

Hippograph

The language for High Performance Parsing of Graphs

Benjamin Lewinter
Manager
`bsl12121`

Irina Mateescu
Language Guru
`im2441`

Harry Smith
System Architect
`hs3061`

Yasunari Watanabe
Test Expert
`yw3239`

December 19, 2018

Contents

1	Introduction	6
2	Tutorial	7
2.1	Basic Syntax	7
2.2	Graphs, Nodes and Edges	7
2.3	Graph Iterators	7
2.4	Hello, world!	7
3	Language Manual	9
3.1	Lexical Conventions	9
3.2	Data Types	9
3.2.1	Primitives	9
3.2.2	Reference Types	9
3.3	Variable Declaration and Assignment	10
3.3.1	Primitive Types	10
3.4	Graph Construction	11
3.4.1	Nodes	11
3.4.2	Edges	11
3.4.3	Graphs	12
3.5	Function Expressions	12
3.6	Arithmetic Operators	13
3.7	Boolean Operators	13
3.8	Comments	13
3.9	Control Flow	13
3.9.1	Conditionals	13
3.9.2	Loops	14
3.10	Program Structure	14
4	Project Plan - Ben	15
4.1	Workflow	15
4.2	Tools and Software	15
4.3	Code frequency graph	16
5	Language Evolution - Irina	17
6	Translator Architecture - Harry	18
7	Test Plan - Yasu	20
7.1	Test Structure and Automation	20
7.2	Examples	20
7.2.1	Positive Case: <code>test-graph-neighbors5.hpg</code>	20
7.2.2	Negative Case: <code>fail-for-node1.hpg</code>	23
8	Conclusions	24

9 Full Code	26
9.1 Scanner	26
9.1.1 scanner.mll	26
9.2 Parser	27
9.2.1 parser.mly	27
9.2.2 parseraux.ml	30
9.3 AST	31
9.3.1 ast.ml	31
9.4 SAST	34
9.4.1 sast.ml	34
9.5 Semantic Checking	36
9.5.1 semant.ml	36
9.6 Code Generation	44
9.6.1 codegen.ml	44
9.6.2 hippograph.ml	62
9.6.3 Makefile	63
9.6.4 run.sh	63
9.6.5 testall.sh	64
9.7 C library	67
9.7.1 graph.c	67
9.8 Example Code	86
9.8.1 Bellman-Ford Algorithm	86
9.8.2 Family Tree	87
9.9 Regression Test Suite - Positive Tests	88
9.9.1 test-anon-func.hpg	88
9.9.2 test-are-neighbors1.hpg	88
9.9.3 test-are-neighbors2.hpg	88
9.9.4 test-bellmanford.hpg	89
9.9.5 test-bfs-path.hpg	90
9.9.6 test-binop-ops.hpg	90
9.9.7 test-binop-ops-str.hpg	91
9.9.8 test-binop-prec.hpg	91
9.9.9 test-create-graph-type-bool.hpg	92
9.9.10 test-create-graph-type-int.hpg	92
9.9.11 test-create-graph-type-string.hpg	92
9.9.12 test-create-node.hpg	92
9.9.13 test-dfs-path.hpg	93
9.9.14 test-fdecls-argnum.hpg	93
9.9.15 test-fdecls-argtype.hpg	93
9.9.16 test-find-data-int1.hpg	94
9.9.17 test-find-data-int2.hpg	94
9.9.18 test-find-data-str.hpg	94
9.9.19 test-for-edge1.hpg	95
9.9.20 test-for-edge2.hpg	95
9.9.21 test-for-edge3.hpg	95
9.9.22 test-for-edge4.hpg	95
9.9.23 test-for-edge5.hpg	96
9.9.24 test-for-node1.hpg	96
9.9.25 test-for-node2.hpg	96

9.9.26	test-for-node3.hpg	97
9.9.27	test-for-node4.hpg	97
9.9.28	test-for.hpg	97
9.9.29	test-get-name1.hpg	98
9.9.30	test-get-node-data.hpg	98
9.9.31	test-get-node1.hpg	98
9.9.32	test-get-node2.hpg	99
9.9.33	test-get-node3.hpg	99
9.9.34	test-get-weight1.hpg	99
9.9.35	test-get-weight2.hpg	99
9.9.36	test-get-weight3.hpg	100
9.9.37	test-graph-neighbors1.hpg	100
9.9.38	test-graph-neighbors2.hpg	100
9.9.39	test-graph-neighbors3.hpg	100
9.9.40	test-graph-neighbors4.hpg	101
9.9.41	test-graph-neighbors5.hpg	101
9.9.42	test-has-node-bool.hpg	102
9.9.43	test-has-node-int.hpg	102
9.9.44	test-has-node-str.hpg	102
9.9.45	test-helloworld.hpg	102
9.9.46	test-if-else.hpg	103
9.9.47	test-if.hpg	103
9.9.48	test-is-empty.hpg	103
9.9.49	test-pass-graph-to-func.hpg	103
9.9.50	test-print-node.hpg	104
9.9.51	test-printbool.hpg	104
9.9.52	test-printint.hpg	104
9.9.53	test-recursion1.hpg	105
9.9.54	test-recursion2.hpg	105
9.9.55	test-remove-edge1.hpg	105
9.9.56	test-remove-node-bool.hpg	106
9.9.57	test-remove-node-int.hpg	106
9.9.58	test-remove-node-str.hpg	106
9.9.59	test-set-data1.hpg	106
9.9.60	test-set-edge-bool-int.hpg	107
9.9.61	test-set-edge-bool-str.hpg	107
9.9.62	test-set-edge-bool.hpg	107
9.9.63	test-set-edge-int-bool.hpg	107
9.9.64	test-set-edge-int-int.hpg	107
9.9.65	test-set-edge-int-str.hpg	108
9.9.66	test-set-edge-int.hpg	108
9.9.67	test-set-edge-str-bool.hpg	108
9.9.68	test-set-edge-str-str.hpg	108
9.9.69	test-set-edge-str.hpg	108
9.9.70	test-set-node1.hpg	109
9.9.71	test-vdecls-global.hpg	109
9.9.72	test-vdecls.hpg	109
9.9.73	test-while1.hpg	109
9.9.74	test-while2.hpg	110

9.10	Regression Test Suite - Negative Tests	110
9.10.1	fail-are-neighbors.hpg	110
9.10.2	fail-create-graph1.hpg	110
9.10.3	fail-create-graph2.hpg	110
9.10.4	fail-create-graph3.hpg	111
9.10.5	fail-create-graph4.hpg	111
9.10.6	fail-create-graph5.hpg	111
9.10.7	fail-create-graph6.hpg	111
9.10.8	fail-fdecls-argnum1.hpg	111
9.10.9	fail-fdecls-argtype1.hpg	112
9.10.10	fail-fdecls-argtype2.hpg	112
9.10.11	fail-find-data.hpg	112
9.10.12	fail-for-edge1.hpg	112
9.10.13	fail-for-edge2.hpg	113
9.10.14	fail-for-node1.hpg	113
9.10.15	fail-for-node2.hpg	113
9.10.16	fail-get-name.hpg	113
9.10.17	fail-get-node-data1.hpg	113
9.10.18	fail-get-weight.hpg	114
9.10.19	fail-graph-neighbors1.hpg	114
9.10.20	fail-graph-neighbors2.hpg	114
9.10.21	fail-if.hpg	114
9.10.22	fail-localvars.hpg	114
9.10.23	fail-nomain.hpg	115
9.10.24	fail-print.hpg	115
9.10.25	fail-printint.hpg	115
9.10.26	fail-remove-edge1.hpg	115
9.10.27	fail-remove-edge2.hpg	115
9.10.28	fail-remove-node1.hpg	115
9.10.29	fail-set-data1.hpg	116
9.10.30	fail-set-edge1.hpg	116
9.10.31	fail-set-node1.hpg	116
9.10.32	fail-var-scope.hpg	116
9.10.33	fail-vdecls-global1.hpg	116
9.10.34	fail-vdecls-global2.hpg	117
9.10.35	fail-vdecls-local1.hpg	117

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:

```
1 int gcd(int a; int b) {
2     while (a != b) {
3         if (a > b) {
4             a = a - b;
5         }
6         else {
7             b = b - a;
8         }
9     }
10    return a;
11 }
12
13 int main() {
14     print_int(gcd(2; 12));
15 }
```

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:

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

An example with int edge weights declared:

```
1 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):

```
1 node<string:int> x = "D":4;
2 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.

```
1 for_node( m : graph ) { }
2 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!

```
1 int main() {
2     graph<int:string> hello = [0:"H" -()> 1:"E" -()> 2:"L" -()> 3:"L" -()> 4:"O" -()>
3         5:" " -()> 6:"World!"];
4     for_node (i : hello) {
5         print(i.get_data());
6     }
7 }
```

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`.

```
1     graph<int, int> g = [1 -(3)- 4; 6; 8];
2     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.

```

1 (* type declaration only *)
2 bool val1;
3 int val2;
4 string val4;
5
6 (* combined type declaration and value assignment *)
7 bool val1 = true;
8 int val2 = 42;
9 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.

```

1 node<int:int> n1;      (* node with name of int type and data of int type *)
2 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.

```

1 (* both nodes with name of int type and data of bool type *)
2 node<int:> n3;
3 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.

```

1 node<int:string> n5 = 123:NULL; (* explicit null string *)
2 node<int:string> n6 = 123;       (* inferred null string *)
3 node<int> n7 = 123:NULL;        (* explicit null bool *)
4 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.

```
1 (* all graphs with node name of int type, node data/edge weight of bool type *)
2 graph<int:bool, bool> g1;
3 graph<int, bool> g2;
4 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.

```
1 (* all graphs with a directed edge from 1 to 2 and another from 2 to 1 *)
2 graph<int, string> g4 = [1-("a")-2];
3 graph<int, string> g5 = [1<"a">2];
4 graph<int, string> g6 = [1<"a">2; 1-("a")>2];
5 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.

```
1 while (condition) {statement}
2 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

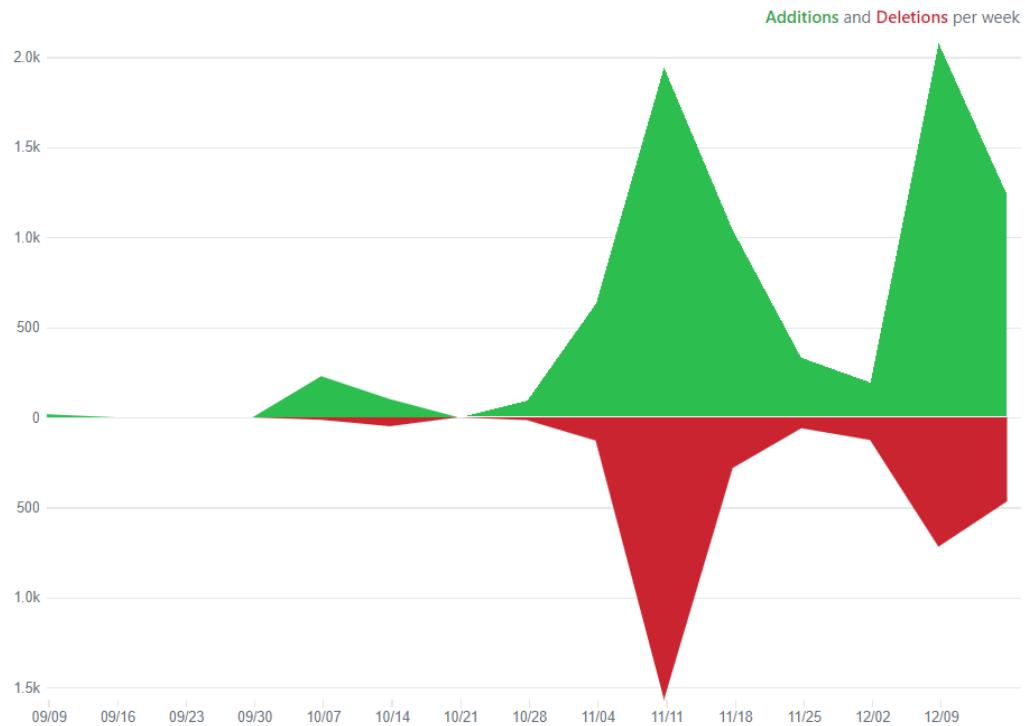
Languages: OCaml and C

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

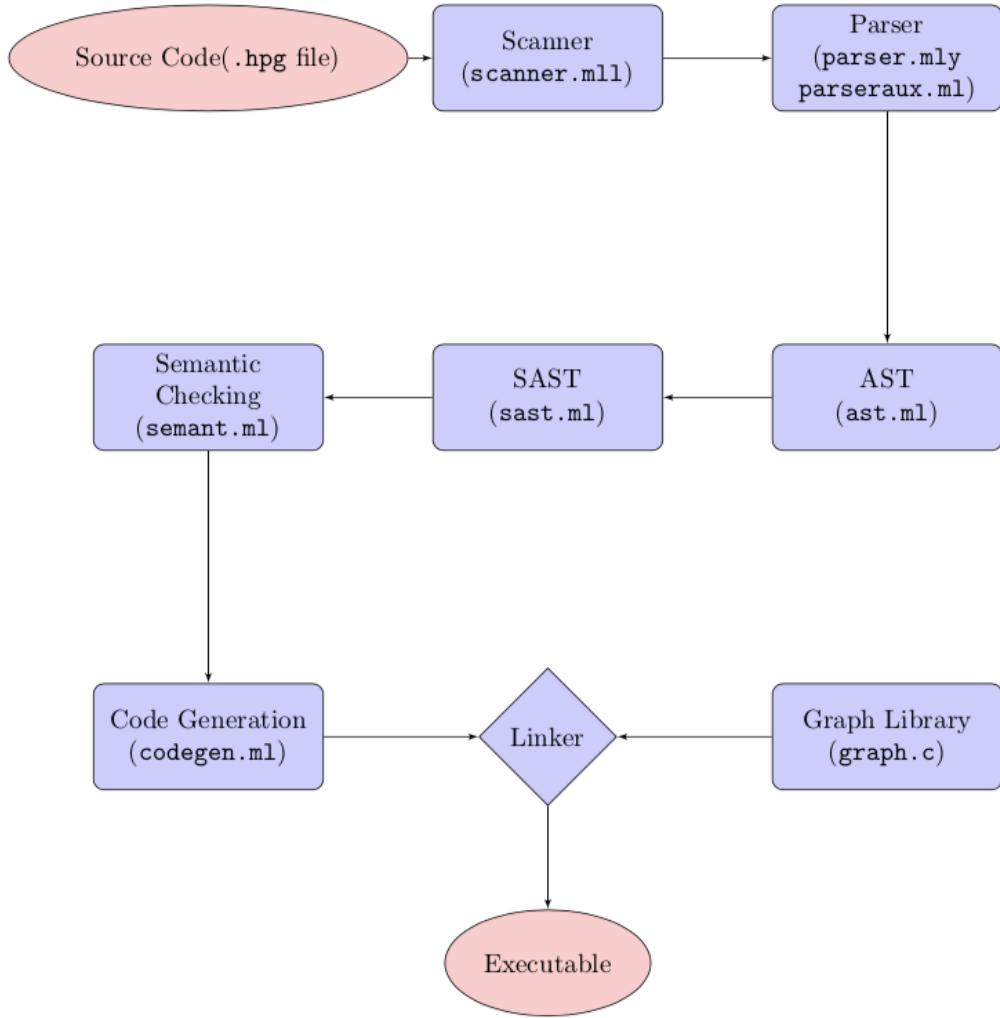


Figure 1: Compiler Architecture

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

```
1 int main() {
2     graph<int> g = [1 -() - 2 -() - 3 -() - 4 -() - 2];
3     graph<int> g_sub1 = g.neighbors(1, 4, false);
4     for_node (n : g_sub1) {
5         n.print();
6         print("");
7     }
8     print("");
9     graph<int> g_sub2 = g.neighbors(1, 4, true);
10    for_node (n : g_sub2) {
11        n.print();
12        print("");
13    }
14 }
```

LLVM IR

```
1 ; ModuleID = 'Hippograph'
2 source_filename = "Hippograph"
3
4 ;;; C library function declarations are omitted for brevity
5
6 define i32 @main() {
7 entry:
8     %g = alloca i8*
9     %create_graph = call i8* (...) @create_graph()
10    %create_node = call i8* (...) @create_node()
11    call void (i8*, i32, ...) @set_node_label_int(i8* %create_node, i32 1)
12    call void (i8*, i1, i1, ...) @set_node_data_bool(i8* %create_node, i1 false, i1
13                                false)
13    %add_node = call i8* (i8*, i8*, ...) @add_node(i8* %create_graph, i8* %create_node
14                                )
14    %create_node1 = call i8* (...) @create_node()
```

```

15  call void (i8*, i32, ...) @set_node_label_int(i8* %create_node1, i32 2)
16  call void (i8*, i1, i1, ...) @set_node_data_bool(i8* %create_node1, i1 false, i1
     false)
17  %add_node2 = call i8* (i8*, i8*, ...) @add_node(i8* %create_graph, i8* %
     create_node1)
18  %create_node3 = call i8* (...) @create_node()
19  call void (i8*, i32, ...) @set_node_label_int(i8* %create_node3, i32 3)
20  call void (i8*, i1, i1, ...) @set_node_data_bool(i8* %create_node3, i1 false, i1
     false)
21  %add_node4 = call i8* (i8*, i8*, ...) @add_node(i8* %create_graph, i8* %
     create_node3)
22  %create_node5 = call i8* (...) @create_node()
23  call void (i8*, i32, ...) @set_node_label_int(i8* %create_node5, i32 4)
24  call void (i8*, i1, i1, ...) @set_node_data_bool(i8* %create_node5, i1 false, i1
     false)
25  %add_node6 = call i8* (i8*, i8*, ...) @add_node(i8* %create_graph, i8* %
     create_node5)
26  %edge = call i8* (...) @create_edge()
27  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge, i1 false, i1 false)
28  %add_edge = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph, i8
     * %edge, i32 1, i32 2)
29  %edge7 = call i8* (...) @create_edge()
30  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge7, i1 false, i1 false)
31  %add_edge8 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph,
     i8* %edge7, i32 2, i32 1)
32  %edge9 = call i8* (...) @create_edge()
33  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge9, i1 false, i1 false)
34  %add_edge10 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph,
     i8* %edge9, i32 2, i32 3)
35  %edge11 = call i8* (...) @create_edge()
36  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge11, i1 false, i1 false)
37  %add_edge12 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph,
     i8* %edge11, i32 3, i32 2)
38  %edge13 = call i8* (...) @create_edge()
39  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge13, i1 false, i1 false)
40  %add_edge14 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph,
     i8* %edge13, i32 3, i32 4)
41  %edge15 = call i8* (...) @create_edge()
42  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge15, i1 false, i1 false)
43  %add_edge16 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph,
     i8* %edge15, i32 4, i32 3)
44  %edge17 = call i8* (...) @create_edge()
45  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge17, i1 false, i1 false)
46  %add_edge18 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph,
     i8* %edge17, i32 4, i32 2)
47  %edge19 = call i8* (...) @create_edge()
48  call void (i8*, i1, i1, ...) @set_edge_w_bool(i8* %edge19, i1 false, i1 false)
49  %add_edge20 = call i8* (i8*, i8*, i32, i32, ...) @add_edge_int(i8* %create_graph,
     i8* %edge19, i32 2, i32 4)
50  store i8* %create_graph, i8** %g
51  %g_sub1 = alloca i8*
52  %g21 = load i8*, i8** %g
53  %get_node_by_label_int = call i8* (i8*, i32, ...) @get_node_by_label_int(i8* %g21,
     i32 1)
54  %neighbors = call i8* (i8*, i32, i1, ...) @neighbors(i8* %get_node_by_label_int,
     i32 4, i1 false)
55  store i8* %neighbors, i8** %g_sub1
56  %g_sub122 = load i8*, i8** %g_sub1
57  %n = alloca i8*
58  %hd_node = call i8* (i8*, ...) @graph_to_node_iterable(i8* %g_sub122)
59  store i8* %hd_node, i8** %n

```

```

60     br label %while
61
62 while:                                ; preds = %while_body, %entry
63     %node_tmp = load i8*, i8** %n
64     %bool_val = icmp ne i8* %node_tmp, null
65     br i1 %bool_val, label %while_body, label %merge
66
67 while_body:                            ; preds = %while
68     %n23 = load i8*, i8** %n
69     call void (i8*, ...) @print_node(i8* %n23)
70     call void (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt,
71         i32 0, i32 0), i8* getelementptr inbounds ([1 x i8], [1 x i8]* @str, i32 0,
72         i32 0))
73     %curr_node = load i8*, i8** %n
74     %next_node = call i8* (i8*, ...) @get_graph_next_node(i8* %curr_node)
75     store i8* %next_node, i8** %n
76     br label %while
77
78 merge:                                ; preds = %while
79     call void (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt,
80         i32 0, i32 0), i8* getelementptr inbounds ([1 x i8], [1 x i8]* @str.2, i32 0,
81         i32 0))
82     %g_sub2 = alloca i8*
83     %g24 = load i8*, i8** %g
84     %get_node_by_label_int25 = call i8* (i8*, i32, ...) @get_node_by_label_int(i8* %
85         g24, i32 1)
86     %neighbors26 = call i8* (i8*, i32, i1, ...) @neighbors(i8* %
87         get_node_by_label_int25, i32 4, i1 true)
88     store i8* %neighbors26, i8** %g_sub2
89     %g_sub227 = load i8*, i8** %g_sub2
90     %n28 = alloca i8*
91     %hd_node29 = call i8* (i8*, ...) @graph_to_node_iterable(i8* %g_sub227)
92     store i8* %hd_node29, i8** %n28
93     br label %while30
94
95 while30:                            ; preds = %while_body31, %merge
96     %node_tmp35 = load i8*, i8** %n28
97     %bool_val36 = icmp ne i8* %node_tmp35, null
98     br i1 %bool_val36, label %while_body31, label %merge37
99
100 while_body31:                         ; preds = %while30
101     %n32 = load i8*, i8** %n28
102     call void (i8*, ...) @print_node(i8* %n32)
103     call void (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt,
104         i32 0, i32 0), i8* getelementptr inbounds ([1 x i8], [1 x i8]* @str.3, i32 0,
105         i32 0))
106     %curr_node33 = load i8*, i8** %n28
107     %next_node34 = call i8* (i8*, ...) @get_graph_next_node(i8* %curr_node33)
108     store i8* %next_node34, i8** %n28
109     br label %while30
110
111 merge37:                            ; preds = %while30
112     ret i32 0
113 }
```

Output

```

1 2:null
2 3:null
3 4:null
4
5 1:null
```

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

7.2.2 Negative Case: fail-for-node1.hpg

Source

```
1 int main() {  
2     int g = 0;  
3     for_node(n : g) {  
4         print_int(n);  
5     }  
6 }
```

Error

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

```
1 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 its 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

```
1 (* Authors:
2   Benjamin Lewinter bsl2121
3   Irina Mateescu     im2441
4   Harry Smith        hs3061
5   Yasunari Watanabe yw3239
6 *)
7
8 { open Parser }
9
10 rule token = parse
11   [ ' ' '\t' '\r' '\n'] { token lexbuf }
12 | '+' { PLUS }
13 | '-' { MINUS }
14 | '*' { TIMES }
15 | '/' { DIVIDE }
16 | ';' { SEQUENCE }
17 | '=' { ASSIGN }
18 | '{' { LBRACE }
19 | '}' { RBRACE }
20 | '.' { DOT }
21 | ',' { COMMA }
22 | '(' { LPAREN }
23 | ')' { RPAREN }
24 | '<' { LANGLE }
25 | '>' { RANGLE }
26 | '[' { LBRAK }
27 | ']' { RBRAK }
28 | '\'' { SQUOTE }
29 | '\"' { DQUOTE }
30 | ':' { COLON }
31 | "==" { EQ }
32 | "!=" { NEQ }
33 | "<=" { LEQ }
34 | ">=" { GEQ }
35 | "and" { AND }
36 | "or" { OR }
37 | "not" { NOT }
38 | "int" { INTTYPE }
39 | "bool" { BOOLTYPE }
40 | "string" { STRINGTYPE }
41 | "fun" { FUNTYPE }
42 | "void" { VOIDTYPE }
43 | "graph" { GRAPHTYPE }
44 | "node" { NODETYPE }
45 | "-(" { LUEDGE }
46 | ")-" { RUEDGE }
47 | "<(" { LDEdge }
48 | ")>" { RDEdge }
49 | "if" { IF }
50 | "else" { ELSE }
51 | "while" { WHILE }
52 | "for" { FOR }
53 | "for_node" { FORNODE }
54 | "for_edge" { FOREDGE }
```

```

55 | "in"      { IN }
56 | "NULL"    { NULL }
57 | "return"   { RETURN }
58 | ['0'-'9']+ as int_lit           { INTLIT(int_of_string int_lit) }
59 | '\"' ([^\""]* as string_lit) '\"' { STRINGLIT(string_lit) }
60 | ("true" | "false") as bool_lit  { BOOLLIT(bool_of_string bool_lit) }
61 | ['a'-'z' 'A'-'Z'][ '0'-'9' 'a'-'z' 'A'-'Z' '_']* as id { VARIABLE(id) }
62 | eof { EOF }
63 | "(*)" { comment lexbuf }
64 and comment = parse
65   "*" { token lexbuf }
66   | _ { comment lexbuf }

