -> L.build_icmp L.Icmp.Sge
            ) e1' e2' "tmp" builder)
      | SFunsig (t, bl, _) ->
        let t_list = List.map fst bl in
        let new_fun_t = L.function_type (ltype_of_typ t) (Array.of_list (List.map ltype_of_typ t_list)) in
        L.define_function "temp" new_fun_t the_module
      | SFCall ("print", [e]) ->
         L.build_call print_func [| str_format_str ; ( expr fdecls vars builder e ) |] "" builder
      | SFCall ("print_int", [e]) | SFCall ("print_bool", [e]) ->
         L.build_call print_func [| int_format_str ; ( expr fdecls vars builder e ) |] "" builder
      | SFCall (f, act) ->
         let (fdef, sfdecl) = lookup_func fdecls f in
         let actuals = List.rev (List.map (expr fdecls vars builder) (List.rev act))
```

```
           in
       let result = (match sfdecl.styp with A.Void -> "" | _ -> f ^ "_result") in
       L.build_call fdef (Array.of_list actuals) result builder
    | SMCall (e, s, args) ->
       handle_mcall_expr fdecls vars builder ty e args s
    | SAsn (s, (t, v)) ->
       (* If e is SNull, change to default value for type s *)
       let v = match v with
       | SNull -> (match t with
         | A.Int -> SIntlit 0
         | A.Bool -> SBoollit false
         | A.String -> SStringlit ""
         | _ -> raise A.Unsupported_constructor)
       | _ -> v
       in
       let e' = expr fdecls vars builder (t, v) in
       (match t with
       | A.Fun(_) -> ignore (L.build_store e' (lookup vars s) builder); e'
       | _ -> ignore (L.build_store e' (lookup vars s) builder); e')
    | SGraphExpr(nlist, elist) ->
       let g = L.build_call create_graph_func [||] "create_graph" builder in
       ignore (List.map (fun n -> L.build_call add_node_func [| g; expr fdecls vars builder n |] "add_node" builder) nlist);
       ignore (List.map (fun e -> let f, src', dst' =
                           match e with
                           | (_, SEdgeExpr(src, dst, _)) ->
                              (match src with
                               | (A.Int, _) -> add_edge_int_func
                               | (A.Bool, _) -> add_edge_bool_func
                               | (A.String, _) -> add_edge_str_func
                               | _ -> raise A.Unsupported_constructor),
                              expr fdecls vars builder src,
                              expr fdecls vars builder dst
                           | _ -> raise A.Unsupported_constructor
                           in L.build_call f [| g; expr fdecls vars builder e; src'; dst' |] "add_edge" builder) elist);
       g
    | SEdgeExpr(_, _, w) ->
       let e = L.build_call create_edge_func [||] "edge" builder in
       let w' = expr fdecls vars builder w in
       (match w with
       | (A.Int, SNull) -> ignore (L.build_call set_edge_w_int_func [| e; w'; L.const_int i1_t 0 |] "" builder)
       | (A.Int, _) -> ignore (L.build_call set_edge_w_int_func [| e; w'; L.const_int i1_t 1 |] "" builder)
       | (A.Bool, SNull) -> ignore (L.build_call set_edge_w_bool_func [| e; w'; L.const_int i1_t 0 |] "" builder)
       | (A.Bool, _) -> ignore (L.build_call set_edge_w_bool_func [| e; w'; L.const_int i1_t 1 |] "" builder)
       | (A.String, SNull) -> ignore (L.build_call set_edge_w_str_func [| e; w'; L.const_int i1_t 0 |] "" builder)
       | (A.String, _) -> ignore (L.build_call set_edge_w_str_func [| e; w'; L.const_int i1_t 1 |] "" builder)
       | _ -> raise A.Unsupported_constructor);
       e
    | SNodeExpr (l, d) ->
       let l' = expr fdecls vars builder l in
       let d' = expr fdecls vars builder d in
       let n = L.build_call create_node_func [||] "create_node" builder in
       (match l with
        | (A.Int, _) -> ignore (L.build_call set_node_label_int_func [| n; l' |]
           "" builder)
        | (A.Bool, _) -> ignore (L.build_call set_node_label_bool_func [| n; l’ |] "" builder)
        | (A.String, _) -> ignore (L.build_call set_node_label_str_func [| n; l’ |] "" builder)
        | _ -> raise A.Unsupported_constructor);
       (match d with
        | (A.Int, v) ->
          if v = SNull
          then ignore (L.build_call set_node_data_int_func [| n; L.const_int i32_t 0; L.const_int i1_t 0 |] "" builder)
          else ignore (L.build_call set_node_data_int_func [| n; d’; L.const_int i1_t 1 |] "" builder)
        | (A.Bool, v) ->
          if v = SNull
          then ignore (L.build_call set_node_data_bool_func [| n; L.const_int i1_t 0; L.const_int i1_t 0 |] "" builder)
          else ignore (L.build_call set_node_data_bool_func [| n; d’; L.const_int i1_t 1 |] "" builder)
        | (A.String, v) ->
          if v = SNull
          then ignore (L.build_call set_node_data_str_func [| n; L.const_null str_t; L.const_int i1_t 0 |] "" builder)
          else ignore (L.build_call set_node_data_str_func [| n; d’; L.const_int i1_t 1 |] "" builder)
        | _ -> raise A.Unsupported_constructor);
       n
    | SNull ->
       (match ty with
        | A.Int -> L.const_null i32_t
        | A.Bool -> L.const_null i1_t
        | A.String -> L.const_null str_t
        | _ -> L.const_null void_ptr_t)
    | SNoexpr ->
       L.undef (L.void_type context) (* placeholder *)

  and handle_mcall_expr fdecls vars builder ty e args = function
  | "set_node" ->
       (match args with
        | ((A.Node(_), _) as n) :: [] ->
          let g_ptr = expr fdecls vars builder e in
          let n_ptr = expr fdecls vars builder n in
          let n_ptr’ = L.build_call clone_node_func [| n_ptr |] "clone_node" builder in
          L.build_call graph_set_node_func [| g_ptr; n_ptr’ |] "tmp_data" builder
        | _ -> raise A.Unsupported_constructor)
  | "remove_node" ->
       (match args with
        | ((l_typ, _) as l) :: [] ->
          let g_ptr = expr fdecls vars builder e in
          let l’ = expr fdecls vars builder l in
          (match l_typ with
           | A.Int -> L.build_call remove_node_int_func [| g_ptr; l’ |] "tmp_data" builder
           | A.String -> L.build_call remove_node_str_func [| g_ptr; l’ |] "tmp_data" builder
           | A.Bool -> L.build_call remove_node_bool_func [| g_ptr; l’ |] "tmp_data" builder
           | _ -> raise A.Unsupported_constructor)
        | _ -> raise A.Unsupported_constructor)
  | "remove_edge" ->
```

```
         (match args with
          | ((src_typ, _) as src) :: dst :: [] ->
             let g_ptr = expr fdecls vars builder e in
             let src_ptr = expr fdecls vars builder src in
             let dst_ptr = expr fdecls vars builder dst in
             let e_ptr = (match src_typ with
               | A.Int -> L.build_call get_edge_by_src_and_dst_int_func [| g_ptr; src_ptr; dst_ptr |] "get_edge_by_src_and_dst_int" builder
               | A.Bool -> L.build_call get_edge_by_src_and_dst_bool_func [| g_ptr; src_ptr; dst_ptr |] "get_edge_by_src_and_dst_bool" builder
               | A.String -> L.build_call get_edge_by_src_and_dst_str_func [| g_ptr; src_ptr; dst_ptr |] "get_edge_by_src_and_dst_str" builder
               | _ -> raise A.Unsupported_constructor) in
             L.build_call remove_edge_func [| g_ptr; e_ptr |] "remove_edge" builder
          | _ -> raise A.Unsupported_constructor)
    | "get_node" ->
         (match e, args with
          | (A.Graph(lt, _, _), _), label :: [] ->
            let g_ptr = expr fdecls vars builder e in
            let label' = expr fdecls vars builder label in
            (match lt with
             | A.Int -> L.build_call get_node_by_label_int_opt_func [| g_ptr; label' |] "get_node_by_label" builder
             | A.Bool -> L.build_call get_node_by_label_bool_opt_func [| g_ptr; label' |] "get_node_by_label" builder
             | A.String -> L.build_call get_node_by_label_str_opt_func [| g_ptr; label' |] "get_node_by_label" builder
             | _ -> raise A.Unsupported_constructor)
          | _ -> raise A.Unsupported_constructor)
    | "get_weight" ->
         (match e, args with
          | (A.Graph(lt, _, wt), _), src :: dst :: [] ->
             let g_ptr = expr fdecls vars builder e in
             let src' = expr fdecls vars builder src in
             let dst' = expr fdecls vars builder dst in
             let e_ptr = (match lt with
               | A.Int -> L.build_call get_edge_by_src_and_dst_int_func [| g_ptr; src'; dst' |] "get_edge_by_src_and_dst_int" builder
               | A.Bool -> L.build_call get_edge_by_src_and_dst_bool_func [| g_ptr; src'; dst' |] "get_edge_by_src_and_dst_bool" builder
               | A.String -> L.build_call get_edge_by_src_and_dst_str_func [| g_ptr; src'; dst' |] "get_edge_by_src_and_dst_str" builder
               | _ -> raise A.Unsupported_constructor) in
             (match wt with
               | A.Int -> L.build_call get_edge_w_int_func [| e_ptr |] "get_edge_w" builder
               | A.Bool -> L.build_call get_edge_w_bool_func [| e_ptr |] "get_edge_w" builder
               | A.String -> L.build_call get_edge_w_str_func [| e_ptr |] "get_edge_w" builder
               | _ -> raise A.Unsupported_constructor)
          | _ -> raise A.Unsupported_constructor)
    | "get_name" ->
         let n_ptr = expr fdecls vars builder e in
         let ret = L.build_call get_node_label_func [| n_ptr |] "tmp_data" builder in
         (match ty with
         | A.String -> ret
         | A.Int -> L.build_load (L.build_bitcast ret i32_ptr_t "bitcast" builder) "deref" builder
         | A.Bool -> L.build_load (L.build_bitcast ret i32_ptr_t "bitcast" builder)
```

```
            "deref" builder
         | _ -> raise A.Unsupported_constructor)
    | "has_node" ->
         (match args with
          | ((n_typ, _) as n) :: [] ->
            let g_ptr = expr fdecls vars builder e in
            let n' = expr fdecls vars builder n in
            (match n_typ with
             | A.Int -> L.build_call graph_has_node_int_func [| g_ptr; n' |] "tmp_data" builder
             | A.String -> L.build_call graph_has_node_str_func [| g_ptr; n' |] "tmp_data" builder
             | A.Bool -> L.build_call graph_has_node_bool_func [| g_ptr; n' |] "tmp_data" builder
             | _ -> raise A.Unsupported_constructor)
          | _ -> raise A.Unsupported_constructor)
    | "get_data" ->
         let n_ptr = expr fdecls vars builder e in
         let ret = L.build_call get_node_data_func [| n_ptr |] "tmp_data" builder in
         (match ty with
         | A.String -> ret
         | A.Bool -> L.build_load (L.build_bitcast ret i32_ptr_t "bitcast" builder) "deref" builder
         | A.Int -> L.build_load (L.build_bitcast ret i32_ptr_t "bitcast" builder) "deref" builder
         | _ -> raise A.Unsupported_constructor)
    | "set_edge" ->
         (match args with
          | ((src_typ, _) as src) :: dst :: ((w_typ, _) as w) :: [] ->
             let g_ptr = expr fdecls vars builder e in
             let src' = expr fdecls vars builder src in
             let dst' = expr fdecls vars builder dst in
             let w' = expr fdecls vars builder w in
             (match (src_typ, w_typ) with
             | (A.Int, A.Bool) -> L.build_call graph_set_edge_int_bool_func [| g_ptr; src'; dst'; w' |] "tmp_data" builder
             | (A.Int, A.Int) -> L.build_call graph_set_edge_int_int_func [| g_ptr; src'; dst'; w' |] "tmp_data" builder
             | (A.Bool, A.Bool) -> L.build_call graph_set_edge_bool_bool_func [| g_ptr; src'; dst'; w' |] "tmp_data" builder
             | (A.Bool, A.Int) -> L.build_call graph_set_edge_bool_int_func [| g_ptr; src'; dst'; w' |] "tmp_data" builder
             | (A.String, A.Bool) -> L.build_call graph_set_edge_str_bool_func [| g_ptr; src'; dst'; w' |] "tmp_data" builder
             | (A.Bool, A.String) -> L.build_call graph_set_edge_bool_str_func [| g_ptr; src'; dst'; w' |] "tmp_data" builder
             | (A.String, A.Int) -> L.build_call graph_set_edge_str_int_func [| g_ptr; src'; dst'; w' |] "tmp_data" builder
             | (A.Int, A.String) -> L.build_call graph_set_edge_int_str_func [| g_ptr; src'; dst'; w' |] "tmp_data" builder
             | (A.String, A.String) -> L.build_call graph_set_edge_str_str_func [| g_ptr; src'; dst'; w' |] "tmp_data" builder
             | _ -> raise A.Unsupported_constructor)
          | ((src_typ, _) as src) :: dst :: [] ->
             let g_ptr = expr fdecls vars builder e in
             let src' = expr fdecls vars builder src in
             let dst' = expr fdecls vars builder dst in
             (match src_typ with
              | A.Int -> L.build_call graph_set_edge_int_func [| g_ptr; src'; dst' |] "tmp_data" builder
              | A.String -> L.build_call graph_set_edge_str_func [| g_ptr; src'; dst
```

