giraph

a language for manipulating graphs

Jessie Liu	Seth Benjamin	Daniel Benett	Jennifer Bi
Manager	Language Guru	System Architect	Tester
jll2219	sjb2190	deb2174	jb3495

Fall 2017

Contents

1 Introduction **3**
1.1 Motivation . . . 3

2 Language Tutorial **4**
2.1 Basic Syntax . . . 4
2.2 Graphs, Nodes, Edges . . . 4
2.3 Graph Iterators . . . 6
2.4 Generics in Maps and Graphs . . . 8

3 Language Manual **9**
3.1 Lexical Conventions . . . 9
3.2 Data Types . . . 9
3.2.1 Primitive types . . . 9
3.2.2 Graph types . . . 9
3.2.3 Maps . . . 10
3.3 Operators and Expressions . . . 10
3.3.1 Variable Assignment . . . 10
3.3.2 Arithmetic Operators . . . 10
3.3.3 Logical and Relational Operators . . . 10
3.3.4 Graph construction . . . 10
3.3.5 Methods . . . 11
3.4 Control Flow . . . 12
3.4.1 Conditionals . . . 12
3.4.2 Loops . . . 13
3.5 Scoping . . . 13
3.6 Program Structure . . . 13

4 Project Plan **14**
4.1 Work Process . . . 14
4.2 Software Development Environment . . . 14
4.3 Programming Style Guide . . . 14
4.4 Project Timeline . . . 14
4.5 Team Roles and Responsibilities . . . 15
4.6 Project Log . . . 15

5 Architectural Design **16**
5.1 Components . . . 16
5.2 Interfaces between Components . . . 16
5.2.1 Lexer (all contributed) . . . 16
5.2.2 Parser (Seth, Jennifer, Jessie) . . . 16
5.2.3 Semantic checking (Jessie, Jennifer, Seth) . . . 16
5.2.4 Code generation (Seth, Daniel) . . . 16
5.2.5 C Libraries (Daniel, Seth) . . . 17

6 Test Plan **18**
6.1 Unit testing . . . 18
6.2 Regression testing . . . 18

7 Lessons Learned **20**
7.1 Danny . . . 20
7.2 Jennifer . . . 20
7.3 Jessie . . . 20
7.4 Seth . . . 20

8 Appendix **22**
8.1 Scanner . . . 22
8.2 Parser . . . 23
8.3 Semantic Checking . . . 31
8.4 Code Generation . . . 60
8.5 C Libraries . . . 80
8.6 Testing . . . 104
8.7 Example code . . . 108
8.8 Project Log . . . 109

1 | Introduction

1.1 Motivation

Graphs are a fundamental method of representing the world we live in. Though graphs are composed of simple nodes and edges, they are powerful enough to be applied to real areas like economics and chemistry, with applications ranging from airline scheduling to linguistic modeling. However, graph creation and algorithms in common programming languages can be verbose and tedious to write. *giraph* is a programming language with the goal of simplifying the creation of graphs and the implementation of graph algorithms.

2 Language Tutorial

2.1 Basic Syntax

Our program uses C-like syntax. Executables run from their main function. All functions must have a return. This example program increments a counter 10 times and then prints the result:

```
1 void main() {
2   int counter = 0;
3   while (counter < 10) {
4     counter = counter + 1;
5   }
6   print(counter);
7   return;
8 }
```

2.2 Graphs, Nodes, Edges

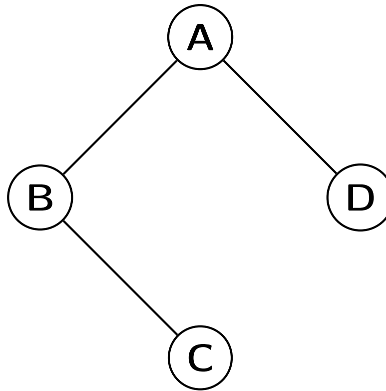
There are 4 graph types: graphs, digraphs, wegraphs, and wedigraphs.

	weighted?	directed?
graph	✗	✗
wegraph	✓	✗
digraph	✗	✓
wedigraph	✓	✓

Consider the following code snippet:

```
1 graph<int> g = [A:1 -- B:2 -- C:3 ; D:4 -- A];
```

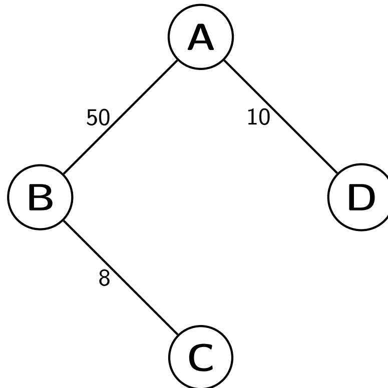
This creates a graph (unweighted and undirected) with the following structure, initializing nodes A, B, C, and D with int data values 1, 2, 3, and 4 (respectively):



To create a wegraph, use the following syntax:

```
1 wegraph<int> g = [A:1 -{50}- B:2 -{8}- C:3 ; D:4 -{10}- A];
```

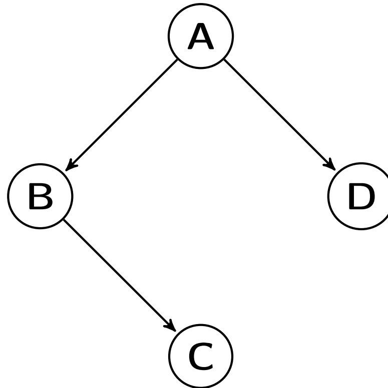
This creates a graph with the following structure:



To create a digraph, use the following syntax:

```
1 digraph<int> g = [A:1 -> B:2 -> C:3 ; D:4 <- A];
```

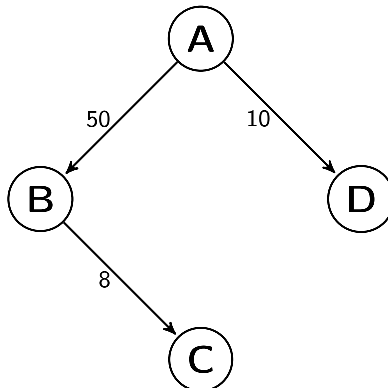
This creates a graph with the following structure:



To create a wedigraph, use the following syntax:

```
1 wedigraph<int> g = [A:1 -{50}-> B:2 -{8}-> C:3 ; D:4 <-{10}- A];
```

This creates a graph with the following structure:



In each of the above cases, the four nodes A, B, C, D are defined as part of the inline graph definition. It is also possible, however, to create nodes and set their data outside of graph definitions, as in this example:

```
node<int> n;
n.set_data(1);
graph<int> g = [n];
```

After line 3, graph 'g' contains a single node 'n' with integer data 1.

If you assign a previously declared and assigned node in a graph definition, its old data will be overwritten, as in this example:

```
node<int> n;
n.set_data(1);
graph<int> g = [n:5];
```

After line 3, graph 'g' contains a single node 'n' with integer data 5

Multiple graphs can share the same nodes. When a node's data is updated, this update occurs across all graphs. Consider the following example:

```
graph<int> g = [A:1 -- B:2 -- C:3];
graph<int> g_subgraph = [A -- B];
A.set_data(5);
```

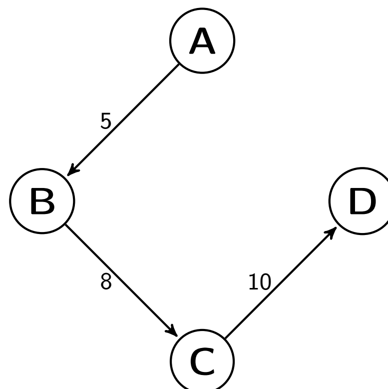
After line 3, in both 'g' and 'g_subgraph', node A will have data 5.

It is also possible to add edges and set edge weights outside of the initial graph definition, as in the following snippet:

```
wedigraph<int> g = [A:1 -{50}-> B:2 -{8}-> C:3 ; D:4 <-{10}- A];

g.remove_edge(A, D);
g.set_edge_weight(A, B, 5);
g.add_edge(C, D, 10);
```

This creates a graph with the following structure:



2.3 Graph Iterators

for_node

Consider example:

```
void main() {
  int counter = 0;
  graph<int> g = [A:1 -- B:2 -- C:3];
  for_node(n : g) {
    counter = counter + n.data();
  }
  print(counter);
  return;
}
```

The above program will print 6. It iterates through each node in graph g, assigning each to variable n, and executing the for_node body with each n. The iteration order is consistent across program runs, but is not predictable from the structure of the graph.

for_edge

Consider example:

```
void main() {
  int counter = 0;
  wegraph<int> g = [A:1 -{1}- B:2 -{2}- C:3];
  for_edge(e : g) {
    counter = counter + e.weight();
  }
  print(counter);
  return;
}
```

The above program will print 3. It iterates through each edge in graph g, assigning each to variable e, and executing the for_edge body with each e. The iteration order is consistent across program runs, but is not predictable from the structure of the graph.

bfs

Consider example:

```
void main() {
  int counter = 0;
  digraph<int> g = [A:1 -> B:2 -> C:3];
  bfs(n : g ; B) {
    counter = counter + n.data();
  }
  print(counter);
  return;
}
```

The above program will print 5. It visits through each node in graph g in bfs order starting from node B, assigning each to variable n, and executing the bfs body with each n. In this example, node A is never visited.

dfs

The dfs iterator works just as the bfs iterator above, but visits nodes in dfs order.

2.4 Generics in Maps and Graphs

Maps

Maps pair nodes (serving as unique keys) to values. We may use maps to store and access data. Consider example:

```
1 node<int> C;
2 map<int> m;
3
4 m.put(C, C.data());
5
6 printb(m.contains(C));
```

The program prints true, since the map contains our data.

Graphs

Generics allow for nesting in graphs. Consider example:

```
1 graph<string> g = [A:"gir" -- B:"aph"];
2 graph<graph<string>> G = [N:g];
3
4 for_node(ng : G){
5     for_node(n : ng.data()) {
6             prints(n.data());
7     }
8 }
```

Here, the graph G contains a node which contains a graph (ng.data()), which we access and iterate over using for_node.

3 | Language Manual

3.1 Lexical Conventions

Tokens fall into the following categories: identifiers, keywords, literals, expression operators. Whitespace (of any amount) may be used to separate tokens.

- Comments, single-line or multi-line, are indicated by !~ the comment characters ~!
- Identifiers are sequences of alphanumeric characters, including underscore: _. All identifiers must begin with an alphabetic character. Identifiers may not be the same as reserved keywords.
- Reserved keywords are

```
bool       int       float     string
graph      node      wegraph   digraph
wedigraph  map       void      return
if         then      else      for
while      for_node  for_edge  bfs
dfs        true      false
```

3.2 Data Types

giraph is a statically typed language, and supports the following:

3.2.1 Primitive types

- bool - Boolean data, which can be true or false.
- int - Ints are signed 8-byte literals. Represents a number as a sequence of digits.
- float - Floats are signed single-precision floating point numbers.
- string - A sequence of ASCII characters. Literals are enclosed in double quotes: string s = "graphs are cool"

3.2.2 Graph types

A graph consists of nodes and optionally edges. The following distinct subtypes of graph are supported, with the following hierarchical structure:

- graph - A graph whose edges are unweighted and undirected
- digraph - A graph whose edges are directed
- wegraph - A graph whose edges are weighted with positive or negative integer weights
- wedigraph - A graph whose edges are directed and weighted with positive or negative weights

Graphs and nodes are generically implemented in giraph. The type of data the nodes in a graph can store may be any of the data types in the language. A graph can only have a single type, which must be specified (e.g. digraph<int> is the type of a directed graph whose nodes store ints). A graph's data type can also be a generic type (e.g. graph<graph<int»).