```

9.2 Parser

9.2.1 parser.mly

```

1  /* Authors:
2   Benjamin Lewinter bsl2121
3   Irina Mateescu im2441
4   Harry Smith hs3061
5   Yasunari Watanabe yw3239
6 */
7
8 %{
9  open Ast
10 open Parseraux
11 %}
12
13 %token PLUS MINUS TIMES DIVIDE SEQUENCE ASSIGN EOF
14 %token LBRACE RBRACE DOT COMMA LPAREN RPAREN LANGLE RANGLE LBRAK RBRAK SQUOTE DQUOTE
15 %token COLON
16 %token EQ NEQ LEQ GEQ AND OR NOT
17 %token INTTYPE BOOLTYPE STRINGTYPE FUNTYPE GRAPHTYPE NODETYPE VOIDTYPE
18 %token LUEDGE RUEDGE LDEDGE RDEDGE
19 %token IF ELSE NOELSE WHILE FOR FORNODE FOREdge IN NULL RETURN
20 %token <int> INTLIT
21 %token <string> STRINGLIT
22 %token <bool> BOOLLIT
23 %token <string> VARIABLE
24
25 %left SEQUENCE
26 %left DOT
27 %right ASSIGN
28 %left AND OR
29 %right COLON
30 %nonassoc EQ NEQ
31 %nonassoc LEQ GEQ LANGLE RANGLE
32 %left PLUS MINUS
33 %left TIMES DIVIDE
34 %right NEG NOT
35 %nonassoc LPAREN RPAREN
36 %nonassoc NOELSE
37 %nonassoc ELSE
38
39 %start program
40 %type <Ast.program> program
41 %%
42
43 program: decls EOF { $1 }

```

```

44
45 decls:
46   { [], [] }
47 | decls vdecl { ($2 :: fst $1), snd $1 }
48 | decls fdecl { fst $1, ($2 :: snd $1) }
49
50 vdecl:
51   typ VARIABLE SEQUENCE { ($1, $2) }
52
53 fdecl:
54   typ VARIABLE LPAREN args_opt RPAREN LBRACE stmt_list RBRACE
55     { { typ = $1; fname = $2; args = $4; body = List.rev $7 } }
56
57 args_opt:
58   { [] }
59 | args_list { List.rev $1 }
60
61 args_list:
62   typ VARIABLE           { [($1, $2)] }
63 | args_list SEQUENCE typ VARIABLE { ($3, $4) :: $1 }
64
65
66
67 actuals_opt:
68   { [] }
69 | actuals_list { List.rev $1 }
70
71 actuals_list:
72   expr           { [$1] }
73 | actuals_list SEQUENCE expr { $3 :: $1 }
74
75 typ:
76   VOIDTYPE    { Void }
77 | INTTYPE     { Int }
78 | BOOLTYPE    { Bool }
79 | STRINGTYPE { String }
80 | FUNTYPE LANGLE typ COLON typ_list_opt RANGLE      { Fun($3, $5) }
81 | GRAPHTYPE LANGLE typ COLON typ COMMA typ RANGLE { Graph($3, $5, $7) }
82 | GRAPHTYPE LANGLE typ COMMA typ RANGLE            { Graph($3, Bool, $5) }
83 | GRAPHTYPE LANGLE typ COLON typ RANGLE           { Graph($3, $5, Bool) }
84 | GRAPHTYPE LANGLE typ RANGLE                     { Graph($3, Bool, Bool) }
85 | NODETYPE   LANGLE typ COLON typ RANGLE           { Node($3, $5) }
86 | NODETYPE   LANGLE typ RANGLE                     { Node($3, Bool) }
87
88 typ_list_opt:
89   { [] }
90 | typ_list { List.rev $1 }
91
92 typ_list:
93   typ           { [$1] }
94 | typ_list COMMA typ { $3 :: $1 }
95
96 stmt_list:
97   { [] }
98 | stmt_list stmt { $2 :: $1 }
99
100 stmt:
101   expr SEQUENCE           { Expr $1 }
102 | FOR LPAREN expr SEQUENCE expr SEQUENCE expr RPAREN stmt { For($3, $5, $7, $9) }
103 | FORNODE LPAREN VARIABLE COLON VARIABLE RPAREN stmt       { ForNode($3, Var($5), $7)
    ) }

```

```

104 | FOREdge LPAREN VARIABLE COMMA VARIABLE COMMA VARIABLE COLON VARIABLE RPAREN stmt
     { ForEdge($3, $5, $7, Var($9), $11) }
105 | WHILE LPAREN expr RPAREN stmt                                { While($3, $5) }
106 | IF LPAREN expr RPAREN stmt %prec NOELSE                      { If($3, $5, Block([])) }
107 | IF LPAREN expr RPAREN stmt ELSE stmt                         { If($3, $5, $7) }
108 | LBRACE stmt_list RBRACE                                     { Block(List.rev $2) }
109 | typ VARIABLE SEQUENCE                                     { Vdecl($1, $2, Noexpr) }
110 | typ VARIABLE ASSIGN expr SEQUENCE                         { Vdecl($1, $2, Asn($2,
     $4)) }
111 | RETURN SEQUENCE                                         { Return Null }
112 | RETURN expr SEQUENCE                                    { Return $2 }
113
114 expr:
115   INTLIT                           { Intlit($1) }
116   STRINGLIT                        { Stringlit($1) }
117   BOOLLIT                           { Boollit($1) }
118   typ LPAREN args_opt RPAREN LPAREN expr RPAREN { Funsig($1, $3, $6) }
119   NULL                             { Null }
120   VARIABLE                          { Var($1) }
121   LPAREN expr RPAREN               { $2 }
122   expr PLUS      expr             { Binop($1, Add, $3) }
123   expr MINUS      expr             { Binop($1, Sub, $3) }
124   expr TIMES      expr             { Binop($1, Mul, $3) }
125   expr DIVIDE      expr            { Binop($1, Div, $3) }
126   expr AND      expr              { Binop($1, And, $3) }
127   expr OR       expr              { Binop($1, Or, $3) }
128   expr EQ        expr              { Binop($1, Eq, $3) }
129   expr NEQ       expr              { Binop($1, Neq, $3) }
130   expr LEQ       expr              { Binop($1, Leq, $3) }
131   expr GEQ       expr              { Binop($1, Geq, $3) }
132   expr LANGLE     expr             { Binop($1, Lt, $3) }
133   expr RANGLE     expr             { Binop($1, Gt, $3) }
134   expr MINUS      %prec NEG    { Unop(Neg, $2) }
135   expr NOT        expr              { Unop(Not, $2) }
136   VARIABLE ASSIGN expr           { Asn($1, $3) }
137   VARIABLE LPAREN actuals_opt RPAREN { FCall($1, $3) }
138   expr DOT VARIABLE LPAREN actuals_opt RPAREN { MCall($1, $3, $5) }
139   expr COLON      expr              { NodeExpr($1, $3) }
140   LBRAK graph_item_opt RBRAK          { match $2 with (node_list, edge_list)
     ->
141                                         GraphExpr(node_list, edge_list) }
142
143 graph_item_opt:
144   [] , []
145 | graph_item_list { match $1 with (node_list, edge_list) ->
146   List.rev node_list, List.rev edge_list }
147
148 graph_item_list:
149   node_edge_list                  { $1 }
150 | graph_item_list SEQUENCE        { $1 }
151 | graph_item_list SEQUENCE node_edge_list { merge_node_edge_lists $1 $3 }
152
153 node_edge_list:
154   expr                            { [construct_node_expr $1], [] }
155 | node_edge_list LUEDGE expr RUEDGE expr { update_node_edge_list_with_edge $1 $3 (
     construct_node_expr $5) }
156 | node_edge_list LDEDGE expr RDEDGE expr { update_node_edge_list_with_edge $1 $3 (
     construct_node_expr $5) }
157 | node_edge_list LUEDGE expr RDEDGE expr { update_node_edge_list_with_redge $1 $3 (
     construct_node_expr $5) }
158 | node_edge_list LDEDGE expr RUEDGE expr { update_node_edge_list_with_ledge $1 $3 (

```

```

        construct_node_expr $5) }
159 | node_edge_list LUEDGE RUEDGE expr      { update_node_edge_list_with_edge $1 Null (
    construct_node_expr $4) }               { update_node_edge_list_with_edge $1 Null (
160 | node_edge_list LDEdge RDEdge expr      { update_node_edge_list_with_redge $1 Null
    construct_node_expr $4) }               { update_node_edge_list_with_redge $1 Null
161 | node_edge_list LUEDGE RDEdge expr      { update_node_edge_list_with_ledge $1 Null
    (construct_node_expr $4) }              { update_node_edge_list_with_ledge $1 Null
162 | node_edge_list LDEdge RUEDGE expr      { update_node_edge_list_with_ledge $1 Null
    (construct_node_expr $4) }

```

9.2.2 parseraux.ml

```

1 open Ast;;
2
3 let construct_node_expr expr =
4   (* if already a NodeExpr, keep it that way; otherwise create a NodeExpr with Null
5    data *)
6   match expr with
7   | NodeExpr(_, _) -> expr
8   | _ -> NodeExpr(expr, Null)
8 ;;
9
10 let node_list_append_opt n_expr n_list =
11   (* add to list if key doesn't exist; ignore otherwise *)
12   let label, _ = unwrap_node_expr n_expr in
13   if List.exists (fun n -> let l, _ = unwrap_node_expr n in l = label) n_list
14   then n_list
15   else n_expr :: n_list
16 ;;
17
18 let edge_list_append_opt e_expr e_list =
19   (* add to list if src-dst pair doesn't exist; ignore otherwise regardless of
      weight *)
20   let src, dst, _ = unwrap_edge_expr e_expr in
21   if List.exists (fun e -> let s, d, _ = unwrap_edge_expr e in s = src && d = dst)
22     e_list
23   else e_expr :: e_list
24 ;;
25
26 let update_node_edge_list_with_edge (n_list, e_list) weight n_expr =
27   let prev_n_label, _ = unwrap_node_expr (List.hd n_list) in
28   let n_label, _ = unwrap_node_expr n_expr in
29   let n_list' = node_list_append_opt n_expr n_list in
30   let e_list' = edge_list_append_opt (EdgeExpr(n_label, prev_n_label, weight))
31           (edge_list_append_opt (EdgeExpr(prev_n_label,
32                                         n_label, weight))
33                                         e_list)
34   in
35   (n_list', e_list')
36 ;;
37
38 let update_node_edge_list_with_redge (n_list, e_list) weight n_expr =
39   let prev_n_label, _ = unwrap_node_expr (List.hd n_list) in
40   let n_label, _ = unwrap_node_expr n_expr in
41   let n_list' = node_list_append_opt n_expr n_list in
42   let e_list' = edge_list_append_opt (EdgeExpr(prev_n_label, n_label, weight))
43           e_list in
44   (n_list', e_list')
43 ;;
44

```

```

45 let update_node_edge_list_with_ledge (n_list, e_list) weight n_expr =
46   let prev_n_label, _ = unwrap_node_expr (List.hd n_list) in
47   let n_label, _ = unwrap_node_expr n_expr in
48   let n_list' = node_list_append_opt n_expr n_list in
49   let e_list' = edge_list_append_opt (EdgeExpr(n_label, prev_n_label, weight))
      e_list in
50   (n_list', e_list')
51 ;;
52
53 let merge_node_edge_lists (n_list1, e_list1) (n_list2, e_list2) =
54   let n_list' = List.fold_right node_list_append_opt n_list2 n_list1 in
55   let e_list' = List.fold_right edge_list_append_opt e_list2 e_list1 in
56   (n_list', e_list')
57 ;;

```

9.3 AST

9.3.1 ast.ml

```

1 (* Authors:
2   Benjamin Lewinter bsl2121
3   Irina Mateescu im2441
4   Harry Smith hs3061
5   Yasunari Watanabe yw3239
6 *)
7
8 type typ =
9   | Int
10  | Fun of typ * (typ list)
11  | String
12  | Bool
13  | Void
14  | Graph of typ * typ * typ
15  | Node of typ * typ
16  | Edge of typ
17
18 type binding = typ * string
19
20 type binop =
21  | Add
22  | Sub
23  | Mul
24  | Div
25  | And
26  | Or
27  | Eq
28  | Neq
29  | Leq
30  | Geq
31  | Lt
32  | Gt
33
34 type unop =
35  | Not
36  | Neg
37
38 type expr =
39  | Intlit of int
40  | Stringlit of string
41  | Boollit of bool
42  | Funsig of typ * binding list * expr

```

```

43  | Null
44  | Var of string
45  | Binop of expr * binop * expr
46  | Unop of unop * expr
47  | Asn of string * expr
48  | FCall of string * expr list
49  | MCall of expr * string * expr list
50  | NodeExpr of expr * expr
51  | EdgeExpr of expr * expr * expr
52  | GraphExpr of node_list * edge_list
53  | Noexpr
54 and node_list = expr list
55 and edge_list = expr list
56
57 type stmt =
58  | Expr of expr
59  | For of expr * expr * expr * stmt
60  | ForNode of string * expr * stmt
61  | ForEdge of string * string * string * expr * stmt
62  | While of expr * stmt
63  | If of expr * stmt * stmt
64  | Block of stmt list
65  | Vdecl of typ * string * expr
66  | Return of expr
67
68 type fdecl = {typ: typ; fname:string; args:binding list; body:stmt list}
69
70 type program = binding list * fdecl list
71
72 exception Unsupported_constructor;;
73
74 let unwrap_node_expr n_expr =
75   match n_expr with
76   | NodeExpr(label, data) -> (label, data)
77   | _ -> raise Unsupported_constructor
78 ;;
79
80 let unwrap_edge_expr e_expr =
81   match e_expr with
82   | EdgeExpr(src, dst, w) -> (src, dst, w)
83   | _ -> raise Unsupported_constructor
84 ;;
85
86 let string_of_op = function
87   | Add -> "+"
88   | Sub -> "-"
89   | Mul -> "*"
90   | Div -> "/"
91   | Eq -> "=="
92   | Neq -> "!="
93   | Lt -> "<"
94   | Leq -> "<="
95   | Gt -> ">"
96   | Geq -> ">="
97   | And -> "&&"
98   | Or -> "||"
99
100 let string_of_uop = function
101   | Not -> "!"
102   | Neg -> "-"
103

```

```

104
105 let rec string_of_typ = function
106   | Int      -> "int"
107   | Bool     -> "bool"
108   | Void     -> "void"
109   | Fun(_)   -> "fun"
110   | String   -> "string"
111   | Node(nl, nd) -> "node<" ^ string_of_typ nl ^ ", " ^ string_of_typ nd ^ ">"
112   | Edge(vl)    -> "edge<" ^ string_of_typ vl ^ ">"
113   | Graph(nl, nd, ew) -> "graph<" ^ string_of_typ nl ^ ", " ^ string_of_typ nd ^ ", "
114   |           " " ^ string_of_typ ew ^ ">""
115
116 let string_of_vdecl (t, var) = string_of_typ t ^ " " ^ var ^ "; "
117
118 let rec string_of_expr = function
119   | Intlit(l) -> string_of_int l
120   | Boollit(true) -> "true"
121   | Boollit(false) -> "false"
122   | Var(s) -> s
123   | Stringlit(l) -> l
124   | Funsig(typ, bl, e) -> " fun: " ^ (string_of_typ typ) ^ " (" ^ (String.concat ""
125   |           (List.map string_of_vdecl bl)) ^ ") { " ^ string_of_expr e ^ " }"
126   | Null -> "null"
127   | Binop(e1, o, e2) ->
128     string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
129   | Unop(o, e) -> string_of_uop o ^ string_of_expr e
130   | Asn(v, e) -> v ^ " = " ^ string_of_expr e
131   | FCall(f, el) ->
132     f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
133   | MCall(caller, f, el) ->
134     string_of_expr caller ^ "." ^ f ^ "(" ^ String.concat ", " (List.map
135     string_of_expr el) ^ ")"
136   | NodeExpr(e1, e2) -> string_of_expr e1 ^ ":" ^ string_of_expr e2
137   | EdgeExpr(src, dst, w) -> "(" ^ (string_of_expr src) ^ ", " ^ (string_of_expr dst
138   |           ) ^ ", " ^ (string_of_expr w) ^ ")"
139   | GraphExpr(node_list, edge_list) ->
140     "[nodes: [" ^ String.concat ", " (List.map string_of_expr node_list) ^
141     "], edges: [" ^ String.concat ", " (List.map string_of_expr edge_list) ^ "]]"
142   | Noexpr -> ""
143
144 let string_of_vdecl (t, var) = string_of_typ t ^ " " ^ var ^ ";\n"
145
146 let rec string_of_stmt = function
147   | Block(stmts) ->
148     "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
149   | Expr(expr) -> string_of_expr expr ^ ";\n";
150   | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
151   | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
152   | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n" ^
153     string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
154   | For(e1, e2, e3, s) ->
155     "for (" ^ string_of_expr e1 ^ " ; " ^ string_of_expr e2 ^ " ; " ^
156     string_of_expr e3 ^ ") " ^ string_of_stmt s
157   | ForNode(n, g, body) ->
158     "for (" ^ n ^ " : " ^ string_of_expr g ^ ") " ^ string_of_stmt body
159   | ForEdge(src, dst, w, g, body) ->
160     "for (" ^ src ^ ", " ^ dst ^ ", " ^ w ^ " : " ^ string_of_expr g ^ ") " ^
161     string_of_stmt body
162   | While(e, body) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt body
163   | Vdecl(t, var, expr) ->

```

```

160     match expr with
161     | Noexpr ->
162         string_of_vdecl (t, var)
163     | _ ->
164         string_of_typ t ^ " " ^ string_of_expr expr ^ ";\n"
165
166
167 let string_of_vdecl (t, var) = string_of_typ t ^ " " ^ var ^ ";\n"
168
169 let string_of_fdecl fdecl =
170   string_of_typ fdecl.typ ^ " " ^
171   fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.args) ^
172   ")\n\n" ^
173   String.concat "" (List.map string_of_stmt fdecl.body) ^
174   "}\n"
175
176 let string_of_program (vars, funcs) =
177   String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
178   String.concat "\n" (List.map string_of_fdecl funcs)

```

9.4 SAST

9.4.1 sast.ml

```

1 (* Authors:
2 Benjamin Lewinter bsl2121
3 Irina Mateescu im2441
4 Harry Smith hs3061
5 Yasunari Watanabe yw3239
6 *)
7 (*Semantically checked abstract syntax tree*)
8
9 open Ast
10
11 type sexpr = typ * sx
12 and sx =
13   SIntlit of int
14 | SStringlit of string
15 | SBoollit of bool
16 | SFunsig of typ * binding list * sexpr
17 | SNull
18 | SVar of string
19 | SBinop of sexpr * binop * sexpr
20 | SUNop of unop * sexpr
21 | SASn of string * sexpr
22 | SFCall of string * sexpr list
23 | SMCall of sexpr * string * sexpr list
24 | SNodeExpr of sexpr * sexpr
25 | SEdgeExpr of sexpr * sexpr * sexpr
26 | SGraphExpr of node_list * edge_list
27 | SNoexpr
28 and node_list = sexpr list
29 and edge_list = sexpr list
30
31 type sstmt =
32   SExpr of sexpr
33 | SFor of sexpr * sexpr * sexpr * sstmt
34 | SForNode of string * sexpr * sstmt
35 | SForEdge of string * string * string * sexpr * sstmt
36 | SWhile of sexpr * sstmt
37 | SIf of sexpr * sstmt * sstmt

```

```

38 | SBlock of sstmt list
39 | SVdecl of typ * string * sexpr
40 | SReturn of sexpr
41
42 type sfdecl = {
43   styp: typ;
44   sfname: string;
45   sargs: binding list;
46   sbody: sstmt list;
47 }
48
49 type sprogram = binding list * sfdecl list
50
51 let string_of_svdecl (t, var) = string_of_typ t ^ " " ^ var ^ "; "
52
53 let rec string_of_sexpr (t, e) =
54   match e with
55   | SIntlit(l) -> string_of_int l
56   | SBoollit(true) -> "true"
57   | SBoollit(false) -> "false"
58   | SVar(s) -> s
59   | SStringlit(l) -> l
60   | SFunsig(typ, bl, e) -> " fun: " ^ (string_of_typ typ) ^ " (" ^ (String.concat ""
61     (List.map string_of_svdecl bl)) ^ ") { " ^ string_of_sexpr e ^ " }"
62   | SNull -> string_of_typ t ^ " null"
63   | SBinop(e1, o, e2) ->
64     string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
65   | SUNop(o, e) -> string_of_uop o ^ string_of_sexpr e
66   | SAsn(v, e) -> v ^ " = " ^ string_of_sexpr e
67   | SFCall(f, el) ->
68     f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
69   | SMCall(caller, f, el) ->
70     string_of_sexpr caller ^ "." ^ f ^ "(" ^ String.concat ", " (List.map
71       string_of_sexpr el) ^ ")"
72   | SNodeExpr(e1, e2) -> string_of_sexpr e1 ^ ": " ^ string_of_sexpr e2
73   | SEdgeExpr(src, dst, w) -> "(" ^ (string_of_sexpr src) ^ ", " ^ (string_of_sexpr
74     dst) ^ ", " ^ (string_of_sexpr w) ^ ")"
75   | SGGraphExpr(node_list, edge_list) ->
76     "[nodes: [" ^ String.concat ", " (List.map string_of_sexpr node_list) ^
77     "], edges: [" ^ String.concat ", " (List.map string_of_sexpr edge_list) ^ "]]"
78   | SNoexpr -> ""
79
80 let rec string_of_sstmt = function
81   SBlock(stmts) ->
82     "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
83   | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
84   | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
85   | SIIf(e, s, SBlock([])) ->
86     "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
87   | SIIf(e, s1, s2) -> "if (" ^ string_of_sexpr e ^ ")\n" ^
88     string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
89   | SFor(e1, e2, e3, s) ->
90     "for (" ^ string_of_sexpr e1 ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
91     string_of_sexpr e3 ^ ")\n" ^ string_of_sstmt s
92   | SForNode(n, g, body) ->
93     "for (" ^ n ^ " : " ^ string_of_sexpr g ^ ")\n" ^ string_of_sstmt body
94   | SForEdge(src, dst, w, g, body) ->
95     "for (" ^ src ^ ", " ^ dst ^ ", " ^ w ^ " : " ^ string_of_sexpr g ^ ")\n" ^
96     string_of_sstmt body
97   | SWhile(e, body) -> "while (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt body
98   | SVdecl(t, var, expr) ->

```

```

95      match expr with
96      | (_, SNoexpr) ->
97          string_of_svdecl (t, var)
98      | _ ->
99          string_of_typ t ^ " " ^ string_of_sexpr expr ^ ";\n"
100
101 let string_of_sfdecl fdecl = string_of_typ fdecl.styp ^ " " ^
102     fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sargs) ^ "
103     ")\n{ \n" ^
104     String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
105     "}\n"
106
107
108 let string_of_sprogram (vars, funcs) =
109     (match vars with
110     | [] -> ""
111     | _ -> String.concat "" (List.map string_of_svdecl vars) ^ "\n") ^
112     String.concat "\n" (List.map string_of_sfdecl funcs)

```

9.5 Semantic Checking

9.5.1 semant.ml

```

1 (* Authors:
2   Benjamin Lewinter bsl2121
3   Irina Mateescu im2441
4   Harry Smith hs3061
5   Yasunari Watanabe yw3239
6 *)
7
8 open Ast
9 open Sast
10
11 module StringMap = Map.Make(String)
12
13 let check (globals, funcs) =
14   (* Verify a list of bindings has no void types or duplicate names *)
15   let check_binds (kind : string) (binds : binding list) =
16       List.iter (function
17       | Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
18       | _ -> () binds;
19       let rec dups = function
20           [] -> ()
21           | ((_,n1) :: (_ ,n2) :: _) when n1 = n2 ->
22               raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
23           | _ :: t -> dups t
24           in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
25   in
26
27   (**** Check global variables ****)
28   check_binds "global" globals;
29
30   (**** Check functions and methods ****)
31
32   let built_in_fdecls =
33       let add_bind map (name, (ty, args)) =
34           StringMap.add name
35               { typ = ty; fname = name; args = args; body = [] }
36               map in
37   let mappings = [
38       "print_int", (Void, [(Int, "x")]);