```
                  ' |] "tmp_data" builder
              | A.Bool -> L.build_call graph_set_edge_bool_func [| g_ptr; src'; dst' |] "tmp_data" builder
              | _ -> raise A.Unsupported_constructor)
          | _ -> raise A.Unsupported_constructor)
    | "set_data" ->
         (match args with
          | ((dt, dv) as d) :: [] ->
             let n_ptr = expr fdecls vars builder e in
             let d_ptr = expr fdecls vars builder d in
             (match dt with
              | A.Int ->
                 if dv = SNull
                 then L.build_call set_node_data_int_func [| n_ptr; L.const_int i32_t 0; L.const_int i1_t 0 |] "" builder
                 else L.build_call set_node_data_int_func [| n_ptr; d_ptr; L.const_int i1_t 1 |] "" builder
              | A.Bool ->
                 if dv = SNull
                 then L.build_call set_node_data_bool_func [| n_ptr; L.const_int i1_t 0; L.const_int i1_t 0 |] "" builder
                 else L.build_call set_node_data_bool_func [| n_ptr; d_ptr; L.const_int i1_t 1 |] "" builder
              | A.String ->
                 if dv = SNull
                 then L.build_call set_node_data_str_func [| n_ptr; L.const_null str_t; L.const_int i1_t 0 |] "" builder
                 else L.build_call set_node_data_str_func [| n_ptr; d_ptr; L.const_int i1_t 1 |] "" builder
              | _ -> raise A.Unsupported_constructor)
          | _ -> raise A.Unsupported_constructor)
    | "are_neighbors" ->
         (match e, args with
          | (A.Graph(lt, _, _), _), src :: dst :: [] ->
             let g_ptr = expr fdecls vars builder e in
             let src' = expr fdecls vars builder src in
             let dst' = expr fdecls vars builder dst in
             (match lt with
              | A.Int ->
                L.build_call are_neighbors_int_func [| g_ptr; src'; dst' |] "are_neighbors" builder
              | A.Bool ->
                L.build_call are_neighbors_bool_func [| g_ptr; src'; dst' |] "are_neighbors" builder
              | A.String ->
                L.build_call are_neighbors_str_func [| g_ptr; src'; dst' |] "are_neighbors" builder
              | _ -> raise A.Unsupported_constructor)
          | _ -> raise A.Unsupported_constructor)
    | "is_empty" ->
         (match e with
          | (A.Graph(_), _) ->
             let g_ptr = expr fdecls vars builder e in
             L.build_call is_empty_func [| g_ptr |] "is_empty" builder
          | _ -> raise A.Unsupported_constructor)
    | "print" ->
         (match e with
          | (A.Graph(_), _) ->
             let g_ptr = expr fdecls vars builder e in
             L.build_call print_graph_func [| g_ptr |] "" builder
          | (A.Node(_), _) ->
```

```
             let n_ptr = expr fdecls vars builder e in
             L.build_call print_node_func [| n_ptr |] "" builder
          | _ -> raise A.Unsupported_constructor)
    | "neighbors" ->
         (match e, args with
          | (A.Graph(_), _), ((nlt, _) as nl) :: [] ->
             let g_ptr = expr fdecls vars builder e in
             let nl' = expr fdecls vars builder nl in
             let n_ptr = (match nlt with
                         | A.Int | A.Bool -> L.build_call get_node_by_label_int_func [| g_ptr; nl' |] "get_node_by_label_int" builder
                         | A.String -> L.build_call get_node_by_label_str_func [| g_ptr; nl' |] "get_node_by_label_str" builder
                         | _ -> raise A.Unsupported_constructor) in
             L.build_call neighbors_one_arg_func [| n_ptr |] "neihghbors_one_arg" builder
          | (A.Graph(_), _), ((nlt, _) as nl) :: level :: include_current :: [] ->
             let g_ptr = expr fdecls vars builder e in
             let nl' = expr fdecls vars builder nl in
             let level' = expr fdecls vars builder level in
             let include_current' = expr fdecls vars builder include_current in
             let n_ptr = (match nlt with
                         | A.Int | A.Bool -> L.build_call get_node_by_label_int_func [| g_ptr; nl' |] "get_node_by_label_int" builder
                         | A.String -> L.build_call get_node_by_label_str_func [| g_ptr; nl' |] "get_node_by_label_str" builder
                         | _ -> raise A.Unsupported_constructor) in
             L.build_call neighbors_func [| n_ptr; level'; include_current' |] "neighbors" builder
          | _ -> raise A.Unsupported_constructor)
    | "find" ->
         (match e, args with
          | (A.Graph(_, dt, _), _), d :: [] ->
            let g_ptr = expr fdecls vars builder e in
            let d' = expr fdecls vars builder d in
            (match dt with
             | A.Int -> L.build_call find_data_int_func [| g_ptr; d' |] "find_data" builder
             | A.Bool -> L.build_call find_data_bool_func [| g_ptr; d' |] "find_data" builder
             | A.String -> L.build_call find_data_str_func [| g_ptr; d' |] "find_data" builder
             | _ -> raise A.Unsupported_constructor)
          | _ -> raise A.Unsupported_constructor)
    | "dfs" ->
         (match e, args with
          | (A.Graph(lt, _, _), _), l :: [] ->
            let g_ptr = expr fdecls vars builder e in
            let l' = expr fdecls vars builder l in
            (match lt with
             | A.Int -> L.build_call dfs_int_func [|g_ptr; l'|] "dfs_int" builder
             | A.Bool -> L.build_call dfs_int_func [|g_ptr; l'|] "dfs_int" builder
             | A.String -> L.build_call dfs_str_func [|g_ptr; l'|] "dfs_str" builder
             | _ -> raise A.Unsupported_constructor)
          | _ -> raise A.Unsupported_constructor)
    | "bfs" ->
         (match e, args with
          | (A.Graph(lt, _, _), _), l :: [] ->
            let g_ptr = expr fdecls vars builder e in
            let l' = expr fdecls vars builder l in
            (match lt with
             | A.Int -> L.build_call bfs_int_func [|g_ptr; l'|] "bfs_int" builder
             | A.Bool -> L.build_call bfs_int_func [|g_ptr; l'|] "bfs_int" builder
             | A.String -> L.build_call bfs_str_func [|g_ptr; l'|] "bfs_str" builder
             | _ -> raise A.Unsupported_constructor)
          | _ -> raise A.Unsupported_constructor)
    | _ -> raise A.Unsupported_constructor
    in

    let add_terminal builder instr =
      (* The current block where we're inserting instr *)
      match L.block_terminator (L.insertion_block builder) with
      | Some _ -> ()
      | None -> ignore (instr builder)
    in

    let rec stmt (fdecls, vars, builder) = function
      | SBlock sl ->
        List.fold_left stmt (fdecls, vars, builder) sl
      (* Generate code for this expression, return resulting builder *)
      | SExpr e ->
        let _ = expr fdecls vars builder e in (fdecls, vars, builder)
      (* fun f = ... (...) (...) *)
      | SVdecl (ty, s, e) ->
        (match e with
        | (A.Fun(_), SAsn(var_name, (A.Fun(_), SFunsig(t, bl, e')))) ->
          (* Make the function's signature*)
          let sfdecl = {styp = t; sfname = var_name; sargs = bl; sbody = [SReturn(e')]} in
          (* Get the function's llvalue*)
          let vars' = add_local_var builder vars (ty, s) in
          let ll_fun_val = expr fdecls vars' builder e in
          let fdecls' = StringMap.add var_name (ll_fun_val, sfdecl) fdecls in
          let builder' = L.builder_at_end context (L.entry_block ll_fun_val) in
          let new_locals = List.fold_left2 (add_arg builder') StringMap.empty sfdecl.sargs (Array.to_list (L.params ll_fun_val)) in
          let (_, _, builder'') = stmt (fdecls', new_locals, builder') (SBlock sfdecl.sbody) in

          (add_terminal builder'' (match sfdecl.styp with
                            A.Void -> L.build_ret_void
                          | t -> L.build_ret (L.const_int (ltype_of_typ t) 0)));

          (fdecls', vars', builder)
        | _ ->
          let vars' = add_local_var builder vars (ty, s) in
          let fdecls' = (match ty with A.Fun(_) -> add_local_fdecl vars' fdecls (ty, s) | _ -> fdecls) in
          let _ = expr fdecls vars' builder e in (fdecls', vars', builder))

      | SReturn e ->
        let _ = match sfdecl.styp with
                (* Special "return nothing" instr *)
                | A.Void -> L.build_ret_void builder
                (* Build return statement *)
                | _ -> L.build_ret (expr fdecls vars builder e) builder
        in (fdecls, vars, builder)
      | SIf (p, then_stmt, else_stmt) ->
        let bool_val = expr fdecls vars builder p in
        let merge_bb = L.append_block context "merge" the_function in

        let then_bb = L.append_block context "then" the_function in
```

```
        let _, _, builder' = stmt (fdecls, vars, L.builder_at_end context then_bb) then_stmt in
        add_terminal builder' (L.build_br merge_bb);

        let else_bb = L.append_block context "else" the_function in
        let _, _, builder' = stmt (fdecls, vars, L.builder_at_end context else_bb) else_stmt in
        add_terminal builder' (L.build_br merge_bb);

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

      | SWhile (p, body) ->
        let p_bb = L.append_block context "while" the_function in
        ignore (L.build_br p_bb builder);

        let body_bb = L.append_block context "while_body" the_function in
        let _, _, builder' = stmt (fdecls, vars, L.builder_at_end context body_bb) body in
        add_terminal builder' (L.build_br p_bb);

        let p_builder = L.builder_at_end context p_bb in
        let bool_val = expr fdecls vars p_builder p in

        let merge_bb = L.append_block context "merge" the_function in
        ignore (L.build_cond_br bool_val body_bb merge_bb p_builder);
        (fdecls, vars, L.builder_at_end context merge_bb)

      | SFor (e1, p, e2, body) -> stmt (fdecls, vars, builder)
         (SBlock [SExpr e1 ; SWhile (p, SBlock [body ; SExpr e2]) ] )
      | SForNode (n, g, body) ->
        (match g with
         | (A.Graph(lt, dt, _), _) ->
           let graph_ptr = expr fdecls vars builder g in

           (* allocate space for n, add to symbol table, and initially set to head of node linked list *)
           let n_ptr = L.build_alloca (ltype_of_typ (A.Node(lt, dt))) n builder in
           let vars = StringMap.add n n_ptr vars in
           let hd_node = L.build_call graph_to_node_iterable_func [| graph_ptr |] "hd_node" builder in
           ignore(L.build_store hd_node n_ptr builder);

           (* create predicate block *)
           let p_bb = L.append_block context "while" the_function in
           ignore (L.build_br p_bb builder);

           (* while body block *)
           let body_bb = L.append_block context "while_body" the_function in
           let body_builder = L.builder_at_end context body_bb in
           let _, _, builder' = stmt (fdecls, vars, body_builder) body in
           (* change curr_node to be pointer to next node *)
           let curr_node = L.build_load n_ptr "curr_node" builder' in
           let next_node = L.build_call get_graph_next_node_func [| curr_node |] "next_node" builder' in
           ignore(L.build_store next_node n_ptr builder');
           add_terminal builder' (L.build_br p_bb);

           (* define predicate *)
           let p_builder = L.builder_at_end context p_bb in
           let n_val = L.build_load n_ptr "node_tmp" p_builder in
```

```
             let bool_val = L.build_is_not_null n_val "bool_val" p_builder in

             (* merge *)
             let merge_bb = L.append_block context "merge" the_function in
             ignore (L.build_cond_br bool_val body_bb merge_bb p_builder);
             (fdecls, vars, L.builder_at_end context merge_bb)
           | _ -> raise A.Unsupported_constructor)
        | SForEdge (src, dst, w, g, body) ->
          (match g with
           | (A.Graph(lt, dt, wt), _) ->
             let graph_ptr = expr fdecls vars builder g in