3.2.3 Maps

A map stores key-value pairs keyed on nodes. Maps are also generically implemented: keys are always nodes, but values can be any type (including generic types). Like graphs, map can only store a single type of value, which must be specified (e.g. `map<int>`). However, the keys of the map can be nodes with any type of data. For example, you can put a `node<string>` and a `node<int>` as keys in the same map.

3.3 Operators and Expressions

3.3.1 Variable Assignment

Variables are assigned using the = operator. The left hand side must be an identifier while the right hand side must be a value or another identifier. The LHS and RHS must have the same type, as conversions or promotions are not supported. The variable assignment operator groups right-to-left.

3.3.2 Arithmetic Operators

Arithmetic operator precedence is as in standard PEMDAS. Operator binding is as follows:

Additive operators $+, -$ group left-to-right.
Multiplicative operators $*, \backslash, \%$ group left-to-right.
Parentheses have the highest precedence, and therefore can be used to override the default operator precedence.

3.3.3 Logical and Relational Operators

Relational operators `(<, >, <= , >=)` and logical operators `(&&, ||)` also group left-to-right. So, a statement like `a < b && b < c && c < d` can simply be accomplished with `a < b < c < d`. Expressions with relational and or logical operators return 0 or 1 for true or false respectively.

Equality operators also group left-to-right, but have lower precedence than relational ones. Logical operators have the lowest precedence of the three.

3.3.4 Graph construction

The subtype of graph and type of its nodes' data are designated when a new graph is declared. A graph can be initialized with a graph literal, which is enclosed in square brackets ([ ]). A graph literal consists of nodes, edges, and node initializations.

New nodes may be declared and optionally initialized in graphs by including a new identifier in a graph literal; this identifier becomes visible in the most local scoped block. Existing nodes may be added to graphs by referencing their identifier. A node may be referenced (but not initialized) multiple times within one graph construction.

Edges between nodes are defined through intuitive edge tokens, the available edges being undirected (`--`), directed (`->, <-, <->`), weighted undirected (`-{0}-`), weighted directed (`-{0}->`). The type of edge corresponds to the subtype of graph and must be consistent across all edges in the graph expression. A graph literal without edges can be assigned to any of the graph subtypes.

Separate parts of the same graph can written in the same literal by separating them with semicolons; the final graph is the union of all parts. This can be useful for including separate components that are

not connected by edges in the same graph.

Nodes can be initialized with data using the `:` operator. A node identifier followed by a colon followed by an expression in a graph literal initializes the data in the node to be the value of the expression. The expression type for any node in a graph literal must match the data type of the graph it is being assigned to.
A graph literal must either contain at least one previously-declared node, or at least one initialized node.

For example, the following declarations/initializations are all separately valid:

```
node<int> n; !~ a node declaration, with no data initialized ~!
graph<int> g = [A:1 -- B:2 -- C:3 -- A ; D:4 -- A ; E:5];
digraph<float> g = [A:1.0 <-> B:2.0 ; E:5.0 <- A];
digraph<bool> g = []; !~ creates an empty graph with no nodes ~!
wegraph<string> g = [A:"hi" -{1}- B:"there"];
wedigraph<int> g = [A:1 -{1}-> B:2 <-{2}- C:3 <-{3}-> D:4];
```

3.3.5 Methods

Methods can be invoked as follows.

```
id . method()
```

Methods are the primary mode of working with graphs and the nodes/edges belonging to a graph. While a node may exist independently of graphs, construction as part of a graph is preferable. Edges may not be initialized independently of graphs, as they represent a relation between two nodes in a graph. Thus, nodes and edges are manipulated through built-in methods.

- `node<t>` - Nodes hold data within a graph and this data may be accessed and modified.
  - *node*`.data()` - returns data stored in *node*
  - *node*`.set_data(t` *data*`)` - stores data in *node*
- `edge` - An edge connects two nodes within a graph. While iterating over the edges in a graph with the for_edge iterator, the following methods can be called on an edge.
  - *edge*`.from()` - returns node *edge* is coming from. In an undirected graph, *edge*`.from() !=` *edge*`.to()`, except in a self-loop. Beyond that, there is no guarantee as to which node will be returned by *edge*`.from()`
  - *edge*`.to()` - returns node *edge* is going to. In an undirected graph, *edge*`.from() !=` *edge*`.to()`, except in a self-loop. Beyond that, there is no guarantee as to which node will be returned by *edge*`.to()`

    In wegraphs and wedigraphs, the following methods may also be called on edges during for_edge iteration:
    - *edge*`.weight()` - returns weight of *edge*
    - *edge*`.set_weight(int` *i*`)` - sets weight of *edge* to int *i*.
- `graph<t>` and all subtypes - graph methods are as follows.
  - *graph*`.add_node(node<t>` *node*`)` - add *node* to *graph*. If *node* is already in *graph*, does nothing.

- *graph*.add_edge(node<t> *from_node*, node<t> *to_node* - add edge between *from_node* and *to_node* to *graph*. If either of *from_node* or *to_node* are not already in *graph*, they are added, and then connected by an edge.
- *graph*.remove_node(node<t> *node*) - remove *node* from *graph*. If *node* is not already in *graph*, does nothing.
- *graph*.remove_edge(node<t> *from_node*, node<t> *to_node*) - remove edge between *from_node* and *to_node* from *graph*. If there is not an edge between *from_node* or *to_node* in *graph*, or if either node is not in the graph, does nothing.
- *graph*.has_node(node<t> *node*) - returns true if *graph* has node *node*, false otherwise
- *graph*.has_edge(node<t> *from_node*, node<t> *to_node*) - returns true if *graph* has an edge from *from_node* to *to_node*, false otherwise
- *graph*.neighbors(node<t> *node*) - returns edgeless graph containing all neighbors of *node* as nodes
- *graph*.print() - prints graph nodes, node data, and adjacencies.

• wegraph<t> and wegraph<t> - in addition to the above, additional weighted graph methods are as follows.

- *graph*.add_edge(node<t> *from_node*, node<t> *to_node*, int *weight*) - add edge with weight *weight* between *from_node* and *to_node* to *graph*. If either of *from_node* or *to_node* are not already in *graph*, they are added, and then connected by an edge. This method replaces the 2-argument add_edge listed above.
- *graph*.get_edge_weight(node<t> *from_node*, node<t> *to_node*) - returns weight of the edge between *from_node* and *to_node* in *graph*
- *graph*.set_edge_weight(node<t> *from_node*, node<t> *to_node*, int *i*) - sets weight of the edge between *from_node* and *to_node* in *graph* to int *i*

• map<t> - map methods are as follows.

- *map*.put(node<?> *key*, t *val*) - add *key* to map with associated value *val*. If *key* is already in map, replace its old associated value with *val*. *key* can be a node of any type.
- *map*.get(t *key*) - get the value associated with *key* from the map.
- *map*.contains(t *key*) - returns true if *map* contains *key* and false if it does not. It is best to call this prior to get in order to insure that the map contains the desired *key*.

3.4 Control Flow

3.4.1 Conditionals

If-else statements are allowed, in the following formats:

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

The else block is optional for any if statement, and any number of else if's can be appended to any if statement.

3.4.2 Loops

C-style while loops and for loops are provided, such as the following:

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

They can either be followed by a single statement to be looped, or by a sequence of statements enclosed within brackets. Graph-specific iteration over nodes and edges is also allowed, using "for each" loops, which take the following format:

for_node(*node* : *graph*) {statements}
for_edge(*edge* : *graph*) {statements}

These iterate over all the nodes/edges of *graph* respectively, executing the looped statements at every node/edge. For example, the following loop can be used to print the data at every node in some graph g:

```
for_node(n : g) {print(n.data());}
```

Each of `for_node` and `for_edge` iterate over their respective graph components in an unspecified order. However, one can also iterate over graph components in a specific order using the following loop constructions:

bfs(*node* : *graph* ; *root*) {statements}
dfs(*node* : *graph* ; *root*) {statements}

These iterate over the nodes of a graph using breadth-first search and depth-first search respectively, starting at *root* and executing the looped statements at every subsequently reached node.

3.5 Scoping

A variable exists within the scope in which it is declared. This scope could be global (outside of any function) or local, within a function, if block, then block, else block, any loop, or a plain scope delimited by braces: `{ !~this is a scope~! }`.

3.6 Program Structure

Programs in *giraph* consist of a list of declarations which includes either variable or function declarations. The declarations must include a `main()` function which is the entry point of a compiled executable *giraph* program. Functions are defined with the following signatures:

```
return_type function_name(type arg, type arg, ...)  {body}
```

Program execution begins in the `main()` method, and the program exits upon arriving at its return statement.

4 | Project Plan

4.1 Work Process

Our team met to work on the project around once a week at the beginning of the semester, and almost every day the last month. We met with our TA Lizzie before major deadlines (getting the LRM, hello world) along with a few other check ins to run ideas past her, discuss issues, and get encouragement. Our main form of communication was Facebook Messenger, over which we communicated almost every day and pretty much every hour in the final stages of the project. We also would meet to work together in groups, frequently drinking green tea in EC or cappuccinos in the CS lounge as we pushed code late into the night.

4.2 Software Development Environment

To build *giraph*, we used these languages and development tools:

- **OCaml version 4.05.0**: for scanning, parsing, and semantic checking
- **C**: for building graphs
- **Makefile**: for compiling and linking
- **Git and Github** for version control and hosting our git repository, respectively
- **Bash Shell scripting**: for automating testing

4.3 Programming Style Guide

Our team used a very simple style guide (which doesn't quite warrant a whole page):

- Spaces, not tabs
- Functions and variables should be named with under_scores, not camelCase
- Indentation should be consistent
- Code should be as clear and concise as possible, with comments added for clarity

4.4 Project Timeline

10/16: Language Reference Manual submitted

10/30: Hello World program compiled

11/17: First parsed graph

12/2: Graphs in Codegen

12/16: Digraphs end to end

12/17: Semantic checking added - sast types integrated into codegen

12/18: Max Flow program works

12/19: Maps added end to end

12/20: Final presentation

4.5 Team Roles and Responsibilities

Though we each technically had an assigned role, boundaries between them faded as we continued to work on the project. We often found ourselves stepping into parts of different roles, picking up slack in places and also allowing other members of the team to help us out when we needed it. We all contributed to components of the language design, compiler, and test suite, and also made a point to check in with each other frequently.