```

```

39     "print_bool", (Void, [(Bool, "x")]));
40     "print", (Void, [(String, "x")])
41   ] in
42   List.fold_left add_bind StringMap.empty mappings
43 in
44
45 (* Add function name to symbol table *)
46 let add_func map fd =
47   let built_in_err = "function " ^ fd.fname ^ " may not be defined"
48   and dup_err = "duplicate function " ^ fd.fname
49   and make_err er = raise (Failure er)
50   and n = fd.fname (* Name of the function *)
51   in match fd with (* No duplicate functions or redefinitions of built-ins *)
52     _ when StringMap.mem n built_in_fdecls -> make_err built_in_err
53     | _ when StringMap.mem n map -> make_err dup_err
54     | _ -> StringMap.add n fd map
55 in
56
57 (* Collect all function names into one symbol table *)
58 let fdecls = List.fold_left add_func built_in_fdecls funcs
59 in
60
61 (* Return a function from our symbol table *)
62 let find_func local_fdecls s =
63   try StringMap.find s local_fdecls
64   with Not_found ->
65     try StringMap.find s local_fdecls
66     with Not_found -> raise (Failure ("unrecognized function " ^ s))
67 in
68
69 (* Return a method from our symbol table *)
70 let find_method libtyp s margs=
71   try match libtyp with
72     | Node(lt, dt) ->
73       (match s with
74        | "get_data" ->
75          { typ = dt; fname = s; args = []; body = [] }
76        | "set_data" ->
77          { typ = Void; fname = s; args = [(dt, "d")]; body = [] }
78        | "get_name" ->
79          { typ = lt; fname = s; args = []; body = [] }
80        | "print" ->
81          { typ = Void; fname = s; args = []; body = [] }
82        | _ ->
83          raise Not_found)
84     | Graph(lt, dt, wt) ->
85       (match s with
86        | "set_node" ->
87          { typ = Int; fname = s; args = [(Node(lt, dt), "x")]; body = [] }
88        | "set_edge" ->
89          if List.length margs = 2 then
90            { typ = Int; fname = s; args = [(lt, "src"); (lt, "dst")]; body = [] }
91          else
92            { typ = Int; fname = s; args = [(lt, "src"); (lt, "dst"); (wt, "w")];
93              body = [] }
94        | "remove_node" ->
95          { typ = Int; fname = s; args = [(lt, "x")]; body = [] }
96        | "remove_edge" ->
97          { typ = Int; fname = s; args = [(lt, "src"); (lt, "dst")]; body = [] }
98        | "get_node" ->
99          { typ = Node(lt, dt); fname = s; args = [(lt, "l")]; body = [] })

```

```

99      | "get_weight" ->
100         { typ = wt; fname = s; args = [(lt, "src"); (lt, "dst")]; body = [] }
101      | "print" ->
102         { typ = Void; fname = s; args = []; body = [] }
103      | "has_node" ->
104         { typ = Int; fname = s; args = [(lt, "src")]; body = [] }
105      | "are_neighbors" ->
106         { typ = Bool; fname = s; args = [(lt, "src"); (lt, "dst")]; body = [] }
107      | "is_empty" ->
108         { typ = Bool; fname = s; args = []; body = [] }
109      | "neighbors" ->
110         if List.length margs = 1 then
111             { typ = Graph(lt, dt, wt); fname = s; args = [(lt, "label")]; body = []
112                 }
113             else
114                 { typ = Graph(lt, dt, wt); fname = s; args = [(lt, "label"); (Int, "level");
115                     (Bool, "include_current")]; body = [] }
116      | "find" ->
117         { typ = Graph(lt, dt, wt); fname = s; args = [(dt, "data")]; body = []
118             }
119      | "dfs" ->
120         { typ = Graph(lt, dt, wt); fname = s; args = [(lt, "label")]; body = []
121             }
122         | _ -> raise Not_found
123         | _ -> raise Not_found
124     with Not_found -> raise (Failure ("unrecognized method " ^ string_of_typ libtyp
125                                         ^ "." ^ s))
126   in
127 let _ = find_func fdecls "main" in (*Ensure "main" is defined*)
128
129 let check_function func =
130   (* Make sure no args are void or duplicates *)
131   check_binds "args" func.args;
132
133   (* Raise an exception if the given rvalue type cannot be assigned to
134      the given lvalue type *)
135   let check_asn lvaluet rvaluet err =
136     if lvaluet = rvaluet then lvaluet else raise (Failure err)
137   in
138
139   (* Build global and local symbol table of variables for this function *)
140
141   let global_vars = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
142                                         StringMap.empty (globals)
143   in
144
145   let local_vars = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
146                                         StringMap.empty (func.args)
147   in
148
149   let funcs' = List.map (fun (ty, name) ->
150                           match ty with
151                             | Fun(ret_t, args_t) ->
152                               {typ = ret_t; fname = name; args = List.map (fun t ->
153                                     (t, "x")) args_t; body = []}
154                             | _ -> raise Unsupported_constructor)
155                           (List.filter (fun (ty, _) -> match ty with Fun(_) -> true |

```

```

153                                     _ -> false) func.args) in
154
155 let local_fdecls = List.fold_left add_func StringMap.empty funcs' in
156
157 (* Return a variable from our local symbol table *)
158 let type_of_variable vars s =
159   try StringMap.find s vars
160   with Not_found ->
161     try StringMap.find s global_vars
162     with Not_found ->
163       raise (Failure ("undeclared variable " ^ s))
164
165 in
166
167 (* Return a semantically-checked expression, i.e., with a type *)
168 let rec expr fdecls vars = function
169   | Intlit l -> (Int, SIntlit l)
170   | Boollit l -> (Bool, SBoollit l)
171   | Stringlit l -> (String, SStringlit l)
172   | Null -> (Bool, SNull)
173   | Funsig (t, bl, e) ->
174     check_anon_func_expr fdecls vars t bl e
175   | Noexpr -> (Void, SNoexpr)
176   | Var s -> (type_of_variable vars s, SVar s)
177   | NodeExpr (l, d) ->
178     let (lt, _) as l' = expr fdecls vars l in
179     let (dt, _) as d' = expr fdecls vars d in
180     (Node(lt, dt), SNodeExpr(l', d'))
181   | EdgeExpr (src, dst, w) ->
182     let (wt, _) as w' = expr fdecls vars w in
183     (Edge(wt), SEdgeExpr(expr fdecls vars src, expr fdecls vars dst, w'))
184   | Asn(var, e) ->
185     check_asn_expr fdecls vars var e
186   | Unop(op, e) ->
187     check_unop_expr fdecls vars op e
188   | Binop(e1, op, e2) ->
189     check_binop_expr fdecls vars e1 op e2
190   | FCall(fname, args) ->
191     check_fcall_expr fdecls vars fname args
192   | MCall(instance, mname, args) ->
193     check_mcall_expr fdecls vars instance mname args
194   | GraphExpr(node_list, edge_list) ->
195     check_graph_expr fdecls vars node_list edge_list
196
197 and coerce_null_to_typ new_typ e =
198   match e with
199   | (Bool, SNull) -> (new_typ, SNull)
200   | _ -> e
201
202 and check_asn_expr fdecls vars var e =
203   let lvt = type_of_variable vars var in
204   let (rvt, e') = coerce_null_to_typ lvt (expr fdecls vars e) in
205   let err = "illegal assignment " ^ string_of_typ lvt ^ " = " ^
206     string_of_typ rvt ^ " in " ^ string_of_expr (Asn(var, e))
207   in
208
209   match lvt, rvt, e' with
210   (* If left expression is a node with bool type data, wrap right expression
      in a SNodeExpr *)
211   | Node(l1t, l1d), _, SNodeExpr((l2t, _), d) ->
      let (dt, _) = coerce_null_to_typ l1d d in

```

```

212     let lt = check_asn llt lt err in
213     let dt = check_asn ldt dt err in
214     (Node(lt, dt), SAsn(var, (lvt, e')))
215   | Node(_, Node(_), _) ->
216     let lt = check_asn lvt rvt err in
217     (lt, SAsn(var, (lt, e')))
218   | Node(llt, ldt), _, _ ->
219     let lt = check_asn llt rvt err in
220     (Node(llt, ldt), SAsn(var, (lvt, SNodeExpr((llt, e'), (ldt, SNull)))))  

221   | Graph(llt, ldt, lwt), Graph(_, _, _), SGGraphExpr(nl, el) ->
222     (* Coerce (Bool, SNull) to correct type then check type equality *)
223     let nl' = List.map (fun (_, e) -> match e with
224                           | SNodeExpr((lt, le), d) ->
225                             let (dt, de) = coerce_null_to_typ ldt
226                               d in
227                             let lt = check_asn llt lt err in
228                             let dt = check_asn ldt dt err in
229                               (Node(lt, dt), SNodeExpr((lt, le), (dt, de)))
230                         | _ -> raise Unsupported_constructor) nl
231                           in
232     let el' = List.map (fun (_, e) -> match e with
233                           | SEdgeExpr(src, dst, w) ->
234                             let (wt, we) = coerce_null_to_typ lwt
235                               w in
236                             let wt = check_asn lwt wt err in
237                             (Edge(lwt), SEdgeExpr(src, dst, (wt, we)))
238                           | _ -> raise Unsupported_constructor) el
239                           in
240
241     let t = Graph(llt, ldt, lwt) in
242     (t, SAsn(var, (t, SGGraphExpr(nl', el'))))
243   | Fun(_, Fun(_, SFunsig(_)) ->
244     let (_, new_expr) = expr fdecls vars e in
245       (check_asn lvt rvt err, SAsn(var, (rvt, new_expr)))
246
247   | _ ->
248     (check_asn lvt rvt err, SAsn(var, (rvt, e')))

249 and check_unop_expr fdecls vars op e =
250   let (t, e') = expr fdecls vars e in
251   let ty =
252     match op with
253       | Not when t = Bool -> Bool
254       | Neg when t = Int -> Int
255       | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op ^
256                               string_of_typ t ^ " in " ^ string_of_expr (Unop(op, e))))
257   in (ty, SUunop(op, (t, e')))

258 and check_binop_expr fdecls vars e1 op e2 =
259   let (t1, e1') = expr fdecls vars e1
260   and (t2, e2') = expr fdecls vars e2 in
261   (* All binary operators require operands of the same type *)
262   let same = t1 = t2 in
263   (* Determine expression type based on operator and operand types *)
264   let ty = match op with
265     | Add | Sub | Mul | Div when same && t1 = Int -> Int
266     | Eq | Neq when same -> Bool
267     | Lt | Leq | Gt | Geq when same && (t1 = Int || t1 = String) -> Bool

```

```

266     | And | Or when same && t1 = Bool -> Bool
267     | _ ->
268         raise (Failure ("illegal binary operator " ^
269                         string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
270                         string_of_typ t2 ^ " in " ^ string_of_expr (Binop
271                                         (e1, op, e2))))
272     in (ty, SBinop((t1, e1'), op, (t2, e2'))))
273
274 and check_fcall_expr fdecls vars fname args =
275   let fd = find_func fdecls fname in
276   let param_length = List.length fd.args in
277   if List.length args != param_length
278     then raise (Failure ("expecting " ^ string_of_int param_length ^
279                           " arguments in " ^ string_of_expr (FCall(fname, args))))
280   else let check_call (ft, _) e =
281     let (et, e') = expr fdecls vars e in
282     let err = "illegal argument found " ^ string_of_typ et ^
283                   " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
284                   in
285     (check_asn ft et err, e')
286   in
287   let args' = List.map2 check_call fd.args args in
288   (fd.typ, SFCall(fname, args'))
289
290 and check_mcall_expr fdecls vars instance mname args =
291   let (instance_typ, _) as instance' = expr fdecls vars instance in
292   let md = find_method instance_typ mname args in
293   let param_length = List.length md.args in
294   if List.length args != param_length
295     then raise (Failure ("expecting " ^ string_of_int param_length ^
296                           " arguments in method " ^ string_of_expr (MCall(instance,
297                                         mname, args))))
298   else let check_call (ft, _) e =
299     let (et, e') = expr fdecls vars e in
300     let err = "illegal argument found " ^ string_of_typ et ^
301                   " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
302                   in
303     (check_asn ft et err, e')
304   in
305   let args' = List.map2 check_call md.args args in
306   (md.typ, SMCall(instance', mname, args'))
307
308 and check_graph_expr fdecls vars node_list edge_list =
309   (* infer node label/data types from first nodes in list if any,
310      and check that all items have the same type *)
311   let node_label_typ, node_data_typ, s_node_list =
312     if node_list = []
313     then (Bool, Bool, [])
314     else let err = "type mismatch in graph nodes" in
315       let check_node_typ (lt_opt, dt_opt) n =
316         match n with
317         | (Node(lt, dt), SNodeExpr(_, d)) ->
318           (* check matching node label *)
319           let lt_opt = (match lt_opt with
320             | None -> Some(lt)
321             | Some(lt') -> if lt = lt'
322               then lt_opt
323               else raise (Failure err)) in
324           (* check matching node data *)
325           let dt_opt = (match d with
326             | (Bool, SNull) -> dt_opt

```

```

323         | _ -> match dt_opt with
324             | None -> Some(dt)
325             | Some(dt') -> if dt = dt'
326                 then dt_opt
327                 else raise (Failure err))
328             in (lt_opt, dt_opt)
329         | _ -> raise Unsupported_constructor
330     in
331     let node_list' = List.map (expr fdecls vars) node_list in
332     match List.fold_left check_node_typ (None, None) node_list' with
333     | None, _ -> raise (Failure "graph node names are required")
334     | Some(lt), None -> (lt, Bool, node_list')
335     | Some(lt), Some(dt) -> (lt, dt, node_list')
336   in
337   (* infer edge weight types from first edge in list if any,
338      and check that all items have the same type *)
339   let edge_typ, s_edge_list =
340     if edge_list = []
341     then (Bool, [])
342     else let err = "type mismatch in graph edges" in
343       let check_edge_typ wt_opt e =
344         match e with
345         | (Edge(wt), SEdgeExpr(_, _, w)) ->
346             (match w with
347              | (Bool, SNull) -> wt_opt
348              | _ -> (match wt_opt with
349                  | None -> Some(wt)
350                  | Some(wt') -> if wt = wt'
351                      then wt_opt
352                      else raise (Failure err)))
353             | _ -> raise Unsupported_constructor
354       in
355       let edge_list' = List.map (expr fdecls vars) edge_list in
356       match List.fold_left check_edge_typ None edge_list' with
357       | None -> (Bool, edge_list')
358       | Some(wt) -> (wt, edge_list')
359   in
360   (Graph(node_label_typ, node_data_typ, edge_typ), SGExpr(s_node_list,
361               s_edge_list))
362
363 and check_anon_func_expr fdecls vars typ b_list ex =
364   let vars' = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
365                                         vars (b_list) in
366   let (ty, sx) = expr fdecls vars' ex in
367   let err = "type mismatch in result of anonymous function" in
368   let checked_type = check_asn typ ty err in
369   let typ_list = List.map (fun (ty, _) -> ty) b_list in
370   (Fun(typ, typ_list), SFunsig(checked_type, b_list, (ty, sx)))
371 in
372
373 let check_bool_expr fdecls vars e =
374   let (t', e') = expr fdecls vars e
375   and err = "expected Boolean expression in " ^ string_of_expr e
376   in if t' != Bool then raise (Failure err) else (t', e')
377 in
378
379 (* Return a semantically-checked statement i.e. containing sexprs *)
380 let rec check_stmt fdecls vars = function
381   Expr e -> (*
382     (match e with

```

```

383         Asn(s, Funsig(ty, bl, body)) ->
384             let new_fdecl = {typ = ty; fname = s; args = bl; body = [Expr(
385                 body)]} in
386                 let fdecls' = add_func fdecls new_fdecl in
387                     (fdecls', vars, SExpr (expr fdecls' vars e))
388             | _ ->
389                 *) (fdecls, vars, SExpr (expr fdecls vars e))
390             | For (e1, e2, e3, st) ->
391                 let (_, _, st') = check_stmt fdecls vars st in
392                     (fdecls, vars, SFor (expr fdecls vars e1, check_bool_expr fdecls vars e2,
393                         expr fdecls vars e3, st'))
394             | ForNode (n, g, st) ->
395                 (match expr fdecls vars g with
396                  | (Graph(lt, dt, _), _) as ge ->
397                      let vars' = StringMap.add n (Node (lt, dt)) vars in
398                      let (_, _, st') = check_stmt fdecls vars' st in
399                          (fdecls, vars', SForNode (n, ge, st'))
400                  | (ty, _) -> raise (Failure ("illegal argument found: expected graph, got
401                      " ^ string_of_typ ty)))
402             | ForEdge (src, dst, w, g, st) ->
403                 (match expr fdecls vars g with
404                  | (Graph(lt, dt, wt), _) as ge ->
405                      let nt = Node(lt, dt) in
406                          let vars' = StringMap.add src nt (StringMap.add dst nt (StringMap.add
407                              w wt vars)) in
408                              let (_, _, st') = check_stmt fdecls vars' st in
409                                  (fdecls, vars', SFORedge (src, dst, w, ge, st'))
410                  | (ty, _) -> raise (Failure ("illegal argument found: expected graph, got
411                      " ^ string_of_typ ty)))
412             | While (p, st) ->
413                 let (_, _, st') = check_stmt fdecls vars st in
414                     (fdecls, vars, SWhile (check_bool_expr fdecls vars p, st'))
415             | If (p, b1, b2) ->
416                 let (_, _, b1') = check_stmt fdecls vars b1 in
417                 let (_, _, b2') = check_stmt fdecls vars b2 in
418                     (fdecls, vars, SIf (check_bool_expr fdecls vars p, b1', b2'))
419             | Vdecl (ty, s, e) ->
420                 if ty = Void
421                     then raise (Failure ("variable '" ^ s ^ "' declared void"))
422                 else let vars' = StringMap.add s ty vars in
423                     (match ty, e with
424                      | Fun(_), Asn(_, Var(var_name)) ->
425                          (print_string "here\n";
426                           let new_fun = {(find_func fdecls var_name) with fname = s} in
427                               let fdecls' = add_func fdecls new_fun in
428                                   (fdecls', vars', SVdecl (ty, s, expr fdecls vars' e))
429                           | _ ->
430                               (fdecls, vars', SVdecl (ty, s, expr fdecls vars' e)))
431             | Return e ->
432                 let (t, e') = coerce_null_to_typ func.typ (expr fdecls vars e) in
433                     if t = func.typ then (fdecls, vars, SReturn (t, e'))
434                     else raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
435                         string_of_typ func.typ ^ " in " ^ string_of_expr e))
436             (* A block is correct if each statement is correct and nothing
437                 follows any Return statement. Nested blocks are flattened. *)
438             | Block sl ->
439                 let rec check_stmt_list fdecls vars = function
440                     [Return _ as s] ->
441                         let (_, _, s') = check_stmt fdecls vars s in [s']
442                     | Return _ :: _ -> raise (Failure "nothing may follow a return")
443                     | Block sl :: ss -> check_stmt_list fdecls vars (sl @ ss) (* Flatten

```

```

        blocks *)
439 | s :: ss          ->
440   (match s with
441     | Vdecl(_, var_name, Asn(_, Funsig(ty, bl, body))) ->
442       let new_fdecl = {typ = ty; fname = var_name; args = bl; body = [
443         Expr(body)]} in
444       let fdecls' = add_func fdecls new_fdecl in
445       let (fdecls'', vars', s') = check_stmt fdecls' vars s in
446       s' :: check_stmt_list fdecls'' vars' ss
447     | _ ->
448       let (fdecls', vars', s') = check_stmt fdecls vars s in
449       s' :: check_stmt_list fdecls' vars' ss)
450   | []              -> []
451   in (fdecls, vars, SBlock(check_stmt_list fdecls vars sl))
452
453   in (* body of check_function *)
454   { styp = func.typ;
455     sfname = func.fname;
456     sargs = func.args;
457     sbody = let (_, _, st) = check_stmt local_fdecls local_vars (Block func.body)
458             in match st with
459               SBlock(sl) -> sl
460             | _ -> raise (Failure ("internal error: block didn't become a block?"))
461   }
461 in (globals, List.map (check_function) funcs)

```

9.6 Code Generation

9.6.1 codegen.ml

```

1 (* Authors:
2   Benjamin Lewinter bsl2121
3   Irina Mateescu im2441
4   Harry Smith hs3061
5   Yasunari Watanabe yw3239
6 *)
7
8 module L = Llvm
9 module A = Ast
10 open Sast
11
12 module StringMap = Map.Make(String)
13
14 let translate (globals, functions) =
15   let context = L.global_context () in
16   let the_module = L.create_module context "Hippograph" in
17
18   let void_t = L.void_type context
19   and i32_t = L.i32_type context
20   and i32_ptr_t = L.pointer_type (L.i32_type context)
21   and i8_t = L.i8_type context
22   and i1_t = L.i1_type context
23   and str_t = L.pointer_type (L.i8_type context)
24   and void_ptr_t = L.pointer_type (L.i8_type context)
25   in
26
27   let rec ltype_of_typ = function
28     | A.Void    -> void_t
29     | A.Int     -> i32_t
30     | A.Bool    -> i1_t

```

```

31     | A.String  -> str_t
32     | A.Fun(ret_t, args) ->
33         let formal_types = Array.of_list (List.map ltype_of_typ args) in
34         L.pointer_type (L.function_type (ltype_of_typ ret_t) formal_types)
35     | A.Node(_, _) -> void_ptr_t
36     | A.Edge(_) -> void_ptr_t
37     | A.Graph(_, _, _) -> void_ptr_t
38   in
39
40 (* Declare each global variable; remember its value in a map *)
41 let global_vars =
42   let global_var m (t, n) =
43     let init =
44       match t with
45       | A.String -> L.const_bitcast (L.const_stringz context "") str_t
46       | _ -> L.const_int (ltype_of_typ t) 0
47     in StringMap.add n (L.define_global n init the_module) m in
48   List.fold_left global_var StringMap.empty globals in
49
50 (* C Functions *)
51
52 let print_t : L.lltype = L.var_arg_function_type void_t [| L.pointer_type i8_t |]
53   in
54 let print_func : L.llvalue = Ldeclare_function "printf" print_t the_module in
55
56 let strcmp_t : L.lltype = L.var_arg_function_type i32_t [| str_t; str_t |] in
57 let strcmp_func : L.llvalue = Ldeclare_function "strcmp" strcmp_t the_module in
58
59 let create_graph_t : L.lltype = L.var_arg_function_type void_ptr_t [| |] in
60 let create_graph_func : L.llvalue = Ldeclare_function "create_graph"
61   create_graph_t the_module in
62
63 let add_node_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t;
64   void_ptr_t |] in
65 let add_node_func : L.llvalue = Ldeclare_function "add_node" add_node_t
66   the_module in
67
68 let create_node_t : L.lltype = L.var_arg_function_type void_ptr_t [| |] in
69 let create_node_func : L.llvalue = Ldeclare_function "create_node" create_node_t
70   the_module in
71
72 let clone_node_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |]
73   in
74 let clone_node_func : L.llvalue = Ldeclare_function "clone_node" clone_node_t
75   the_module in
76
77 let create_edge_t : L.lltype = L.var_arg_function_type void_ptr_t [| |] in
78 let create_edge_func : L.llvalue = Ldeclare_function "create_edge" create_edge_t
79   the_module in
80
81 let add_edge_int_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t;
82   void_ptr_t; i32_t; i32_t |] in
83 let add_edge_int_func : L.llvalue = Ldeclare_function "add_edge_int"
84   add_edge_int_t the_module in
85
86 let add_edge_bool_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t;
87   void_ptr_t; i1_t; i1_t |] in
88 let add_edge_bool_func : L.llvalue = Ldeclare_function "add_edge_bool"
89   add_edge_bool_t the_module in
90
91 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
80 let add_edge_str_func : L.llvalue = Ldeclare_function "add_edge_str"
    add_edge_str_t the_module in
81
82 let set_node_label_int_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t
    ; i32_t []] in
83 let set_node_label_int_func : L.llvalue = Ldeclare_function "set_node_label_int"
    set_node_label_int_t the_module in
84
85 let set_node_label_bool_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t
    void_ptr_t; i1_t []] in
86 let set_node_label_bool_func : L.llvalue = Ldeclare_function "set_node_label_bool"
    set_node_label_bool_t the_module in
87
88 let set_node_label_str_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t
    ; str_t []] in
89 let set_node_label_str_func : L.llvalue = Ldeclare_function "set_node_label_str"
    set_node_label_str_t the_module in
90
91 let set_node_data_int_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t;
    i32_t; i1_t []] in
92 let set_node_data_int_func : L.llvalue = Ldeclare_function "set_node_data_int"
    set_node_data_int_t the_module in
93
94 let set_node_data_bool_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t
    ; i1_t; i1_t []] in
95 let set_node_data_bool_func : L.llvalue = Ldeclare_function "set_node_data_bool"
    set_node_data_bool_t the_module in
96
97 let set_node_data_str_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t;
    str_t; i1_t []] in
98 let set_node_data_str_func : L.llvalue = Ldeclare_function "set_node_data_str"
    set_node_data_str_t the_module in
99
100 let get_node_label_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t
    []] in
101 let get_node_label_func : L.llvalue = Ldeclare_function "get_node_label"
    get_node_label_t the_module in
102
103 let get_node_data_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t
    []] in
104 let get_node_data_func : L.llvalue = Ldeclare_function "get_node_data"
    get_node_data_t the_module in
105
106 let graph_has_node_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t;
    i32_t []] in
107 let graph_has_node_int_func : L.llvalue = Ldeclare_function "graph_has_node_int"
    graph_has_node_int_t the_module in
108
109 let graph_has_node_str_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t;
    str_t []] in
110 let graph_has_node_str_func : L.llvalue = Ldeclare_function "graph_has_node_str"
    graph_has_node_str_t the_module in
111
112 let graph_has_node_bool_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t
    ; i1_t []] in
113 let graph_has_node_bool_func : L.llvalue = Ldeclare_function "graph_has_node_int"
    graph_has_node_bool_t the_module in
114
115 let graph_set_edge_int_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t
    void_ptr_t; i32_t; i32_t; i32_t []] in