             (* allocate space for edge variables, add to symbol table, and initially set to head of edge linked list *)
             let edge_ptr = L.build_alloca void_ptr_t "edge" builder in
             let src_ptr = L.build_alloca (ltype_of_typ (A.Node(lt, dt))) "src" builder in
             let dst_ptr = L.build_alloca (ltype_of_typ (A.Node(lt, dt))) "dst" builder in
             let w_ptr = L.build_alloca (ltype_of_typ wt) "w" builder in
             let vars = StringMap.add src src_ptr (StringMap.add dst dst_ptr (StringMap.add w w_ptr vars)) in
             let hd_edge = L.build_call graph_to_edge_iterable_func [| graph_ptr |] "hd_edge" builder in
             let hd_edge_src = L.build_call get_edge_src_func [| hd_edge |] "hd_edge_src" builder in
             let hd_edge_dst = L.build_call get_edge_dst_func [| hd_edge |] "hd_edge_dst" builder in
             let hd_edge_w = (match wt with
              | A.Int -> L.build_call get_edge_w_int_func [| hd_edge |] "hd_edge_w" builder
              | A.Bool -> L.build_call get_edge_w_bool_func [| hd_edge |] "hd_edge_w" builder
              | A.String -> L.build_call get_edge_w_str_func [| hd_edge |] "hd_edge_w" builder
              | _ -> raise A.Unsupported_constructor) in
             ignore(L.build_store hd_edge edge_ptr builder);
             ignore(L.build_store hd_edge_src src_ptr builder);
             ignore(L.build_store hd_edge_dst dst_ptr builder);
             ignore(L.build_store hd_edge_w w_ptr builder);

             (* create predicate block *)
             let p_bb = L.append_block context "while" the_function in
             ignore (L.build_br p_bb builder);

             (* while body block *)
             let body_bb = L.append_block context "while_body" the_function in
             let body_builder = L.builder_at_end context body_bb in
             let _, _, builder' = stmt (fdecls, vars, body_builder) body in
             (* change curr_edge to be pointer to next edge *)
             let curr_edge = L.build_load edge_ptr "curr_edge" builder' in
             let next_edge = L.build_call get_graph_next_edge_func [| curr_edge |] "next_edge" builder' in
             let next_edge_src = L.build_call get_edge_src_func [| next_edge |] "next_edge_src" builder' in
             let next_edge_dst = L.build_call get_edge_dst_func [| next_edge |] "next_edge_dst" builder' in
             let next_edge_w = (match wt with
              | A.Int -> L.build_call get_edge_w_int_func [| next_edge |] "next_edge_w" builder'
              | A.Bool -> L.build_call get_edge_w_bool_func [| next_edge |] "
```

```
next_edge_w" builder'
            | A.String -> L.build_call get_edge_w_str_func [| next_edge |] "next_edge_w" builder'
            | _ -> raise A.Unsupported_constructor) in
           ignore(L.build_store next_edge edge_ptr builder');
           ignore(L.build_store next_edge_src src_ptr builder');
           ignore(L.build_store next_edge_dst dst_ptr builder');
           ignore(L.build_store next_edge_w w_ptr builder');
           add_terminal builder' (L.build_br p_bb);

           (* define predicate *)
           let p_builder = L.builder_at_end context p_bb in
           let e_val = L.build_load edge_ptr "edge_tmp" p_builder in
           let bool_val = L.build_is_not_null e_val "bool_val" p_builder in

           (* merge *)
           let merge_bb = L.append_block context "merge" the_function in
           ignore (L.build_cond_br bool_val body_bb merge_bb p_builder);
           (fdecls, vars, L.builder_at_end context merge_bb)
         | _ -> raise A.Unsupported_constructor)
    in

    let (_, _, builder) = stmt (local_fdecls, local_vars, builder) (SBlock sfdecl.sbody) in

    add_terminal builder (match sfdecl.styp with
                            A.Void -> L.build_ret_void
                          | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in
  List.iter (build_function_body StringMap.empty) functions;
  the_module
```

9.6.2 hippograph.ml

```

type action = Ast | Sast | LLVM_IR | Compile

let _ =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./microc.native [-a|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Scanner.token lexbuf in
  match !action with
  | Ast -> print_string (Ast.string_of_program ast)
  | _ ->
    let sast = Semant.check ast in
    match !action with
      Ast -> ()
    | Sast -> print_string (Sast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
    | Compile -> let m = Codegen.translate sast in
```

```
      Llvm_analysis.assert_valid_module m;
      print_string (Llvm.string_of_llmodule m)
```

9.6.3 Makefile

```
hippograph.native:
  opam config exec -- \
  ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \
    hippograph.native


.PHONY : test
test : all testall.sh
  ./testall.sh

.PHONY : all
all : hippograph.native graph.o

.PHONY: clean
clean:
  ocamlbuild -clean
  rm -rf ocamlllvm *.diff

redo:
  ocamlbuild -clean
  rm -rf ocamlllvm *.diff
  make hippograph.native

FILE=""
run: all run.sh
  ./run.sh $(FILE)
```

9.6.4 run.sh

```
#!/bin/sh

# script for running one test

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

# Path to the LLVM compiler
LLC="/usr/local/opt/llvm/bin/llc"

# Path to the C compiler
CC="cc"

# Path to the hippograph compiler.  Usually "./hippograph.native"
# Try "_build/hippograph.native" if ocamlbuild was unable to create a symbolic link.
HIPPOGRAPH="./hippograph.native"
#HIPPOGRAPH="_build/hippograph.native"

HPG=".hpg"
BASENAME="${1%$HPG}"

if [ "$#" -ne 1 ]; then
    echo "usage: ./run.sh filename.hpg"
  exit
fi

```

```
generatedfiles="$generatedfiles $BASENAME.ll $BASENAME.s $BASENAME.exe $BASENAME.out" &&
"$HIPPOGRAPH" "$1" > "$BASENAME.ll"
"$LLC" "-relocation-model=pic" "$BASENAME.ll" > "$BASENAME.s"
"$CC" "-o" "$BASENAME.exe" "$BASENAME.s" "graph.o"
"./$BASENAME.exe"
rm -f $generatedfiles
```

9.6.5 testall.sh

```
#!/bin/sh

# Regression testing script for Hippograph
# 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 LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the hippograph compiler.  Usually "./hippograph.native"
# Try "_build/hippograph.native" if ocamlbuild was unable to create a symbolic link.
HIPPOGRAPH="./hippograph.native"
#HIPPOGRAPH="_build/hippograph.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] [.hpg 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/.hpg//'`
    reffile=`echo $1 | sed 's/.hpg$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe ${basename}.out" &&
    Run "$HIPPOGRAPH" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "graph.o" &&
    Run "./${basename}.exe" > "${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/.hpg//'`
    reffile=`echo $1 | sed 's/.hpg$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$HIPPOGRAPH" "<" $1 "2>" "${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 [ ! -f graph.o ]
then
    echo "Could not find graph.o"
    echo "Try \"make graph.o\""
    exit 1
```

```
fi

if [ $# -ge 1 ]
then
    files=$@
else
    files="test/test-*.hpg test/fail-*.hpg"
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
```

9.7 C library

9.7.1 graph.c

```
/* Authors:
  Benjamin Lewinter bsl2121
  Irina Mateescu    im2441
  Harry Smith       hs3061
  Yasunari Watanabe yw3239
*/

#include <string.h>
#include <stdlib.h>
#include <stdio.h>

/* constants */

int VOIDTYPE = 1;
int INTTYPE  = 2;
int STRTYPE  = 3;
int BOOLTYPE = 4;

/* data structures */

typedef union primitive {
  int *i;
  char *s;
  void *v;
} primitive;

typedef struct node node;

typedef struct edge {
  node *src;
  node *dst;
  primitive *w;
  int w_typ;
  struct edge *next;
  int has_val;
} edge;

typedef struct neighbor_list_item {
  edge *edge;
  struct neighbor_list_item *next;
} neighbor_list_item;

typedef struct neighbor_list {
  neighbor_list_item *hd;
} neighbor_list;

struct node {
  primitive *label;
  int label_typ;
  primitive *data;
  int data_typ;
  int has_val;
  neighbor_list *neighbor_list;
  node *next;
};

typedef struct node_list {
  node *hd;
} node_list;

typedef struct edge_list {
  edge *hd;
} edge_list;

typedef struct graph {
  node_list *node_list;
  edge_list *edge_list;
} graph;

typedef struct q_item {
  node *n;
  struct q_item *next;
} q_item;

typedef struct queue {
  q_item *hd;
  q_item *tl;
} queue;

/* create primitive of type */

void *create_prim_int(int i) {
  primitive *p = (primitive *) malloc(sizeof(primitive));
  p -> i = (int *) malloc(sizeof(int));
  *(p -> i) = i;
  return (void *) p;
}

void *create_prim_str(char *s) {
  primitive *p = (primitive *) malloc(sizeof(primitive));
  p -> s = s;
  return (void *) p;
}

primitive *clone_primitive(primitive *p) {
  if (p == NULL) return NULL;

  primitive *p_cp = (primitive *) malloc(sizeof(primitive));
  memcpy(p_cp, p, sizeof(primitive));
  return p_cp;
}

/* NODES */

void *create_neighbor_list_item(edge *e) {
  neighbor_list_item *nli = (neighbor_list_item *) malloc(sizeof(neighbor_list_item));
  nli -> edge = e;
  nli -> next = NULL;
  return (void *) nli;
}

void *create_neighbor_list() {
  neighbor_list *nl = (neighbor_list *) malloc(sizeof(neighbor_list));
  nl -> hd = NULL;
  return (void *) nl;
}

void *create_node() {
  node *n = (node *) malloc(sizeof(node));
  n -> label = NULL;
  n -> data = 0;
  n -> has_val = 0;
  n -> neighbor_list = create_neighbor_list();
  n -> neighbor_list -> hd = NULL;
  n -> next = NULL;
  return (void *) n;
}

int cmp_node_label(node *n1, node *n2) {
  // return 0 if equal
  int lt = n1 -> label_typ;
  if (lt == INTTYPE || lt == BOOLTYPE) {
    if (*(n1 -> label -> i) == *(n2 -> label -> i)) return 0;
    else return -1;
  } else if (lt == STRTYPE) {
    return strcmp(n1 -> label -> s, n2 -> label -> s);
  } else {
    return -1;
  }
}

node *clone_node(node *n) {
  if (n == NULL) return NULL;

  node *n_cp = create_node();
  n_cp -> label = clone_primitive(n -> label);
  n_cp -> label_typ = n -> label_typ;
  n_cp -> data = clone_primitive(n -> data);
  n_cp -> data_typ = n -> data_typ;
  n_cp -> has_val = n -> has_val;
  return n_cp;
```

```
}

void set_node_label_int(node *n, int i) {
  if (n -> label != NULL) {
    free(n -> label);
  }
  n -> label = create_prim_int(i);
  n -> label_typ = INTTYPE;
}

void set_node_label_bool(node *n, int i) {
  if (n -> label != NULL) {
    free(n -> label);
  }
  n -> label = create_prim_int(i);
  n -> label_typ = BOOLTYPE;
}

void set_node_label_str(node *n, char *s) {
  if (n -> label != NULL) {
    free(n -> label);
  }
  n -> label = create_prim_str(s);
  n -> label_typ = STRTYPE;
}

void set_node_data_int(node *n, int i, int has_val) {
  if (n -> data != NULL) {
    free(n -> data);
  }
  n -> data = create_prim_int(i);
  n -> data_typ = INTTYPE;
  n -> has_val = has_val; //flag
}

void set_node_data_bool(node *n, int i, int has_val) {
  if (n -> data != NULL) {
    free(n -> data);
  }
  n -> data = create_prim_int(i);
  n -> data_typ = BOOLTYPE;
  n -> has_val = has_val; //flag
}

void set_node_data_str(node *n, char *s, int has_val) {
  if (n -> data != NULL) {
    free(n -> data);
  }
  n -> data = create_prim_str(s);
  n -> data_typ = STRTYPE;
  n -> has_val = has_val;
}

void *get_node_label(node *n) {
  int typ = n -> label_typ;
  void *label = NULL;

  if (typ == INTTYPE || typ == BOOLTYPE) {
    label = (void *) n -> label -> i;
  } else if (typ == STRTYPE) {
    label = (void *) n -> label -> s;
  } else if (typ == VOIDTYPE) {
    label = (void *) n -> label -> v;
  }
  return label;
}

void *get_node_data(node *n) {
  int typ = n -> data_typ;
  void *data = NULL;

  if (typ == INTTYPE || typ == BOOLTYPE) {
    data = (void *) n -> data -> i;
  } else if (typ == STRTYPE) {
    data = (void *) n -> data -> s;
  } else if (typ == VOIDTYPE) {
    data = (void *) n -> data -> v;
  }
  return data;  // not guaranteed to return valid value if not has_val
}