Jessie Liu: Semantic checking, parser, lexer, tests

Seth Benjamin: Code generation, C libraries, parser, semantic checking, lexer, tests

Daniel Benett: C libraries, code generation, lexer, parser, tests

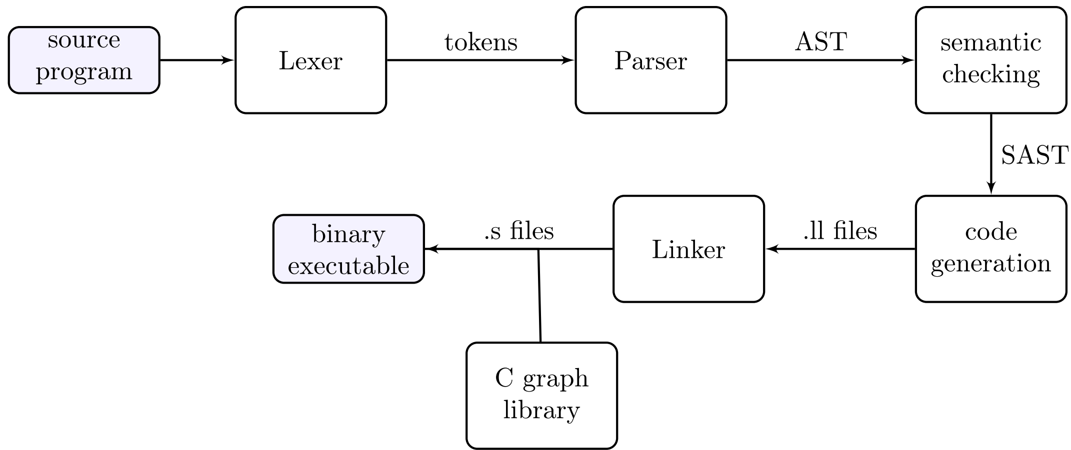
Jennifer Bi: Test suite, semantic checking, parser, lexer

4.6 Project Log

See appendix 8.8 for our project log.

5 | Architectural Design

5.1 Components



5.2 Interfaces between Components

5.2.1 Lexer (all contributed)

The lexer reads in a program string and returns tokens representing the program.

5.2.2 Parser (Seth, Jennifer, Jessie)

The parser reads in tokens from the lexer, and returns an abstract syntax tree. The structure of the tree is determined by the grammar. A program not accepted by the grammar is thus rejected in the parser. In our case, the parser did a lot of heavy-lifting for graph construction. In particular, the parser extracts node names, node initializations, and edge relationships and weights directly from graph literals. Additionally, determining the subtype of a graph literal (weightedness/directedness) is traditionally a type-related check and could have been the responsibility of the semantic checker. However, we found that we could make use of grammar rules to immediately enforce consistency in weightedness/directedness of edges. So, the AST produced by the parser has a subtype by the time it reaches semantic checking. The node initialization types are not checked by the parser.

5.2.3 Semantic checking (Jessie, Jennifer, Seth)

In semantic checking, we perform type checking and enforce static scoping. Our semantic checker keeps an environment variable that corresponds to each scoped block. The environment includes a symbol table for local variables, a map of visible functions, along with several other fields. The recursive nature of OCaml was conducive to using applicative order evaluation, in which the innermost expression were evaluated first, semantically checked, and its environment updated. As each token of the AST is checked, valid expressions are tagged with types, while anything invalid returns a Failure. These type-tagged expressions are then added to a SAST (semantically-checked AST).

5.2.4 Code generation (Seth, Daniel)

In code generation, we step through the SAST produced after semantic checking and converts it to an LLVM module, which is then outputted. We start at the top level of the SAST (containing a list of globals and a list of functions) and unpacking the heavily nested structure down to the terminals.

First, the globals are allocated and added to a symbol table, mapping their identifiers to their allocated registers. For each function, the formals are first processed and added to this symbol table. Then, we enter the body of the function, which is a scoped block. Whenever we enter a scoped block, we create a new symbol table, containing the outer block's variables, as well as allocating and mapping any new variables declared within the block. Then, all of the statements in the block are mapped to LLVM instructions which are outputted in the module. The semantic type information included in the SAST is heavily used in codegen, especially with graphs. Given that the different graph types all have fairly similar user-facing functionalities (e.g. the methods we provide) that have different behaviors (e.g. add_edge(from, to) should add an undirected edge in an undirected graph, but a directed edge in a directed graph), codegen is able to solve this problem by using the type information to determine what instructions to output / what C functions to call. The code generation stage also declares function headers for all of the functions implemented in the C graph library, so that they appear in the outputted LLVM and can be linked from the library's object files.

5.2.5 C Libraries (Daniel, Seth)

There are a series of C structs used to internally store graphs and maps. These most frequently take linked list form. C functions are called from code generation, taking in void pointers that, when casted and dereferenced, contain the structs and data for graphs and nodes. For some methods, such as *graph*.has_edge, code generation can simply pattern match on the SAST, call the proper C function, and use its return as necessary. For functionality such as the bfs and dfs iterators, LLVM code and C code are interlaced. Each iteration of the bfs/dfs loop requires predicate testing and assignment of the current node in the graph traversal, which is occurring in the C library, to the node variable, which is needed in the loop body's statement list. As such, significant coordination between code generation and C library design was required.

6 Test Plan

We tested components separately in the early stages of development when we were still defining the stages of our compiler. Once the architecture was established and once our language had more functionality, regression testing became ideal. Unit testing served as a secondary debugging method.

6.1 Unit testing

- Lexer and parser

  We tested the parser by running programs with the parser trace option `OCAMLRUNPARAM=p` and manually comparing its output with expected output.
  Also useful for early stage parser testing were the debugging options in Menhir, an alternative LR(1) parser generator, which printed out a parse tree.

```
menhir -interpret -interpret-show-cst parser.mly
```

6.2 Regression testing

A test script adapted from MicroC ran tests that should pass and tests that should fail, with a general rule that for each feature tested as a pass case, a corresponding fail case was also tested. This was useful for keeping semantic checking and code generation in sync, that is, at any point in development all code generation functionality would not go unchecked. The regression testing was useful for tracking ripple effects of small changes in semantic checking or codegen. Furthermore, testing provided a sanity check on basic control flow, variable and function declarations, primitive data assignment operations.

- Semantic checking

  Most semantic errors were caught by regression testing. Most of our target programs were simple to isolate certain features. A few additional test programs had more complexity to test that features could work together.

fail-graphdecl6.gir: test to ensure checking against impossible graph declarations were prevented

```
1 int main() {
2   wedigraph g = [A:5 -{5}-> B:6 <-{4}- A];
3   return 0;
4 }
```

```
Fatal error: exception Failure("graph literal cannot feature the same edge with different weights")
```

fail-bfs1.gir: test to ensure checking against interference with breadth-first search

```
1 int main() {
2   graph g = [A:1 -- B:2 -- C:3 -- D:4 -- E:5 -- F:6  -- G:7];
3   bfs(b : g ; A) {
4                 g.add_edge(A,b);
5   }
6   return 0;
7 }
```

```
Fatal error: exception Failure("concurrent modification of graph in bfs")
```

- Code generation

test-foredge2.gir: one of the tests to ensure for_edge functions properly

```
int main() {
  graph<int> g = [A:1 -- B:2; C:3 -- D:4];
  graph<int> g2 = [];
  for_edge(e : g) {
    g2.add_edge(e.from(), e.to());
  }
  for_edge(e : g2) {
    print(e.from().data());
    print(e.to().data());
  }
  return 0;
}
```

out:
```
2
1
4
3
```

test-neighbors3.gir: tests neighbors method in case where there are no neighbors (nothing should print)

```
int main() {
  graph<int> g = [A:5];
  graph<int> g2 = g.neighbors(A);

  for_node(n : g2) {
    print(n.data());
  }

  return 0;
}
```

out:

test-bfs3.gir: tests bfs iteration in disconnected graph

```
!~ should only print A and B, as C and D are not
accessible from bfs root A ~!
int main() {
  graph<int> g = [A:1 -- B:2 ; C:3 -- D:4];
  bfs(b : g ; A) {
    print(b.data());
  }
  return 0;
}
```

out:
```
1
2
```

These tests are indicative of our testing style. test-foredge2.gir is an example of a test that thoroughly tests the core intended functionality of a feature (for-edge). test-neighbors3.gir is an example of a test that tests an edge case (when there are no neighbors). test-bfs3.gir is an example of a test that tests bfs in the specific case of disconnected graphs. We tried to write small, specific tests that thoroughly cover the cases a particular feature could encounter.

7 | Lessons Learned

Working on *giraph* was an intense learning experience for us in several different ways. Below are some lessons and advice we have for future groups.

7.1 Danny

Set yourself clear goals and then work at them in a quiet room until they have been accomplished. Communicate what you need from everybody else in order to implement your goal end-to-end. Talk out and debate tricky implementations and difficult design decisions. Don't talk too long: eventually just make a decision. Ask for help when you get stuck. Test frequently and comprehensively. When you feel lazy and don't want to write more code, bake your team cookies. The physical warmth of the fresh-out-the-oven cookies will combine with the sentimental warmth of the moment and will imbue your language with the sense of joy that all programming languages truly need to succeed. Make sure to continue work after baking and eating cookies.

7.2 Jennifer

I learned to embrace functional programming and forget about pointers and tons of control flow. It was convenient to be able to use variables and apply functions freely without cumbersome type declarations. I learned a lot about version control since it was my first time using github for development. I'm glad that our team communicated and consulted each other frequently–often we would start writing different pieces of code and find ourselves overlapping or thinking about the same problem. This made the process a lot more enjoyable. Testing became habitual bordering on compulsive, which I'm really glad for. Pushing and pulling on github was a lot easier knowing that *it compiles* and all the cases pass. My advice would be to start early and set more milestones between Hello World and the project due date– building a compiler is pretty much boundless with respect to adding functionality and optimization so you can literally never start too early.

7.3 Jessie

Firstly, I now have the utmost respect for the OCaml language. OCaml is the greatest for doing this kind of compiler building stuff. Scanning and parsing was intuitive given pattern matching, and utilizing OCaml's natural scoping made semantic checking so so much easier. Don't try to fight functional programming - I spent a good amount of time in an imperative mindset trying to write OCaml, and it was not fun. Secondly, test often, especially before pushing code. Thirdly, maintaining good communication, relations, and trust with your teammates is absolutely crucial - you want to know they'll always have your back. And lastly, have fun. This is probably the coolest project I've worked on, ever!

7.4 Seth

1. Write design docs. You do not want to end up 70% of the way through the project and realize that your design was fundamentally flawed. You also do not want to waste time arguing with your teammates about your half-baked ideas. For me, writing design docs solved both these problems: it forced me to organize my thoughts coherently and consider edge cases before proposing an idea. It also produces a paper trail: 3 weeks down the line, when everything is breaking, the design doc is there to remind you why you thought x was a good idea.
2. Work incrementally. Get a super dumb and useless compiler working end-to-end first. Add a single new feature, test, repeat. This thing is way too big to be done in any other way.
3. The majority of my contribution to this language was written in a series of fear-driven code-vomiting

sessions between the hours of 2 and 7 am. I don't *recommend* my approach per se, but it did the job. If you're terrified that the project will crash and burn, you'll go the extra mile to make sure it doesn't.