```

```

116 let graph_set_edge_int_int_func : L.llvalue = L.declare_function "
117     graph_set_edge_int_int" graph_set_edge_int_int_t the_module in
118 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
119 let graph_set_edge_bool_int_func : L.llvalue = L.declare_function "
120     graph_set_edge_bool_int" graph_set_edge_bool_int_t the_module in
121 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
122 let graph_set_edge_str_bool_func : L.llvalue = L.declare_function "
123     graph_set_edge_str_int" graph_set_edge_str_bool_t the_module in
124 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
125 let graph_set_edge_bool_str_func : L.llvalue = L.declare_function "
126     graph_set_edge_int_str" graph_set_edge_bool_str_t the_module in
127 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
128 let graph_set_edge_int_bool_func : L.llvalue = L.declare_function "
129     graph_set_edge_int_int" graph_set_edge_int_bool_t the_module in
130 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
131 let graph_set_edge_bool_bool_func : L.llvalue = L.declare_function "
132     graph_set_edge_int_int" graph_set_edge_bool_bool_t the_module in
133 let graph_set_edge_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i32_t; i32_t |] in
134 let graph_set_edge_int_func : L.llvalue = L.declare_function "graph_set_edge_int"
135     graph_set_edge_int_t the_module in
136 let graph_set_edge_str_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; str_t; str_t |] in
137 let graph_set_edge_str_func : L.llvalue = L.declare_function "graph_set_edge_str"
138     graph_set_edge_str_t the_module in
139 let graph_set_edge_bool_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i1_t; i1_t |] in
140 let graph_set_edge_bool_func : L.llvalue = L.declare_function "graph_set_edge_int"
141     graph_set_edge_bool_t the_module in
142 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
143 let graph_set_edge_str_int_func : L.llvalue = L.declare_function "
144     graph_set_edge_str_int" graph_set_edge_str_int_t the_module in
145 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
146 let graph_set_edge_int_str_func : L.llvalue = L.declare_function "
147     graph_set_edge_int_str" graph_set_edge_int_str_t the_module in
148 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
149 let graph_set_edge_str_str_func : L.llvalue = L.declare_function "
150     graph_set_edge_str_str" graph_set_edge_str_str_t the_module in
151 let get_node_by_label_int_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; i32_t |] in
152 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
153 let get_node_by_label_int_opt_func : L.llvalue = Ldeclare_function "
        get_node_by_label_int_opt" get_node_by_label_int_t the_module in
154
155 let get_node_by_label_bool_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; i1_t |] in
156 let get_node_by_label_bool_opt_func : L.llvalue = Ldeclare_function "
        get_node_by_label_bool_opt" get_node_by_label_bool_t the_module in
157
158 let get_node_by_label_str_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; str_t |] in
159 let get_node_by_label_str_func : L.llvalue = Ldeclare_function "
        get_node_by_label_str" get_node_by_label_str_t the_module in
160 let get_node_by_label_str_opt_func : L.llvalue = Ldeclare_function "
        get_node_by_label_str_opt" get_node_by_label_str_t the_module in
161
162 let print_node_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t |] in
163 let print_node_func : L.llvalue = Ldeclare_function "print_node" print_node_t
    the_module in
164
165 let print_graph_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t |] in
166 let print_graph_func : L.llvalue = Ldeclare_function "print_graph" print_graph_t
    the_module in
167
168 let set_edge_w_int_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t; i32_t; i1_t |] in
169 let set_edge_w_int_func : L.llvalue = Ldeclare_function "set_edge_w_int"
    set_edge_w_int_t the_module in
170
171 let set_edge_w_bool_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t; i1_t; i1_t |] in
172 let set_edge_w_bool_func : L.llvalue = Ldeclare_function "set_edge_w_bool"
    set_edge_w_bool_t the_module in
173
174 let set_edge_w_str_t : L.lltype = L.var_arg_function_type void_t [| void_ptr_t; str_t; i1_t |] in
175 let set_edge_w_str_func : L.llvalue = Ldeclare_function "set_edge_w_str"
    set_edge_w_str_t the_module in
176
177 let get_edge_src_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
178 let get_edge_src_func : L.llvalue = Ldeclare_function "get_edge_src"
    get_edge_src_t the_module in
179
180 let get_edge_dst_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
181 let get_edge_dst_func : L.llvalue = Ldeclare_function "get_edge_dst"
    get_edge_dst_t the_module in
182
183 let get_edge_w_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t |]
    in
184 let get_edge_w_int_func : L.llvalue = Ldeclare_function "get_edge_w_int"
    get_edge_w_int_t the_module in
185
186 let get_edge_w_bool_t : L.lltype = L.var_arg_function_type i1_t [| void_ptr_t |]
    in
187 let get_edge_w_bool_func : L.llvalue = Ldeclare_function "get_edge_w_int"
    get_edge_w_bool_t the_module in
188
189 let get_edge_w_str_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in

```

```

190 let get_edge_w_str_func : L.llvalue = L.declare_function "get_edge_w_str"
    get_edge_w_str_t the_module in
191
192 let graph_to_node_iterable_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
193 let graph_to_node_iterable_func : L.llvalue = L.declare_function "graph_to_node_iterable" graph_to_node_iterable_t the_module in
194
195 let graph_to_edge_iterable_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
196 let graph_to_edge_iterable_func : L.llvalue = L.declare_function "graph_to_edge_iterable" graph_to_edge_iterable_t the_module in
197
198 let get_graph_next_node_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
199 let get_graph_next_node_func : L.llvalue = L.declare_function "get_graph_next_node" "get_graph_next_node_t" the_module in
200
201 let get_graph_next_edge_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
202 let get_graph_next_edge_func : L.llvalue = L.declare_function "get_graph_next_edge" "get_graph_next_edge_t" the_module in
203
204 let graph_set_node_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; void_ptr_t |] in
205 let graph_set_node_func : L.llvalue = L.declare_function "graph_set_node" graph_set_node_t the_module in
206
207 let remove_edge_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; void_ptr_t |] in
208 let remove_edge_func : L.llvalue = L.declare_function "remove_edge" remove_edge_t the_module in
209
210 let remove_node_int_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i32_t |] in
211 let remove_node_int_func : L.llvalue = L.declare_function "remove_node_int" remove_node_int_t the_module in
212
213 let remove_node_str_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; str_t |] in
214 let remove_node_str_func : L.llvalue = L.declare_function "remove_node_str" remove_node_str_t the_module in
215
216 let remove_node_bool_t : L.lltype = L.var_arg_function_type i32_t [| void_ptr_t; i1_t |] in
217 let remove_node_bool_func : L.llvalue = L.declare_function "remove_node_int" remove_node_bool_t the_module in
218
219 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
220 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
221
222 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
223 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
224
225 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
226 let get_edge_by_src_and_dst_str_func : L.llvalue = L.declare_function "

```

```

227     get_edge_by_src_and_dst_str" get_edge_by_src_and_dst_str_t the_module in
228 let neighbors_one_arg_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t |] in
229 let neighbors_one_arg_func : L.llvalue = Ldeclare_function "neighbors_one_arg" neighbors_one_arg_t the_module in
230
231 let neighbors_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; i32_t; i1_t |] in
232 let neighbors_func : L.llvalue = Ldeclare_function "neighbors" neighbors_t the_module in
233
234 let find_data_int_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; i32_t |] in
235 let find_data_int_func : L.llvalue = Ldeclare_function "find_data_int" find_data_int_t the_module in
236
237 let find_data_bool_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; i1_t |] in
238 let find_data_bool_func : L.llvalue = Ldeclare_function "find_data_int" find_data_bool_t the_module in
239
240 let find_data_str_t : L.lltype = L.var_arg_function_type void_ptr_t [| void_ptr_t; str_t |] in
241 let find_data_str_func : L.llvalue = Ldeclare_function "find_data_str" find_data_str_t the_module in
242
243 let are_neighbors_int_t : L.lltype = L.var_arg_function_type i1_t [| void_ptr_t; i32_t; i32_t |] in
244 let are_neighbors_int_func : L.llvalue = Ldeclare_function "are_neighbors_int" are_neighbors_int_t the_module in
245
246 let are_neighbors_bool_t : L.lltype = L.var_arg_function_type i1_t [| void_ptr_t; i1_t; i1_t |] in
247 let are_neighbors_bool_func : L.llvalue = Ldeclare_function "are_neighbors_int" are_neighbors_bool_t the_module in
248
249 let are_neighbors_str_t : L.lltype = L.var_arg_function_type i1_t [| void_ptr_t; str_t; str_t |] in
250 let are_neighbors_str_func : L.llvalue = Ldeclare_function "are_neighbors_str" are_neighbors_str_t the_module in
251
252 let is_empty_t : L.lltype = L.var_arg_function_type i1_t [| void_ptr_t |] in
253 let is_empty_func : L.llvalue = Ldeclare_function "is_empty" is_empty_t the_module in
254
255 let dfs_int_t : L.lltype = L.function_type void_ptr_t [| void_ptr_t; i32_t |] in
256 let dfs_int_func : L.llvalue = Ldeclare_function "dfs_int" dfs_int_t the_module in
257
258 let dfs_str_t : L.lltype = L.function_type void_ptr_t [| void_ptr_t; str_t |] in
259 let dfs_str_func : L.llvalue = Ldeclare_function "dfs_str" dfs_str_t the_module in
260
261 let bfs_int_t : L.lltype = L.function_type void_ptr_t [| void_ptr_t; i32_t |] in
262 let bfs_int_func : L.llvalue = Ldeclare_function "bfs_int" bfs_int_t the_module in
263
264 let bfs_str_t : L.lltype = L.function_type void_ptr_t [| void_ptr_t; str_t |] in
265 let bfs_str_func : L.llvalue = Ldeclare_function "bfs_str" bfs_str_t the_module in

```

```

266
267 let global_fdecls : (L.llvalue * sfdecl) StringMap.t =
268   let function_decl m (sfdecl : sfdecl) =
269     let name = sfdecl.sfname
270     and formal_types =
271       Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) sfdecl.sargs)
272     in let ftype =
273       L.function_type (ltype_of_typ sfdecl.styp) formal_types in
274     StringMap.add name (L.define_function name ftype the_module, sfdecl) m in
275   List.fold_left function_decl StringMap.empty functions in
276
277 let build_function_body local_fdecls sfdecl =
278   let (the_function, _) = try StringMap.find sfdecl.sfname local_fdecls
279     with Not_found -> StringMap.find sfdecl.sfname
280           global_fdecls
281   in
282   let builder = L.builder_at_end context (L.entry_block the_function) in
283   let str_format_str = L.build_global_stringptr "%s\n" "fmt" builder in
284   let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder in
285
286 (* Construct the function's "locals": formal arguments and locally
287    declared variables. Allocate each on the stack, initialize their
288    value, if appropriate, and remember their values in the "locals" map *)
289 let add_arg builder m (t, n) p = L.set_value_name n p;
290   let local = L.build_alloca (ltype_of_typ t) n builder in
291   ignore (L.build_store p local builder);
292   StringMap.add n local m in
293
294 let add_local_var builder m (t, n) =
295   let local_var = L.build_alloca (ltype_of_typ t) n builder
296   in StringMap.add n local_var m in
297
298 let local_vars =
299   List.fold_left2 (add_arg builder) StringMap.empty sfdecl.sargs
300     (Array.to_list (L.params the_function)) in
301
302 (* Return the value for a variable or formal argument *)
303 let lookup vars n = try StringMap.find n vars
304   with Not_found -> StringMap.find n global_vars in
305
306 let funcs' = List.map (fun (ty, name) ->
307   match ty with
308     | A.Fun(ret_t, args_t) ->
309       (ty, {styp = ret_t; fname = name; sargs = List.map (
310         fun t -> (t, "x")) args_t; sbody = []})
311     | _ -> raise A.Unsupported_constructor)
312   (List.filter (fun (ty, _) -> match ty with A.Fun(_) -> true
313     | _ -> false) sfdecl.sargs) in
314
315 let add_local_fdecl vars fdecls (t, n) =
316   match t with
317     | A.Fun(ret_t, args_t) ->
318       StringMap.add n (lookup vars n, {styp = ret_t; fname = n; sargs = List.map (
319         fun t -> (t, "x")) args_t; sbody = []}) fdecls
320     | _ -> raise A.Unsupported_constructor in
321
322 let local_fdecls =
323   List.fold_left (fun m (_, sfdecl) ->
324     StringMap.add sfdecl.sfname (lookup local_vars sfdecl.sfname
325       , sfdecl) m) StringMap.empty funcs' in

```

```

322
323
324 let lookup_func fdecls n = try StringMap.find n fdecls
325           with Not_found -> StringMap.find n global_fdecls
326 in
327
328 let rec expr fdecls vars builder ((ty,e) : sexpr) = match e with
329   | SStringlit s -> L.build_global_stringptr s "str" builder
330   | SIntlit i -> L.const_int i32_t i
331   | SBoollit b -> L.const_int i1_t (if b then 1 else 0)
332   | SVar s -> L.build_load (lookup vars s) s builder
333   | SUop(op, e) ->
334     let e' = expr fdecls vars builder e in
335     (match op with
336      | A.Neg      -> L.build_neg
337      | A.Not      -> L.build_not) e' "tmp" builder
338   | SBinop ((e1_t, e1), op, e2) ->
339     let e1' = expr fdecls vars builder (e1_t, e1)
340     and e2' = expr fdecls vars builder e2 in
341     (match e1_t with
342      | A.String ->
343        let e1' = L.build_call strcmp_func [| e1'; e2' |] "strcmp" builder in
344        let e2' = L.const_int i32_t 0 in
345        (match op with
346          | A.Eq       -> L.build_icmp L.Icmp.Eq
347          | A.Neq      -> L.build_icmp L.Icmp.Ne
348          | A.Lt       -> L.build_icmp L.Icmp.Slt
349          | A.Leq      -> L.build_icmp L.Icmp.Sle
350          | A.Gt       -> L.build_icmp L.Icmp.Sgt
351          | A.Geq      -> L.build_icmp L.Icmp.Sge
352          | _          -> raise A.Unsupported_constructor)
353        e1' e2' "tmp" builder
354      | _ ->
355        (match op with
356          | A.Add      -> L.build_add
357          | A.Sub      -> L.build_sub
358          | A.Mul      -> L.build_mul
359          | A.Div      -> L.build_sdiv
360          | A.And      -> L.build_and
361          | A.Or       -> L.build_or
362          | A.Eq       -> L.build_icmp L.Icmp.Eq
363          | A.Neq      -> L.build_icmp L.Icmp.Ne
364          | A.Lt       -> L.build_icmp L.Icmp.Slt
365          | A.Leq      -> L.build_icmp L.Icmp.Sle
366          | A.Gt       -> L.build_icmp L.Icmp.Sgt
367          | A.Geq      -> L.build_icmp L.Icmp.Sge
368        ) e1' e2' "tmp" builder)
369   | SFUnsig (t, bl, _) ->
370     let t_list = List.map fst bl in
371     let new_fun_t = L.function_type (ltype_of_typ t) (Array.of_list (List.map
372       ltype_of_typ t_list)) in
373     L.define_function "temp" new_fun_t the_module
374   | SFCall ("print", [e]) ->
375     L.build_call print_func [| str_format_str ; (expr fdecls vars builder e) |]
376     "" builder
377   | SFCall ("print_int", [e]) | SFCall ("print_bool", [e]) ->
378     L.build_call print_func [| int_format_str ; (expr fdecls vars builder e) |]
379     "" builder
380   | SFCall (f, act) ->
381     let (fdef, sfdecl) = lookup_func fdecls f in
382     let actuals = List.rev (List.map (expr fdecls vars builder) (List.rev act))
```

```

    in
380 let result = (match sfdecl.styp with A.Void -> "" | _ -> f ^ "_result") in
381   L.build_call fdef (Array.of_list actuals) result builder
382 | SMCall (e, s, args) ->
383   handle_mcall_expr fdecls vars builder ty e args s
384 | SAsn (s, (t, v)) ->
385   (* If e is SNull, change to default value for type s *)
386   let v = match v with
387     | SNull -> (match t with
388       | A.Int -> SIntlit 0
389       | A.Bool -> SBoollit false
390       | A.String -> SStringlit ""
391       | _ -> raise A.Unsupported_constructor)
392     | _ -> v
393   in
394   let e' = expr fdecls vars builder (t, v) in
395   (match t with
396   | A.Fun(_) -> ignore (L.build_store e' (lookup vars s) builder); e'
397   | _ -> ignore (L.build_store e' (lookup vars s) builder); e')
398 | SGExpr(nlist, elist) ->
399   let g = L.build_call create_graph_func [||] "create_graph" builder in
400   ignore (List.map (fun n -> L.build_call add_node_func [| g; expr fdecls
401     vars builder n |] "add_node" builder) nlist);
402   ignore (List.map (fun e -> let f, src', dst' =
403     match e with
404     | (_, SEdgeExpr(src, dst, _)) ->
405       (match src with
406         | (A.Int, _) -> add_edge_int_func
407         | (A.Bool, _) -> add_edge_bool_func
408         | (A.String, _) -> add_edge_str_func
409         | _ -> raise A.Unsupported_constructor),
410       expr fdecls vars builder src,
411       expr fdecls vars builder dst
412         | _ -> raise A.Unsupported_constructor
413       in L.build_call f [| g; expr fdecls vars builder e; src
414         ; dst' |] "add_edge" builder) elist);
415   g
416 | SEdgeExpr(_, _, w) ->
417   let e = L.build_call create_edge_func [||] "edge" builder in
418   let w' = expr fdecls vars builder w in
419   (match w with
420   | (A.Int, SNull) -> ignore (L.build_call set_edge_w_int_func [| e; w'; L.
421     const_int i1_t 0 |] "" builder)
422   | (A.Int, _) -> ignore (L.build_call set_edge_w_int_func [| e; w'; L.
423     const_int i1_t 1 |] "" builder)
424   | (A.Bool, SNull) -> ignore (L.build_call set_edge_w_bool_func [| e; w'; L.
425     const_int i1_t 0 |] "" builder)
426   | (A.Bool, _) -> ignore (L.build_call set_edge_w_bool_func [| e; w'; L.
427     const_int i1_t 1 |] "" builder)
428   | (A.String, SNull) -> ignore (L.build_call set_edge_w_str_func [| e; w'; L.
429     const_int i1_t 0 |] "" builder)
430   | (A.String, _) -> ignore (L.build_call set_edge_w_str_func [| e; w'; L.
431     const_int i1_t 1 |] "" builder)
432   | _ -> raise A.Unsupported_constructor);
433   e
434 | SNodeExpr (l, d) ->
435   let l' = expr fdecls vars builder l in
436   let d' = expr fdecls vars builder d in
437   let n = L.build_call create_node_func [||] "create_node" builder in
438   (match l with
439   | (A.Int, _) -> ignore (L.build_call set_node_label_int_func [| n; l' |]

```

```

    "" builder)
432 | (A.Bool, _) -> ignore (L.build_call set_node_label_bool_func [| n; l' |]
    "" builder)
433 | (A.String, _) -> ignore (L.build_call set_node_label_str_func [| n; l'
    [] "" builder])
434 | _ -> raise A.Unsupported_constructor);
435 (match d with
436 | (A.Int, v) ->
437     if v = SNull
438     then ignore (L.build_call set_node_data_int_func [| n; L.const_int i32_t
        0; L.const_int i1_t 0 |] "" builder)
439     else ignore (L.build_call set_node_data_int_func [| n; d'; L.const_int
        i1_t 1 |] "" builder)
440 | (A.Bool, v) ->
441     if v = SNull
442     then ignore (L.build_call set_node_data_bool_func [| n; L.const_int i1_t
        0; L.const_int i1_t 0 |] "" builder)
443     else ignore (L.build_call set_node_data_bool_func [| n; d'; L.const_int
        i1_t 1 |] "" builder)
444 | (A.String, v) ->
445     if v = SNull
446     then ignore (L.build_call set_node_data_str_func [| n; L.const_null
        str_t; L.const_int i1_t 0 |] "" builder)
447     else ignore (L.build_call set_node_data_str_func [| n; d'; L.const_int
        i1_t 1 |] "" builder)
448 | _ -> raise A.Unsupported_constructor);
449
450     n
451 | SNull ->
452     (match ty with
453 | A.Int -> L.const_null i32_t
454 | A.Bool -> L.const_null i1_t
455 | A.String -> L.const_null str_t
456 | _ -> L.const_null void_ptr_t)
457 | SNoexpr ->
458     L.undef (L.void_type context) (* placeholder *)
459
460 and handle_mcall_expr fdecls vars builder ty e args = function
461 | "set_node" ->
462     (match args with
463 | ((A.Node(_), _) as n) :: [] ->
464         let g_ptr = expr fdecls vars builder e in
465         let n_ptr = expr fdecls vars builder n in
466         let n_ptr' = L.build_call clone_node_func [| n_ptr |] "clone_node"
467             builder in
468         L.build_call graph_set_node_func [| g_ptr; n_ptr' |] "tmp_data" builder
469         | _ -> raise A.Unsupported_constructor)
470 | "remove_node" ->
471     (match args with
472 | ((l_typ, _) as l) :: [] ->
473         let g_ptr = expr fdecls vars builder e in
474         let l' = expr fdecls vars builder l in
475         (match l_typ with
476 | A.Int -> L.build_call remove_node_int_func [| g_ptr; l' |] "tmp_data"
477             builder
478 | A.String -> L.build_call remove_node_str_func [| g_ptr; l' |] "
479             tmp_data" builder
480 | A.Bool -> L.build_call remove_node_bool_func [| g_ptr; l' |] "
481             tmp_data" builder
482 | _ -> raise A.Unsupported_constructor)
483 | _ -> raise A.Unsupported_constructor)
484 | "remove_edge" ->

```

```

480     (match args with
481     | ((src_typ, _) as src) :: dst :: [] ->
482         let g_ptr = expr fdecls vars builder e in
483         let src_ptr = expr fdecls vars builder src in
484         let dst_ptr = expr fdecls vars builder dst in
485         let e_ptr = (match src_typ with
486             | A.Int -> L.build_call get_edge_by_src_and_dst_int_func [| g_ptr;
487                 src_ptr; dst_ptr |] "get_edge_by_src_and_dst_int" builder
488             | A.Bool -> L.build_call get_edge_by_src_and_dst_bool_func [| g_ptr;
489                 src_ptr; dst_ptr |] "get_edge_by_src_and_dst_bool" builder
490             | A.String -> L.build_call get_edge_by_src_and_dst_str_func [| g_ptr;
491                 src_ptr; dst_ptr |] "get_edge_by_src_and_dst_str" builder
492             | _ -> raise A.Unsupported_constructor) in
493         L.build_call remove_edge_func [| g_ptr; e_ptr |] "remove_edge" builder
494     | _ -> raise A.Unsupported_constructor)
495 
496 | "get_node" ->
497     (match e, args with
498     | (A.Graph(lt, _, _), _, label :: []) ->
499         let g_ptr = expr fdecls vars builder e in
500         let label' = expr fdecls vars builder label in
501         (match lt with
502             | A.Int -> L.build_call get_node_by_label_int_opt_func [| g_ptr; label';
503                 |] "get_node_by_label" builder
504             | A.Bool -> L.build_call get_node_by_label_bool_opt_func [| g_ptr;
505                 label' |] "get_node_by_label" builder
506             | A.String -> L.build_call get_node_by_label_str_opt_func [| g_ptr;
507                 label' |] "get_node_by_label" builder
508             | _ -> raise A.Unsupported_constructor)
509     | _ -> raise A.Unsupported_constructor)
510 
511 | "get_weight" ->
512     (match e, args with
513     | (A.Graph(lt, _, wt), _, src :: dst :: [] ->
514         let g_ptr = expr fdecls vars builder e in
515         let src' = expr fdecls vars builder src in
516         let dst' = expr fdecls vars builder dst in
517         let e_ptr = (match lt with
518             | A.Int -> L.build_call get_edge_by_src_and_dst_int_func [| g_ptr;
519                 src'; dst' |] "get_edge_by_src_and_dst_int" builder
520             | A.Bool -> L.build_call get_edge_by_src_and_dst_bool_func [| g_ptr;
521                 src'; dst' |] "get_edge_by_src_and_dst_bool" builder
522             | A.String -> L.build_call get_edge_by_src_and_dst_str_func [| g_ptr;
523                 src'; dst' |] "get_edge_by_src_and_dst_str" builder
524             | _ -> raise A.Unsupported_constructor) in
525         (match wt with
526             | A.Int -> L.build_call get_edge_w_int_func [| e_ptr |] "get_edge_w"
527                 builder
528             | A.Bool -> L.build_call get_edge_w_bool_func [| e_ptr |] "get_edge_w"
529                 builder
530             | A.String -> L.build_call get_edge_w_str_func [| e_ptr |] "
531                 get_edge_w" builder
532             | _ -> raise A.Unsupported_constructor)
533     | _ -> raise A.Unsupported_constructor)
534 
535 | "get_name" ->
536     let n_ptr = expr fdecls vars builder e in
537     let ret = L.build_call get_node_label_func [| n_ptr |] "tmp_data" builder
538     in
539     (match ty with
540     | A.String -> ret
541     | A.Int -> L.build_load (L.build_bitcast ret i32_ptr_t "bitcast" builder) "
542         deref" builder
543     | A.Bool -> L.build_load (L.build_bitcast ret i32_ptr_t "bitcast" builder)