/* EDGES */

int cmp_edge_weight(edge *e1, edge *e2) {
  int lt = e1 -> w_typ;
  if (lt == INTTYPE || lt == BOOLTYPE) {
    return *(e1 -> w -> i) == *(e2 -> w -> i);
  } else if (lt == STRTYPE) {
    return strcmp(e1 -> w -> s, e2 -> w -> s);
  } else {
    return -1;
  }
}

void set_edge_w_int(edge *e, int i, int has_val) {
  if (e -> w != NULL) {
    free(e -> w);
  }
  e -> w = create_prim_int(i);
  e -> has_val = has_val;
  e -> w_typ = INTTYPE;
}

void set_edge_w_bool(edge *e, int i, int has_val) {
  if (e -> w != NULL) {
    free(e -> w);
  }
  e -> w = create_prim_int(i);
  e -> has_val = has_val;
  e -> w_typ = BOOLTYPE;
}

void set_edge_w_str(edge *e, char *s, int has_val) {
  if (e -> w != NULL) {
    free(e -> w);
  }
  e -> w = create_prim_str(s);
  e -> has_val = has_val;
  e -> w_typ = STRTYPE;
}

node *get_edge_src(edge *e) {
  if (e == NULL) return NULL;

  return e -> src;
}

node *get_edge_dst(edge *e) {
  if (e == NULL) return NULL;

  return e -> dst;
}

int get_edge_w_int(edge *e) {
  if (e == NULL || e -> has_val == 0) return 0;
  return *(e -> w -> i);
}

char *get_edge_w_str(edge *e) {
  if (e == NULL || e -> has_val == 0) return "";
  return e -> w -> s;
}

void *create_edge() {
  edge *e = (edge *) malloc(sizeof(edge));
  e -> src = NULL;
  e -> dst = NULL;
  e -> w = NULL;
  e -> next = NULL;
  e -> has_val = 0;
  return e;
}

edge *clone_edge(edge *e) {
  if (e == NULL) return NULL;
  edge *e_cp = create_edge();
  e_cp -> src = clone_node(e -> src);
  e_cp -> dst = clone_node(e -> dst);
  e_cp -> w = clone_primitive(e -> w);
  e_cp -> w_typ = e -> w_typ;
  e_cp -> has_val = e -> has_val;
  e_cp -> next = NULL;
  return e_cp;
}

/* GRAPHS */

void *create_node_list() {
  node_list *nl = (node_list *) malloc(sizeof(node_list));
  nl -> hd = NULL;
  return (void *) nl;
}

void *create_edge_list() {
  edge_list *el = (edge_list *) malloc(sizeof(edge_list));
  el -> hd = NULL;
  return (void *) el;
}

void *create_graph() {
  graph *g = (graph *) malloc(sizeof(graph));
  g -> node_list = create_node_list();
  g -> edge_list = create_edge_list();
  return (void *) g;
}

/*
  Given a graph, creates a linked list of copies of its nodes.
  Used to enable node iteration (for_node) without side effects.
*/
node *graph_to_node_iterable(graph *g) {
  node *curr_orig = g -> node_list -> hd;
  node *curr_new = clone_node(curr_orig);
  node *hd_new = curr_new;

  while (curr_orig != NULL) {
    curr_new -> next = clone_node(curr_orig -> next);
    curr_orig = curr_orig -> next;
    curr_new = curr_new -> next;
  }

  return hd_new;
}

/*
  Given a graph, creates a linked list of copies of its edges.
  Used to enable edge iteration (for_edge) without side effects.
*/
edge *graph_to_edge_iterable(graph *g) {
  edge *curr_orig = g -> edge_list -> hd;
  edge *curr_new = clone_edge(curr_orig);
  edge *hd_new = curr_new;

  while (curr_orig != NULL) {
    curr_new -> next = clone_edge(curr_orig -> next);
    curr_orig = curr_orig -> next;
    curr_new = curr_new -> next;
  }

  return hd_new;
}

node *get_graph_next_node(node *n) {
  return n -> next;
}

edge *get_graph_next_edge(edge *e) {
  return e -> next;
}

node *get_node_by_label_int(graph *g, int label) {
  node *curr = g -> node_list -> hd;
  while (curr != NULL) {
    if ((curr -> label_typ == INTTYPE || curr -> label_typ == BOOLTYPE) && *(curr ->
         label -> i) == label) {
      return curr;
    }
    curr = curr -> next;
  }
  return curr;
}

node *get_node_by_label_int_opt(graph *g, int label) {
  // only used in graph.get_node()
  node *n = get_node_by_label_int(g, label);
  if (n == NULL) {
    n = create_node();
    set_node_label_int(n, 0);
  }
  return n;
}

node *get_node_by_label_bool_opt(graph *g, int label) {
  // only used in graph.get_node()
  node *n = get_node_by_label_int(g, label);
  if (n == NULL) {
    n = create_node();
    set_node_label_bool(n, 0);
  }
  return n;
}

node *get_node_by_label_str(graph *g, char *label) {
  node *curr = g -> node_list -> hd;
  while (curr != NULL) {
    if (curr -> label_typ == STRTYPE && strcmp((char *) get_node_label(curr), label) == 0) {
      return curr;
    }
    curr = curr -> next;
  }
  return curr;
}

node *get_node_by_label_str_opt(graph *g, char *label) {
  // only used in graph.get_node()
  node *n = get_node_by_label_str(g, label);
  if (n == NULL) {
    n = create_node();
    set_node_label_str(n, "");
  }
  return n;
}

edge *get_edge_by_src_and_dst_int(graph *g, int src_label, int dst_label) {
  edge *curr = g -> edge_list -> hd;
  while (curr != NULL) {
    if (*(get_edge_src(curr) -> label -> i) == src_label &&
        *(get_edge_dst(curr) -> label -> i) == dst_label) {
      return curr;
    }
    curr = curr -> next;
  }
  return NULL;
}

edge *get_edge_by_src_and_dst_str(graph *g, char *src_label, char *dst_label) {
  edge *curr = g -> edge_list -> hd;
  while (curr != NULL) {
    if (strcmp((char *) (get_edge_src(curr) -> label -> s), src_label) == 0 &&
        strcmp((char *) (get_edge_dst(curr) -> label -> s), dst_label) == 0) {
      return curr;
    }
    curr = curr -> next;
  }
  return NULL;
}

int add_neighbor(node *n, edge *e) {
  if (n != e -> src) return -1;

  if (n -> neighbor_list -> hd == NULL) {}

  if (n -> neighbor_list -> hd == NULL) {
    n -> neighbor_list -> hd = create_neighbor_list_item(e);
  } else if (n -> neighbor_list -> hd -> edge == e) {
    return -1;
  } else {
    neighbor_list_item *curr = n -> neighbor_list -> hd;
    while (curr -> next != NULL) {
      if (curr -> next -> edge == e) return -1;
      curr = curr -> next;
    }
    curr -> next = create_neighbor_list_item(e);
  }
  return 0;
}

int add_edge_to_edge_list(edge *e, edge_list *el) {
  if (el -> hd == NULL) {
    el -> hd = e;
  } else if (el -> hd == e) {
    return -1;
  } else {
    edge *curr = el -> hd;
    while (curr -> next != NULL) {
      if (curr -> next == e) return -1;
      curr = curr -> next;
    }
    curr -> next = e;
  }
  return 0;
}

void *add_edge_int(graph *g, edge *e, int src, int dst) {
  e -> src = get_node_by_label_int(g, src);
  e -> dst = get_node_by_label_int(g, dst);
  e -> next = NULL;

  // add to neighbors
  if (e -> src == NULL || e -> dst == NULL ||
      add_neighbor(e -> src, e) < 0 ||
      add_edge_to_edge_list(e, g -> edge_list) < 0) return NULL;

  return e;
}

void *add_edge_bool(graph *g, edge *e, int src, int dst) {
  return add_edge_int(g, e, src, dst);
}

void *add_edge_str(graph *g, edge *e, char *src, char *dst) {
  e -> src = get_node_by_label_str(g, src);
  e -> dst = get_node_by_label_str(g, dst);
  e -> next = NULL;

  // add to neighbors
  if (e -> src == NULL || e -> dst == NULL ||
      add_neighbor(e -> src, e) < 0 ||
      add_edge_to_edge_list(e, g -> edge_list) < 0) return NULL;

  return e;
}

int add_node(graph *g, node *n) {
  if (g -> node_list -> hd == NULL) {
    g -> node_list -> hd = n;
  } else if (cmp_node_label(g -> node_list -> hd, n) == 0) {
    return -1;
  } else {
    node *curr = g -> node_list -> hd;
    while (curr -> next != NULL) {
      if (cmp_node_label(curr -> next, n) == 0) return -1;
      curr = curr -> next;
    }
    curr -> next = n;
  }
  return 0;
}

int graph_set_node(graph *g, node *n) {
  // try adding node; handle if node w/ name already exists in the graph
  if (add_node(g, n) < 0 && n -> has_val) {
    int lt = n -> label_typ;
    int dt = n -> data_typ;
    node *n_in_g;

    // find the node in the graph
    if (lt == INTTYPE || lt == BOOLTYPE) {
      n_in_g = get_node_by_label_int(g, *(n -> label -> i));
    } else {
      n_in_g = get_node_by_label_str(g, n -> label -> s);
    }

    // set its data to the data of n
    if (dt == INTTYPE || dt == BOOLTYPE) {
      set_node_data_int(n_in_g, *(n -> data -> i), 1);
    } else if (dt == STRTYPE) {
      set_node_data_str(n_in_g, n -> data -> s, 1);
    }
  }

  return 0;
}

int remove_edge(graph *g, edge *e) {
  if (e == NULL) return -1;

  // remove from edge list
  edge *curr_e = g -> edge_list -> hd;
  if (curr_e != NULL && curr_e == e) {
    g -> edge_list -> hd = curr_e -> next;
  } else {
    edge *prev;
    while (curr_e != NULL && curr_e != e) {
      prev = curr_e;
      curr_e = curr_e -> next;
    }

    if (curr_e == NULL) return -1;

    prev -> next = curr_e -> next;
    prev = NULL;
  }

  // remove from neighbors
  neighbor_list_item *curr_nl = e -> src -> neighbor_list -> hd;
  if (curr_nl != NULL && curr_nl -> edge == e) {
    e -> src -> neighbor_list -> hd = curr_nl -> next;
  } else {
    neighbor_list_item *prev;
    while (curr_nl != NULL && curr_nl -> edge != e) {
      prev = curr_nl;
      curr_nl = curr_nl -> next;
    }

    if (curr_nl == NULL) return -1;

    prev -> next = curr_nl -> next;
    free(curr_nl);
    prev = NULL;
  }

  // free
  free(e);
  e = NULL;

  return 0;
}

int graph_set_edge_int(graph *g, int src_label , int dst_label) {
  edge *e = get_edge_by_src_and_dst_int(g, src_label , dst_label);
  if (e != NULL) {
    set_edge_w_int(e, 0, 0);
    return 0;
  }

  edge *new_e = create_edge();
  set_edge_w_int(new_e , 0, 0);
  add_edge_int(g, new_e , src_label , dst_label);

  return 0;
}

int graph_set_edge_str(graph *g, char *src_label , char *dst_label) {
  edge *e = get_edge_by_src_and_dst_str(g, src_label , dst_label);
  if (e != NULL) {
    set_edge_w_str(e, "", 0);
    return 0;
  }

  edge *new_e = create_edge();
  set_edge_w_str(new_e , "", 0);
  add_edge_str(g, new_e , src_label , dst_label);

  return 0;
}
```

```
int graph_set_edge_int_int(graph *g, int src_label, int dst_label, int w) {
  edge *e = get_edge_by_src_and_dst_int(g, src_label, dst_label);
  if (e != NULL) {
    set_edge_w_int(e, w, 1);
    return 0;
  }

  edge *new_e = create_edge();
  set_edge_w_int(new_e, w, 1);
  add_edge_int(g, new_e, src_label, dst_label);

  return 0;
}

int graph_set_edge_str_str(graph *g, char *src_label, char *dst_label, char *w) {
  edge *e = get_edge_by_src_and_dst_str(g, src_label, dst_label);
  if (e != NULL) {
    set_edge_w_str(e, w, 1);
    return 0;
  }

  edge *new_e = create_edge();
  set_edge_w_str(new_e, w, 1);
  add_edge_str(g, new_e, src_label, dst_label);

  return 0;
}

int graph_set_edge_str_int(graph *g, char *src_label, char *dst_label, int w) {
  edge *e = get_edge_by_src_and_dst_str(g, src_label, dst_label);
  if (e != NULL) {
    set_edge_w_int(e, w, 1);
    return 0;
  }

  edge *new_e = create_edge();
  set_edge_w_int(new_e, w, 1);
  add_edge_str(g, new_e, src_label, dst_label);

  return 0;
}

int graph_set_edge_int_str(graph *g, int src_label, int dst_label, char *w) {
  edge *e = get_edge_by_src_and_dst_int(g, src_label, dst_label);
  if (e != NULL) {
    set_edge_w_str(e, w, 1);
    return 0;
  }

  edge *new_e = create_edge();
  set_edge_w_str(new_e, w, 1);
  add_edge_int(g, new_e, src_label, dst_label);