8 | Appendix

8.1 Scanner

```
(* Authors:
Daniel Benett deb2174
Seth Benjamin sjb2190
Jennifer Bi jb3495
Jessie Liu jll2219
*)
(* Ocammllex scanner for giraph *)
{ open Parser }

(* Definitions *)
let digit = ['0'-'9']
let decimal = ((digit+ '.' digit*) | ('.' digit+))
let letter = ['a'-'z' 'A'-'Z']

(* Rules *)
rule token = parse
    [ ' ' '\t' '\r' '\n'] { token lexbuf } (* to ignore whitespace *)
  | "!~" { comment lexbuf }
  | ',' { COMMA }
  | '.' { DOT }
  | ';' { SEMI }
  | ':' { COLON }
  | '\'' { SINGLEQUOTE }
  | '\"' { DOUBLEQUOTE }

  (* scoping *)
  | '(' { LPAREN }
  | ')' { RPAREN }
  | '{' { LBRACE }
  | '}' { RBRACE }
  | '[' { LBRACK }
  | ']' { RBRACK }

  (* keywords *)
  | "for" { FOR }
  | "while" { WHILE }
  | "for_node" {FOR_NODE}
  | "for_edge" {FOR_EDGE}
  | "bfs" {BFS}
  | "dfs" {DFS}
  | "if" { IF }
  | "then" { THEN }
  | "else" { ELSE }
  | "bool" { BOOL }
  | "float" { FLOAT }
  | "int" { INT }
  | "string" { STRING }
  | "graph" { GRAPH }
  | "node" { NODE }
  | "wegraph" { WEGRAPH }
  | "digraph" { DIGRAPH }
  | "wedigraph" { WEDIGRAPH }
```

```
53   | "map" { MAP }
54 (*  | "break" { BREAK }
55   | "continue" { CONTINUE } *)
56   | "return" { RETURN }
57   | "void" { VOID }
58
59   (* operators *)
60   | '+' { PLUS }
61   | '-' { MINUS }
62   | '*' { TIMES }
63   | '/' { DIVIDE }
64   | '=' { ASSIGN }
65   (* | "+=" { PLUSEQ }
66      | "-=" { MINUSEQ }
67      | "*=" { TIMESEQ }
68      | "/=" { DIVEQ } *)
69   | '%' { MOD }
70   | "&&" { AND }
71   | "||" { OR }
72   (* | '&' { INTERSECTION }
73      | '|' { UNION } *)
74   | '!' { NOT }
75   | "==" { EQ }
76   | "!=" { NEQ }
77   | ">=" { GEQ }
78   | "<=" { LEQ }
79   | '>' { GT }
80   | '<' { LT }
81   | "->" { RARROW }
82   | "<-" { LARROW }
83   | "<->" { DIARROW }
84   | "--" { EDGE }
85
86   (* literals and IDs *)
87   | digit+ as lxm                              { INT_LIT(int_of_string lxm) }
88   | decimal as lxm                             { FLOAT_LIT(float_of_string lxm)
     }
89   | ("true" | "false") as lxm                  { BOOL_LIT(bool_of_string lxm) }
90   | letter (letter | digit | '_')* as lxm      { ID(lxm) }
91   | '\"' ([^'\"']* as lxm) '\"'                { STRING_LIT(lxm) }
92   | eof { EOF }
93   | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
94
95
96 and comment = parse
97     "~!" { token lexbuf }
98   | _    { comment lexbuf }
```

Scanner: scanner.mll

8.2 Parser

```
1 /* Authors:
2 Daniel Benett deb2174
3 Seth Benjamin sjb2190
4 Jennifer Bi jb3495
```

```
5  Jessie Liu jll2219 */
6
7  %{ open Ast
8  open Prshelper %}
9
10
11 %token LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE DOT COMMA SEMI COLON
12 %token PLUS MINUS TIMES DIVIDE ASSIGN NOT MOD
13 %token EQ NEQ LT LEQ GT GEQ AND OR
14 %token RETURN IF THEN ELSE FOR WHILE FOR_NODE FOR_EDGE BFS DFS BREAK CONTINUE
15 %token INT BOOL VOID FLOAT STRING NODE EDGE GRAPH WEGRAPH DIGRAPH WEDIGRAPH MAP
16 %token RARROW LARROW DIARROW
17 %token SINGLEQUOTE DOUBLEQUOTE
18 %token <int> INT_LIT
19 %token <float> FLOAT_LIT
20 %token <bool> BOOL_LIT
21 %token <string> STRING_LIT
22 %token <string> ID
23 %token EOF
24
25 %right SEMI
26
27 /*arithmetic ops*/
28 %right ASSIGN
29 %left PLUS MINUS
30 %left TIMES DIVIDE MOD
31 %left OR AND
32 %right NOT NEG
33
34 %nonassoc EQ NEQ
35 %nonassoc GEQ LEQ GT LT
36 %nonassoc NOELSE
37 %nonassoc ELSE
38
39 %left DOT
40
41 %right LARROW RARROW DIARROW EDGE
42 %nonassoc COLON
43
44 %start program
45 %type <Ast.program> program
46 %%
47
48 program:
49   decls EOF { $1 }
50
51 decls:
52   /* nothing */     { [], [] } /* first list has vdecls, second has fdecls*/
53 | decls vdecl { ($2 :: fst $1), snd $1 }
54 | decls fdecl { fst $1, ($2 :: snd $1) }
55
56 fdecl: typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
57         { { f_typ = $1; f_name = $2; f_formals = $4; f_body = List.rev $7 } }
58
59 typ:
60     INT { Int }
61   | FLOAT { Float }
```

```
62    | BOOL { Bool }
63    | VOID { Void }
64    | STRING { String }
65    | NODE LT typ GT { Node($3) }
66    | GRAPH LT typ GT { Graph($3) }
67    | DIGRAPH LT typ GT { Digraph($3) }
68    | WEGRAPH LT typ GT { Wegraph($3) }
69    | WEDIGRAPH LT typ GT { Wedigraph($3) }
70    | MAP LT typ GT { Map($3) }
71
72  formals_opt: /* nothing */  { [] }
73          | formal_list { List.rev $1 }
74
75  formal_list: typ ID { [($1, $2)] }
76          | formal_list COMMA typ ID { ($3,$4) :: $1 }
77
78  vdecl:
79    typ ID SEMI { ($1, $2) }
80
81  stmt_list:
82    /* nothing */  { [] }
83  | stmt_list stmt { $2 :: $1 }
84
85  stmt:
86    expr SEMI     { Expr $1 }
87  | typ ID SEMI { Vdecl($1, $2, Noexpr) }
88  | typ ID ASSIGN expr SEMI { Vdecl($1, $2, Assign($2,$4)) }
89  | RETURN SEMI   { Return Noexpr }
90  | RETURN expr SEMI { Return $2 }
91  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
92  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
93  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
94  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt { For($3, $5, $7, $9) }
95  | FOR_NODE LPAREN ID COLON expr RPAREN stmt { For_Node($3, $5, $7) }
96  | FOR_EDGE LPAREN ID COLON expr RPAREN stmt { For_Edge($3, $5, $7) }
97  | BFS LPAREN ID COLON expr SEMI expr RPAREN stmt { Bfs($3, $5, $7, $9) }
98  | DFS LPAREN ID COLON expr SEMI expr RPAREN stmt { Dfs($3, $5, $7, $9) }
99  | WHILE LPAREN expr RPAREN stmt         { While($3, $5) }
100 | BREAK SEMI  {Break}
101 | CONTINUE SEMI   {Continue}
102
103 expr:
104   INT_LIT           { Int_Lit($1) }
105 | BOOL_LIT          { Bool_Lit($1) }
106 | STRING_LIT        { String_Lit($1) }
107 | FLOAT_LIT         { Float_Lit($1) }
108 | ID                { Id($1) }
109 | expr PLUS expr { Binop($1, Add, $3)  }
110 | expr MINUS expr { Binop($1, Sub, $3) }
111 | expr TIMES expr { Binop($1, Mult, $3) }
112 | expr DIVIDE expr { Binop($1, Div , $3) }
113 | expr MOD expr { Binop($1, Mod, $3) }
114 | expr EQ expr { Binop($1, Eq, $3) }
115 | expr NEQ expr { Binop($1, Neq, $3) }
116 | expr LEQ expr { Binop($1, Leq, $3) }
117 | expr GT expr { Binop($1, Greater, $3) }
118 | expr LT expr { Binop($1, Less, $3) }
```

```
119 | expr AND expr { Binop($1, And, $3) }
120 | expr OR expr { Binop($1, Or, $3) }
121 | MINUS expr %prec NEG { Unop(Neg, $2) }
122 | NOT expr              { Unop(Not, $2) }
123 | ID ASSIGN expr        { Assign($1, $3) }
124 | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
125 | expr DOT ID LPAREN actuals_opt RPAREN { Method($1, $3, $5) }
126 | LPAREN expr RPAREN { $2 }
127 | LBRACK graph_expr_opt RBRACK { match $2 with (n, e, n_i, is_d, is_w) ->
128                                    Graph_Lit(n, e, n_i, is_d, is_w) }
129
130 graph_expr_opt:
131   /* nothing */ { [], [], [], false, false }
132 | single_node_expr { match $1 with (n, e, n_i) -> (List.rev n, List.rev e, List.
      rev n_i, false, false) }
133 | single_node_exprs_list { match $1 with (n, e, n_i) -> (List.rev n, List.rev e,
      List.rev n_i, false, false) }
134 | ugraph_exprs_list { match $1 with (n, e, n_i) -> (List.rev n, List.rev e, List.
      rev n_i, false, false) }
135 | digraph_exprs_list { match $1 with (n, e, n_i) -> (List.rev n, List.rev e, List
      .rev n_i, true, false) }
136 | wegraph_exprs_list { match $1 with (n, e, n_i) -> (List.rev n, List.rev e, List
      .rev n_i, false, true) }
137 | wedigraph_exprs_list { match $1 with (n, e, n_i) -> (List.rev n, List.rev e,
      List.rev n_i, true, true) }
138
139 single_node_exprs_list:
140   single_node_expr SEMI single_node_expr { merge_graph_exprs $1 $3 }
141 | single_node_expr SEMI single_node_exprs_list { merge_graph_exprs $1 $3 }
142
143 ugraph_exprs_list:
144   ugraph_expr { $1 }
145 | ugraph_exprs_list SEMI ugraph_expr { merge_graph_exprs $1 $3 }
146 | ugraph_exprs_list SEMI single_node_expr { merge_graph_exprs $1 $3 }
147 | single_node_expr SEMI ugraph_exprs_list { merge_graph_exprs $1 $3 }
148
149 digraph_exprs_list:
150   digraph_expr { $1 }
151 | digraph_exprs_list SEMI digraph_expr { merge_graph_exprs $1 $3 }
152 | digraph_exprs_list SEMI single_node_expr { merge_graph_exprs $1 $3 }
153 | single_node_expr SEMI digraph_exprs_list { merge_graph_exprs $1 $3 }
154
155 ugraph_expr:
156   single_node_expr EDGE ID { update_graph $1 $3 (Int_Lit(0)) }
157 | single_node_expr EDGE ID COLON expr { update_graph_e $1 $3 $5 (Int_Lit(0)) }
158 | ugraph_expr EDGE ID { update_graph $1 $3 (Int_Lit(0)) }
159 | ugraph_expr EDGE ID COLON expr { update_graph_e $1 $3 $5 (Int_Lit(0)) }
160
161 digraph_expr:
162   single_node_expr RARROW ID { update_digraph $1 $3 (Int_Lit(0)) 0 }
163 | single_node_expr LARROW ID { update_digraph $1 $3 (Int_Lit(0)) 1 }
164 | single_node_expr DIARROW ID { update_digraph_b $1 $3 (Int_Lit(0)) }
165 | single_node_expr RARROW ID COLON expr { update_digraph_e $1 $3 $5 (Int_Lit(0))
      0 }
166 | single_node_expr LARROW ID COLON expr { update_digraph_e $1 $3 $5 (Int_Lit(0))
      1 }
```

```
167 | single_node_expr DIARROW ID COLON expr { update_digraph_be $1 $3 $5 (Int_Lit(0)
      ) }
168 | digraph_expr RARROW ID { update_digraph $1 $3 (Int_Lit(0)) 0 }
169 | digraph_expr LARROW ID { update_digraph $1 $3 (Int_Lit(0)) 1 }
170 | digraph_expr DIARROW ID { update_digraph_b $1 $3 (Int_Lit(0)) }
171 | digraph_expr RARROW ID COLON expr { update_digraph_e $1 $3 $5 (Int_Lit(0)) 0 }
172 | digraph_expr LARROW ID COLON expr { update_digraph_e $1 $3 $5 (Int_Lit(0)) 1 }
173 | digraph_expr DIARROW ID COLON expr { update_digraph_be $1 $3 $5 (Int_Lit(0)) }
174
175 /* weighted graphs */
176 wegraph_exprs_list:
177   wegraph_expr { $1 }
178 | wegraph_exprs_list SEMI wegraph_expr { merge_graph_exprs $1 $3 }
179 | wegraph_exprs_list SEMI single_node_expr { merge_graph_exprs $1 $3 }
180 | single_node_expr SEMI wegraph_exprs_list { merge_graph_exprs $1 $3 }
181
182 wegraph_expr:
183   single_node_expr MINUS LBRACE expr RBRACE MINUS ID %prec EDGE { update_graph $1
       $7 $4 }
184 | single_node_expr MINUS LBRACE expr RBRACE MINUS ID COLON expr %prec EDGE {
      update_graph_e $1 $7 $9 $4 }
185 | wegraph_expr MINUS LBRACE expr RBRACE MINUS ID %prec EDGE { update_graph $1 $7
      $4 }
186 | wegraph_expr MINUS LBRACE expr RBRACE MINUS ID COLON expr %prec EDGE {
      update_graph_e $1 $7 $9 $4 }
187
188 wedigraph_exprs_list:
189   wedigraph_expr { $1 }
190 | wedigraph_exprs_list SEMI wedigraph_expr { merge_graph_exprs $1 $3 }
191 | wedigraph_exprs_list SEMI single_node_expr { merge_graph_exprs $1 $3 }
192 | single_node_expr SEMI wedigraph_exprs_list { merge_graph_exprs $1 $3 }
193
194 wedigraph_expr:
195   single_node_expr MINUS LBRACE expr RBRACE RARROW ID %prec EDGE { update_digraph
       $1 $7 $4 0 }
196 | single_node_expr LARROW LBRACE expr RBRACE MINUS ID %prec EDGE { update_digraph
       $1 $7 $4 1 }
197 | single_node_expr LARROW LBRACE expr RBRACE RARROW ID %prec EDGE {
      update_digraph_b $1 $7 $4 }
198 | single_node_expr MINUS LBRACE expr RBRACE RARROW ID COLON expr %prec EDGE {
      update_digraph_e $1 $7 $9 $4 0 }
199 | single_node_expr LARROW LBRACE expr RBRACE MINUS ID COLON expr %prec EDGE {
      update_digraph_e $1 $7 $9 $4 1 }
200 | single_node_expr LARROW LBRACE expr RBRACE RARROW ID COLON expr %prec EDGE {
      update_digraph_be $1 $7 $9 $4 }
201 | wedigraph_expr MINUS LBRACE expr RBRACE RARROW ID %prec EDGE { update_digraph
      $1 $7 $4 0 }
202 | wedigraph_expr LARROW LBRACE expr RBRACE MINUS ID %prec EDGE { update_digraph
      $1 $7 $4 1 }
203 | wedigraph_expr LARROW LBRACE expr RBRACE RARROW ID %prec EDGE {
      update_digraph_b $1 $7 $4 }
204 | wedigraph_expr MINUS LBRACE expr RBRACE RARROW ID COLON expr %prec EDGE {
      update_digraph_e $1 $7 $9 $4 0 }
205 | wedigraph_expr LARROW LBRACE expr RBRACE MINUS ID COLON expr %prec EDGE {
      update_digraph_e $1 $7 $9 $4 1 }
206 | wedigraph_expr LARROW LBRACE expr RBRACE RARROW ID COLON expr %prec EDGE {
      update_digraph_be $1 $7 $9 $4 }
```

```
207
208
209
210 single_node_expr:
211   ID { [$1], [], [] }
212 | ID COLON expr { [$1], [], [($1, $3)] }
213
214
215 expr_opt:
216     /* nothing */ { Noexpr }
217   | expr { $1 }
218
219
220 actuals_opt:
221     /* nothing */ { [] }
222   | actuals_list  { List.rev $1 }
223
224 actuals_list:
225     expr                    { [$1] }
226   | actuals_list COMMA expr { $3 :: $1 }
```