```

```

        "deref" builder
527    | _ -> raise A.Unsupported_constructor)
528  | "has_node" ->
529    (match args with
530    | ((n_typ, _) as n) :: [] ->
531      let g_ptr = expr fdecls vars builder e in
532      let n' = expr fdecls vars builder n in
533      (match n_typ with
534      | A.Int -> L.build_call graph_has_node_int_func [| g_ptr; n' |] "
535        tmp_data" builder
536      | A.String -> L.build_call graph_has_node_str_func [| g_ptr; n' |] "
537        tmp_data" builder
538      | A.Bool -> L.build_call graph_has_node_bool_func [| g_ptr; n' |] "
539        tmp_data" builder
540      | _ -> raise A.Unsupported_constructor)
541    | _ -> raise A.Unsupported_constructor)
542  | "get_data" ->
543    let n_ptr = expr fdecls vars builder e in
544    let ret = L.build_call get_node_data_func [| n_ptr |] "tmp_data" builder in
545    (match ty with
546    | A.String -> ret
547    | A.Bool -> L.build_load (L.build_bitcast ret i32_ptr_t "bitcast" builder)
548      "deref" builder
549    | A.Int -> L.build_load (L.build_bitcast ret i32_ptr_t "bitcast" builder) "
550      deref" builder
551    | _ -> raise A.Unsupported_constructor)
552  | "set_edge" ->
553    (match args with
554    | ((src_typ, _) as src) :: dst :: ((w_typ, _) as w) :: [] ->
555      let g_ptr = expr fdecls vars builder e in
556      let src' = expr fdecls vars builder src in
557      let dst' = expr fdecls vars builder dst in
558      let w' = expr fdecls vars builder w in
559      (match (src_typ, w_typ) with
560      | (A.Int, A.Bool) -> L.build_call graph_set_edge_int_bool_func [| g_ptr;
561        ; src'; dst'; w' |] "tmp_data" builder
562      | (A.Int, A.Int) -> L.build_call graph_set_edge_int_int_func [| g_ptr;
563        ; src'; dst'; w' |] "tmp_data" builder
564      | (A.Bool, A.Bool) -> L.build_call graph_set_edge_bool_bool_func [| g_ptr;
565        ; src'; dst'; w' |] "tmp_data" builder
566      | (A.Bool, A.Int) -> L.build_call graph_set_edge_bool_int_func [| g_ptr;
567        ; src'; dst'; w' |] "tmp_data" builder
568      | (A.String, A.Bool) -> L.build_call graph_set_edge_str_bool_func [| g_ptr;
569        ; src'; dst'; w' |] "tmp_data" builder
570      | (A.Bool, A.String) -> L.build_call graph_set_edge_bool_str_func [| g_ptr;
571        ; src'; dst'; w' |] "tmp_data" builder
572      | (A.String, A.Int) -> L.build_call graph_set_edge_str_int_func [| g_ptr;
573        ; src'; dst'; w' |] "tmp_data" builder
574      | (A.Int, A.String) -> L.build_call graph_set_edge_int_str_func [| g_ptr;
575        ; src'; dst'; w' |] "tmp_data" builder
576      | (A.String, A.String) -> L.build_call graph_set_edge_str_str_func [| g_ptr;
577        ; src'; dst'; w' |] "tmp_data" builder
578      | _ -> raise A.Unsupported_constructor)
579    | ((src_typ, _) as src) :: dst :: [] ->
580      let g_ptr = expr fdecls vars builder e in
581      let src' = expr fdecls vars builder src in
582      let dst' = expr fdecls vars builder dst in
583      (match src_typ with
584      | A.Int -> L.build_call graph_set_edge_int_func [| g_ptr; src'; dst';
585        |] "tmp_data" builder
586      | A.String -> L.build_call graph_set_edge_str_func [| g_ptr; src'; dst';

```

```

        , [] "tmp_data" builder
572    | A.Bool -> L.build_call graph_set_edge_bool_func [| g_ptr; src'; dst'
573        , [] "tmp_data" builder
574        | _ -> raise A.Unsupported_constructor)
575        | _ -> raise A.Unsupported_constructor)
576    | "set_data" ->
577        (match args with
578        | ((dt, dv) as d) :: [] ->
579            let n_ptr = expr fdecls vars builder e in
580            let d_ptr = expr fdecls vars builder d in
581            (match dt with
582            | A.Int ->
583                if dv = SNull
584                    then L.build_call set_node_data_int_func [| n_ptr; L.const_int
585                        i32_t 0; L.const_int i1_t 0 |] "" builder
586                    else L.build_call set_node_data_int_func [| n_ptr; d_ptr; L.
587                        const_int i1_t 1 |] "" builder
588            | A.Bool ->
589                if dv = SNull
590                    then L.build_call set_node_data_bool_func [| n_ptr; L.const_int
591                        i1_t 0; L.const_int i1_t 0 |] "" builder
592                    else L.build_call set_node_data_bool_func [| n_ptr; d_ptr; L.
593                        const_int i1_t 1 |] "" builder
594            | A.String ->
595                if dv = SNull
596                    then L.build_call set_node_data_str_func [| n_ptr; L.const_null
597                        str_t; L.const_int i1_t 0 |] "" builder
598                    else L.build_call set_node_data_str_func [| n_ptr; d_ptr; L.
599                        const_int i1_t 1 |] "" builder
600            | _ -> raise A.Unsupported_constructor)
601        | _ -> raise A.Unsupported_constructor)
602    | "are_neighbors" ->
603        (match e, args with
604        | (A.Graph(lt, _, _), _), src :: dst :: [] ->
605            let g_ptr = expr fdecls vars builder e in
606            let src' = expr fdecls vars builder src in
607            let dst' = expr fdecls vars builder dst in
608            (match lt with
609            | A.Int ->
610                L.build_call are_neighbors_int_func [| g_ptr; src'; dst' |] "
611                    are_neighbors" builder
612            | A.Bool ->
613                L.build_call are_neighbors_bool_func [| g_ptr; src'; dst' |] "
614                    are_neighbors" builder
615            | A.String ->
616                L.build_call are_neighbors_str_func [| g_ptr; src'; dst' |] "
617                    are_neighbors" builder
618            | _ -> raise A.Unsupported_constructor)
619        | _ -> raise A.Unsupported_constructor)
620    | "is_empty" ->
621        (match e with
622        | (A.Graph(_), _) ->
623            let g_ptr = expr fdecls vars builder e in
624            L.build_call is_empty_func [| g_ptr |] "is_empty" builder
625            | _ -> raise A.Unsupported_constructor)
626    | "print" ->
627        (match e with
628        | (A.Graph(_), _) ->
629            let g_ptr = expr fdecls vars builder e in
630            L.build_call print_graph_func [| g_ptr |] "" builder
631            | (A.Node(_), _) ->

```

```

622         let n_ptr = expr fdecls vars builder e in
623         L.build_call print_node_func [| n_ptr |] "" builder
624         | _ -> raise A.Unsupported_constructor)
625     | "neighbors" ->
626         (match e, args with
627         | (A.Graph(_, _), ((nlt, _) as nl) :: [] ->
628             let g_ptr = expr fdecls vars builder e in
629             let nl' = expr fdecls vars builder nl in
630             let n_ptr = (match nlt with
631                 | A.Int | A.Bool -> L.build_call get_node_by_label_int_func
632                     [| g_ptr; nl' |] "get_node_by_label_int" builder
633                 | A.String -> L.build_call get_node_by_label_str_func [| g_ptr; nl' |] "get_node_by_label_str" builder
634                 | _ -> raise A.Unsupported_constructor) in
635             L.build_call neighbors_one_arg_func [| n_ptr |] "neihghbors_one_arg"
636             builder
637         | (A.Graph(_, _), ((nlt, _) as nl) :: level :: include_current :: [] ->
638             let g_ptr = expr fdecls vars builder e in
639             let nl' = expr fdecls vars builder nl in
640             let level' = expr fdecls vars builder level in
641             let include_current' = expr fdecls vars builder include_current in
642             let n_ptr = (match nlt with
643                 | A.Int | A.Bool -> L.build_call get_node_by_label_int_func
644                     [| g_ptr; nl' |] "get_node_by_label_int" builder
645                 | A.String -> L.build_call get_node_by_label_str_func [| g_ptr; nl' |] "get_node_by_label_str" builder
646                 | _ -> raise A.Unsupported_constructor) in
647             L.build_call neighbors_func [| n_ptr; level'; include_current' |] "
648             neighbors" builder
649             | _ -> raise A.Unsupported_constructor)
650     | "find" ->
651         (match e, args with
652         | (A.Graph(_, dt, _), _, d) :: [] ->
653             let g_ptr = expr fdecls vars builder e in
654             let d' = expr fdecls vars builder d in
655             (match dt with
656                 | A.Int -> L.build_call find_data_int_func [| g_ptr; d' |] "find_data"
657                     builder
658                 | A.Bool -> L.build_call find_data_bool_func [| g_ptr; d' |] "find_data"
659                     " builder
660                 | A.String -> L.build_call find_data_str_func [| g_ptr; d' |] "find_data"
661                     " builder
662                 | _ -> raise A.Unsupported_constructor)
663             | _ -> raise A.Unsupported_constructor)
664     | "dfs" ->
665         (match e, args with
666         | (A.Graph(lt, _, _), l) :: [] ->
667             let g_ptr = expr fdecls vars builder e in
668             let l' = expr fdecls vars builder l in
669             (match lt with
670                 | A.Int -> L.build_call dfs_int_func [| g_ptr; l' |] "dfs_int" builder
671                 | A.Bool -> L.build_call dfs_int_func [| g_ptr; l' |] "dfs_int" builder
672                 | A.String -> L.build_call dfs_str_func [| g_ptr; l' |] "dfs_str" builder
673                 | _ -> raise A.Unsupported_constructor)
674             | _ -> raise A.Unupported_constructor)
675     | "bfs" ->
676         (match e, args with
677         | (A.Graph(lt, _, _), l) :: [] ->
678             let g_ptr = expr fdecls vars builder e in
679             let l' = expr fdecls vars builder l in
680             (match lt with

```

```

674     | A.Int -> L.build_call bfs_int_func [|g_ptr; l'|] "bfs_int" builder
675     | A.Bool -> L.build_call bfs_int_func [|g_ptr; l'|] "bfs_int" builder
676     | A.String -> L.build_call bfs_str_func [|g_ptr; l'|] "bfs_str" builder
677     | _ -> raise A.Unsupported_constructor)
678   | _ -> raise A.Unsupported_constructor)
679 | _ -> raise A.Unsupported_constructor
680 in
681
682 let add_terminal builder instr =
683   (* The current block where we're inserting instr *)
684   match L.block_terminator (L.insertion_block builder) with
685   | Some _ -> ()
686   | None -> ignore (instr builder)
687 in
688
689 let rec stmt (fdecls, vars, builder) = function
690   | SBlock sl ->
691     List.fold_left stmt (fdecls, vars, builder) sl
692   (* Generate code for this expression, return resulting builder *)
693   | SExpr e ->
694     let _ = expr fdecls vars builder e in (fdecls, vars, builder)
695   (* fun f = ... (...) ... *)
696   | SVdecl (ty, s, e) ->
697     (match e with
698     | (A.Fun(_), SAsn(var_name, (A.Fun(_), SFunsig(t, bl, e')))) ->
699       (* Make the function's signature*)
700       let sfdecl = {styp = t; fname = var_name; sargs = bl; sbody = [SReturn(e
701         ')]} in
702       (* Get the function's llvalue*)
703       let vars' = add_local_var builder vars (ty, s) in
704       let ll_fun_val = expr fdecls vars' builder e in
705       let fdecls' = StringMap.add var_name (ll_fun_val, sfdecl) fdecls in
706       let builder' = L.builder_at_end context (L.entry_block ll_fun_val) in
707       let new_locals = List.fold_left2 (add_arg builder') StringMap.empty sfdecl
708         .sargs (Array.to_list (L.params ll_fun_val)) in
709       let (_, _, builder'') = stmt (fdecls', new_locals, builder') (SBlock
710         sfdecl.sbody) in
711
712       (add_terminal builder'') (match sfdecl.styp with
713         A.Void -> L.build_ret_void
714         | t -> L.build_ret (L.const_int (ltype_of_typ t) 0));
715
716       (fdecls', vars', builder)
717     | _ ->
718       let vars' = add_local_var builder vars (ty, s) in
719       let fdecls' = (match ty with A.Fun(_) -> add_local_fdecl vars' fdecls (ty,
720         s) | _ -> fdecls) in
721       let _ = expr fdecls vars' builder e in (fdecls', vars', builder))
722
723   | SReturn e ->
724     let _ = match sfdecl.styp with
725       (* Special "return nothing" instr *)
726       | A.Void -> L.build_ret_void builder
727       (* Build return statement *)
728       | _ -> L.build_ret (expr fdecls vars builder e) builder
729     in (fdecls, vars, builder)
730   | SIf (p, then_stmt, else_stmt) ->
731     let bool_val = expr fdecls vars builder p in
732     let merge_bb = L.append_block context "merge" the_function in
733
734     let then_bb = L.append_block context "then" the_function in

```

```

731     let _, _, builder' = stmt (fdecls, vars, L.builder_at_end context then_bb)
732         then_stmt in
733     add_terminal builder' (L.build_br merge_bb);
734
735     let else_bb = L.append_block context "else" the_function in
736     let _, _, builder' = stmt (fdecls, vars, L.builder_at_end context else_bb)
737         else_stmt in
738     add_terminal builder' (L.build_br merge_bb);
739
740     ignore (L.build_cond_br bool_val then_bb else_bb builder);
741     (fdecls, vars, L.builder_at_end context merge_bb)
742
743 | SWhile (p, body) ->
744     let p_bb = L.append_block context "while" the_function in
745         ignore (L.build_br p_bb builder);
746
747     let body_bb = L.append_block context "while_body" the_function in
748     let _, _, builder' = stmt (fdecls, vars, L.builder_at_end context body_bb)
749         body in
750     add_terminal builder' (L.build_br p_bb);
751
752     let p_builder = L.builder_at_end context p_bb in
753     let bool_val = expr fdecls vars p_builder p in
754
755     let merge_bb = L.append_block context "merge" the_function in
756     ignore (L.build_cond_br bool_val body_bb merge_bb p_builder);
757     (fdecls, vars, L.builder_at_end context merge_bb)
758
759 | SFor (e1, p, e2, body) -> stmt (fdecls, vars, builder)
760     (SBlock [SExpr e1 ; SWhile (p, SBlock [body ; SExpr e2]) ] )
761 | SForNode (n, g, body) ->
762     (match g with
763      | (A.Graph(lt, dt, _, _), _) ->
764          let graph_ptr = expr fdecls vars builder g in
765
766          (* allocate space for n, add to symbol table, and initially set to head
767             of node linked list *)
768          let n_ptr = L.build_alloca (ltype_of_typ (A.Node(lt, dt))) n builder in
769          let vars = StringMap.add n n_ptr vars in
770          let hd_node = L.build_call graph_to_node_iterable_func [| graph_ptr |] "
771              hd_node" builder in
772          ignore(L.build_store hd_node n_ptr builder);
773
774          (* create predicate block *)
775          let p_bb = L.append_block context "while" the_function in
776          ignore (L.build_br p_bb builder);
777
778          (* while body block *)
779          let body_bb = L.append_block context "while_body" the_function in
780          let body_builder = L.builder_at_end context body_bb in
781          let _, _, builder' = stmt (fdecls, vars, body_builder) body in
782          (* change curr_node to be pointer to next node *)
783          let curr_node = L.build_load n_ptr "curr_node" builder' in
784          let next_node = L.build_call get_graph_next_node_func [| curr_node |] "
785              next_node" builder' in
786          ignore(L.build_store next_node n_ptr builder');
787          add_terminal builder' (L.build_br p_bb);
788
789          (* define predicate *)
790          let p_builder = L.builder_at_end context p_bb in
791          let n_val = L.build_load n_ptr "node_tmp" p_builder in

```

```

786     let bool_val = L.build_is_not_null n_val "bool_val" p_builder in
787
788     (* merge *)
789     let merge_bb = L.append_block context "merge" the_function in
790     ignore (L.build_cond_br bool_val body_bb merge_bb p_builder);
791     (fdecls, vars, L.builder_at_end context merge_bb)
792     | _ -> raise A.Unsupported_constructor)
793   | SForEdge (src, dst, w, g, body) ->
794     (match g with
795     | (A.Graph(lt, dt, wt), _) ->
796       let graph_ptr = expr fdecls vars builder g in
797
798       (* allocate space for edge variables, add to symbol table, and initially
799          set to head of edge linked list *)
800       let edge_ptr = L.build_alloca void_ptr_t "edge" builder in
801       let src_ptr = L.build_alloca (ltype_of_typ (A.Node(lt, dt))) "src"
802         builder in
803       let dst_ptr = L.build_alloca (ltype_of_typ (A.Node(lt, dt))) "dst"
804         builder in
805       let w_ptr = L.build_alloca (ltype_of_typ wt) "w" builder in
806       let vars = StringMap.add src src_ptr (StringMap.add dst dst_ptr (
807         StringMap.add w w_ptr vars)) in
808       let hd_edge = L.build_call graph_to_edge_iterable_func [| graph_ptr |] "
809         hd_edge" builder in
810       let hd_edge_src = L.build_call get_edge_src_func [| hd_edge |] "
811         hd_edge_src" builder in
812       let hd_edge_dst = L.build_call get_edge_dst_func [| hd_edge |] "
813         hd_edge_dst" builder in
814       let hd_edge_w = (match wt with
815         | A.Int -> L.build_call get_edge_w_int_func [| hd_edge |] "hd_edge_w"
816           builder
817         | A.Bool -> L.build_call get_edge_w_bool_func [| hd_edge |] "hd_edge_w"
818           builder
819         | A.String -> L.build_call get_edge_w_str_func [| hd_edge |] "hd_edge_w"
820           builder
821         | _ -> raise A.Unsupported_constructor) in
822       ignore(L.build_store hd_edge edge_ptr builder);
823       ignore(L.build_store hd_edge_src src_ptr builder);
824       ignore(L.build_store hd_edge_dst dst_ptr builder);
825       ignore(L.build_store hd_edge_w w_ptr builder);
826
827     (* create predicate block *)
828     let p_bb = L.append_block context "while" the_function in
829     ignore (L.build_br p_bb builder);
830
831     (* while body block *)
832     let body_bb = L.append_block context "while_body" the_function in
833     let body_builder = L.builder_at_end context body_bb in
834     let _, _, builder' = stmt (fdecls, vars, body_builder) body in
835     (* change curr_edge to be pointer to next edge *)
836     let curr_edge = L.build_load edge_ptr "curr_edge" builder' in
837     let next_edge = L.build_call get_graph_next_edge_func [| curr_edge |] "
838       next_edge" builder' in
839     let next_edge_src = L.build_call get_edge_src_func [| next_edge |] "
840       next_edge_src" builder' in
841     let next_edge_dst = L.build_call get_edge_dst_func [| next_edge |] "
842       next_edge_dst" builder' in
843     let next_edge_w = (match wt with
844       | A.Int -> L.build_call get_edge_w_int_func [| next_edge |] "next_edge_w"
845         builder,
846       | A.Bool -> L.build_call get_edge_w_bool_func [| next_edge |] "

```

```

    next_edge_w" builder'
833 | A.String -> L.build_call get_edge_w_str_func [| next_edge |] "
     next_edge_w" builder'
834 | _ -> raise A.Unsupported_constructor) in
835 ignore(L.build_store next_edge edge_ptr builder');
836 ignore(L.build_store next_edge_src src_ptr builder');
837 ignore(L.build_store next_edge_dst dst_ptr builder');
838 ignore(L.build_store next_edge_w w_ptr builder');
839 add_terminal builder' (L.build_br p_bb);
840
841 (* define predicate *)
842 let p_builder = L.builder_at_end context p_bb in
843 let e_val = L.build_load edge_ptr "edge_tmp" p_builder in
844 let bool_val = L.build_is_not_null e_val "bool_val" p_builder in
845
846 (* merge *)
847 let merge_bb = L.append_block context "merge" the_function in
848 ignore (L.build_cond_br bool_val body_bb merge_bb p_builder);
849 (fdecls, vars, L.builder_at_end context merge_bb)
850 | _ -> raise A.Unsupported_constructor)
851 in
852
853 let (_, _, builder) = stmt (local_fdecls, local_vars, builder) (SBlock sfdecl.
854     sbody) in
855
856 add_terminal builder (match sfdecl.styp with
857     A.Void -> L.build_ret_void
858     | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
859 in
860 List.iter (build_function_body StringMap.empty) functions;
the_module

```

9.6.2 hippograph.ml

```

1 type action = Ast | Sast | LLVM_IR | Compile
2
3 let _ =
4   let action = ref Compile in
5   let set_action a () = action := a in
6   let speclist = [
7     ("-a", Arg.Unit (set_action Ast), "Print the AST");
8     ("-s", Arg.Unit (set_action Sast), "Print the SAST");
9     ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
10    ("-c", Arg.Unit (set_action Compile),
11      "Check and print the generated LLVM IR (default)");
12  ] in
13  let usage_msg = "usage: ./microcc.native [-a|-l|-c] [file.mc]" in
14  let channel = ref stdin in
15  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
16  let lexbuf = Lexing.from_channel !channel in
17  let ast = Parser.program Scanner.token lexbuf in
18  match !action with
19  | Ast -> print_string (Ast.string_of_program ast)
20  | _ ->
21    let sast = Semant.check ast in
22    match !action with
23    | Ast -> ()
24    | Sast -> print_string (Sast.string_of_sprogram sast)
25    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
26    | Compile -> let m = Codegen.translate sast in

```

```

28     Llvm_analysis.assert_valid_module m;
29     print_string (Llvm.string_of_llmodule m)

```

9.6.3 Makefile

```

1  hippograph.native:
2      opam config exec -- \
3          ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \
4              hippograph.native
5
6
7 .PHONY : test
8 test : all testall.sh
9 ./testall.sh
10
11 .PHONY : all
12 all : hippograph.native graph.o
13
14 .PHONY: clean
15 clean:
16     ocamlbuild -clean
17     rm -rf ocaml LLVM *.diff
18
19 redo:
20     ocamlbuild -clean
21     rm -rf ocaml LLVM *.diff
22     make hippograph.native
23
24 FILE=""
25 run: all run.sh
26     ./run.sh $(FILE)

```

9.6.4 run.sh

```

1 #!/bin/sh
2
3 # script for running one test
4
5 # Path to the LLVM interpreter
6 #LLI="lli"
7 LLI="/usr/local/opt/llvm/bin/lli"
8
9 # Path to the LLVM compiler
10 LLC="/usr/local/opt/llvm/bin/llc"
11
12 # Path to the C compiler
13 CC="cc"
14
15 # Path to the hippograph compiler. Usually "./hippograph.native"
16 # Try "_build/hippograph.native" if ocamlbuild was unable to create a symbolic link.
17 HIPPOGRAPH="./hippograph.native"
18 #HIPPOGRAPH="_build/hippograph.native"
19
20 HPG=".hpg"
21 BASENAME="${1%$HPG}"
22
23 if [ "$#" -ne 1 ]; then
24     echo "usage: ./run.sh filename.hpg"
25     exit
26 fi
27

```

```

28 generatedfiles="$generatedfiles $BASENAME.ll $BASENAME.s $BASENAME.exe $BASENAME.out
29     " &&
29 "$HIPPOGRAPH" "$1" > "$BASENAME.ll"
30 "$LLC" "-relocation-model=pic" "$BASENAME.ll" > "$BASENAME.s"
31 "$CC" "-o" "$BASENAME.exe" "$BASENAME.s" "graph.o"
32 "./$BASENAME.exe"
33 rm -f $generatedfiles

```

9.6.5 testall.sh

```

1 #!/bin/sh
2
3 # Regression testing script for Hippograph
4 # Step through a list of files
5 # Compile, run, and check the output of each expected-to-work test
6 # Compile and check the error of each expected-to-fail test
7
8 # Path to the LLVM interpreter
9 LLI="lli"
10 #LLI="/usr/local/opt/llvm/bin/lli"
11
12 # Path to the LLVM compiler
13 LLC="llc"
14
15 # Path to the C compiler
16 CC="cc"
17
18 # Path to the hippograph compiler. Usually "./hippograph.native"
19 # Try "_build/hippograph.native" if ocamlbuild was unable to create a symbolic link.
20 HIPPOGRAPH="./hippograph.native"
21 #HIPPOGRAPH="_build/hippograph.native"
22
23 # Set time limit for all operations
24 ulimit -t 30
25
26 globallog=testall.log
27 rm -f $globallog
28 error=0
29 globalerror=0
30
31 keep=0
32
33 Usage() {
34     echo "Usage: testall.sh [options] [.hpg files]"
35     echo "-k      Keep intermediate files"
36     echo "-h      Print this help"
37     exit 1
38 }
39
40 SignalError() {
41     if [ $error -eq 0 ] ; then
42         echo "FAILED"
43         error=1
44     fi
45     echo "    $1"
46 }
47
48 # Compare <outfile> <reffile> <difffile>
49 # Compares the outfile with reffile. Differences, if any, written to difffile
50 Compare() {
51     generatedfiles="$generatedfiles $3"