  return 0;
}

int remove_all_edges(graph *g, node *n) {
  edge *curr_edge = g -> edge_list -> hd;
  edge *temp;
  while (curr_edge != NULL) {
    temp = curr_edge->next;
    if (n -> label_typ == INTTYPE || n -> label_typ == BOOLTYPE) {
      if (*(int *) get_node_label(curr_edge -> src) == *(int *) get_node_label(n) ||
           *(int *) get_node_label(curr_edge -> dst) == *(int *)get_node_label(n)) {
        remove_edge(g, curr_edge);
      }
    }
    if (n -> label_typ == STRTYPE) {
      if ((char *) get_node_label(curr_edge -> src) == (char *)get_node_label(n) ||
          (char *) get_node_label(curr_edge -> dst) == (char *)get_node_label(n)) {
        remove_edge(g, curr_edge);
      }
    }
    curr_edge = temp;
  }
  return 0;
}

int remove_node_int(graph *g, int label){
  node *curr = g -> node_list -> hd;
  if (*(int *) get_node_label(curr) == label) {
    node *n = get_node_by_label_int(g, label);
    remove_all_edges(g, n);
    g -> node_list -> hd = curr -> next;
    free(curr);
    return 0;
  }
  node *prev = curr;
  curr = curr -> next;
  while (curr != NULL) {
    if (*(int *) get_node_label(curr) == label) {
      node *n = get_node_by_label_int(g, label);
      remove_all_edges(g, n);
      prev -> next = curr -> next;
      free(curr);
      return 0;
    }
    prev = curr;
    curr = curr -> next;
  }
  return -1;
}

int remove_node_str(graph *g, char *label){
  node *curr = g -> node_list -> hd;
  if (strcmp((char *) get_node_label(curr), label) == 0) {
    node *n = get_node_by_label_str(g, label);
    remove_all_edges(g, n);
    g -> node_list -> hd = curr -> next;
    free(curr);
    return 0;
  }
  node *prev = curr;
  curr = curr -> next;
  while (curr != NULL) {
    if (strcmp((char *) get_node_label(curr), label) == 0) {
      node *n = get_node_by_label_str(g, label);
      remove_all_edges(g, n);
      prev -> next = curr -> next;
      free(curr);
      return 0;
    }
    prev = curr;
    curr = curr -> next;
  }
  return -1;
}

int graph_has_node_int(graph *g, int name) {
  if (get_node_by_label_int(g, name)) {
    return 0;
  }
  return -1;
}

int graph_has_node_str(graph *g, char *name) {
  if (get_node_by_label_str(g, name)) {
    return 0;
  }
  return -1;
}

int are_neighbors_int(graph *g, int from_name, int to_name) {
  node *src = get_node_by_label_int(g, from_name);
  if (src == NULL || src -> neighbor_list -> hd == NULL) return 0;

  neighbor_list_item *nli = src -> neighbor_list -> hd;
  while (nli != NULL) {
    node *dst = nli -> edge -> dst;
    if (*(dst -> label -> i) == to_name) return 1;
    nli = nli -> next;
  }
  return 0;
}

int are_neighbors_str(graph *g, char *from_name, char *to_name) {
  node *src = get_node_by_label_str(g, from_name);
  if (src == NULL || src -> neighbor_list -> hd == NULL) return 0;

  neighbor_list_item *nli = src -> neighbor_list -> hd;
  while (nli != NULL) {
    node *dst = nli -> edge -> dst;
    if (strcmp(dst -> label -> s, to_name) == 0) return 1;
    nli = nli -> next;
  }
  return 0;
}

int is_empty(graph *g) {
  if (g -> node_list -> hd) {
    return 1; // true
  }
  return 0; // false
}

/* GRAPH TRAVERSAL */

queue *create_queue() {
  queue *Q = (queue *) malloc(sizeof(queue));
  Q -> hd = NULL;
  Q -> tl = NULL;
  return Q;
}
```

```
q_item *create_q_item(node *n) {
  q_item *i = (q_item *) malloc(sizeof(q_item));
  i -> n = n;
  i -> next = NULL;
  return i;
}

void enqueue(queue *Q, node *n) {
  if (Q -> tl == NULL) {
    Q -> hd = create_q_item(n);
    Q -> tl = Q -> hd;
  } else {
    Q -> tl -> next = create_q_item(n);
    Q -> tl = Q -> tl -> next;
  }
}

node *dequeue(queue *Q) {
  if (Q -> hd == NULL) {
    return NULL;
  } else {
    node *n = Q -> hd -> n;
    q_item *tmp = Q -> hd;
    Q -> hd = Q -> hd -> next;
    free(tmp);
    if (Q -> hd == NULL) Q -> tl = NULL;
    return n;
  }
}

void push(queue *Q, node *n) {
  if (Q -> hd == NULL) {
    Q -> hd = create_q_item(n);
  } else {
    q_item *curr = Q -> hd;
    Q -> hd = create_q_item(n);
    Q -> hd -> next = curr;
  }
}

node *pop(queue *Q) {
  if (Q -> hd == NULL) {
    return NULL;
  } else {
    q_item *fst = Q -> hd;
    Q -> hd = fst -> next;
    node *n = fst -> n;
    free(fst);
    return n;
  }
}

int is_empty_q(queue *Q) {
  return (Q -> hd == NULL);
}

void add_neighbors_of_node_to_graph(graph *g_new, node *n_root, node *n_orig, int level) {
  if (level == 0) return;
  queue *Q = create_queue();

  neighbor_list_item *nli = n_orig -> neighbor_list -> hd;
  while (nli != NULL) {
    node *neighbor = nli -> edge -> dst;

    // Don't include neighbor if it is the root node
    if (neighbor == n_root) {
      nli = nli -> next;
      continue;
    }

    // Try to find node with same label as neighbor in g_new
    node *neighbor_copy;
    if (neighbor -> label_typ == INTTYPE || neighbor -> label_typ == BOOLTYPE) {
      neighbor_copy = get_node_by_label_int(g_new, *(neighbor -> label -> i));
    } else if (neighbor -> label_typ == STRTYPE) {
      neighbor_copy = get_node_by_label_str(g_new, neighbor -> label -> s);
    }

    edge *e = create_edge();
    e -> w = clone_primitive(nli -> edge -> w);
    e -> w_typ = nli -> edge -> w_typ;
    e -> has_val = nli -> edge -> has_val;
    if (n_orig == neighbor && neighbor_copy != NULL) {
      // If edge is self-directed, add edge to graph but nothing else
      if (neighbor -> label_typ == INTTYPE || neighbor -> label_typ == BOOLTYPE) {
        add_edge_int(g_new, e, *(neighbor_copy -> label -> i), *(neighbor_copy -> label -> i));
      } else if (neighbor -> label_typ == STRTYPE) {
        add_edge_str(g_new, e, neighbor_copy -> label -> s, neighbor_copy -> label -> s);
      }
    } else {
      // If node doesn't yet exist in g_new, create one and add original neighbor to processing queue
      if (neighbor_copy == NULL) {
        neighbor_copy = clone_node(neighbor);
        add_node(g_new, neighbor_copy);
        enqueue(Q, neighbor);
      }

      if (neighbor -> label_typ == INTTYPE || neighbor -> label_typ == BOOLTYPE) {
        add_edge_int(g_new, e, *(n_orig -> label -> i), *(neighbor_copy -> label -> i));
      } else if (neighbor -> label_typ == STRTYPE) {
        add_edge_str(g_new, e, n_orig -> label -> s, neighbor_copy -> label -> s);
      }
    }

    nli = nli -> next;
  }

  while (Q -> tl != NULL) {
    add_neighbors_of_node_to_graph(g_new, n_root, dequeue(Q), level - 1);
  }

  free(Q);
}

graph *neighbors_one_arg(node *n) {
  graph *g_new = create_graph();
  if (n == NULL) return g_new;
  add_neighbors_of_node_to_graph(g_new, n, n, 1);

  return g_new;
}

graph *neighbors(node *n, int level, int include_current) {
  graph *g_new = create_graph();
  if (level <= 1) level = 1;

  if (n == NULL) return g_new;

  if (include_current != 0) {
    add_node(g_new, clone_node(n));
    add_neighbors_of_node_to_graph(g_new, NULL, n, level);
  } else {
    add_neighbors_of_node_to_graph(g_new, n, n, level);
  }

  return g_new;
}

graph *find_data_int(graph *g, int data) {
  node *n = g -> node_list -> hd;
  graph *g_new = create_graph();
  while (n != NULL) {
    if (n -> has_val == 1 && (n -> data_typ == INTTYPE || n -> data_typ == BOOLTYPE)
         && *(n -> data -> i) == data) {
      node *n_cp = clone_node(n);
      add_node(g_new, n_cp);
    }
    n = n -> next;
  }
  return g_new;
}

graph *find_data_str(graph *g, char *data) {
  node *n = g -> node_list -> hd;
  graph *g_new = create_graph();

  while (n != NULL) {
    if (n -> has_val == 1 && n -> data_typ == STRTYPE && strcmp(n -> data -> s, data
        ) == 0) {
      node *n_cp = clone_node(n);
      add_node(g_new, n_cp);
    }
    n = n -> next;
  }
  return g_new;
}

/* PRINTING */

void print_node(node *n) {
  if (n -> label_typ == INTTYPE) {
    printf("%d:", *(n -> label -> i));
  } else if (n -> label_typ == BOOLTYPE) {
    if (*(n -> label -> i) == 0) printf("false:");
    else printf("true:");
  } else if (n -> label_typ == STRTYPE) {
    printf("\"%s\":", n -> label -> s);
  }

  if (n -> has_val == 0) {
    printf("null");
  } else if (n -> data_typ == INTTYPE) {
    printf("%d", *(n -> data -> i));
  } else if (n -> data_typ == BOOLTYPE) {
    if (*(n -> data -> i) == 0) printf("false");
    else printf("true");
  } else if (n -> data_typ == STRTYPE) {
    printf("\"%s\"", n -> data -> s);
  }

  return;

}

int search_node_list(node_list *nl, node *n) {
  node *curr = nl -> hd;
  while (curr != NULL) {
    if ((curr -> label) == (n -> label)) {return 1;}
    curr = curr -> next;
  }
  return 0;
}

void add_node_to_list(node_list *nl, node *n) {
  node *curr = nl -> hd;
  n -> next = curr;
  nl -> hd = n;
}


void print_edge_weight(edge *e) {
  if (e -> has_val == 0) {
    printf("(null)");
  } else if (e -> w_typ == INTTYPE) {
    printf("(%d)", *(int *) e -> w -> i );
  } else if (e -> w_typ == BOOLTYPE) {
    if (*(int *) e -> w -> i == 1) printf("(true)");
    else printf("(false)");
  } else if (e -> w_typ == STRTYPE) {
    printf("(%s)", (char *)e -> w -> s );
  }
  return;
}

void print_graph(graph *g) {
  node *n = g -> node_list -> hd;
  while (n) {
    print_node(n);
    printf(" -> [");
    neighbor_list_item *nli = n -> neighbor_list -> hd;
    if (nli) {
      print_node(nli -> edge -> dst);
      printf(" ");
      print_edge_weight(nli -> edge);
      while (nli -> next) {
        printf(", ");
        print_node(nli -> next -> edge -> dst);
        printf(" ");
        print_edge_weight(nli -> next -> edge);
        nli = nli -> next;
      }
    }

    printf("]\n");
    n = n -> next;
  }

  return;
}

graph *bfs_int(graph *g, int name) {
  queue *Q = create_queue();
  node_list *seen = create_node_list();
  graph *bfs_graph = create_graph();
  node *start = get_node_by_label_int(g, name);

  enqueue(Q, start);
  while (is_empty_q(Q) == 0) {
    node *next = dequeue(Q);
    if (search_node_list(seen, next) == 0) {
      add_node_to_list(seen, next);
      add_node(bfs_graph, clone_node(next));
      neighbor_list *neighbors = next -> neighbor_list;
      neighbor_list_item *neighbor = neighbors -> hd;
      while (neighbor != NULL) {
        enqueue(Q, neighbor -> edge -> dst);
        neighbor = neighbor -> next;
      }
    }
  }
  return bfs_graph;
}

graph *bfs_str(graph *g, char *name) {
  queue *Q = create_queue();
  node_list *seen = create_node_list();
  graph *bfs_graph = create_graph();
  node *start = get_node_by_label_str(g, name);

  enqueue(Q, start);
  while (is_empty_q(Q) == 0) {
    node *next = dequeue(Q);
    if (search_node_list(seen, next) == 0) {
      add_node_to_list(seen, next);
      add_node(bfs_graph, clone_node(next));
      neighbor_list *neighbors = next -> neighbor_list;
      neighbor_list_item *neighbor = neighbors -> hd;
      while (neighbor != NULL) {
        enqueue(Q, neighbor -> edge -> dst);
        neighbor = neighbor -> next;
      }
    }
  }
  return bfs_graph;
}

graph *dfs_int(graph *g, int name) {
  queue *Q = create_queue();
  node_list *seen = create_node_list();
  graph *dfs_graph = create_graph();
  node *start = get_node_by_label_int(g, name);

  push(Q, start);
  while (is_empty_q(Q) == 0) {
    node *next = pop(Q);
    if (search_node_list(seen, next) == 0) {
      add_node_to_list(seen, next);
      add_node(dfs_graph, clone_node(next));
      neighbor_list *neighbors = next -> neighbor_list;
      neighbor_list_item *neighbor = neighbors -> hd;
      while (neighbor != NULL) {
        push(Q, neighbor -> edge -> dst);
        neighbor = neighbor -> next;
      }
    }
  }
  return dfs_graph;
}

graph *dfs_str(graph *g, char *name) {
  queue *Q = create_queue();
  node_list *seen = create_node_list();
  graph *dfs_graph = create_graph();
  node *start = get_node_by_label_str(g, name);

  push(Q, start);
  while (is_empty_q(Q) == 0) {
    node *next = pop(Q);
    if (search_node_list(seen, next) == 0) {
      add_node_to_list(seen, next);
      add_node(dfs_graph, clone_node(next));
      neighbor_list *neighbors = next -> neighbor_list;
      neighbor_list_item *neighbor = neighbors -> hd;
      while (neighbor != NULL) {
        push(Q, neighbor -> edge -> dst);
        neighbor = neighbor -> next;
      }
    }
  }
  return dfs_graph;
}
```

9.8 Example Code

9.8.1 Bellman-Ford Algorithm