Parser: parser.mly

```
1  (* Authors:
2  Seth Benjamin sjb2190
3  Jennifer Bi jb3495
4  *)
5
6  let merge_graph_exprs (n1, e1, n_i1) (n2, e2, n_i2) =
7    (* essentially, take the union of node/edge/node_init lists. *)
8    let add_if_missing list elem = if (List.mem elem list) then
9        list
10     else
11       elem :: list
12   in (List.fold_left add_if_missing n1 (List.rev n2),
13       List.fold_left add_if_missing e1 (List.rev e2),
14       List.fold_left add_if_missing n_i1 (List.rev n_i2))
15
16 let update_graph graph n weight =  match graph with
17     (nodes, edges, nodes_init) ->
18     let nodes = if (List.mem n nodes) then (* if next node is already in this
       graph, *)
19         (n :: List.filter (fun x -> x <> n) nodes) (* move to front of nodelist
       so edges work *)
20       else
21         n :: nodes (* otherwise just add to front *)
22     and edges =
23       let new_edge = ((List.hd nodes), n, weight)
24       and new_edge_rev = (n, (List.hd nodes), weight) in
25       (* only add this edge if it's not already there *)
26       if (List.mem new_edge edges || List.mem new_edge_rev edges) then
27         edges
28       else if (new_edge_rev = new_edge) then (* check for self-loop *)
29         new_edge :: edges
30       else
31         new_edge_rev :: new_edge :: edges (* add in both directions for undir.
       graph *)
32     in (nodes, edges, nodes_init)
```

```
33
34 let update_graph_e graph n expr weight =  match graph with
35     (nodes, edges, nodes_init) ->
36     let nodes = if (List.mem n nodes) then (* if next node is already in this
    graph, *)
37         (n :: List.filter (fun x -> x <> n) nodes) (* move to front of nodelist
    so edges work *)
38       else
39         n :: nodes (* otherwise just add to front *)
40     and edges =
41       let new_edge = ((List.hd nodes), n, weight)
42       and new_edge_rev = (n, (List.hd nodes), weight) in
43       (* only add this edge if it's not already there *)
44       if (List.mem new_edge edges || List.mem new_edge_rev edges) then
45         edges
46       else if (new_edge_rev = new_edge) then (* check for self-loop *)
47         new_edge :: edges
48       else
49         new_edge_rev :: new_edge :: edges (* add in both directions for undir.
    graph *)
50     and nodes_init = (n, expr) :: nodes_init (* add node name/data pair to
    nodes_init *)
51     in (nodes, edges, nodes_init)
52
53
54 let update_digraph graph n weight l =  match graph with
55     (nodes, edges, nodes_init) ->
56     let nodes = if (List.mem n nodes) then (* if next node is already in this
    graph, *)
57         (n :: List.filter (fun x -> x <> n) nodes) (* move to front of nodelist
    so edges work *)
58       else
59         n :: nodes (* otherwise just add to front *)
60     and edges =
61       let new_edge =
62         if l==0 then
63           ((List.hd nodes), n, weight)
64         else
65           (n, List.hd nodes, weight) in
66       (* only add this edge if it's not already there *)
67       if (List.mem new_edge edges) then
68         edges
69       else
70         new_edge :: edges
71     in (nodes, edges, nodes_init)
72
73 let update_digraph_e graph n expr weight l =  match graph with
74     (nodes, edges, nodes_init) ->
75     let nodes = if (List.mem n nodes) then (* if next node is already in this
    graph, *)
76         (n :: List.filter (fun x -> x <> n) nodes) (* move to front of nodelist
    so edges work *)
77       else
78         n :: nodes (* otherwise just add to front *)
79     and edges =
80       let new_edge =
81         if l==0 then
```

```
82           ((List.hd nodes), n, weight)
83         else
84           (n, List.hd nodes, weight) in
85       (* only add this edge if it's not already there *)
86       if (List.mem new_edge edges) then
87         edges
88       else
89         new_edge :: edges
90     and nodes_init = (n, expr) :: nodes_init (* add node name/data pair to
    nodes_init *)
91     in (nodes, edges, nodes_init)
92
93 let update_digraph_b graph n weight = match graph with
94     (nodes, edges, nodes_init) ->
95     let nodes = if (List.mem n nodes) then (* if next node is already in this
    graph, *)
96         (n :: List.filter (fun x -> x <> n) nodes) (* move to front of nodelist
    so edges work *)
97       else
98         n :: nodes (* otherwise just add to front *)
99     and edges =
100       let new_edge = ((List.hd nodes), n, weight)
101       and new_edge_rev = (n, (List.hd nodes), weight) in
102       (* only add this edge if it's not already there *)
103       if (List.mem new_edge edges && List.mem new_edge_rev edges) then
104         edges
105       else if (List.mem new_edge edges) then
106         new_edge_rev :: edges
107       else if (List.mem new_edge_rev edges || new_edge_rev = new_edge) then
108         new_edge :: edges
109       else
110         new_edge :: new_edge_rev :: edges
111     in (nodes, edges, nodes_init)
112
113 let update_digraph_be graph n expr weight = match graph with
114     (nodes, edges, nodes_init) ->
115     let nodes = if (List.mem n nodes) then (* if next node is already in this
    graph, *)
116         (n :: List.filter (fun x -> x <> n) nodes) (* move to front of nodelist
    so edges work *)
117       else
118         n :: nodes (* otherwise just add to front *)
119     and edges =
120       let new_edge = ((List.hd nodes), n, weight)
121       and new_edge_rev = (n, (List.hd nodes), weight) in
122       (* only add this edge if it's not already there *)
123       if (List.mem new_edge edges && List.mem new_edge_rev edges) then
124         edges
125       else if (List.mem new_edge edges) then
126         new_edge_rev :: edges
127       else if (List.mem new_edge_rev edges || new_edge_rev = new_edge) then
128         new_edge :: edges
129       else
130         new_edge :: new_edge_rev :: edges
131     and nodes_init = (n, expr) :: nodes_init (* add node name/data pair to
    nodes_init *)
132     in (nodes, edges, nodes_init)
```