```

```

52     echo diff -b $1 $2 ">" $3 1>&2
53     diff -b "$1" "$2" > "$3" 2>&1 || {
54     SignalError "$1 differs"
55     echo "FAILED $1 differs from $2" 1>&2
56     }
57 }
58
59 # Run <args>
60 # Report the command, run it, and report any errors
61 Run() {
62     echo $* 1>&2
63     eval $* || {
64     SignalError "$1 failed on $*"
65     return 1
66     }
67 }
68
69 # RunFail <args>
70 # Report the command, run it, and expect an error
71 RunFail() {
72     echo $* 1>&2
73     eval $* && {
74     SignalError "failed: $* did not report an error"
75     return 1
76     }
77     return 0
78 }
79
80 Check() {
81     error=0
82     basename='echo $1 | sed 's/.*\\///
83                           s/.hpg//,'
84     reffile='echo $1 | sed 's/.hpg$//,'
85     basedir="'echo $1 | sed 's/\\/[^\\/]*$//','."
86
87     echo -n "$basename..."
88
89     echo 1>&2
90     echo ##### Testing $basename" 1>&2
91
92     generatedfiles=""
93
94     generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe ${{
95             basename}.out" &&
96     Run "$HIPPOGRAPH" "$1" ">" "${basename}.ll" &&
97     Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
98     Run "$CC" "-o" "${basename}.exe" "${basename}.s" "graph.o" &&
99     Run "./${basename}.exe" > "${basename}.out" &&
100    Compare ${basename}.out ${reffile}.out ${basename}.diff
101
102    # Report the status and clean up the generated files
103
104    if [ $error -eq 0 ] ; then
105        if [ $keep -eq 0 ] ; then
106            rm -f $generatedfiles
107        fi
108        echo "OK"
109        else
110        echo "##### FAILED" 1>&2
111        globalerror=$error

```

```

112     fi
113 }
114
115 CheckFail() {
116     error=0
117     basename='echo $1 | sed 's/.*\/\/\/
118                         s/.hpg//,'
119     reffile='echo $1 | sed 's/.hpg$//,'
120     basedir="'echo $1 | sed 's/\//[^\/]*$/','."
121
122     echo -n "$basename..."
123
124     echo 1>&2
125     echo "##### Testing $basename" 1>&2
126
127     generatedfiles=""
128
129     generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
130     RunFail "$HIPPOGRAPH" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
131     Compare ${basename}.err ${reffile}.err ${basename}.diff
132
133     # Report the status and clean up the generated files
134
135     if [ $error -eq 0 ] ; then
136     if [ $keep -eq 0 ] ; then
137         rm -f $generatedfiles
138     fi
139     echo "OK"
140     echo "##### SUCCESS" 1>&2
141     else
142     echo "##### FAILED" 1>&2
143     globalerror=$error
144     fi
145 }
146
147 while getopts kdps h c; do
148     case $c in
149     k) # Keep intermediate files
150         keep=1
151         ;;
152     h) # Help
153         Usage
154         ;;
155     esac
156 done
157
158 shift `expr $OPTIND - 1`
159
160 LLIFail() {
161     echo "Could not find the LLVM interpreter \"\$LLI\"."
162     echo "Check your LLVM installation and/or modify the LLI variable in testall.sh"
163     exit 1
164 }
165
166 which "$LLI" >> $globallog || LLIFail
167
168 if [ ! -f graph.o ]
169 then
170     echo "Could not find graph.o"
171     echo "Try \"make graph.o\""
172     exit 1

```

```

173 fi
174
175 if [ $# -ge 1 ]
176 then
177     files=$@
178 else
179     files="test/test-*.hpg test/fail-*.hpg"
180 fi
181
182 for file in $files
183 do
184     case $file in
185     *test-*)
186         Check $file 2>> $globallog
187         ;;
188     *fail-*)
189         CheckFail $file 2>> $globallog
190         ;;
191     *)
192         echo "unknown file type $file"
193         globalerror=1
194         ;;
195     esac
196 done
197
198 exit $globalerror

```

9.7 C library

9.7.1 graph.c

```

1 /* Authors:
2  Benjamin Lewinter bsl2121
3  Irina Mateescu im2441
4  Harry Smith hs3061
5  Yasunari Watanabe yw3239
6 */
7
8 #include <string.h>
9 #include <stdlib.h>
10 #include <stdio.h>
11
12 /* constants */
13
14 int VOIDTYPE = 1;
15 int INTTYPE = 2;
16 int STRTYPE = 3;
17 int BOOLTYPE = 4;
18
19 /* data structures */
20
21 typedef union primitive {
22     int *i;
23     char *s;
24     void *v;
25 } primitive;
26
27 typedef struct node node;
28
29 typedef struct edge {
30     node *src;

```

```

31     node *dst;
32     primitive *w;
33     int w_typ;
34     struct edge *next;
35     int has_val;
36 } edge;
37
38 typedef struct neighbor_list_item {
39     edge *edge;
40     struct neighbor_list_item *next;
41 } neighbor_list_item;
42
43 typedef struct neighbor_list {
44     neighbor_list_item *hd;
45 } neighbor_list;
46
47 struct node {
48     primitive *label;
49     int label_typ;
50     primitive *data;
51     int data_typ;
52     int has_val;
53     neighbor_list *neighbor_list;
54     node *next;
55 };
56
57 typedef struct node_list {
58     node *hd;
59 } node_list;
60
61 typedef struct edge_list {
62     edge *hd;
63 } edge_list;
64
65 typedef struct graph {
66     node_list *node_list;
67     edge_list *edge_list;
68 } graph;
69
70 typedef struct q_item {
71     node *n;
72     struct q_item *next;
73 } q_item;
74
75 typedef struct queue {
76     q_item *hd;
77     q_item *tl;
78 } queue;
79
80 /* create primitive of type */
81
82 void *create_prim_int(int i) {
83     primitive *p = (primitive *) malloc(sizeof(primitive));
84     p -> i = (int *) malloc(sizeof(int));
85     *(p -> i) = i;
86     return (void *) p;
87 }
88
89 void *create_prim_str(char *s) {
90     primitive *p = (primitive *) malloc(sizeof(primitive));
91     p -> s = s;

```

```

92     return (void *) p;
93 }
94
95 primitive *clone_primitive(primitive *p) {
96     if (p == NULL) return NULL;
97
98     primitive *p_cp = (primitive *) malloc(sizeof(primitive));
99     memcpy(p_cp, p, sizeof(primitive));
100    return p_cp;
101 }
102
103 /* NODES */
104
105 void *create_neighbor_list_item(edge *e) {
106     neighbor_list_item *nli = (neighbor_list_item *) malloc(sizeof(neighbor_list_item)
107         );
108     nli -> edge = e;
109     nli -> next = NULL;
110     return (void *) nli;
111 }
112
113 void *create_neighbor_list() {
114     neighbor_list *nl = (neighbor_list *) malloc(sizeof(neighbor_list));
115     nl -> hd = NULL;
116     return (void *) nl;
117 }
118
119 void *create_node() {
120     node *n = (node *) malloc(sizeof(node));
121     n -> label = NULL;
122     n -> data = 0;
123     n -> has_val = 0;
124     n -> neighbor_list = create_neighbor_list();
125     n -> neighbor_list -> hd = NULL;
126     n -> next = NULL;
127     return (void *) n;
128 }
129
130 int cmp_node_label(node *n1, node *n2) {
131     // return 0 if equal
132     int lt = n1 -> label_typ;
133     if (lt == INTTYPE || lt == BOOLTYPE) {
134         if (*(n1 -> label -> i) == *(n2 -> label -> i)) return 0;
135         else return -1;
136     } else if (lt == STRTYPE) {
137         return strcmp(n1 -> label -> s, n2 -> label -> s);
138     } else {
139         return -1;
140     }
141 }
142
143 node *clone_node(node *n) {
144     if (n == NULL) return NULL;
145
146     node *n_cp = create_node();
147     n_cp -> label = clone_primitive(n -> label);
148     n_cp -> label_typ = n -> label_typ;
149     n_cp -> data = clone_primitive(n -> data);
150     n_cp -> data_typ = n -> data_typ;
151     n_cp -> has_val = n -> has_val;
152     return n_cp;

```

```

152 }
153
154 void set_node_label_int(node *n, int i) {
155     if (n -> label != NULL) {
156         free(n -> label);
157     }
158     n -> label = create_prim_int(i);
159     n -> label_typ = INTTYPE;
160 }
161
162 void set_node_label_bool(node *n, int i) {
163     if (n -> label != NULL) {
164         free(n -> label);
165     }
166     n -> label = create_prim_int(i);
167     n -> label_typ = BOOLTYPE;
168 }
169
170 void set_node_label_str(node *n, char *s) {
171     if (n -> label != NULL) {
172         free(n -> label);
173     }
174     n -> label = create_prim_str(s);
175     n -> label_typ = STRTYPE;
176 }
177
178 void set_node_data_int(node *n, int i, int has_val) {
179     if (n -> data != NULL) {
180         free(n -> data);
181     }
182     n -> data = create_prim_int(i);
183     n -> data_typ = INTTYPE;
184     n -> has_val = has_val; //flag
185 }
186
187 void set_node_data_bool(node *n, int i, int has_val) {
188     if (n -> data != NULL) {
189         free(n -> data);
190     }
191     n -> data = create_prim_int(i);
192     n -> data_typ = BOOLTYPE;
193     n -> has_val = has_val; //flag
194 }
195
196 void set_node_data_str(node *n, char *s, int has_val) {
197     if (n -> data != NULL) {
198         free(n -> data);
199     }
200     n -> data = create_prim_str(s);
201     n -> data_typ = STRTYPE;
202     n -> has_val = has_val;
203 }
204
205 void *get_node_label(node *n) {
206     int typ = n -> label_typ;
207     void *label = NULL;
208
209     if (typ == INTTYPE || typ == BOOLTYPE) {
210         label = (void *) n -> label -> i;
211     } else if (typ == STRTYPE) {
212         label = (void *) n -> label -> s;

```

```

213     } else if (typ == VOIDTYPE) {
214         label = (void *) n -> label -> v;
215     }
216     return label;
217 }
218
219 void *get_node_data(node *n) {
220     int typ = n -> data_typ;
221     void *data = NULL;
222
223     if (typ == INTTYPE || typ == BOOLTYPE) {
224         data = (void *) n -> data -> i;
225     } else if (typ == STRTYPE) {
226         data = (void *) n -> data -> s;
227     } else if (typ == VOIDTYPE) {
228         data = (void *) n -> data -> v;
229     }
230     return data; // not guaranteed to return valid value if not has_val
231 }
232
233 /* EDGES */
234
235 int cmp_edge_weight(edge *e1, edge *e2) {
236     int lt = e1 -> w_typ;
237     if (lt == INTTYPE || lt == BOOLTYPE) {
238         return *(e1 -> w -> i) == *(e2 -> w -> i);
239     } else if (lt == STRTYPE) {
240         return strcmp(e1 -> w -> s, e2 -> w -> s);
241     } else {
242         return -1;
243     }
244 }
245
246 void set_edge_w_int(edge *e, int i, int has_val) {
247     if (e -> w != NULL) {
248         free(e -> w);
249     }
250     e -> w = create_prim_int(i);
251     e -> has_val = has_val;
252     e -> w_typ = INTTYPE;
253 }
254
255 void set_edge_w_bool(edge *e, int i, int has_val) {
256     if (e -> w != NULL) {
257         free(e -> w);
258     }
259     e -> w = create_prim_int(i);
260     e -> has_val = has_val;
261     e -> w_typ = BOOLTYPE;
262 }
263
264 void set_edge_w_str(edge *e, char *s, int has_val) {
265     if (e -> w != NULL) {
266         free(e -> w);
267     }
268     e -> w = create_prim_str(s);
269     e -> has_val = has_val;
270     e -> w_typ = STRTYPE;
271 }
272
273 node *get_edge_src(edge *e) {

```

```

274     if (e == NULL) return NULL;
275
276     return e -> src;
277 }
278
279 node *get_edge_dst(edge *e) {
280     if (e == NULL) return NULL;
281
282     return e -> dst;
283 }
284
285 int get_edge_w_int(edge *e) {
286     if (e == NULL || e -> has_val == 0) return 0;
287     return *(e -> w -> i);
288 }
289
290 char *get_edge_w_str(edge *e) {
291     if (e == NULL || e -> has_val == 0) return "";
292     return e -> w -> s;
293 }
294
295 void *create_edge() {
296     edge *e = (edge *) malloc(sizeof(edge));
297     e -> src = NULL;
298     e -> dst = NULL;
299     e -> w = NULL;
300     e -> next = NULL;
301     e -> has_val = 0;
302
303     return e;
304 }
305
306 edge *clone_edge(edge *e) {
307     if (e == NULL) return NULL;
308     edge *e_cp = create_edge();
309     e_cp -> src = clone_node(e -> src);
310     e_cp -> dst = clone_node(e -> dst);
311     e_cp -> w = clone_primitive(e -> w);
312     e_cp -> w_typ = e -> w_typ;
313     e_cp -> has_val = e -> has_val;
314     e_cp -> next = NULL;
315
316     return e_cp;
317 }
318
319 /* GRAPHS */
320
321 void *create_node_list() {
322     node_list *nl = (node_list *) malloc(sizeof(node_list));
323     nl -> hd = NULL;
324
325     return (void *) nl;
326 }
327
328 void *create_edge_list() {
329     edge_list *el = (edge_list *) malloc(sizeof(edge_list));
330     el -> hd = NULL;
331
332     return (void *) el;
333 }
334
335 void *create_graph() {
336     graph *g = (graph *) malloc(sizeof(graph));
337     g -> node_list = create_node_list();
338     g -> edge_list = create_edge_list();

```

```

335     return (void *) g;
336 }
337 /*
338  * Given a graph, creates a linked list of copies of its nodes.
339  * Used to enable node iteration (for_node) without side effects.
340  */
341 node *graph_to_node_iterable(graph *g) {
342     node *curr_orig = g -> node_list -> hd;
343     node *curr_new = clone_node(curr_orig);
344     node *hd_new = curr_new;
345
346     while (curr_orig != NULL) {
347         curr_new -> next = clone_node(curr_orig -> next);
348         curr_orig = curr_orig -> next;
349         curr_new = curr_new -> next;
350     }
351
352     return hd_new;
353 }
354 */
355 /*
356  * Given a graph, creates a linked list of copies of its edges.
357  * Used to enable edge iteration (for_edge) without side effects.
358  */
359 edge *graph_to_edge_iterable(graph *g) {
360     edge *curr_orig = g -> edge_list -> hd;
361     edge *curr_new = clone_edge(curr_orig);
362     edge *hd_new = curr_new;
363
364     while (curr_orig != NULL) {
365         curr_new -> next = clone_edge(curr_orig -> next);
366         curr_orig = curr_orig -> next;
367         curr_new = curr_new -> next;
368     }
369
370     return hd_new;
371 }
372 */
373
374 node *get_graph_next_node(node *n) {
375     return n -> next;
376 }
377
378 edge *get_graph_next_edge(edge *e) {
379     return e -> next;
380 }
381
382 node *get_node_by_label_int(graph *g, int label) {
383     node *curr = g -> node_list -> hd;
384     while (curr != NULL) {
385         if ((curr -> label_typ == INTTYPE || curr -> label_typ == BOOLTYPE) && *(curr ->
386             label -> i) == label) {
387             return curr;
388         }
389         curr = curr -> next;
390     }
391     return curr;
392 }
393
394 node *get_node_by_label_int_opt(graph *g, int label) {
395     // only used in graph.get_node()

```

```

395     node *n = get_node_by_label_int(g, label);
396     if (n == NULL) {
397         n = create_node();
398         set_node_label_int(n, 0);
399     }
400     return n;
401 }
402
403 node *get_node_by_label_bool_opt(graph *g, int label) {
404     // only used in graph.get_node()
405     node *n = get_node_by_label_int(g, label);
406     if (n == NULL) {
407         n = create_node();
408         set_node_label_bool(n, 0);
409     }
410     return n;
411 }
412
413 node *get_node_by_label_str(graph *g, char *label) {
414     node *curr = g -> node_list -> hd;
415     while (curr != NULL) {
416         if (curr -> label_typ == STRTYPE && strcmp((char *) get_node_label(curr), label)
417             == 0) {
418             return curr;
419         }
420         curr = curr -> next;
421     }
422     return curr;
423 }
424
425 node *get_node_by_label_str_opt(graph *g, char *label) {
426     // only used in graph.get_node()
427     node *n = get_node_by_label_str(g, label);
428     if (n == NULL) {
429         n = create_node();
430         set_node_label_str(n, "");
431     }
432     return n;
433 }
434
435 edge *get_edge_by_src_and_dst_int(graph *g, int src_label, int dst_label) {
436     edge *curr = g -> edge_list -> hd;
437     while (curr != NULL) {
438         if ((*get_edge_src(curr) -> label -> i) == src_label &&
439             (*get_edge_dst(curr) -> label -> i) == dst_label) {
440             return curr;
441         }
442         curr = curr -> next;
443     }
444     return NULL;
445 }
446
447 edge *get_edge_by_src_and_dst_str(graph *g, char *src_label, char *dst_label) {
448     edge *curr = g -> edge_list -> hd;
449     while (curr != NULL) {
450         if (strcmp((char *) (get_edge_src(curr) -> label -> s), src_label) == 0 &&
451             strcmp((char *) (get_edge_dst(curr) -> label -> s), dst_label) == 0) {
452             return curr;
453         }
454         curr = curr -> next;
455     }

```

```

455     return NULL;
456 }
457
458 int add_neighbor(node *n, edge *e) {
459     if (n != e -> src) return -1;
460
461     if (n -> neighbor_list -> hd == NULL) {}
462
463     if (n -> neighbor_list -> hd == NULL) {
464         n -> neighbor_list -> hd = create_neighbor_list_item(e);
465     } else if (n -> neighbor_list -> hd -> edge == e) {
466         return -1;
467     } else {
468         neighbor_list_item *curr = n -> neighbor_list -> hd;
469         while (curr -> next != NULL) {
470             if (curr -> next -> edge == e) return -1;
471             curr = curr -> next;
472         }
473         curr -> next = create_neighbor_list_item(e);
474     }
475     return 0;
476 }
477
478 int add_edge_to_edge_list(edge *e, edge_list *el) {
479     if (el -> hd == NULL) {
480         el -> hd = e;
481     } else if (el -> hd == e) {
482         return -1;
483     } else {
484         edge *curr = el -> hd;
485         while (curr -> next != NULL) {
486             if (curr -> next == e) return -1;
487             curr = curr -> next;
488         }
489         curr -> next = e;
490     }
491     return 0;
492 }
493
494 void *add_edge_int(graph *g, edge *e, int src, int dst) {
495     e -> src = get_node_by_label_int(g, src);
496     e -> dst = get_node_by_label_int(g, dst);
497     e -> next = NULL;
498
499     // add to neighbors
500     if (e -> src == NULL || e -> dst == NULL ||
501         add_neighbor(e -> src, e) < 0 ||
502         add_edge_to_edge_list(e, g -> edge_list) < 0) return NULL;
503
504     return e;
505 }
506
507 void *add_edge_bool(graph *g, edge *e, int src, int dst) {
508     return add_edge_int(g, e, src, dst);
509 }
510
511 void *add_edge_str(graph *g, edge *e, char *src, char *dst) {
512     e -> src = get_node_by_label_str(g, src);
513     e -> dst = get_node_by_label_str(g, dst);
514     e -> next = NULL;
515

```

```

516 // add to neighbors
517 if (e -> src == NULL || e -> dst == NULL ||
518     add_neighbor(e -> src, e) < 0 ||
519     add_edge_to_edge_list(e, g -> edge_list) < 0) return NULL;
520
521 return e;
522 }
523
524 int add_node(graph *g, node *n) {
525 if (g -> node_list -> hd == NULL) {
526     g -> node_list -> hd = n;
527 } else if (cmp_node_label(g -> node_list -> hd, n) == 0) {
528     return -1;
529 } else {
530     node *curr = g -> node_list -> hd;
531     while (curr -> next != NULL) {
532         if (cmp_node_label(curr -> next, n) == 0) return -1;
533         curr = curr -> next;
534     }
535     curr -> next = n;
536 }
537 return 0;
538 }
539
540 int graph_set_node(graph *g, node *n) {
541 // try adding node; handle if node w/ name already exists in the graph
542 if (add_node(g, n) < 0 && n -> has_val) {
543     int lt = n -> label_typ;
544     int dt = n -> data_typ;
545     node *n_in_g;
546
547     // find the node in the graph
548     if (lt == INTTYPE || lt == BOOLTYPE) {
549         n_in_g = get_node_by_label_int(g, *(n -> label -> i));
550     } else {
551         n_in_g = get_node_by_label_str(g, n -> label -> s);
552     }
553
554     // set its data to the data of n
555     if (dt == INTTYPE || dt == BOOLTYPE) {
556         set_node_data_int(n_in_g, *(n -> data -> i), 1);
557     } else if (dt == STRTYPE) {
558         set_node_data_str(n_in_g, n -> data -> s, 1);
559     }
560 }
561
562 return 0;
563 }
564
565 int remove_edge(graph *g, edge *e) {
566 if (e == NULL) return -1;
567
568 // remove from edge list
569 edge *curr_e = g -> edge_list -> hd;
570 if (curr_e != NULL && curr_e == e) {
571     g -> edge_list -> hd = curr_e -> next;
572 } else {
573     edge *prev;
574     while (curr_e != NULL && curr_e != e) {
575         prev = curr_e;
576         curr_e = curr_e -> next;

```

```

577     }
578
579     if (curr_e == NULL) return -1;
580
581     prev -> next = curr_e -> next;
582     prev = NULL;
583 }
584
585 // remove from neighbors
586 neighbor_list_item *curr_nl = e -> src -> neighbor_list -> hd;
587 if (curr_nl != NULL && curr_nl -> edge == e) {
588     e -> src -> neighbor_list -> hd = curr_nl -> next;
589 } else {
590     neighbor_list_item *prev;
591     while (curr_nl != NULL && curr_nl -> edge != e) {
592         prev = curr_nl;
593         curr_nl = curr_nl -> next;
594     }
595
596     if (curr_nl == NULL) return -1;
597
598     prev -> next = curr_nl -> next;
599     free(curr_nl);
600     prev = NULL;
601 }
602
603 // free
604 free(e);
605 e = NULL;
606
607 return 0;
608 }
609
610 int graph_set_edge_int(graph *g, int src_label, int dst_label) {
611     edge *e = get_edge_by_src_and_dst_int(g, src_label, dst_label);
612     if (e != NULL) {
613         set_edge_w_int(e, 0, 0);
614         return 0;
615     }
616
617     edge *new_e = create_edge();
618     set_edge_w_int(new_e, 0, 0);
619     add_edge_int(g, new_e, src_label, dst_label);
620
621     return 0;
622 }
623
624 int graph_set_edge_str(graph *g, char *src_label, char *dst_label) {
625     edge *e = get_edge_by_src_and_dst_str(g, src_label, dst_label);
626     if (e != NULL) {
627         set_edge_w_str(e, "", 0);
628         return 0;
629     }
630
631     edge *new_e = create_edge();
632     set_edge_w_str(new_e, "", 0);
633     add_edge_str(g, new_e, src_label, dst_label);
634
635     return 0;
636 }
637

```

```

638 int graph_set_edge_int_int(graph *g, int src_label, int dst_label, int w) {
639     edge *e = get_edge_by_src_and_dst_int(g, src_label, dst_label);
640     if (e != NULL) {
641         set_edge_w_int(e, w, 1);
642         return 0;
643     }
644
645     edge *new_e = create_edge();
646     set_edge_w_int(new_e, w, 1);
647     add_edge_int(g, new_e, src_label, dst_label);
648
649     return 0;
650 }
651
652 int graph_set_edge_str_str(graph *g, char *src_label, char *dst_label, char *w) {
653     edge *e = get_edge_by_src_and_dst_str(g, src_label, dst_label);
654     if (e != NULL) {
655         set_edge_w_str(e, w, 1);
656         return 0;
657     }
658
659     edge *new_e = create_edge();
660     set_edge_w_str(new_e, w, 1);
661     add_edge_str(g, new_e, src_label, dst_label);
662
663     return 0;
664 }
665
666 int graph_set_edge_str_int(graph *g, char *src_label, char *dst_label, int w) {
667     edge *e = get_edge_by_src_and_dst_str(g, src_label, dst_label);
668     if (e != NULL) {
669         set_edge_w_int(e, w, 1);
670         return 0;
671     }
672
673     edge *new_e = create_edge();
674     set_edge_w_int(new_e, w, 1);
675     add_edge_str(g, new_e, src_label, dst_label);
676
677     return 0;
678 }
679
680 int graph_set_edge_int_str(graph *g, int src_label, int dst_label, char *w) {
681     edge *e = get_edge_by_src_and_dst_int(g, src_label, dst_label);
682     if (e != NULL) {
683         set_edge_w_str(e, w, 1);
684         return 0;
685     }
686
687     edge *new_e = create_edge();
688     set_edge_w_str(new_e, w, 1);
689     add_edge_int(g, new_e, src_label, dst_label);
690
691     return 0;
692 }
693
694 int remove_all_edges(graph *g, node *n) {
695     edge *curr_edge = g -> edge_list -> hd;
696     edge *temp;
697     while (curr_edge != NULL) {
698         temp = curr_edge->next;