```
int main() {

    bool valid = true; (*negative edge cycle check*)

    (*Initial graph*)
    graph<string:int, int> g = ["S":500 -(10)> "A":500 -(2)> "C":500 -(2)> "B":500 -(1)> "A"; "S" -(8)>"E":500 -(1)> "D":500 -(1)>"C"; "D" -(4)> "A"];

    (*Shortest path graph*)
    graph<string:int, int> shortest_path = [];

    (*Initialize distances to infinity*)
    for_node(n : g) {
```

```
    node<string:int> n1 = n.get_name():999999;
    shortest_path.set_node(n1);
  }

  (*copy in weights*)
  for_edge(src, dst, w: g) {
    shortest_path.set_edge(src.get_name(); dst.get_name(); w);
  }

  (*Initialize start node to 0*)
  node<string:int> source_n = "S":0;
  shortest_path.set_node(source_n);

  print("ORIGINAL GRAPH:");
  g.print();

  (*Relax edges n times*)
  for_node(n : shortest_path) {
   for_edge(src, dst, w : shortest_path) {
     int src_data = src.get_data();
     int dst_data = dst.get_data();

     if (src_data + w < dst_data) {
      int new_dst_data = src_data + w;
      node<string:int> new_dst = dst.get_name():new_dst_data;
      shortest_path.set_node(new_dst);
     }
   }
  }

  (*Negative edge weight cycle check*)
  for_edge(src, dst, w : shortest_path) {
    int src_data = src.get_data();
    int dst_data = dst.get_data();

    if (src_data + w < dst_data) {
      print("negative edge weight cycle");
      valid = false;
    }
    else if (src_data + w > dst_data) {
      shortest_path.remove_edge(src.get_name(); dst.get_name());
    }
  }

  if (valid) {
    print("SHORTEST PATH:");
    shortest_path.print();
  }

  return 0;
}
```

9.8.2 Family Tree

```
int main() {
 graph<string:string, string> family = ["Joe":"Grandfather" <("Spouses")> "Mary":"
   Grandmother"; "Joe" <("Brothers")> "Charlie":"Great Uncle"; "Joe" -("Son")> "
   Mufasa":"Father"; "Mary" -("Son")> "Mufasa" <("Spouses")> "Sirabi":"Mother" -("
   Son")> "Simba":"The Prince" <("Son")- "Mufasa"];

 family.print();
```

```

  print("");
  print("Simba got married!");
  node<string:string> nala = "Nala":"Daughter in law";
  family.set_node(nala);
  family.set_edge("Simba"; "Nala"; "Spouses");
  family.set_edge("Nala"; "Simba"; "Spouses");
  family.print();
}
```

9.9 Regression Test Suite - Positive Tests

9.9.1 test-anon-func.hpg

```
int main() {
  fun<int:int,int> f = int (int x; int y) (x + y);
  fun<int:bool> g = int (bool q) (0);
  print_int( f(3; 4) );
  return 0;
}
```

Expected output:

```
7
```

9.9.2 test-are-neighbors1.hpg

```
int main() {
  graph<int, int> g = [1 <(5)> 2 -()> 3];
  print_bool(g.are_neighbors(1; 2));
  print_bool(g.are_neighbors(2; 1));
  print_bool(g.are_neighbors(2; 3));
  print_bool(g.are_neighbors(3; 2));
  return 0;
}
```

Expected output:

```
1
1
1
0
```

9.9.3 test-are-neighbors2.hpg

```
int main() {
  graph<string, int> g = ["1" <(5)> "2" -()> "3"];
  print_bool(g.are_neighbors("1"; "2"));
  print_bool(g.are_neighbors("2"; "1"));
  print_bool(g.are_neighbors("2"; "3"));
  print_bool(g.are_neighbors("3"; "2"));
  return 0;
}
```

Expected output:

```
1
1
1
0
```

9.9.4 test-bellmanford.hpg

```
int main() {

    bool valid = true; (*negative edge cycle check*)

    (*Initial graph*)
    graph<string:int, int> g = ["S":500 -(10)> "A":500 -(2)> "C":500 -(2)> "B":500
        -(1)> "A"; "S" -(8)>"E":500 -(1)> "D":500 -(1)>"C"; "D" -(4)> "A"];

    (*Shortest path graph*)
    graph<string:int, int> shortest_path = [];

    (*Initialize distances to infinity*)
    for_node(n : g) {
        node<string:int> n1 = n.get_name():999999;
        shortest_path.set_node(n1);
    }

    (*copy in weights*)
    for_edge(src, dst, w: g) {
        shortest_path.set_edge(src.get_name(); dst.get_name(); w);
    }

    (*Initialize start node to 0*)
    node<string:int> source_n = "S":0;
    shortest_path.set_node(source_n);

    print("ORIGINAL GRAPH:");
    g.print();

    (*Relax edges n times*)
    for_node(n : shortest_path) {
      for_edge(src, dst, w : shortest_path) {
          int src_data = src.get_data();
          int dst_data = dst.get_data();

          if (src_data + w < dst_data) {
            int new_dst_data = src_data + w;
            node<string:int> new_dst = dst.get_name():new_dst_data;
            shortest_path.set_node(new_dst);
          }
      }
    }

    (*Negative edge weight cycle check*)
    for_edge(src, dst, w : shortest_path) {
        int src_data = src.get_data();
        int dst_data = dst.get_data();

        if (src_data + w < dst_data) {
            print("negative edge weight cycle");
            valid = false;
        }
        else if (src_data + w > dst_data) {
            shortest_path.remove_edge(src.get_name(); dst.get_name());
        }
    }

    if (valid) {
        print("SHORTEST PATH:");
        shortest_path.print();
    }

    return 0;
}
```

Expected output:

```
ORIGINAL GRAPH:
”S”:500 ->[”A”:500 (10), ”E”:500 (8)]
”A”:500 ->[”C”:500 (2)]
”C”:500 ->[”B”:500 (2)]
”B”:500 ->[”A”:500 (1)]
”E”:500 ->[”D”:500 (1)]
”D”:500 ->[”C”:500 (1), ”A”:500 (4)]
SHORTEST PATH:
”S”:0 ->[”A”:10 (10), ”E”:8 (8)]
”A”:10 ->[]
”C”:10 ->[”B”:12 (2)]
”B”:12 ->[]
”E”:8 ->[”D”:9 (1)]
”D”:9 ->[”C”:10 (1)]
```

9.9.5 test-bfs-path.hpg

```
int main() {
  graph<int> g = [1 -()- 2 -()- 3; 1 -()- 4];
  graph<int> g_sub1 = g.bfs(1);
  g_sub1.print();
}
```

Expected output:

```
1:null ->[]
2:null ->[]
4:null ->[]
3:null ->[]
```

9.9.6 test-binop-ops.hpg

```
int main() {
  print_bool(4 < 5);
  print_bool(5 < 5);
  print_bool(6 < 5);

  print_bool(4 <= 5);
  print_bool(5 <= 5);
  print_bool(6 <= 5);

  print_bool(4 == 5);
  print_bool(5 == 5);
  print_bool(6 == 5);

  print_bool(4 >= 5);
  print_bool(5 >= 5);
  print_bool(6 >= 5);

  print_bool(4 > 5);
  print_bool(5 > 5);
  print_bool(6 > 5);

  return 0;
}
```

Expected output:

```
1
0
0
1
1
0
0
1
0
0
1
1
0
0
1
```

9.9.7 test-binop-ops-str.hpg

```
int main() {
  print_bool("foo" == "foo");
  print_bool("foo" == "bar");
  print_bool("foo" > "bar");
  print_bool("foo" < "bar");
  return 0;
}
```

Expected output:

```
1
0
1
0
```

9.9.8 test-binop-prec.hpg

```
int main() {
  print_bool(5 == 4 + 1);
  print_bool(4 + 1 == 5);
  print_bool(2 * 4 + 1 == 9);
  print_bool(1 + 2 * 4 == 9);
  print_bool(2 * (4 + 1) == 10);
  print_bool((2 * (4 + 1) + 2) * 3 == 18 * 2);
}
```

Expected output:

```
1
1
1
```

```
1
1
1
```

9.9.9 test-create-graph-type-bool.hpg

```
int main() {
  graph<bool:bool, bool> g0 = [true: false -(true)> false: true];

(*  graph<bool:bool, bool> g0 = [true: false <(true)> false: true];

  graph<int:bool, bool> g1 = [];
  graph<int:bool, bool> g2 = [1];
  graph<int:bool, bool> g3 = [1:NULL];
  graph<int:bool, bool> g4 = [1:true];
  graph<int:bool, bool> g5 = [1:true <()> 2:NULL];
  graph<int:bool, bool> g6 = [1:NULL <()> 2:true];
  graph<int:bool, bool> g7 = [1:true <(NULL)> 2:NULL];
  graph<int:bool, bool> g8 = [1:false <()> 2:true];
  graph<int:bool, bool> g9 = [1 <()> 2:true];
  graph<int:bool, bool> g10 = [1:true <(false)> 2:NULL -(true)> 3:true; 3 <()- 4];
  graph<int:bool, bool> g11 = [1 <()> 2:NULL -()> 3:NULL; 3 <()- 4];*)

  return 0;
}
```

9.9.10 test-create-graph-type-int.hpg

```
int main() {
  graph<int:int, int> g1 = [];
  graph<int:int, int> g2 = [1];
  graph<int:int, int> g3 = [1:NULL];
  graph<int:int, int> g4 = [1:2 <()> 3:NULL];
  graph<int:int, int> g5 = [1:2 <(NULL)> 3:NULL];
  graph<int:int, int> g6 = [1:NULL <()> 2:3];
  graph<int:int, int> g7 = [1 <()> 2:3];
  graph<int:int, int> g8 = [1:2 <(3)> 4:NULL -()> 5:42; 5 <()- 6];
  graph<int:int, int> g9 = [1 <()> 2:NULL -()> 3:NULL; 3 <()- 4];

  return 0;
}
```

9.9.11 test-create-graph-type-string.hpg

```
int main() {
  graph<string:string, string> g1 = [];
  graph<string:string, string> g2 = ["foo"];
  graph<string:string, string> g3 = ["foo":NULL];
  graph<string:string, string> g4 = ["foo":"bar" <()> "baz":NULL];
  graph<string:string, string> g5 = ["foo":"bar" <(NULL)> "baz":NULL];
  graph<string:string, string> g6 = ["foo":NULL <()> "bar":"baz"];
  graph<string:string, string> g7 = ["foo" <()> "bar":"baz"];
  graph<string:string, string> g8 = ["a":"b" <("c")> "d":NULL -()> "e":"f"; "g" <()-
      "h"];
  graph<string:string, string> g9 = ["a" <()> "b":NULL -()> "c":NULL; "c" <()- "d"];

  return 0;
}
```

9.9.12 test-create-node.hpg

```
int main() {
  node<string:string> n1 = "abc":"def";
  node<string:int> n2 = "abc":42;
  node<int:string> n3 = 42:"abc";
  node<int:int> n4 = 42:1;
  node<string:bool> n5 = "abc":NULL;
  node<int:bool> n6 = 42:NULL;
  node<string> n7 = "abc";
  node<int> n8 = 42;
  return 0;
}
```

9.9.13 test-dfs-path.hpg

```
int main() {
  graph<int> g = [1 -()- 2 -()- 3 -()- 4];
  graph<int> g_sub1 = g.dfs(1);
  g_sub1.print();
}
```

Expected output:

```
1:null ->[]
2:null ->[]
3:null ->[]
4:null ->[]
```

9.9.14 test-fdecls-argnum.hpg