Parser: parser helper functions prshelper.ml

8.3 Semantic Checking

```
(* Authors:
Daniel Benett deb2174
Seth Benjamin sjb2190
Jennifer Bi jb3495
Jessie Liu jll2219
*)
type binop = Add | Sub | Mult | Div | Mod | Eq | Neq |
             Less | Leq | Greater | Geq | And | Or

type unop = Neg | Not

type typ = Int | Float | Bool | Void | String
         | Graph of typ
         | Digraph of typ
         | Wegraph of typ
         | Wedigraph of typ
         | Node of typ
         | Edge of typ
         | Wedge of typ
         | Diwedge of typ
         | Map of typ

type bind = typ * string

type expr =
    Id of string
  | Binop of expr * binop * expr
  | Unop of unop * expr
  | Assign of string * expr
  | Call of string * expr list
  | Method of expr * string * expr list
  | Bool_Lit of bool
  | Int_Lit of int
  | Float_Lit of float
  | String_Lit of string
  (* first bool is true if graph is directed; second bool is true if graph is
    weighted *)
  | Graph_Lit of string list * (string * string * expr) list * (string * expr)
    list * bool * bool
  | Noexpr

type stmt =
    Block of stmt list
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | For_Node of string * expr * stmt
  | For_Edge of string * expr * stmt
  | Bfs of string * expr * expr * stmt
  | Dfs of string * expr * expr * stmt
```

```
49    | Break
50    | Continue
51    | Expr of expr
52    | Vdecl of typ * string * expr
53    | Return of expr
54
55
56  type fdecl = {
57    f_typ : typ;
58    f_name : string;
59    f_formals : bind list;
60    f_body : stmt list;
61  }
62
63
64  type program = bind list * fdecl list
65
66  (* Pretty-printing functions *)
67
68  let string_of_op = function
69      Add -> "+"
70    | Sub -> "-"
71    | Mult -> "*"
72    | Div -> "/"
73    | Mod -> "%"
74    | Eq -> "=="
75    | Neq -> "!="
76    | Less -> "<"
77    | Leq -> "<="
78    | Greater -> ">"
79    | Geq -> ">="
80    | And -> "&&"
81    | Or -> "||"
82
83
84  let string_of_uop = function
85      Neg -> "-"
86    | Not -> "!"
87
88
89  let rec string_of_typ = function
90      Int -> "int"
91    | Float -> "float"
92    | Bool -> "bool"
93    | String -> "str"
94    | Node(t) -> "node<" ^ string_of_typ t ^ ">"
95    | Graph(t) -> "graph<" ^ string_of_typ t ^ ">"
96    | Digraph(t) -> "digraph<" ^ string_of_typ t ^ ">"
97    | Wegraph(t) -> "wegraph<" ^ string_of_typ t ^ ">"
98    | Wedigraph(t) -> "wedigraph<" ^ string_of_typ t ^ ">"
99    | Edge(t) -> "edge<" ^ string_of_typ t ^ ">"
100   | Wedge(t) -> "wedge<" ^ string_of_typ t ^ ">"
101   | Diwedge(t) -> "diwedge<" ^ string_of_typ t ^ ">"
102   | Map(t) -> "map<" ^ string_of_typ t ^ ">"
103   | Void -> "void"
104
105
```

```
106 let rec string_of_expr = function
107       Bool_Lit(true) -> "true"
108     | Bool_Lit(false) -> "false"
109     | Int_Lit(l) -> string_of_int l
110     | String_Lit(l) -> l
111     | Float_Lit(l) -> string_of_float l
112     | Id(s) -> s
113     | Binop(e1, o, e2) ->
114         string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
115     | Unop(o, e) -> string_of_uop o ^ string_of_expr e
116     | Assign(v, e) -> v ^ " = " ^ string_of_expr e
117     | Call(f, el) ->
118       f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
119     | Method(e, m, el) -> string_of_expr e ^ "." ^ m ^ "(" ^ String.concat ", " (
      List.map string_of_expr el) ^ ")"
120     | Graph_Lit(node_l, edge_l, node_init_l, _, _) ->
121       "[" ^ String.concat ", " node_l ^ "] " ^
122       "[" ^ String.concat ", " (List.map (fun(f,t,w) -> "(" ^ f ^ "," ^ t ^ "," ^
      string_of_expr w ^ ")") edge_l) ^ "] " ^
123       "[" ^ String.concat ", " (List.map (fun(n,e) -> "(" ^ n ^ "," ^
      string_of_expr e ^ ")") node_init_l) ^ "]"
124     | Noexpr -> ""
125
126 let rec string_of_stmt = function
127       Block(stmts) ->
128         "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
129     | Expr(expr) -> string_of_expr expr ^ ";\n";
130     | Vdecl(t, id, Noexpr) -> string_of_typ t ^ " " ^ id ^ ";\n"
131     | Vdecl(t, id, e) -> string_of_typ t ^ " " ^ string_of_expr e ^  ";\n"
132     | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
133     | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
134     | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
135         string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
136     | For(e1, e2, e3, s) ->
137         "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
138         string_of_expr e3  ^ ") " ^ string_of_stmt s
139     | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
140     | For_Node(n, g, sl) -> "for_node (" ^ n ^ " : " ^ string_of_expr g ^ ") " ^
      string_of_stmt sl
141     | For_Edge(e, g, sl) -> "for_edge (" ^ e ^ " : " ^ string_of_expr g ^ ") " ^
      string_of_stmt sl
142     | Bfs(n, g, src, sl) -> "bfs (" ^ n ^ " : " ^ string_of_expr g ^ " ; " ^
143                             string_of_expr src ^ ") " ^ string_of_stmt sl
144
145
146 let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"
147
148 let string_of_fdecl fdecl =
149   string_of_typ fdecl.f_typ ^ " " ^
150   fdecl.f_name ^ "(" ^ String.concat ", " (List.map snd fdecl.f_formals) ^
151   ")\n{\n" ^
152   String.concat "" (List.map string_of_stmt fdecl.f_body) ^
153   "}\n"
154
155 let string_of_program (vars, funcs) =
156   String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
157   String.concat "\n" (List.map string_of_fdecl funcs)
```

AST: ast.ml

```
(* Authors:
Seth Benjamin sjb2190
Jennifer Bi jb3495
Jessie Liu jll2219
*)

(* Semantically checked AST *)

open Ast

type sexpr =
    SId of string * typ
  | SBinop of sexpr * binop * sexpr * typ
  | SUnop of unop * sexpr * typ
  | SAssign of string * sexpr * typ
  | SMethod of sexpr * string * sexpr list * typ
  | SCall of string * sexpr list * typ
  | SBool_Lit of bool
  | SInt_Lit of int
  | SFloat_Lit of float
  | SString_Lit of string
  (* 1st typ is graph subtype (graph, digraph, wegraph, wedigraph), 2nd is the
    type of node data *)
  | SGraph_Lit of string list * (string * string * sexpr) list * (string * sexpr)
     list * typ * typ
  | SNoexpr

type svdecl = {
(* do we want this *)
  sv_name : string;
  sv_type : typ;
  sv_init : sexpr;
}

(* type expr_det = sexpr * typ *)
(* remove because will prob want to use sexpr in build
in codegen*)

type sstmt =
    SBlock of sstmt list
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt
  | SFor_Node of string * sexpr * sstmt
  | SFor_Edge of string * sexpr * sstmt
  | SBfs of string * sexpr * sexpr * sstmt
  | SDfs of string * sexpr * sexpr * sstmt
  | SBreak
  | SContinue
  | SExpr of sexpr * typ
  | SVdecl of typ * string * sexpr
  | SReturn of sexpr


```

```
53
54  type sfdecl = {
55    sf_typ : typ;
56    sf_name : string;
57    sf_formals : bind list;
58    sf_body : sstmt list;
59  }
60
61  type sprogram = bind list * sfdecl list
62
63
64  (* Pretty-printing functions *)
65
66  let rec string_of_sexpr = function
67      SBool_Lit(true) -> "true"
68    | SBool_Lit(false) -> "false"
69    | SInt_Lit(l) -> string_of_int l
70    | SString_Lit(l) -> l
71    | SFloat_Lit(l) -> string_of_float l
72    | SId(s, t) -> s ^ ":" ^ string_of_typ t
73    | SBinop(e1, o, e2, t) ->
74        string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^
75        string_of_sexpr e2 ^ ":" ^ string_of_typ t
76    | SUnop(o, e, t) -> string_of_uop o ^ string_of_sexpr e ^ ":" ^ string_of_typ t
77    | SAssign(v, e, t) -> v ^ " = " ^ string_of_sexpr e^ ":" ^ string_of_typ t
78    | SCall(f, el, t) ->
79        f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"^ ":" ^
     string_of_typ t
80    | SGraph_Lit(node_l, edge_l, node_init_l, g_typ, n_typ) ->
81      "[" ^ String.concat ", " node_l ^ "] " ^
82      "[" ^ String.concat ", " (List.map (fun(f,t,w) -> "(" ^ f ^ "," ^ t ^ "," ^
     string_of_sexpr w ^ ")") edge_l) ^ "] " ^
83      "[" ^ String.concat ", " (List.map (fun(n,e) -> "(" ^ n ^ "," ^
     string_of_sexpr e ^ ")") node_init_l) ^
84      "] : " ^ string_of_typ g_typ ^ ", " ^ string_of_typ n_typ
85    | SMethod(i, m, e, r_typ) -> string_of_sexpr i ^ "." ^ m ^ "(" ^ String.concat
     ", " (List.map string_of_sexpr e) ^ "):" ^ string_of_typ r_typ
86    | SNoexpr -> ""
87
88
89  let rec string_of_sstmt = function
90      SBlock(stmts) ->
91        "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
92    | SExpr(expr, t) -> string_of_sexpr expr ^ ";\n";
93    | SVdecl(t, id, SNoexpr) -> string_of_typ t ^ " " ^ id ^ ";\n"
94    | SVdecl(t, id, a_expr) -> string_of_typ t ^ " " ^ string_of_sexpr a_expr ^ ";\
     n"
95    | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
96    | SIf(e, s, SBlock([])) -> "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt
      s
97    | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
98        string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
99    | SFor(e1, e2, e3, s) ->
100       "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
101       string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
102   | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s
103   | SFor_Node(n, g, sl) -> "for_node (" ^ n ^ " : " ^ string_of_sexpr g ^ ") " ^
```

```
    string_of_sstmt sl
104   | SFor_Edge(e, g, sl) -> "for_edge (" ^ e ^ " : " ^ string_of_sexpr g ^ ") " ^
    string_of_sstmt sl
105   | SBfs(n, g, src, sl) -> "bfs (" ^ n ^ " : " ^ string_of_sexpr g ^ " ; " ^
106                            string_of_sexpr src ^ ") " ^ string_of_sstmt sl
107
108
109
110 let string_of_svdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"
111
112 let string_of_sfdecl fdecl =
113   string_of_typ fdecl.sf_typ ^ " " ^
114   fdecl.sf_name ^ "(" ^ String.concat ", " (List.map snd fdecl.sf_formals) ^
115   ")\n{\n" ^
116   String.concat "" (List.map string_of_sstmt fdecl.sf_body) ^
117   "}\n"
118
119 let string_of_sprogram (vars, funcs) =
120   String.concat "" (List.map string_of_svdecl vars) ^ "\n" ^
121   String.concat "\n" (List.map string_of_sfdecl funcs)
```