```

```

699     if (n -> label_typ == INTTYPE || n -> label_typ == BOOLTYPE) {
700         if (*(int *) get_node_label(curr_edge -> src) == *(int *) get_node_label(n) ||
701             *(int *) get_node_label(curr_edge -> dst) == *(int *) get_node_label(n)) {
702             remove_edge(g, curr_edge);
703         }
704     if (n -> label_typ == STRTYPE) {
705         if ((char *) get_node_label(curr_edge -> src) == (char *) get_node_label(n) ||
706             (char *) get_node_label(curr_edge -> dst) == (char *) get_node_label(n)) {
707             remove_edge(g, curr_edge);
708         }
709         curr_edge = temp;
710     }
711     return 0;
712 }
713
714 int remove_node_int(graph *g, int label){
715     node *curr = g -> node_list -> hd;
716     if (*(int *) get_node_label(curr) == label) {
717         node *n = get_node_by_label_int(g, label);
718         remove_all_edges(g, n);
719         g -> node_list -> hd = curr -> next;
720         free(curr);
721         return 0;
722     }
723     node *prev = curr;
724     curr = curr -> next;
725     while (curr != NULL) {
726         if (*(int *) get_node_label(curr) == label) {
727             node *n = get_node_by_label_int(g, label);
728             remove_all_edges(g, n);
729             prev -> next = curr -> next;
730             free(curr);
731             return 0;
732         }
733         prev = curr;
734         curr = curr -> next;
735     }
736     return -1;
737 }
738
739 int remove_node_str(graph *g, char *label){
740     node *curr = g -> node_list -> hd;
741     if (strcmp((char *) get_node_label(curr), label) == 0) {
742         node *n = get_node_by_label_str(g, label);
743         remove_all_edges(g, n);
744         g -> node_list -> hd = curr -> next;
745         free(curr);
746         return 0;
747     }
748     node *prev = curr;
749     curr = curr -> next;
750     while (curr != NULL) {
751         if (strcmp((char *) get_node_label(curr), label) == 0) {
752             node *n = get_node_by_label_str(g, label);
753             remove_all_edges(g, n);
754             prev -> next = curr -> next;
755             free(curr);
756             return 0;
757         }

```

```

758     prev = curr;
759     curr = curr -> next;
760 }
761 return -1;
762 }
763
764 int graph_has_node_int(graph *g, int name) {
765     if (get_node_by_label_int(g, name)) {
766         return 0;
767     }
768     return -1;
769 }
770
771 int graph_has_node_str(graph *g, char *name) {
772     if (get_node_by_label_str(g, name)) {
773         return 0;
774     }
775     return -1;
776 }
777
778 int are_neighbors_int(graph *g, int from_name, int to_name) {
779     node *src = get_node_by_label_int(g, from_name);
780     if (src == NULL || src -> neighbor_list -> hd == NULL) return 0;
781
782     neighbor_list_item *nli = src -> neighbor_list -> hd;
783     while (nli != NULL) {
784         node *dst = nli -> edge -> dst;
785         if (*(dst -> label -> i) == to_name) return 1;
786         nli = nli -> next;
787     }
788     return 0;
789 }
790
791 int are_neighbors_str(graph *g, char *from_name, char *to_name) {
792     node *src = get_node_by_label_str(g, from_name);
793     if (src == NULL || src -> neighbor_list -> hd == NULL) return 0;
794
795     neighbor_list_item *nli = src -> neighbor_list -> hd;
796     while (nli != NULL) {
797         node *dst = nli -> edge -> dst;
798         if (strcmp(dst -> label -> s, to_name) == 0) return 1;
799         nli = nli -> next;
800     }
801     return 0;
802 }
803
804 int is_empty(graph *g) {
805     if (g -> node_list -> hd) {
806         return 1; // true
807     }
808     return 0; // false
809 }
810
811 /* GRAPH TRAVERSAL */
812
813 queue *create_queue() {
814     queue *Q = (queue *) malloc(sizeof(queue));
815     Q -> hd = NULL;
816     Q -> tl = NULL;
817     return Q;
818 }

```

```

819 q_item *create_q_item(node *n) {
820     q_item *i = (q_item *) malloc(sizeof(q_item));
821     i -> n = n;
822     i -> next = NULL;
823     return i;
824 }
825 }
826
827 void enqueue(queue *Q, node *n) {
828     if (Q -> tl == NULL) {
829         Q -> hd = create_q_item(n);
830         Q -> tl = Q -> hd;
831     } else {
832         Q -> tl -> next = create_q_item(n);
833         Q -> tl = Q -> tl -> next;
834     }
835 }
836
837 node *dequeue(queue *Q) {
838     if (Q -> hd == NULL) {
839         return NULL;
840     } else {
841         node *n = Q -> hd -> n;
842         q_item *tmp = Q -> hd;
843         Q -> hd = Q -> hd -> next;
844         free(tmp);
845         if (Q -> hd == NULL) Q -> tl = NULL;
846         return n;
847     }
848 }
849
850 void push(queue *Q, node *n) {
851     if (Q -> hd == NULL) {
852         Q -> hd = create_q_item(n);
853     } else {
854         q_item *curr = Q -> hd;
855         Q -> hd = create_q_item(n);
856         Q -> hd -> next = curr;
857     }
858 }
859
860 node *pop(queue *Q) {
861     if (Q -> hd == NULL) {
862         return NULL;
863     } else {
864         q_item *fst = Q -> hd;
865         Q -> hd = fst -> next;
866         node *n = fst -> n;
867         free(fst);
868         return n;
869     }
870 }
871
872 int is_empty_q(queue *Q) {
873     return (Q -> hd == NULL);
874 }
875
876 void add_neighbors_of_node_to_graph(graph *g_new, node *n_root, node *n_orig, int
877     level) {
878     if (level == 0) return;

```

```

879     queue *Q = create_queue();
880
881     neighbor_list_item *nli = n_orig -> neighbor_list -> hd;
882     while (nli != NULL) {
883         node *neighbor = nli -> edge -> dst;
884
885         // Don't include neighbor if it is the root node
886         if (neighbor == n_root) {
887             nli = nli -> next;
888             continue;
889         }
890
891         // Try to find node with same label as neighbor in g_new
892         node *neighbor_copy;
893         if (neighbor -> label_typ == INTTYPE || neighbor -> label_typ == BOOLTYPE) {
894             neighbor_copy = get_node_by_label_int(g_new, *(neighbor -> label -> i));
895         } else if (neighbor -> label_typ == STRTYPE) {
896             neighbor_copy = get_node_by_label_str(g_new, neighbor -> label -> s);
897         }
898
899         edge *e = create_edge();
900         e -> w = clone_primitive(nli -> edge -> w);
901         e -> w_typ = nli -> edge -> w_typ;
902         e -> has_val = nli -> edge -> has_val;
903         if (n_orig == neighbor && neighbor_copy != NULL) {
904             // If edge is self-directed, add edge to graph but nothing else
905             if (neighbor -> label_typ == INTTYPE || neighbor -> label_typ == BOOLTYPE) {
906                 add_edge_int(g_new, e, *(neighbor_copy -> label -> i), *(neighbor_copy ->
907                     label -> i));
908             } else if (neighbor -> label_typ == STRTYPE) {
909                 add_edge_str(g_new, e, neighbor_copy -> label -> s, neighbor_copy -> label
910                     -> s);
911             }
912         } else {
913             // If node doesn't yet exist in g_new, create one and add original neighbor to
914             // processing queue
915             if (neighbor_copy == NULL) {
916                 neighbor_copy = clone_node(neighbor);
917                 add_node(g_new, neighbor_copy);
918                 enqueue(Q, neighbor);
919             }
920             if (neighbor -> label_typ == INTTYPE || neighbor -> label_typ == BOOLTYPE) {
921                 add_edge_int(g_new, e, *(n_orig -> label -> i), *(neighbor_copy -> label ->
922                     i));
923             } else if (neighbor -> label_typ == STRTYPE) {
924                 add_edge_str(g_new, e, n_orig -> label -> s, neighbor_copy -> label -> s);
925             }
926             nli = nli -> next;
927         }
928     while (Q -> tl != NULL) {
929         add_neighbors_of_node_to_graph(g_new, n_root, dequeue(Q), level - 1);
930     }
931     free(Q);
932 }
933
934 graph *neighbors_one_arg(node *n) {

```

```

936     graph *g_new = create_graph();
937     if (n == NULL) return g_new;
938     add_neighbors_of_node_to_graph(g_new, n, n, 1);
939
940     return g_new;
941 }
942
943 graph *neighbors(node *n, int level, int include_current) {
944     graph *g_new = create_graph();
945     if (level <= 1) level = 1;
946
947     if (n == NULL) return g_new;
948
949     if (include_current != 0) {
950         add_node(g_new, clone_node(n));
951         add_neighbors_of_node_to_graph(g_new, NULL, n, level);
952     } else {
953         add_neighbors_of_node_to_graph(g_new, n, n, level);
954     }
955
956     return g_new;
957 }
958
959 graph *find_data_int(graph *g, int data) {
960     node *n = g -> node_list -> hd;
961     graph *g_new = create_graph();
962     while (n != NULL) {
963         if (n -> has_val == 1 && (n -> data_typ == INTTYPE || n -> data_typ == BOOLTYPE)
964             && *(n -> data -> i) == data) {
965             node *n_cp = clone_node(n);
966             add_node(g_new, n_cp);
967         }
968         n = n -> next;
969     }
970     return g_new;
971 }
972
973 graph *find_data_str(graph *g, char *data) {
974     node *n = g -> node_list -> hd;
975     graph *g_new = create_graph();
976     while (n != NULL) {
977         if (n -> has_val == 1 && n -> data_typ == STRTYPE && strcmp(n -> data -> s, data
978             ) == 0) {
979             node *n_cp = clone_node(n);
980             add_node(g_new, n_cp);
981         }
982         n = n -> next;
983     }
984     return g_new;
985 }
986 /* PRINTING */
987
988 void print_node(node *n) {
989     if (n -> label_typ == INTTYPE) {
990         printf("%d:", *(n -> label -> i));
991     } else if (n -> label_typ == BOOLTYPE) {
992         if (*(n -> label -> i) == 0) printf("false:");
993         else printf("true:");
994     } else if (n -> label_typ == STRTYPE) {

```

```

995     printf("\"%s\":\"", n -> label -> s);
996 }
997
998 if (n -> has_val == 0) {
999     printf("null");
1000 } else if (n -> data_typ == INTTYPE) {
1001     printf("%d", *(n -> data -> i));
1002 } else if (n -> data_typ == BOOLTYPE) {
1003     if (*(n -> data -> i) == 0) printf("false");
1004     else printf("true");
1005 } else if (n -> data_typ == STRTYPE) {
1006     printf("\"%s\"", n -> data -> s);
1007 }
1008
1009 return;
1010
1011 }
1012
1013 int search_node_list(node_list *nl, node *n) {
1014     node *curr = nl -> hd;
1015     while (curr != NULL) {
1016         if ((curr -> label) == (n -> label)) {return 1;}
1017         curr = curr -> next;
1018     }
1019     return 0;
1020 }
1021
1022 void add_node_to_list(node_list *nl, node *n) {
1023     node *curr = nl -> hd;
1024     n -> next = curr;
1025     nl -> hd = n;
1026 }
1027
1028
1029 void print_edge_weight(edge *e) {
1030     if (e -> has_val == 0) {
1031         printf("(null)");
1032     } else if (e -> w_typ == INTTYPE) {
1033         printf("(%d)", *(int *) e -> w -> i );
1034     } else if (e -> w_typ == BOOLTYPE) {
1035         if (*(int *) e -> w -> i == 1) printf("(true)");
1036         else printf("(false)");
1037     } else if (e -> w_typ == STRTYPE) {
1038         printf("(%s)", (char *)e -> w -> s );
1039     }
1040     return;
1041 }
1042
1043 void print_graph(graph *g) {
1044     node *n = g -> node_list -> hd;
1045     while (n) {
1046         print_node(n);
1047         printf(" -> [");
1048         neighbor_list_item *nli = n -> neighbor_list -> hd;
1049         if (nli) {
1050             print_node(nli -> edge -> dst);
1051             printf(" ");
1052             print_edge_weight(nli -> edge);
1053             while (nli -> next) {
1054                 printf(", ");
1055                 print_node(nli -> next -> edge -> dst);

```

```

1056     printf(" ");
1057     print_edge_weight(nli -> next -> edge);
1058     nli = nli -> next;
1059 }
1060 }
1061
1062     printf("]\n");
1063     n = n -> next;
1064 }
1065
1066     return;
1067 }
1068
1069 graph *bfs_int(graph *g, int name) {
1070     queue *Q = create_queue();
1071     node_list *seen = create_node_list();
1072     graph *bfs_graph = create_graph();
1073     node *start = get_node_by_label_int(g, name);
1074
1075     enqueue(Q, start);
1076     while (is_empty_q(Q) == 0) {
1077         node *next = dequeue(Q);
1078         if (search_node_list(seen, next) == 0) {
1079             add_node_to_list(seen, next);
1080             add_node(bfs_graph, clone_node(next));
1081             neighbor_list *neighbors = next -> neighbor_list;
1082             neighbor_list_item *neighbor = neighbors -> hd;
1083             while (neighbor != NULL) {
1084                 enqueue(Q, neighbor -> edge -> dst);
1085                 neighbor = neighbor -> next;
1086             }
1087         }
1088     }
1089     return bfs_graph;
1090 }
1091
1092 graph *bfs_str(graph *g, char *name) {
1093     queue *Q = create_queue();
1094     node_list *seen = create_node_list();
1095     graph *bfs_graph = create_graph();
1096     node *start = get_node_by_label_str(g, name);
1097
1098     enqueue(Q, start);
1099     while (is_empty_q(Q) == 0) {
1100         node *next = dequeue(Q);
1101         if (search_node_list(seen, next) == 0) {
1102             add_node_to_list(seen, next);
1103             add_node(bfs_graph, clone_node(next));
1104             neighbor_list *neighbors = next -> neighbor_list;
1105             neighbor_list_item *neighbor = neighbors -> hd;
1106             while (neighbor != NULL) {
1107                 enqueue(Q, neighbor -> edge -> dst);
1108                 neighbor = neighbor -> next;
1109             }
1110         }
1111     }
1112     return bfs_graph;
1113 }
1114
1115 graph *dfs_int(graph *g, int name) {
1116     queue *Q = create_queue();

```

```

1117 node_list *seen = create_node_list();
1118 graph *dfs_graph = create_graph();
1119 node *start = get_node_by_label_int(g, name);
1120
1121 push(Q, start);
1122 while (is_empty_q(Q) == 0) {
1123     node *next = pop(Q);
1124     if (search_node_list(seen, next) == 0) {
1125         add_node_to_list(seen, next);
1126         add_node(dfs_graph, clone_node(next));
1127         neighbor_list *neighbors = next -> neighbor_list;
1128         neighbor_list_item *neighbor = neighbors -> hd;
1129         while (neighbor != NULL) {
1130             push(Q, neighbor -> edge -> dst);
1131             neighbor = neighbor -> next;
1132         }
1133     }
1134 }
1135 return dfs_graph;
1136 }
1137
1138 graph *dfs_str(graph *g, char *name) {
1139     queue *Q = create_queue();
1140     node_list *seen = create_node_list();
1141     graph *dfs_graph = create_graph();
1142     node *start = get_node_by_label_str(g, name);
1143
1144 push(Q, start);
1145 while (is_empty_q(Q) == 0) {
1146     node *next = pop(Q);
1147     if (search_node_list(seen, next) == 0) {
1148         add_node_to_list(seen, next);
1149         add_node(dfs_graph, clone_node(next));
1150         neighbor_list *neighbors = next -> neighbor_list;
1151         neighbor_list_item *neighbor = neighbors -> hd;
1152         while (neighbor != NULL) {
1153             push(Q, neighbor -> edge -> dst);
1154             neighbor = neighbor -> next;
1155         }
1156     }
1157 }
1158 return dfs_graph;
1159 }
```

9.8 Example Code

9.8.1 Bellman-Ford Algorithm

```

1 int main() {
2
3     bool valid = true; (*negative edge cycle check*)
4
5     (*Initial graph*)
6     graph<string:int, int> g = [{"S":500 -(10)> "A":500 -(2)> "C":500 -(2)> "B":500
7         -(1)> "A"; "S" -(8)>"E":500 -(1)> "D":500 -(1)>"C"; "D" -(4)> "A"] ;
8
9     (*Shortest path graph*)
10    graph<string:int, int> shortest_path = [];
11
12    (*Initialize distances to infinity*)
13    for_node(n : g) {
```

```

13         node<string:int> n1 = n.get_name():999999;
14         shortest_path.set_node(n1);
15     }
16
17     (*copy in weights*)
18     for_edge(src, dst, w: g) {
19         shortest_path.set_edge(src.get_name(); dst.get_name(); w);
20     }
21
22     (*Initialize start node to 0*)
23     node<string:int> source_n = "S":0;
24     shortest_path.set_node(source_n);
25
26     print("ORIGINAL GRAPH:");
27     g.print();
28
29     (*Relax edges n times*)
30     for_node(n : shortest_path) {
31         for_edge(src, dst, w : shortest_path) {
32             int src_data = src.get_data();
33             int dst_data = dst.get_data();
34
35             if (src_data + w < dst_data) {
36                 int new_dst_data = src_data + w;
37                 node<string:int> new_dst = dst.get_name():new_dst_data;
38                 shortest_path.set_node(new_dst);
39             }
40         }
41     }
42
43     (*Negative edge weight cycle check*)
44     for_edge(src, dst, w : shortest_path) {
45         int src_data = src.get_data();
46         int dst_data = dst.get_data();
47
48         if (src_data + w < dst_data) {
49             print("negative edge weight cycle");
50             valid = false;
51         }
52         else if (src_data + w > dst_data) {
53             shortest_path.remove_edge(src.get_name(); dst.get_name());
54         }
55     }
56
57     if (valid) {
58         print("SHORTEST PATH:");
59         shortest_path.print();
60     }
61
62     return 0;
63 }

```

9.8.2 Family Tree

```

1 int main() {
2     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"];
3
4     family.print();

```

```

5
6     print("");
7     print("Simba got married!");
8     node<string:string> nala = "Nala":"Daughter in law";
9     family.set_node(nala);
10    family.set_edge("Simba"; "Nala"; "Spouses");
11    family.set_edge("Nala"; "Simba"; "Spouses");
12    family.print();
13 }

```

9.9 Regression Test Suite - Positive Tests

9.9.1 test-anon-func.hpg

```

1 int main() {
2     fun<int:int,int> f = int (int x; int y) (x + y);
3     fun<int:bool> g = int (bool q) (0);
4     print_int( f(3; 4) );
5     return 0;
6 }

```

Expected output:

7

9.9.2 test-are-neighbors1.hpg

```

1 int main() {
2     graph<int, int> g = [1 <(5)> 2 -()> 3];
3     print_bool(g.are_neighbors(1; 2));
4     print_bool(g.are_neighbors(2; 1));
5     print_bool(g.are_neighbors(2; 3));
6     print_bool(g.are_neighbors(3; 2));
7     return 0;
8 }

```

Expected output:

1
1
1
0

9.9.3 test-are-neighbors2.hpg

```

1 int main() {
2     graph<string, int> g = ["1" <(5)> "2" -()> "3"];
3     print_bool(g.are_neighbors("1"; "2"));
4     print_bool(g.are_neighbors("2"; "1"));
5     print_bool(g.are_neighbors("2"; "3"));
6     print_bool(g.are_neighbors("3"; "2"));
7     return 0;
8 }

```

Expected output:

1
1
1
0

9.9.4 test-bellmanford.hpg

```
1 int main() {
2
3     bool valid = true; (*negative edge cycle check*)
4
5     (*Initial graph*)
6     graph<string:int, int> g = [{"S":500 -(10)> "A":500 -(2)> "C":500 -(2)> "B":500
7         -(1)> "A"; "S" -(8)>"E":500 -(1)> "D":500 -(1)>"C"; "D" -(4)> "A"}];
8
9     (*Shortest path graph*)
10    graph<string:int, int> shortest_path = [];
11
12    (*Initialize distances to infinity*)
13    for_node(n : g) {
14        node<string:int> n1 = n.get_name():999999;
15        shortest_path.set_node(n1);
16    }
17
18    (*copy in weights*)
19    for_edge(src, dst, w: g) {
20        shortest_path.set_edge(src.get_name(); dst.get_name(); w);
21    }
22
23    (*Initialize start node to 0*)
24    node<string:int> source_n = "S":0;
25    shortest_path.set_node(source_n);
26
27    print("ORIGINAL GRAPH:");
28    g.print();
29
30    (*Relax edges n times*)
31    for_node(n : shortest_path) {
32        for_edge(src, dst, w : shortest_path) {
33            int src_data = src.get_data();
34            int dst_data = dst.get_data();
35
36            if (src_data + w < dst_data) {
37                int new_dst_data = src_data + w;
38                node<string:int> new_dst = dst.get_name():new_dst_data;
39                shortest_path.set_node(new_dst);
40            }
41        }
42    }
43
44    (*Negative edge weight cycle check*)
45    for_edge(src, dst, w : shortest_path) {
46        int src_data = src.get_data();
47        int dst_data = dst.get_data();
48
49        if (src_data + w < dst_data) {
50            print("negative edge weight cycle");
51            valid = false;
52        }
53        else if (src_data + w > dst_data) {
54            shortest_path.remove_edge(src.get_name(); dst.get_name());
55        }
56    }
57
58    if (valid) {
59        print("SHORTEST PATH:");
60    }
61}
```

```

59         shortest_path.print();
60     }
61
62     return 0;
63 }
```

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

```

1 int main() {
2     graph<int> g = [1 -() - 2 -() - 3; 1 -() - 4];
3     graph<int> g_sub1 = g.bfs(1);
4     g_sub1.print();
5 }
```

Expected output:

```

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

9.9.6 test-binop-ops.hpg

```

1 int main() {
2     print_bool(4 < 5);
3     print_bool(5 < 5);
4     print_bool(6 < 5);
5
6     print_bool(4 <= 5);
7     print_bool(5 <= 5);
8     print_bool(6 <= 5);
9
10    print_bool(4 == 5);
11    print_bool(5 == 5);
12    print_bool(6 == 5);
13
14    print_bool(4 >= 5);
15    print_bool(5 >= 5);
16    print_bool(6 >= 5);
17}
```

```
18     print_bool(4 > 5);
19     print_bool(5 > 5);
20     print_bool(6 > 5);
21
22     return 0;
23 }
```

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

```
1 int main() {
2     print_bool("foo" == "foo");
3     print_bool("foo" == "bar");
4     print_bool("foo" > "bar");
5     print_bool("foo" < "bar");
6     return 0;
7 }
```

Expected output:

```
1
0
1
0
```

9.9.8 test-binop-prec.hpg

```
1 int main() {
2     print_bool(5 == 4 + 1);
3     print_bool(4 + 1 == 5);
4     print_bool(2 * 4 + 1 == 9);
5     print_bool(1 + 2 * 4 == 9);
6     print_bool(2 * (4 + 1) == 10);
7     print_bool((2 * (4 + 1) + 2) * 3 == 18 * 2);
8 }
```

Expected output:

```
1
1
1
```

```
1  
1  
1
```

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

```
1 int main() {  
2     graph<bool:bool, bool> g0 = [true: false -(true)> false: true];  
3  
4     (* graph<bool:bool, bool> g0 = [true: false <(true)> false: true];  
5  
6     graph<int:bool, bool> g1 = [];  
7     graph<int:bool, bool> g2 = [1];  
8     graph<int:bool, bool> g3 = [1:NULL];  
9     graph<int:bool, bool> g4 = [1:true];  
10    graph<int:bool, bool> g5 = [1:true <()> 2:NULL];  
11    graph<int:bool, bool> g6 = [1:NULL <()> 2:true];  
12    graph<int:bool, bool> g7 = [1:true <(NULL)> 2:NULL];  
13    graph<int:bool, bool> g8 = [1:false <()> 2:true];  
14    graph<int:bool, bool> g9 = [1 <()> 2:true];  
15    graph<int:bool, bool> g10 = [1:true <(false)> 2:NULL -(true)> 3:true; 3 <()- 4];  
16    graph<int:bool, bool> g11 = [1 <()> 2:NULL -()> 3:NULL; 3 <()- 4];*)  
17  
18    return 0;  
19 }
```

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

```
1 int main() {  
2     graph<int:int, int> g1 = [];  
3     graph<int:int, int> g2 = [1];  
4     graph<int:int, int> g3 = [1:NULL];  
5     graph<int:int, int> g4 = [1:2 <()> 3:NULL];  
6     graph<int:int, int> g5 = [1:2 <(NULL)> 3:NULL];  
7     graph<int:int, int> g6 = [1:NULL <()> 2:3];  
8     graph<int:int, int> g7 = [1 <()> 2:3];  
9     graph<int:int, int> g8 = [1:2 <(3)> 4:NULL -()> 5:42; 5 <()- 6];  
10    graph<int:int, int> g9 = [1 <()> 2:NULL -()> 3:NULL; 3 <()- 4];  
11  
12    return 0;  
13 }
```

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

```
1 int main() {  
2     graph<string:string, string> g1 = [];  
3     graph<string:string, string> g2 = ["foo"];  
4     graph<string:string, string> g3 = ["foo":NULL];  
5     graph<string:string, string> g4 = ["foo":"bar" <()> "baz":NULL];  
6     graph<string:string, string> g5 = ["foo":"bar" <(NULL)> "baz":NULL];  
7     graph<string:string, string> g6 = ["foo":NULL <()> "bar":"baz"];  
8     graph<string:string, string> g7 = ["foo" <()> "bar":"baz"];  
9     graph<string:string, string> g8 = ["a":"b" <("c")> "d":NULL -()> "e":"f"; "g" <()-  
    "h"];  
10    graph<string:string, string> g9 = ["a" <()> "b":NULL -()> "c":NULL; "c" <()- "d"];  
11  
12    return 0;  
13 }
```

9.9.12 test-create-node.hpg