```
int main() {
  print_int(41 + one());
  print_int(addone(41));
  print_int(add(40; 2));
  return 0;
}

(* no args *)
int one() {
  return 1;
}

(* one arg *)
int addone(int x) {
  return x + 1;
}

(* multiple args *)
int add(int x; int y) {
  return x + y;
}
```

Expected output:

```
42
42
42
```

9.9.15 test-fdecls-argtype.hpg

```
int main() {
  void_func1();
  void_func2();
  int v1 = int_func();
  string v2 = str_func();
  print_int(v1);
  print(v2);
  return 0;
}

void void_func1() {}
void void_func2() { return; }

int int_func() { return 42;}
string str_func() { return "hello"; }
```

Expected output:
42
hello

9.9.16 test-find-data-int1.hpg

```
int main() {
  graph<string:int> g = ["a":10 -()> "b":20 <()> "c":20; "a" <()- "d"];
  graph<string:int> g_new = g.find(20);
  for_node(n : g_new) {
    n.print();
    print("");
  }
}
```

Expected output:
”b”:20
”c”:20

9.9.17 test-find-data-int2.hpg

```
int main() {
  graph<string:int> g = ["a":10 -()> "b":20 <()> "c":20; "a" <()- "d"];
  graph<string:int> g_new = g.find(40);
  for_node(n : g_new) {
    n.print();
    print("");
  }
}
```

9.9.18 test-find-data-str.hpg

```
int main() {
  graph<int:string> g = [1:"a" -()> 2:"b" <()> 3:"b"; 1 <()- 4];
  graph<int:string> g_new = g.find("b");
  for_node(n : g_new) {
    n.print();
    print("");
  }
}
```

Expected output:
2:”b”

3:”b”

9.9.19 test-for-edge1.hpg

```
int main() {
  graph<int:int, int> g = [1:1-(3)>2:2];
  for_edge(src, dst, w : g) {
    print_int(src.get_data());
    print_int(dst.get_data());
    print_int(w);
  }
  return 0;
}
```

Expected output:
1
2
3

9.9.20 test-for-edge2.hpg

```
int main() {
  graph<int:int, int> g = [];
  for_edge(src, dst, w : g) {
    print_int(src.get_data());
    print_int(dst.get_data());
    print_int(w);
  }
  return 0;
}
```

9.9.21 test-for-edge3.hpg

```
int main() {
  graph<int:int, int> g = [1:1<(3)>2:2];
  for_edge(src, dst, w : g) {
    print_int(src.get_data());
    print_int(dst.get_data());
    print_int(w);
  }
  return 0;
}
```

Expected output:
1
2
3
2
1
3

9.9.22 test-for-edge4.hpg

```
int main() {
  graph<int:int, int> g = [1:1-(3)>2:2<(5)>4:4; 7:7<(8)-6:6];
  for_edge(src, dst, w : g) {
    print_int(src.get_data());
    print_int(dst.get_data());
    print_int(w);
  }
  return 0;
}
```

Expected output:

```
1
2
3
2
4
5
4
2
5
6
7
8
```

9.9.23 test-for-edge5.hpg

```
int main() {
  graph<int:int, int> g = [1:1-()>2:2];
  for_edge(src, dst, w : g) {
    print_int(src.get_data());
    print_int(dst.get_data());
    print_int(w);
  }
  return 0;
}
```

Expected output:

```
1
2
0
```

9.9.24 test-for-node1.hpg

```
int main() {
  graph<int:int, int> g = [1:1];
  for_node(n : g) {
    print_int(n.get_data());
  }
}
```

Expected output:

```
1
```

9.9.25 test-for-node2.hpg

```
int main() {
  graph<int:int, int> g = [];
  for_node(n : g) {
    print_int(n.get_data());
  }
}
```

9.9.26 test-for-node3.hpg

```
int main() {
  graph<int:int, int> g = [1:1 -()> 2:2 -()> 3:3 -()> 4:4 -()> 5:5 -()> 6:6 -()>
      7:7];
  for_node(n : g) {
    print_int(n.get_data());
  }
}
```

Expected output:

```
1
2
3
4
5
6
7
```

9.9.27 test-for-node4.hpg

```
int main() {
  graph<string:int, int> g = ["10":1 -()> "20":2 -()> "30":3 -()> "40":4 -()> "50":5
      -()> "60":6 -()> "70":7];
  for_node(n : g) {
    print_int(n.get_data());
  }
}
```

Expected output:

```
1
2
3
4
5
6
7
```

9.9.28 test-for.hpg

```
int main() {
  int x;
  for ( x = 0 ; x <= 3 ; x = x + 1 ) {
    print("True!");
  }
}
```

Expected output:

```
True!
True!
True!
True!
```

9.9.29 test-get-name1.hpg

```
int main() {
  node<int> n = 1;
  print_int(n.get_name());
  node<bool> n = true;
  print_bool(n.get_name());
  node<string> n = "foo";
  print(n.get_name());
}
```

Expected output:

```
1
1
foo
```

9.9.30 test-get-node-data.hpg

```
int main() {
  node<string:string> n1 = "abc":"def";
  node<string:int> n2 = "abc":42;
  node<int:string> n3 = 42:"abc";
  node<int:int> n4 = 42:1;

  string d1 = n1.get_data();
  print(d1);

  int d2 = n2.get_data();
  print_int(d2);

  string d3 = n3.get_data();
  print(d3);

  int d4 = n4.get_data();
  print_int(d4);

  return 0;
}
```

Expected output:

```
def
42
abc
1
```

9.9.31 test-get-node1.hpg

```
int main() {
  graph<int:string> g = [1:"foo"; 2:"bar"];
  node<int:string> n1 = g.get_node(1);
  node<int:string> n2 = g.get_node(2);
  node<int:string> n3 = g.get_node(3);

  n1.print();
  print("");
  n2.print();
  print("");
  n3.print();
}
```

Expected output:

```
1:”foo”
2:”bar”
0:null
```

9.9.32 test-get-node2.hpg

```
int main() {
  graph<string:int> g = ["foo":1; "bar":2];
  node<string:int> n1 = g.get_node("foo");
  node<string:int> n2 = g.get_node("bar");
  node<string:int> n3 = g.get_node("baz");

  n1.print();
  print("");
  n2.print();
  print("");
  n3.print();
}
```

Expected output:

```
”foo”:1
”bar”:2
””:null
```

9.9.33 test-get-node3.hpg

```
int main() {
  graph<bool:string> g = [true:"foo"];
  node<bool:string> n1 = g.get_node(true);
  node<bool:string> n2 = g.get_node(false);

  n1.print();
  print("");
  n2.print();
}
```

Expected output:

```
true:”foo”
false:null
```

9.9.34 test-get-weight1.hpg

```
int main() {
  graph<int:string, int> g = [1:"foo" -(10)> 2:"bar" -()> 3];
  print_int(g.get_weight(1; 2));
  print_int(g.get_weight(2; 1));
  print_int(g.get_weight(2; 3));
}
```

Expected output:

```
10
0
0
```

9.9.35 test-get-weight2.hpg

```
int main() {
  graph<int:string, string> g = [1:"foo" -("10")> 2:"bar" -()> 3];
  print(g.get_weight(1; 2));
  print(g.get_weight(2; 1));
  print(g.get_weight(2; 3));
}
```

Expected output:
10

9.9.36 test-get-weight3.hpg

```
int main() {
  graph<int:string, bool> g = [1:"foo" -(true)> 2:"bar" -()> 3];
  print_bool(g.get_weight(1; 2));
  print_bool(g.get_weight(2; 1));
  print_bool(g.get_weight(2; 3));
}
```

Expected output:
1
0
0

9.9.37 test-graph-neighbors1.hpg

```
int main() {
  graph<int, int> g = [];
  graph<int, int> g_sub = g.neighbors(5; 2; true);
}
```

9.9.38 test-graph-neighbors2.hpg

```
int main() {
  graph<int, int> g = [5];
  graph<int, int> g_sub = g.neighbors(5; 2; true);
}
```

9.9.39 test-graph-neighbors3.hpg

```
int main() {
  graph<int> g = [1 -()- 2 -()- 3; 1 -()- 4 -()- 5; 4 -()- 6 -()- 8; 6 -()- 7];
  graph<int> g_sub1 = g.neighbors(1; 2; false);
  for_node (n : g_sub1) {
    n.print();
    print("");
  }
  print("");
  graph<int> g_sub2 = g.neighbors(1; 3; true);
  for_node (n : g_sub2) {
    n.print();
    print("");
  }
}
```

Expected output:
2:null
4:null
3:null

```
5:null
6:null
1:null
2:null
4:null
3:null
5:null
6:null
8:null
7:null
```

9.9.40 test-graph-neighbors4.hpg

```
int main() {
  graph<int> g = [1 -()- 2 -()- 3 -()- 1];
  graph<int> g_sub1 = g.neighbors(1; 3; false);
  for_node (n : g_sub1) {
    n.print();
    print("");
  }
  print("");
  graph<int> g_sub2 = g.neighbors(1; 3; true);
  for_node (n : g_sub2) {
    n.print();
    print("");
  }
}
```

Expected output:

```
2:null
3:null
1:null
2:null
3:null
```

9.9.41 test-graph-neighbors5.hpg

```
int main() {
  graph<int> g = [1 -()- 2 -()- 3 -()- 4 -()- 2];
  graph<int> g_sub1 = g.neighbors(1; 4; false);
  for_node (n : g_sub1) {
    n.print();
    print("");
  }
  print("");
  graph<int> g_sub2 = g.neighbors(1; 4; true);
  for_node (n : g_sub2) {
    n.print();
    print("");
  }
}
```

Expected output:

```
2:null
3:null
4:null
```

1:null
2:null
3:null
4:null

9.9.42 test-has-node-bool.hpg

```
int main() {
  graph<bool, int> g = [true <(10)> false];
  int result1 = g.has_node(true);
  print_int(result1);
  g.remove_node(true);
  int result2 = g.has_node(true);
  print_int(result2);
}
```

Expected output:
0
-1

9.9.43 test-has-node-int.hpg

```
int main() {
  graph<int, int> g = [1 <(10)> 2; 3];
  int result1 = g.has_node(1);
  print_int(result1);
  int result2 = g.has_node(5);
  print_int(result2);
}
```

Expected output:
0
-1

9.9.44 test-has-node-str.hpg

```
int main() {
  graph<string, int> g = ["hello" <(10)> "there"; "!"];
  int result1 = g.has_node("!!");
  print_int(result1);
  int result2 = g.has_node("hello");
  print_int(result2);
}
```

Expected output:
-1
0

9.9.45 test-helloworld.hpg

```
int main() {
  print("Hello, world");
  return 0;
}
```

Expected output:
Hello, world

9.9.46 test-if-else.hpg

```
int main() {
  if (true) {
    print("True!");
  }

  int foo = 5;

  if (foo == 1) {
    print("True!");
  } else {
    print("False!");
  }

  return 0;
}
```

Expected output:
True!
False!

9.9.47 test-if.hpg

```
int main() {
  if (true) {
    print("True!");
  }

  return 0;
}
```

Expected output:
True!

9.9.48 test-is-empty.hpg

```
int main() {
  graph<int:int, int> g1 = [];
  print_bool(g1.is_empty());
  graph<int:int, int> g2 = [1];
  print_bool(g2.is_empty());
}
```

Expected output:
0
1

9.9.49 test-pass-graph-to-func.hpg

```
int main() {
  graph<int, string> g = [1 -("a")- 2];
  add_node_remotely(g; 3);
  add_node_remotely(g; 4);
  add_node_remotely(g; 5);
  g.print();
  return 0;
}
```

```

void add_node_remotely(graph<int, string> g; int label) {
  node<int> n = label;
  g.set_node(n);
}
```

Expected output:

```
1:null -¿ [2:null (a)]
2:null -¿ [1:null (a)]
3:null -¿ []
4:null -¿ []
5:null -¿ []
```

9.9.50 test-print-node.hpg

```
int main() {
  node<int:string> n1 = 500:"foo";
  node<string:bool> n2 = "bar":true;
  node<bool:string> n3 = false;
  node<int:int> n4 = 12345:6789;
  n1.print();
  print("");
  n2.print();
  print("");
  n3.print();
  print("");
  n4.print();
}
```

Expected output:

```
500:”foo”
”bar”:true
false:null
12345:6789
```

9.9.51 test-printbool.hpg

```
int main() {
  print_bool(true);
  print_bool(false);
  print_bool(2 + 4 == 6);
  print_bool(2 + 4 == 5);
  return 0;
}
```

Expected output:

```
1
0
1
0
```

9.9.52 test-printint.hpg

```
int main() {
  print_int(123);
  return 0;
}
```

Expected output:
123

9.9.53 test-recursion1.hpg

```
void count(int i) {
  if (i <= 0) {
    print_int(i);
  } else {
    print_int(i);
    count(i - 1);
  }
  return;
}

int main() {
  count(5);
  return 0;
}
```

Expected output:
5
4
3
2
1
0

9.9.54 test-recursion2.hpg