SAST

```
1  (* Authors:
2  Daniel Benett deb2174
3  Seth Benjamin sjb2190
4  Jennifer Bi jb3495
5  Jessie Liu jll2219
6  *)
7  open Ast
8  open Sast
9  open Exception
10
11 module StringMap = Map.Make(String)
12
13 type env = {
14     env_name : string; (* name of fn*)
15     env_return_type : typ; (* return type *)
16     env_fmap : fdecl StringMap.t; (* function map *)
17     env_sfmap : sfdecl StringMap.t;
18     env_globals : typ StringMap.t; (* global vars *)
19     env_flocals : typ StringMap.t; (* function locals *)
20     env_fformals : typ StringMap.t; (* function formals *)
21     env_in_loop : bool;
22     (* todo: built in methods? *)
23 }
24
25 let rec convert_expr e env = match e with
26     Id(str)                       -> (check_id str env, env)
27   | Binop(e1, op, e2)             -> (check_binop e1 op e2 env, env)
28   | Unop(op, e)                   -> (check_unop op e env, env)
29   | Assign(str, e)                -> (check_assign str e env)
30   | Method (e, "from", e_lst)     -> (check_edgemtd e "from" e_lst env)
31   | Method (e, "to", e_lst)     -> (check_edgemtd e "to" e_lst env)
32   | Method (e, "weight", e_lst)   -> (check_edgemtd e "weight" e_lst env)
33   | Method (e, "set_weight", e_lst)   -> (check_edgemtd e "set_weight" e_lst env)
34   | Method(e, "data", e_lst)     -> (check_data e e_lst env)
35   | Method (e, "set_data", e_lst) -> (check_sdata e e_lst env)
```

```
36    | Method (e, "print", e_lst) -> (check_graphmtd e "print" 0 e_lst Void env)
37    | Method (e, "add_node", e_lst) -> (check_graphmtd e "add_node" 1 e_lst Void
      env)
38    | Method (e, "remove_node", e_lst) -> (check_graphmtd e "remove_node" 1 e_lst
      Void env)
39    | Method (e, "has_node", e_lst) -> (check_graphmtd e "has_node" 1 e_lst Bool
      env)
40    | Method (e, "add_edge", [f;t]) -> (check_graphmtd e "add_edge" 2 [f;t] Void
      env)
41    | Method (e, "add_edge", [f;t;w]) -> (check_graphmtd e "add_edge" 3 [f;t;w]
      Void  env)
42    | Method (e, "remove_edge", e_lst) -> (check_graphmtd e "remove_edge" 2 e_lst
      Void env)
43    | Method (e, "has_edge", e_lst) -> (check_graphmtd e "has_edge" 2 e_lst Bool
      env)
44    | Method (e, "neighbors", e_lst) -> (check_graphmtd e "neighbors" 1 e_lst (
      Graph(Int)) env)
45    | Method (e, "get_edge_weight", e_lst) -> (check_graphmtd e "get_edge_weight" 2
       e_lst Int env)
46    | Method (e, "set_edge_weight", e_lst) -> (check_graphmtd e "set_edge_weight" 3
       e_lst Int env)
47    | Method (e, "put", e_lst) -> (check_mapmtd e "put" 2 e_lst env)
48    | Method (e, "get", e_lst) -> (check_mapmtd e "get" 1 e_lst env)
49    | Method (e, "contains", e_lst) -> (check_mapmtd e "contains" 1 e_lst env)
50    | Method (e, s2, e_lst)        -> (report_meth_not_found s2)
51    | Call("print", e_lst)          -> (check_print e_lst env)
52    | Call("printb", e_lst)         -> (check_print e_lst env)
53    | Call("prints", e_lst)         -> (check_print e_lst env)
54    | Call(str, e_lst)              -> (check_call str e_lst env)
55    | Int_Lit(i)                    -> (SInt_Lit(i), env)
56    | Float_Lit(f)                  -> (SFloat_Lit(f), env)
57    | String_Lit(str)               -> (SString_Lit(str), env)
58    | Bool_Lit(bool)                -> (SBool_Lit(bool), env)
59    | Graph_Lit(str_lst, ed_lst, n_lst, is_d, is_w) ->
60      (check_graph str_lst ed_lst n_lst is_d is_w env)
61    | Noexpr                        -> (SNoexpr, env)
62
63
64  and convert_expr_list expr_lst env =
65      match expr_lst with
66      [] -> [], env
67      | _ ->
68            let rec add_sexpr acc ex_lst env = match ex_lst with
69              [] -> acc, env
70              | e :: e_lst ->
71                    let sexpr, new_env = convert_expr e env in
72                    let new_acc = sexpr::acc in add_sexpr new_acc e_lst new_env
73            in
74            let sexpr_lst, nenv = add_sexpr [] expr_lst env in
75            let new_env = {
76              env_name = env.env_name;
77              env_return_type = env.env_return_type;
78              env_fmap = env.env_fmap;
79              env_sfmap = nenv.env_sfmap;
80              env_globals = env.env_globals;
81              env_flocals = env.env_flocals;
82              env_fformals = env.env_fformals;
```

```
83              env_in_loop = env.env_in_loop;
84              }
85      in
86      List.rev sexpr_lst, new_env
87
88
89  and get_sexpr_type sexpr = match sexpr with
90      SId(_, typ)                 -> typ
91    | SBinop(_, _, _, typ)        -> typ
92    | SUnop(_, _, typ)            -> typ
93    | SAssign(_, _, typ)          -> typ
94    | SCall(_, _, typ)            -> typ
95    | SMethod(_,_,_, typ)         -> typ
96    | SBool_Lit(_)                -> Bool
97    | SInt_Lit(_)                 -> Int
98    | SFloat_Lit(_)               -> Float
99    | SString_Lit(_)              -> String
100   | SGraph_Lit(_,_,_,subtype,_) -> subtype
101   | SNoexpr                     -> Void
102
103
104 and get_sexpr_lst_type sexpr_lst =
105     List.map (fun sexpr -> get_sexpr_type sexpr) sexpr_lst
106
107
108 (* s is the id being checked *)
109 and check_id str env =
110     try let typ = StringMap.find str env.env_flocals in
111         SId(str, typ) with Not_found ->
112     (try let typ = StringMap.find str env.env_fformals in
113         SId(str, typ) with Not_found ->
114     (try let typ = StringMap.find str env.env_globals in
115         SId(str, typ) with Not_found ->
116                 raise (Failure("undeclared identifier " ^ str))))
117
118 and check_id_typ str typ env =
119     if not (StringMap.mem str env.env_flocals || StringMap.mem str env.
    env_fformals
120             || StringMap.mem str env.env_globals)
121     then raise(Failure("node " ^ str ^ " is undefined"))
122     else
123       let lval = try StringMap.find str env.env_flocals with
124           Not_found -> (try StringMap.find str env.env_fformals with
125               Not_found -> StringMap.find str env.env_globals) in
126       if lval <> typ then raise(Failure("variable " ^ str ^ " is of type " ^
    string_of_typ lval ^ " when " ^ string_of_typ typ ^ " was expected"));
127
128
129 and check_binop e1 op e2 env =
130     let (s1, _) = convert_expr e1 env
131     and (s2, _) = convert_expr e2 env in
132     let t1 = get_sexpr_type s1 and t2 = get_sexpr_type s2 in
133     (* TODO add checking if types are graph!! *)
134     match op with
135         Add ->
136             (match t1, t2 with
137                 Int, Int        -> SBinop(s1, op, s2, Int)
```

```
138                 | Float, Float  -> SBinop(s1, op, s2, Float)
139                 (*TODO: string concat? *)
140                 | _             -> report_bad_binop t1 op t2
141             )
142         | Sub | Mult | Div
143             when t1 = t2 && (t1 = Int || t1 = Float) -> SBinop(s1, op, s2, t1)
144         | Mod
145             when t1 = Int && t2 = Int -> SBinop(s1, op, s2, t1)
146         | Eq | Neq ->
147             (match t1, t2 with
148                 Int, Int        -> SBinop(s1, op, s2, Bool)
149                 | Bool, Bool    -> SBinop(s1, op, s2, Bool)
150                 | Float, Float  -> SBinop(s1, op, s2, Bool)
151                 | Node(dt1), Node(dt2) when dt1 = dt2 -> SBinop(s1, op, s2, Bool)
152                 | _             -> report_bad_binop t1 op t2
153                  (* TODO: add string compare? *)
154             )
155         | Less | Leq | Greater | Geq
156             when t1 = t2 && (t1 = Int || t1 = Float) -> SBinop(s1, op, s2, Bool)
157         | And | Or
158             when t1 = Bool && t2 = Bool -> SBinop(s1, op, s2, Bool)
159                 | _             -> report_bad_binop t1 op t2
160
161
162 and check_unop op e env =
163     let (s, _) = convert_expr e env in
164     let t = get_sexpr_type s in
165     match op with
166         Neg when (t = Int || t = Float) -> SUnop(op, s, t)
167         | Not when t = Bool             -> SUnop(op, s, t)
168         | _ -> report_bad_unop op t
169
170
171 and check_print e_lst env =
172     if ((List.length e_lst) != 1) then raise(Failure("wrong number of arguments")
    )
173     else
174         let (s, nenv) = convert_expr (List.hd e_lst) env in
175         let t = get_sexpr_type s in
176         let strcall =
177             match t with
178                 Int         -> "print"
179                 | String    -> "prints"
180                 | Bool      -> "printb"
181                 | _         ->
182                     raise(Failure("type " ^ string_of_typ t ^ " is unsupported
    for this function"))
183         in
184         SCall(strcall, [s], Void), nenv
185
186
187 and check_assign str e env =
188     let (r, env) = convert_expr e env in
189     let rvaluet = get_sexpr_type r in
190
191     (* check if the id has been declared *)
192     if StringMap.mem str env.env_flocals || StringMap.mem str env.env_fformals ||
```

```
193         StringMap.mem str env.env_globals then
194         let lvaluet = try StringMap.find str env.env_flocals with
195             Not_found -> (try StringMap.find str env.env_fformals with
196                 Not_found -> StringMap.find str env.env_globals) in
197         (* if types match *)
198         if lvaluet = rvaluet then SAssign(str, r, lvaluet), env
199         else
200           (* The parser always says an edgeless graph literal is of type Graph,
201              but it is a valid rvalue for Digraph, Wegraph, and Wedigraph too -
202              if this is why lvaluet <> rvaluet, this is fine.
203              Similarly, semant always says the graph literal  is of type Graph<
    Void>,
204              but it is always a valid rvalue for any type of Graph with any data
    type. *)
205           (match (lvaluet, r) with
206              (Digraph(t1), SGraph_Lit (_, [], _, Graph(t2), _)) when t1 = t2 ->
    SAssign(str, r, lvaluet), env
207            | (Wegraph(t1), SGraph_Lit (_, [], _, Graph(t2), _)) when t1 = t2 ->
    SAssign(str, r, lvaluet), env
208            | (Wedigraph(t1), SGraph_Lit (_, [], _, Graph(t2), _)) when t1 = t2 ->
     SAssign(str, r, lvaluet), env
209            | (Graph(_), SGraph_Lit (_, [], _, Graph(Void), _))
210            | (Digraph(_), SGraph_Lit (_, [], _, Graph(Void), _))
211            | (Wegraph(_), SGraph_Lit (_, [], _, Graph(Void), _))
212            | (Wedigraph(_), SGraph_Lit (_, [], _, Graph(Void), _)) -> SAssign(str
    , r, lvaluet), env
213            | _ -> report_bad_assign lvaluet rvaluet
214           )
215     else report_undeclared_id_assign str
216
217
218 and check_data e e_lst env =
219     let len = List.length e_lst in
220     if (len != 0) then (raise(Failure("data takes 0 arguments but " ^
    string_of_int len ^ " arguments given")))
221     else
222       let (s,env) = convert_expr e env in
223       let t = get_sexpr_type s in
224       match t with
225         Node(dt) -> SMethod(s, "data", [], dt), env
226       | _ -> raise(Failure("data() called on type " ^ string_of_typ t ^ " when
    node was expected"))
227
228 and check_sdata e e_lst env =
229     let len = List.length e_lst in
230     if (len != 1) then raise(Failure("set_data() takes 1 arguments but " ^
    string_of_int len ^ " arguments given"))
231     else
232       let (id,env) = convert_expr e env in
233       (* TODO: GET NEW ENVIRONMENT HERE! like in check_mapmtd *)
234       let id_t = get_sexpr_type id in
235       let (arg, env) = convert_expr (List.hd e_lst) env in
236       let arg_t = get_sexpr_type arg in
237       match (id_t, arg_t) with