```

1 int main() {
2     node<string:string> n1 = "abc":"def";
3     node<string:int> n2 = "abc":42;
4     node<int:string> n3 = 42:"abc";
5     node<int:int> n4 = 42:1;
6     node<string:bool> n5 = "abc":NULL;
7     node<int:bool> n6 = 42:NULL;
8     node<string> n7 = "abc";
9     node<int> n8 = 42;
10    return 0;
11 }
```

9.9.13 test-dfs-path.hpg

```

1 int main() {
2     graph<int> g = [1 -() - 2 -() - 3 -() - 4];
3     graph<int> g_sub1 = g.dfs(1);
4     g_sub1.print();
5 }
```

Expected output:

```

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

9.9.14 test-fdecls-argnum.hpg

```

1 int main() {
2     print_int(41 + one());
3     print_int(addone(41));
4     print_int(add(40, 2));
5     return 0;
6 }
7
8 (* no args *)
9 int one() {
10    return 1;
11 }
12
13 (* one arg *)
14 int addone(int x) {
15    return x + 1;
16 }
17
18 (* multiple args *)
19 int add(int x, int y) {
20    return x + y;
21 }
```

Expected output:

```

42
42
42
```

9.9.15 test-fdecls-argtype.hpg

```

1 int main() {
```

```

2     void_func1();
3     void_func2();
4     int v1 = int_func();
5     string v2 = str_func();
6     print_int(v1);
7     print(v2);
8     return 0;
9 }
10
11 void void_func1() {}
12 void void_func2() { return; }
13
14 int int_func() { return 42; }
15 string str_func() { return "hello"; }

```

Expected output:

```

42
hello

```

9.9.16 test-find-data-int1.hpg

```

1 int main() {
2     graph<string:int> g = ["a":10 -()> "b":20 <()> "c":20; "a" <()- "d"];
3     graph<string:int> g_new = g.find(20);
4     for_node(n : g_new) {
5         n.print();
6         print("");
7     }
8 }

```

Expected output:

```

"b":20
"c":20

```

9.9.17 test-find-data-int2.hpg

```

1 int main() {
2     graph<string:int> g = ["a":10 -()> "b":20 <()> "c":20; "a" <()- "d"];
3     graph<string:int> g_new = g.find(40);
4     for_node(n : g_new) {
5         n.print();
6         print("");
7     }
8 }

```

9.9.18 test-find-data-str.hpg

```

1 int main() {
2     graph<int:string> g = [1:"a" -()> 2:"b" <()> 3:"b"; 1 <()- 4];
3     graph<int:string> g_new = g.find("b");
4     for_node(n : g_new) {
5         n.print();
6         print("");
7     }
8 }

```

Expected output:

```

2:"b"

```

3:"b"

9.9.19 test-for-edge1.hpg

```
1 int main() {
2     graph<int:int, int> g = [1:1-(3)>2:2];
3     for_edge(src, dst, w : g) {
4         print_int(src.get_data());
5         print_int(dst.get_data());
6         print_int(w);
7     }
8     return 0;
9 }
```

Expected output:

```
1
2
3
```

9.9.20 test-for-edge2.hpg

```
1 int main() {
2     graph<int:int, int> g = [];
3     for_edge(src, dst, w : g) {
4         print_int(src.get_data());
5         print_int(dst.get_data());
6         print_int(w);
7     }
8     return 0;
9 }
```

9.9.21 test-for-edge3.hpg

```
1 int main() {
2     graph<int:int, int> g = [1:1<(3)>2:2];
3     for_edge(src, dst, w : g) {
4         print_int(src.get_data());
5         print_int(dst.get_data());
6         print_int(w);
7     }
8     return 0;
9 }
```

Expected output:

```
1
2
3
2
1
3
```

9.9.22 test-for-edge4.hpg

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

```
5     print_int(dst.get_data());
6     print_int(w);
7 }
8 return 0;
9 }
```

Expected output:

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

9.9.23 test-for-edge5.hpg

```
1 int main() {
2     graph<int:int, int> g = [1:1->2:2];
3     for_edge(src, dst, w : g) {
4         print_int(src.get_data());
5         print_int(dst.get_data());
6         print_int(w);
7     }
8     return 0;
9 }
```

Expected output:

```
1
2
0
```

9.9.24 test-for-node1.hpg

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

Expected output:

```
1
```

9.9.25 test-for-node2.hpg

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

```
6 }
```

9.9.26 test-for-node3.hpg

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

Expected output:

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

9.9.27 test-for-node4.hpg

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

Expected output:

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

9.9.28 test-for.hpg

```
1 int main() {
2     int x;
3     for (x = 0 ; x <= 3 ; x = x + 1) {
4         print("True!");
5     }
6 }
```

Expected output:

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

9.9.29 test-get-name1.hpg

```
1 int main() {
2     node<int> n = 1;
3     print_int(n.get_name());
4     node<bool> n = true;
5     print_bool(n.get_name());
6     node<string> n = "foo";
7     print(n.get_name());
8 }
```

Expected output:

```
1
1
foo
```

9.9.30 test-get-node-data.hpg

```
1 int main() {
2     node<string:string> n1 = "abc":"def";
3     node<string:int> n2 = "abc":42;
4     node<int:string> n3 = 42:"abc";
5     node<int:int> n4 = 42:1;
6
7     string d1 = n1.get_data();
8     print(d1);
9
10    int d2 = n2.get_data();
11    print_int(d2);
12
13    string d3 = n3.get_data();
14    print(d3);
15
16    int d4 = n4.get_data();
17    print_int(d4);
18
19    return 0;
20 }
```

Expected output:

```
def
42
abc
1
```

9.9.31 test-get-node1.hpg

```
1 int main() {
2     graph<int:string> g = [1:"foo"; 2:"bar"];
3     node<int:string> n1 = g.get_node(1);
4     node<int:string> n2 = g.get_node(2);
5     node<int:string> n3 = g.get_node(3);
6
7     n1.print();
8     print("");
9     n2.print();
10    print("");
11    n3.print();
12 }
```

Expected output:

```
1:"foo"  
2:"bar"  
0:null
```

9.9.32 test-get-node2.hpg

```
1 int main() {  
2     graph<string:int> g = ["foo":1, "bar":2];  
3     node<string:int> n1 = g.get_node("foo");  
4     node<string:int> n2 = g.get_node("bar");  
5     node<string:int> n3 = g.get_node("baz");  
6  
7     n1.print();  
8     print("");  
9     n2.print();  
10    print("");  
11    n3.print();  
12 }
```

Expected output:

```
"foo":1  
"bar":2  
"":null
```

9.9.33 test-get-node3.hpg

```
1 int main() {  
2     graph<bool:string> g = [true:"foo"];  
3     node<bool:string> n1 = g.get_node(true);  
4     node<bool:string> n2 = g.get_node(false);  
5  
6     n1.print();  
7     print("");  
8     n2.print();  
9 }
```

Expected output:

```
true:"foo"  
false:null
```

9.9.34 test-get-weight1.hpg

```
1 int main() {  
2     graph<int:string, int> g = [1:"foo" -(10)> 2:"bar" -(> 3];  
3     print_int(g.get_weight(1, 2));  
4     print_int(g.get_weight(2, 1));  
5     print_int(g.get_weight(2, 3));  
6 }
```

Expected output:

```
10  
0  
0
```

9.9.35 test-get-weight2.hpg

```

1 int main() {
2     graph<int:string, string> g = [1:"foo" -(10)> 2:"bar" -()> 3];
3     print(g.get_weight(1; 2));
4     print(g.get_weight(2; 1));
5     print(g.get_weight(2; 3));
6 }
```

Expected output:

10

9.9.36 test-get-weight3.hpg

```

1 int main() {
2     graph<int:string, bool> g = [1:"foo" -(true)> 2:"bar" -()> 3];
3     print_bool(g.get_weight(1; 2));
4     print_bool(g.get_weight(2; 1));
5     print_bool(g.get_weight(2; 3));
6 }
```

Expected output:

1
0
0

9.9.37 test-graph-neighbors1.hpg

```

1 int main() {
2     graph<int, int> g = [];
3     graph<int, int> g_sub = g.neighbors(5; 2; true);
4 }
```

9.9.38 test-graph-neighbors2.hpg

```

1 int main() {
2     graph<int, int> g = [5];
3     graph<int, int> g_sub = g.neighbors(5; 2; true);
4 }
```

9.9.39 test-graph-neighbors3.hpg

```

1 int main() {
2     graph<int> g = [1 -() - 2 -() - 3; 1 -() - 4 -() - 5; 4 -() - 6 -() - 8; 6 -() - 7];
3     graph<int> g_sub1 = g.neighbors(1; 2; false);
4     for_node (n : g_sub1) {
5         n.print();
6         print("");
7     }
8     print("");
9     graph<int> g_sub2 = g.neighbors(1; 3; true);
10    for_node (n : g_sub2) {
11        n.print();
12        print("");
13    }
14 }
```

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

```
1 int main() {  
2     graph<int> g = [1 -()- 2 -()- 3 -()- 1];  
3     graph<int> g_sub1 = g.neighbors(1; 3; false);  
4     for_node (n : g_sub1) {  
5         n.print();  
6         print("");  
7     }  
8     print("");  
9     graph<int> g_sub2 = g.neighbors(1; 3; true);  
10    for_node (n : g_sub2) {  
11        n.print();  
12        print("");  
13    }  
14 }
```

Expected output:

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

9.9.41 test-graph-neighbors5.hpg

```
1 int main() {  
2     graph<int> g = [1 -()- 2 -()- 3 -()- 4 -()- 2];  
3     graph<int> g_sub1 = g.neighbors(1; 4; false);  
4     for_node (n : g_sub1) {  
5         n.print();  
6         print("");  
7     }  
8     print("");  
9     graph<int> g_sub2 = g.neighbors(1; 4; true);  
10    for_node (n : g_sub2) {  
11        n.print();  
12        print("");  
13    }  
14 }
```

Expected output:

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

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

9.9.42 test-has-node-bool.hpg

```
1 int main() {  
2     graph<bool, int> g = [true <(10)> false];  
3     int result1 = g.has_node(true);  
4     print_int(result1);  
5     g.remove_node(true);  
6     int result2 = g.has_node(true);  
7     print_int(result2);  
8 }
```

Expected output:

```
0  
-1
```

9.9.43 test-has-node-int.hpg

```
1 int main() {  
2     graph<int, int> g = [1 <(10)> 2; 3];  
3     int result1 = g.has_node(1);  
4     print_int(result1);  
5     int result2 = g.has_node(5);  
6     print_int(result2);  
7 }
```

Expected output:

```
0  
-1
```

9.9.44 test-has-node-str.hpg

```
1 int main() {  
2     graph<string, int> g = ["hello" <(10)> "there"; "!"];  
3     int result1 = g.has_node("!!");  
4     print_int(result1);  
5     int result2 = g.has_node("hello");  
6     print_int(result2);  
7 }
```

Expected output:

```
-1  
0
```

9.9.45 test-helloworld.hpg

```
1 int main() {  
2     print("Hello, world");  
3     return 0;  
4 }
```

Expected output:

```
Hello, world
```

9.9.46 test-if-else.hpg

```
1 int main() {
2     if (true) {
3         print("True!");
4     }
5
6     int foo = 5;
7
8     if (foo == 1) {
9         print("True!");
10    } else {
11        print("False!");
12    }
13
14    return 0;
15 }
```

Expected output:

True!

False!

9.9.47 test-if.hpg

```
1 int main() {
2     if (true) {
3         print("True!");
4     }
5
6     return 0;
7 }
```

Expected output:

True!

9.9.48 test-is-empty.hpg

```
1 int main() {
2     graph<int:int, int> g1 = [];
3     print_bool(g1.is_empty());
4     graph<int:int, int> g2 = [1];
5     print_bool(g2.is_empty());
6 }
```

Expected output:

0

1

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

```
1 int main() {
2     graph<int, string> g = [1 -(a)- 2];
3     add_node_remotely(g; 3);
4     add_node_remotely(g; 4);
5     add_node_remotely(g; 5);
6     g.print();
7     return 0;
8 }
```

```
9
10 void add_node_remotely(graph<int, string> g; int label) {
11     node<int> n = label;
12     g.set_node(n);
13 }
```

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

```
1 int main() {
2     node<int:string> n1 = 500:"foo";
3     node<string:bool> n2 = "bar":true;
4     node<bool:string> n3 = false;
5     node<int:int> n4 = 12345:6789;
6     n1.print();
7     print("");
8     n2.print();
9     print("");
10    n3.print();
11    print("");
12    n4.print();
13 }
```

Expected output:

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

9.9.51 test-printbool.hpg

```
1 int main() {
2     print_bool(true);
3     print_bool(false);
4     print_bool(2 + 4 == 6);
5     print_bool(2 + 4 == 5);
6     return 0;
7 }
```

Expected output:

```
1
0
1
0
```

9.9.52 test-printint.hpg

```
1 int main() {
2     print_int(123);
3     return 0;
4 }
```

Expected output:

123

9.9.53 test-recursion1.hpg

```
1 void count(int i) {
2     if (i <= 0) {
3         print_int(i);
4     } else {
5         print_int(i);
6         count(i - 1);
7     }
8     return;
9 }
```

10

```
11 int main() {
12     count(5);
13     return 0;
14 }
```

Expected output:

5
4
3
2
1
0

9.9.54 test-recursion2.hpg

```
1 int main() {
2     print_int(fib(7));
3     return 0;
4 }
5
6 int fib(int i) {
7     if (i < 2) {
8         return 1;
9     } else {
10        return fib(i - 2) + fib(i - 1);
11    }
12 }
```

Expected output:

21

9.9.55 test-remove-edge1.hpg

```
1 int main() {
2     graph<int, int> g = [1 <(10)> 2; 3];
3     print_int(g.remove_edge(1; 2));
4     print_int(g.remove_edge(2; 1));
5     print_int(g.remove_edge(2; 1));
6 }
```

Expected output:

0

```
0  
-1
```

9.9.56 test-remove-node-bool.hpg

```
1 int main() {  
2     graph<bool, int> g = [true <(10)> false];  
3     int result1 = g.remove_node(true);  
4     print_int(result1);  
5     int result2 = g.remove_node(true);  
6     print_int(result2);  
7 }
```

Expected output:

```
0  
-1
```

9.9.57 test-remove-node-int.hpg

```
1 int main() {  
2     graph<int, int> g = [1 <(10)> 2; 3];  
3     int result1 = g.remove_node(1);  
4     print_int(result1);  
5     int result2 = g.remove_node(5);  
6     print_int(result2);  
7 }
```

Expected output:

```
0  
-1
```

9.9.58 test-remove-node-str.hpg

```
1 int main() {  
2     graph<string, int> g = ["hello" <(10)> "there"; "!"];  
3     int result1 = g.remove_node("!!");  
4     print_int(result1);  
5     int result2 = g.remove_node("hello");  
6     print_int(result2);  
7 }
```

Expected output:

```
-1  
0
```

9.9.59 test-set-data1.hpg

```
1 int main() {  
2     node<int:int> n = 1:1;  
3     print_int(n.get_data());  
4     n.set_data(2);  
5     print_int(n.get_data());  
6     return 0;  
7 }
```

Expected output:

```
1
```

2

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

```
1 int main() {
2     graph<bool:int, int> g = [true:1 <(3)> false:9];
3     g.set_edge(true; false; 1);
4     print_int(g.get_weight(true; false));
5     return 0;
6 }
```

Expected output:

1

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

```
1 int main() {
2     graph<bool:int, string> g = [true:1 <("bla")> false:9];
3     g.set_edge(false; true; "bla");
4     print(g.get_weight(false; true));
5     return 0;
6 }
```

Expected output:

bla

9.9.62 test-set-edge-bool.hpg

```
1 int main() {
2     graph<bool:bool, string> g = [true:true; false: false];
3     g.set_edge(true; false); (* set weight to empty *)
4     g.print();
5     return 0;
6 }
```

Expected output:

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

false:false ->[]

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

```
1 int main() {
2     graph<int:int, bool> g = [1:1 <(true)> 2:9];
3     g.set_edge(1; 2; false);
4     print_bool(g.get_weight(1; 2));
5     return 0;
6 }
```

Expected output:

0

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

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

```
5     return 0;
6 }
```

Expected output:

```
1
```

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

```
1 int main() {
2     graph<int:bool, string> g = [1:true; 2: false];
3     g.set_edge(1; 2; "foo");
4     print(g.get_weight(1; 2));
5 }
```

Expected output:

```
foo
```

9.9.66 test-set-edge-int.hpg

```
1 int main() {
2     graph<int:bool, string> g = [1:true; 2: false];
3     g.set_edge(1; 2); (* set weight to empty *)
4     g.print();
5 }
```

Expected output:

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

```
2:false ->[]
```

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

```
1 int main() {
2     graph<string:int, bool> g = ["hi":1 <(true)> "there":9];
3     g.set_edge("hi"; "there"; false);
4     print_bool(g.get_weight("hi"; "there"));
5     return 0;
6 }
```

Expected output:

```
0
```

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

```
1 int main() {
2     graph<string:int, string> g = ["hello":1 <"yes"> "good":9];
3     g.set_edge("hello"; "good"; "now");
4     print(g.get_weight("hello"; "good"));
5     return 0;
6 }
```

Expected output:

```
now
```

9.9.69 test-set-edge-str.hpg

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

```
3     g.set_edge("1"; "2"); (* set weight to empty *)
4     g.print();
5 }
```

Expected output:

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

9.9.70 test-set-node1.hpg

```
1 int main() {
2     graph<int:int, int> g = [1:1];
3     node<int:int> n = 2:2;
4     g.set_node(n);
5     return 0;
6 }
```

9.9.71 test-vdecls-global.hpg

```
1 int a;
2 int b;
3 string c;
4
5 int main() {
6     b = 40;
7     a = b + 2;
8     string c = "foo";
9     print_int(a);
10    print_int(b);
11    print(c);
12
13    return 0;
14 }
```

Expected output:

```
42
40
foo
```

9.9.72 test-vdecls.hpg

```
1 int res;
2
3 int main() {
4     res = 7;
5     int x = res + 2;
6     int y = x - 1;
7     print_int(y);
8     return 0;
9 }
```

Expected output:

```
8
```

9.9.73 test-while1.hpg

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

```

3
4     while (x <= 2) {
5         print ("True!");
6         x = x + 1;
7     }
8
9     return 0;
10 }

```

Expected output:

True!

True!

True!

9.9.74 test-while2.hpg

```

1 int main() {
2     int x = 0;
3
4     while (x < 3) {
5         print ("True!");
6         x = x + 1;
7     }
8
9     return 0;
10 }

```

Expected output:

True!

True!

True!

9.10 Regression Test Suite - Negative Tests

9.10.1 fail-are-neighbors.hpg

```

1 int main() {
2     graph<int, int> g = [1 <(5)> 2 -()> 3];
3     print_bool(g.are_neighbors(1, "2"));
4 }

```

Expected error:

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

9.10.2 fail-create-graph1.hpg

```

1 int main() {
2     graph<int:int, bool> g = ["foo"];
3     return 0;
4 }

```

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

```

1 int main() {
2     graph<int, bool> g = [1:2];
3     return 0;
4 }
```

Expected error:

Fatal error: exception Failure("illegal assignment graph<int, bool> = graph<int, int, bool> in g = [nodes: [1: 2], edges: []]"")

9.10.4 fail-create-graph3.hpg

```

1 int main() {
2     graph<int:int> g = [NULL:5];
3     return 0;
4 }
```

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

```

1 int main() {
2     graph<int:bool, int> g = [1:"foo"];
3     return 0;
4 }
```

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

```

1 int main() {
2     graph<int:bool, int> g = [1:true <("foo")> 2:false];
3     return 0;
4 }
```

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

```

1 int main() {
2     graph<int:bool, int> g = [1:true <()> 2:"foo"];
3     return 0;
4 }
```

Expected error:

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

9.10.8 fail-fdecls-argnum1.hpg

```

1 int main() {
2     print_int(addone(41, 1));
3     return 0;
```

```

4 }
5
6 int addone(int x) {
7     return x + 1;
8 }
```

Expected error:

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

9.10.9 fail-fdecls-argtype1.hpg

```

1 int main() {
2     return 0;
3 }
4
5 int void_arg(void foo) {
6     return 1;
7 }
```

Expected error:

Fatal error: exception Failure("illegal void args foo")

9.10.10 fail-fdecls-argtype2.hpg

```

1 int main() {
2     return 0;
3 }
4
5 int void_arg(void foo) {
6     return 1;
7 }
```

Expected error:

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

9.10.11 fail-find-data.hpg

```

1 int main() {
2     graph<string:int> g = ["a":10 -()> "b":20 <()> "c":20; "a" <()-> "d"];
3     graph<string:string> g_new = g.find("20");
4     for_node(n : g_new) {
5         n.print();
6         print("");
7     }
8 }
```

Expected error:

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

9.10.12 fail-for-edge1.hpg

```

1 int main() {
2     int g = 0;
3     for_edge(src, dst, w : g) {
4         print("foo");
5     }
6     return 0;
7 }
```

Expected error:

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

9.10.13 fail-for-edge2.hpg

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

Expected error:

Fatal error: exception Failure("illegal argument found node:int, int; expected int in src")

9.10.14 fail-for-node1.hpg

```
1 int main() {
2     int g = 0;
3     for_node(n : g) {
4         print_int(n);
5     }
6 }
```

Expected error:

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

9.10.15 fail-for-node2.hpg

```
1 int main() {
2     int g = 0;
3     for_node(n : g) {
4         print_int(n);
5     }
6 }
```

Expected error:

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

9.10.16 fail-get-name.hpg

```
1 int main() {
2     node<int> n = 1;
3     print(n.get_name());
4 }
```

Expected error:

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

9.10.17 fail-get-node-data1.hpg

```
1 int main() {
2     node<string:int> n1 = "abc":123;
3     string d1 = n1.get_data();
4     return 0;
5 }
```

Expected error:

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

9.10.18 fail-get-weight.hpg

```
1 int main() {
2     graph<int:string, string> g = [1:"foo" -(10)> 2:"bar" -(>) 3];
3     print(g.get_weight("1"; "2"));
4 }
```

Expected error:

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

9.10.19 fail-graph-neighbors1.hpg

```
1 int main() {
2     graph<int, int> g = [];
3     graph<int, string> g_sub = g.neighbors(5; 2; true);
4 }
```

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

```
1 int main() {
2     graph<int, int> g = [];
3     graph<int, int> g_sub = g.neighbors("5"; "2"; true);
4 }
```

Expected error:

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

9.10.21 fail-if.hpg

```
1 int main() {
2     if (1) {
3         print("Foo");
4     }
5     return 0;
6 }
```

Expected error:

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

9.10.22 fail-localvars.hpg

```
1 int res;
2
3 int main() {
4     res = 7;
5     int x = res + 2;
6     y = x - 1;
7     print_int(y);
8     return 0;
9 }
```

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

```
1 int main() {
2     print(1);
3     return 0;
4 }
```

Expected error:

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

9.10.25 fail-printint.hpg

```
1 int main() {
2     print_int("Hello, world");
3     return 0;
4 }
```

Expected error:

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

9.10.26 fail-remove-edge1.hpg

```
1 int main() {
2     graph<int, int> g = [1 <(10)> 2; 3];
3     g.remove_edge(1, "foo");
4 }
```

Expected error:

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

9.10.27 fail-remove-edge2.hpg

```
1 int main() {
2     graph<int, int> g = [1 <(10)> 2; 3];
3     string ret = g.remove_edge(1, 2);
4 }
```

Expected error:

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

9.10.28 fail-remove-node1.hpg

```
1 int main() {
2     graph<int, int> g = [1 <(10)> 2; 3];
3     int result1 = g.remove_node("1");
4 }
```

Expected error:

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

9.10.29 fail-set-data1.hpg

```
1 int main() {
2     node<int:string> n = 1:"foo";
3     n.set_data(1);
4     return 0;
5 }
```

Expected error:

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

9.10.30 fail-set-edge1.hpg

```
1 int main() {
2     graph<int:int, string> g = [1:1 <("foo")> 2:9];
3     g.set_edge(1; 2; 1);
4     return 0;
5 }
```

Expected error:

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

9.10.31 fail-set-node1.hpg

```
1 int main() {
2     graph<int:int, int> g = [1:1];
3     node<int:bool> n = 2:true;
4     g.set_node(n);
5     return 0;
6 }
```

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

```
1 int main() {
2     int a = 1;
3     if (a == 2) {
4         int b = 4;
5         if (b == 4) {
6             int c;
7         }
8         print_int(c);
9     }
10 }
```

Expected error:

Fatal error: exception Failure("undeclared variable c")

9.10.33 fail-vdecls-global1.hpg

```
1 string a;
2
```

```
3 int main() {
4     a = 4;
5     return 0;
6 }
```

Expected error:

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

9.10.34 fail-vdecls-global2.hpg

```
1 int a;
2
3 int main() {
4     print(a);
5     return 0;
6 }
```

Expected error:

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

9.10.35 fail-vdecls-local1.hpg

```
1 int main() {
2     void foo;
3 }
```

Expected error:

Fatal error: exception Failure("variable 'foo' declared void")