```
int main() {
  print_int(fib(7));
  return 0;
}

int fib(int i) {
  if (i < 2) {
    return 1;
  } else {
    return fib(i - 2) + fib(i - 1);
  }
}
```

Expected output:
21

9.9.55 test-remove-edge1.hpg

```
int main() {
  graph<int, int> g = [1 <(10)> 2; 3];
  print_int(g.remove_edge(1; 2));
  print_int(g.remove_edge(2; 1));
  print_int(g.remove_edge(2; 1));
}
```

Expected output:
0

0
-1

9.9.56 test-remove-node-bool.hpg

```
int main() {
  graph<bool, int> g = [true <(10)> false];
  int result1 = g.remove_node(true);
  print_int(result1);
  int result2 = g.remove_node(true);
  print_int(result2);
}
```

Expected output:
0
-1

9.9.57 test-remove-node-int.hpg

```
int main() {
  graph<int, int> g = [1 <(10)> 2; 3];
  int result1 = g.remove_node(1);
  print_int(result1);
  int result2 = g.remove_node(5);
  print_int(result2);
}
```

Expected output:
0
-1

9.9.58 test-remove-node-str.hpg

```
int main() {
  graph<string, int> g = ["hello" <(10)> "there"; "!"];
  int result1 = g.remove_node("!!");
  print_int(result1);
  int result2 = g.remove_node("hello");
  print_int(result2);
}
```

Expected output:
-1
0

9.9.59 test-set-data1.hpg

```
int main() {
  node<int:int> n = 1:1;
  print_int(n.get_data());
  n.set_data(2);
  print_int(n.get_data());
  return 0;
}
```

Expected output:
1

2

9.9.60 test-set-edge-bool-int.hpg

```
int main() {
  graph<bool:int, int> g = [true:1 <(3)> false:9];
  g.set_edge(true; false; 1);
  print_int(g.get_weight(true; false));
  return 0;
}
```

Expected output:
1

9.9.61 test-set-edge-bool-str.hpg

```
int main() {
  graph<bool:int, string> g = [true:1 <("bla")> false:9];
  g.set_edge(false; true; "bla");
  print(g.get_weight(false; true));
  return 0;
}
```

Expected output:
bla

9.9.62 test-set-edge-bool.hpg

```
int main() {
  graph<bool:bool, string> g = [true:true; false: false];
  g.set_edge(true; false); (* set weight to empty *)
  g.print();
  return 0;
}
```

Expected output:
true:true ->[false:false (null)]
false:false ->[]

9.9.63 test-set-edge-int-bool.hpg

```
int main() {
  graph<int:int, bool> g = [1:1 <(true)> 2:9];
  g.set_edge(1; 2; false);
  print_bool(g.get_weight(1; 2));
  return 0;
}
```

Expected output:
0

9.9.64 test-set-edge-int-int.hpg

```
int main() {
  graph<int:int, int> g = [1:1 <(3)> 2:9];
  g.set_edge(1; 2; 1);
  print_int(g.get_weight(1; 2));
  return 0;
}
```

Expected output:

```
1
```

9.9.65 test-set-edge-int-str.hpg

```
int main() {
  graph<int:bool, string> g = [1:true; 2: false];
  g.set_edge(1; 2; "foo");
  print(g.get_weight(1; 2));
}
```

Expected output:

```
foo
```

9.9.66 test-set-edge-int.hpg

```
int main() {
  graph<int:bool, string> g = [1:true; 2: false];
  g.set_edge(1; 2); (* set weight to empty *)
  g.print();
}
```

Expected output:

```
1:true ->[2:false (null)]
2:false ->[]
```

9.9.67 test-set-edge-str-bool.hpg

```
int main() {
  graph<string:int, bool> g = ["hi":1 <(true)> "there":9];
  g.set_edge("hi"; "there"; false);
  print_bool(g.get_weight("hi"; "there"));
  return 0;
}
```

Expected output:

```
0
```

9.9.68 test-set-edge-str-str.hpg

```
int main() {
  graph<string:int, string> g = ["hello":1 <("yes")> "good":9];
  g.set_edge("hello"; "good"; "now");
  print(g.get_weight("hello"; "good"));
  return 0;
}
```

Expected output:

```
now
```

9.9.69 test-set-edge-str.hpg

```
int main() {
  graph<string:bool, string> g = ["1":true; "2": false];
  g.set_edge("1"; "2"); (* set weight to empty *)
  g.print();
}
```

Expected output:
”1”:true ->[”2”:false (null)]
”2”:false ->[ ]

9.9.70 test-set-node1.hpg

```
int main() {
  graph<int:int, int> g = [1:1];
  node<int:int> n = 2:2;
  g.set_node(n);
  return 0;
}
```

9.9.71 test-vdecls-global.hpg

```
int a;
int b;
string c;

int main() {
  b = 40;
  a = b + 2;
  string c = "foo";
  print_int(a);
  print_int(b);
  print(c);

  return 0;
}
```

Expected output:
42
40
foo

9.9.72 test-vdecls.hpg

```
int res;

int main() {
  res = 7;
  int x = res + 2;
  int y = x - 1;
  print_int(y);
  return 0;
}
```

Expected output:
8

9.9.73 test-while1.hpg

```
int main() {
  int x = 0;

  while (x <= 2) {
    print ("True!");
    x = x + 1;
  }

  return 0;
}
```

Expected output:
True!
True!
True!

9.9.74 test-while2.hpg

```
int main() {
  int x = 0;

  while (x < 3) {
    print ("True!");
    x = x + 1;
  }

  return 0;
}
```

Expected output:
True!
True!
True!

9.10 Regression Test Suite - Negative Tests

9.10.1 fail-are-neighbors.hpg

```
int main() {
  graph<int, int> g = [1 <(5)> 2 -()> 3];
  print_bool(g.are_neighbors(1; "2"));
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found string expected int in 2”)

9.10.2 fail-create-graph1.hpg

```
int main() {
  graph<int:int, bool> g = ["foo"];
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal assignment graph¡int, int, bool¿ = graph¡string, bool, bool¿ in g = [nodes: [foo: null], edges: []]”)

9.10.3 fail-create-graph2.hpg

```
int main() {
  graph<int, bool> g = [1:2];
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal assignment graph¡int, bool, bool¿ = graph¡int, int, bool¿ in g = [nodes: [1: 2], edges: []]”)

9.10.4 fail-create-graph3.hpg

```
int main() {
  graph<int:int> g = [NULL:5];
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal assignment graph¡int, int, bool¿ = graph¡bool, int, bool¿ in g = [nodes: [null: 5], edges: []]”)

9.10.5 fail-create-graph4.hpg

```
int main() {
  graph<int:bool, int> g = [1:"foo"];
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal assignment graph¡int, bool, int¿ = graph¡int, string, bool¿ in g = [nodes: [1: foo], edges: []]”)

9.10.6 fail-create-graph5.hpg

```
int main() {
  graph<int:bool, int> g = [1:true <("foo")> 2:false];
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal assignment graph¡int, bool, int¿ = graph¡int, bool, string¿ in g = [nodes: [1: true, 2: false], edges: [(1, 2, foo), (2, 1, foo)]]”)

9.10.7 fail-create-graph6.hpg

```
int main() {
  graph<int:bool, int> g = [1:true <()> 2:"foo"];
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”type mismatch in graph nodes”)

9.10.8 fail-fdecls-argnum1.hpg

```
int main() {
  print_int(addone(41; 1));
  return 0;
}

int addone(int x) {
  return x + 1;
}
```

Expected error:
Fatal error: exception Failure(”expecting 1 arguments in addone(41, 1)”)

9.10.9 fail-fdecls-argtype1.hpg

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

int void_arg(void foo) {
  return 1;
}
```

Expected error:
Fatal error: exception Failure(”illegal void args foo”)

9.10.10 fail-fdecls-argtype2.hpg

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

int void_arg(void foo) {
  return 1;
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found string expected int in hello”)

9.10.11 fail-find-data.hpg

```
int main() {
  graph<string:int> g = ["a":10 -()> "b":20 <()> "c":20; "a" <()- "d"];
  graph<string:string> g_new = g.find("20");
  for_node(n : g_new) {
    n.print();
    print("");
  }
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found string expected int in 20”)

9.10.12 fail-for-edge1.hpg

```
int main() {
  int g = 0;
  for_edge(src, dst, w : g) {
    print("foo");
  }
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found: expected graph, got int”)

9.10.13 fail-for-edge2.hpg

```
int main() {
  graph<int:int, int> g = [1:1-(3)>2:2];
  for_edge(src, dst, w : g) {
    print_int(src);
  }
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found node¡int, int¿ expected int in src”)

9.10.14 fail-for-node1.hpg

```
int main() {
  int g = 0;
  for_node(n : g) {
    print_int(n);
  }
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found: expected graph, got int”)

9.10.15 fail-for-node2.hpg

```
int main() {
  int g = 0;
  for_node(n : g) {
    print_int(n);
  }
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found node¡int, int¿ expected int in n”)

9.10.16 fail-get-name.hpg

```
int main() {
  node<int> n = 1;
  print(n.get_name());
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found int expected string in n.get_name()”)

9.10.17 fail-get-node-data1.hpg

```
int main() {
  node<string:int> n1 = "abc":123;
  string d1 = n1.get_data();
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal assignment string = int in d1 = n1.get_data()”)

9.10.18 fail-get-weight.hpg

```
int main() {
  graph<int:string, string> g = [1:"foo" -("10")> 2:"bar" -()> 3];
  print(g.get_weight("1"; "2"));
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found string expected int in 1”)

9.10.19 fail-graph-neighbors1.hpg

```
int main() {
  graph<int, int> g = [];
  graph<int, string> g_sub = g.neighbors(5; 2; true);
}
```

Expected error:
Fatal error: exception Failure(”illegal assignment graph¡int, bool, string¿ = graph¡int, bool, int¿ in g_sub = g.neighbors(5, 2, true)”)

9.10.20 fail-graph-neighbors2.hpg

```
int main() {
  graph<int, int> g = [];
  graph<int, int> g_sub = g.neighbors("5"; "2"; true);
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found string expected int in 5”)

9.10.21 fail-if.hpg

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

Expected error:
Fatal error: exception Failure(”expected Boolean expression in 1”)

9.10.22 fail-localvars.hpg

```
int res;

int main() {
  res = 7;
  int x = res + 2;
  y = x - 1;
  print_int(y);
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”undeclared variable y”)

9.10.23 fail-nomain.hpg

Expected error:
Fatal error: exception Failure(”unrecognized function main”)

9.10.24 fail-print.hpg

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

Expected error:
Fatal error: exception Failure(”illegal argument found int expected string in 1”)

9.10.25 fail-printint.hpg

```
int main() {
  print_int("Hello, world");
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found string expected int in Hello, world”)

9.10.26 fail-remove-edge1.hpg

```
int main() {
  graph<int, int> g = [1 <(10)> 2; 3];
  g.remove_edge(1; "foo");
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found string expected int in foo”)

9.10.27 fail-remove-edge2.hpg

```
int main() {
  graph<int, int> g = [1 <(10)> 2; 3];
  string ret = g.remove_edge(1; 2);
}
```

Expected error:
Fatal error: exception Failure(”illegal assignment string = int in ret = g.remove_edge(1, 2)”)

9.10.28 fail-remove-node1.hpg

```
int main() {
  graph<int, int> g = [1 <(10)> 2; 3];
  int result1 = g.remove_node("1");
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found string expected int in 1”)

9.10.29 fail-set-data1.hpg

```
int main() {
  node<int:string> n = 1:"foo";
  n.set_data(1);
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found int expected string in 1”)

9.10.30 fail-set-edge1.hpg

```
int main() {
  graph<int:int, string> g = [1:1 <("foo")> 2:9];
  g.set_edge(1; 2; 1);
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found int expected string in 1”)

9.10.31 fail-set-node1.hpg

```
int main() {
  graph<int:int, int> g = [1:1];
  node<int:bool> n = 2:true;
  g.set_node(n);
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal argument found node¡int, bool¿ expected node¡int, int¿ in n”)

9.10.32 fail-var-scope.hpg

```
int main() {
  int a = 1;
  if (a == 2) {
    int b = 4;
    if (b == 4) {
      int c;
    }
    print_int(c);
  }
}
```

Expected error:
Fatal error: exception Failure(”undeclared variable c”)

9.10.33 fail-vdecls-global1.hpg

```
string a;

```

```
int main() {
  a = 4;
  return 0;
}
```

Expected error:
Fatal error: exception Failure(”illegal assignment string = int in a = 4”)

9.10.34 fail-vdecls-global2.hpg

```
int a;

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

Expected error:
Fatal error: exception Failure(”illegal argument found int expected string in a”)

9.10.35 fail-vdecls-local1.hpg

```
int main() {
  void foo;
}
```

Expected error:
Fatal error: exception Failure(”variable ’foo’ declared void”)