238         (Node(dt), at) when dt = at -> SMethod(id, "set_data", [arg], Void), env
239       | (Node(dt), at) -> raise(Failure("set_data() called on type " ^
    string_of_typ id_t ^ " with parameter " ^
```

```
240                                          string_of_typ at ^ " when " ^
    string_of_typ dt ^ " was expected"))
241       | (_, _) -> raise(Failure("set_data() called on type " ^ string_of_typ id_t
      ^ " when node was expected"))
242
243 and check_graphmtd g name args e_lst ret_typ env =
244     let id,env = convert_expr g env in
245     let t = get_sexpr_type id in
246     let data_type = match t with Graph(dt) -> dt
247                                | Digraph(dt) -> dt
248                                | Wegraph(dt) -> dt
249                                | Wedigraph(dt) -> dt
250                                | _ ->
251                               raise(Failure(name ^ " called on type " ^
    string_of_typ t ^ " when graph was expected")) in
252     let len = List.length e_lst in
253     if (len != args) then raise(Failure( name ^ " takes " ^ string_of_int args ^
    " arguments but " ^ string_of_int len ^ " arguments given"));
254
255     let sexpr,env =
256       match args with
257       0 (* print *) -> (match data_type with
258           | Int | Float | Bool | String -> (SMethod(id, name, [], ret_typ)), env
259           | _ -> raise(Failure("print cannot be called on graphs with generic
    data types")))
260     | 1 -> (
261           let e1 = (List.hd e_lst) in
262           let (ex,nenv) = convert_expr e1 env in
263           let t1 = get_sexpr_type ex in
264           if ( t1 <> Node(data_type) )
265           then raise(Failure("graph method " ^ name ^ " may not be called on
    graph of type " ^ string_of_typ t ^
266                                " with parameter " ^ string_of_typ t1));
267                 let new_env = {
268                   env_name = env.env_name;
269                   env_return_type = env.env_return_type;
270                   env_fmap = env.env_fmap;
271                   env_sfmap = nenv.env_sfmap;
272                   env_globals = env.env_globals;
273                   env_flocals = env.env_flocals;
274                   env_fformals = env.env_fformals;
275                   env_in_loop = env.env_in_loop;
276                 } in
277                 let ret_typ = if (ret_typ = Graph(Int)) then Graph(data_type) (*
    neighbors *)
278                   else ret_typ in
279           (SMethod(id, name, [ex], ret_typ)), new_env)
280
281     | 2 ->
282       let e1 = (List.hd e_lst) and e2 = (List.nth e_lst 1) in
283       let (ex,env1) = convert_expr e1 env in
284       let (ex2,env2) = convert_expr e2 env1 in
285       let t1 = get_sexpr_type ex and t2 = get_sexpr_type ex2 in
286       if ( t1 <> Node(data_type) || t2 <> Node(data_type) )
287       then raise(Failure("graph method " ^ name ^ " may not be called on graph of
     type " ^ string_of_typ t ^
288                         " with parameters " ^ string_of_typ t1 ^ ", " ^
```

```
    string_of_typ t2));
289       (* make sure this method can be called on this type *)
290       if (name = "get_edge_weight" && (t = Graph(data_type) || t = Digraph(
    data_type)))
291       then raise(Failure(name ^ " may not be called on unweighted graphs"));
292       if (name = "add_edge" && (t = Wegraph(data_type) || t = Wedigraph(data_type
    )))
293       then raise(Failure(name ^ " may not be called on weighted graphs without a
    weight argument"));
294               let new_env = {
295                 env_name = env.env_name;
296                 env_return_type = env.env_return_type;
297                 env_fmap = env.env_fmap;
298                 env_sfmap = env2.env_sfmap;
299                 env_globals = env.env_globals;
300                 env_flocals = env.env_flocals;
301                 env_fformals = env.env_fformals;
302                 env_in_loop = env.env_in_loop;
303             } in
304       (SMethod(id, name, [ex; ex2], ret_typ)), new_env
305     | 3 ->
306       let e1 = (List.hd e_lst)
307       and e2 = (List.nth e_lst 1)
308       and e3 = (List.nth e_lst 2) in
309       let (ex,env1) = convert_expr e1 env in
310       let (ex2,env2) = convert_expr e2 env1 in
311       let (ex3,env3) = convert_expr e3 env2 in
312       let t1 = get_sexpr_type ex
313       and t2 = get_sexpr_type ex2
314       and t3 = get_sexpr_type ex3 in
315       if ( t1 <> Node(data_type) || t2 <> Node(data_type) || t3 <> Int)
316       then raise(Failure("graph method " ^ name ^ " may not be called on graph of
     type " ^ string_of_typ t ^
317                          " with parameters " ^ string_of_typ t1 ^ ", " ^
    string_of_typ t2 ^ ", " ^ string_of_typ t3));
318       (* all 3-argument graph methods can only be called on we(di)graphs *)
319       if (t = Graph(data_type) || t = Digraph(data_type)) then
320         (if (name = "add_edge") then
321           raise(Failure(name ^ " may not be called on unweighted graphs with a
    weight argument"));
322          raise(Failure(name ^ " may not be called on unweighted graphs"));) ;
323               let new_env = {
324                 env_name = env.env_name;
325                 env_return_type = env.env_return_type;
326                 env_fmap = env.env_fmap;
327                 env_sfmap = env3.env_sfmap;
328                 env_globals = env.env_globals;
329                 env_flocals = env.env_flocals;
330                 env_fformals = env.env_fformals;
331                 env_in_loop = env.env_in_loop;
332             } in
333       (SMethod(id, name, [ex; ex2; ex3], ret_typ)), new_env
334   in sexpr, env
335
336
337 and check_edgemtd e n e_lst env =
338     let len = List.length e_lst in
```

```
339
340     let rec add_sexpr acc e_lst env = match e_lst with
341       [] -> acc, env
342       | e :: e_lst ->
343             let se, new_env = convert_expr e env in
344             let new_acc = se::acc in add_sexpr new_acc e_lst new_env
345     in
346     let e_lst_checked, nenv = add_sexpr [] e_lst env in
347     let new_env = {
348           env_name = env.env_name;
349           env_return_type = env.env_return_type;
350           env_fmap = env.env_fmap;
351           env_sfmap = nenv.env_sfmap;
352           env_globals = env.env_globals;
353           env_flocals = env.env_flocals;
354           env_fformals = env.env_fformals;
355           env_in_loop = env.env_in_loop;
356         }
357     in
358
359     let correct_len = match n with "set_weight" -> 1 | _ -> 0 in
360     if (len != correct_len) then
361       raise(Failure(n ^ " takes " ^ string_of_int correct_len ^ " arguments but "
      ^ string_of_int len ^ " arguments given")) else
362       let se, nenv = convert_expr e new_env in
363       let t = get_sexpr_type se in
364       let new_env = {
365               env_name = env.env_name;
366               env_return_type = env.env_return_type;
367               env_fmap = env.env_fmap;
368               env_sfmap = nenv.env_sfmap;
369               env_globals = env.env_globals;
370               env_flocals = env.env_flocals;
371               env_fformals = env.env_fformals;
372               env_in_loop = env.env_in_loop;
373       }
374       in
375       match t with
376         Diwedge(dt) | Wedge(dt) ->
377         (match n with
378            "from" -> SMethod(se, n, List.rev e_lst_checked, Node(dt)), new_env
379          | "to" -> SMethod(se, n, List.rev e_lst_checked, Node(dt)), new_env
380          | "weight" -> SMethod(se, n, List.rev e_lst_checked, Int), new_env
381          | "set_weight" -> SMethod(se, n, List.rev e_lst_checked, Void), new_env)
382       | Edge(dt) ->
383         (match n with
384            "from" -> SMethod(se, n, List.rev e_lst_checked, Node(dt)), new_env
385          | "to" -> SMethod(se, n, List.rev e_lst_checked, Node(dt)), new_env
386          | "weight" -> raise(Failure("weight() cannot be called on edges of
     unweighted graphs"));
387          | "set_weight" -> raise(Failure("set_weight() cannot be called on edges
     of unweighted graphs"));)
388       | _ -> raise(Failure("Edge method " ^ n ^ " called on type " ^
     string_of_typ t));
389
390 and check_mapmtd m name args e_lst env =
391   let id,env = convert_expr m env in
```

```
392    let t = get_sexpr_type id in
393    let value_typ = match t with
394      Map(v) -> v
395    | _ -> raise(Failure(name ^ " called on type " ^ string_of_typ t ^ " when map
         was expected")) in
396    let len = List.length e_lst in
397    if (len != args) then raise(Failure( name ^ " takes " ^ string_of_int args ^ "
         arguments but " ^ string_of_int len ^ " arguments given")) ;
398
399    let sexpr,env =
400      match args with
401        1 -> ( (* get(k), contains(k) *)
402          let e1 = (List.hd e_lst) in
403          let (ex,nenv) = convert_expr e1 env in
404          let t1 = get_sexpr_type ex in
405          (match t1 with
406            Node(dt) -> ()
407          | _ -> raise(Failure("map method " ^ name ^ " may not be called on type "
       ^ string_of_typ t1)));
408          let new_env = {
409            env_name = env.env_name;
410            env_return_type = env.env_return_type;
411            env_fmap = env.env_fmap;
412            env_sfmap = nenv.env_sfmap;
413            env_globals = env.env_globals;
414            env_flocals = env.env_flocals;
415            env_fformals = env.env_fformals;
416            env_in_loop = env.env_in_loop;
417          } in
418          let ret_typ = (match name with "contains" -> Bool | _ -> value_typ) in
419          (SMethod(id, name, [ex], ret_typ)), new_env)
420
421      | 2 -> (* put(k,v) *)
422        let e1 = (List.hd e_lst) and e2 = (List.nth e_lst 1) in
423        let (ex,env1) = convert_expr e1 env in
424        let (ex2,env2) = convert_expr e2 env1 in
425        let t1 = get_sexpr_type ex and t2 = get_sexpr_type ex2 in
426        (match t1 with
427            Node(dt) -> ()
428          | _ -> raise(Failure("map method " ^ name ^ " may not be called with key
     type "
429                           ^ string_of_typ t1)));
430        if ( t2 <> value_typ )
431        then raise(Failure("map method " ^ name ^ " called with value type "
432                           ^ string_of_typ t2 ^ " on map of type " ^ string_of_typ
     value_typ ));
433        let new_env = {
434          env_name = env.env_name;
435          env_return_type = env.env_return_type;
436          env_fmap = env.env_fmap;
437          env_sfmap = env2.env_sfmap;
438          env_globals = env.env_globals;
439          env_flocals = env.env_flocals;
440          env_fformals = env.env_fformals;
441          env_in_loop = env.env_in_loop;
442        } in
443        (SMethod(id, name, [ex; ex2], Void)), new_env
```

```
444     in sexpr, env
445 
446   (* TODO *)
447   and check_graph str_lst ed_lst n_lst is_d is_w env =
448       let graph_type data_type = match (is_d, is_w) with
449           (true, true) -> Wedigraph(data_type)
450         | (true, false) -> Digraph(data_type)
451         | (false, true) -> Wegraph(data_type)
452         | (false, false) -> Graph(data_type)
453       in
454       (* convert each weight expression *)
455       let ed_lst_checked = List.map (fun (f,t,w) -> (f, t, fst (convert_expr w env)
      )) ed_lst in
456       let weight_types = List.map (fun (_,_,w_sexpr) -> get_sexpr_type w_sexpr)
      ed_lst_checked in
457       List.iter (fun t -> if t <> Int then
458                   raise (Failure("edge weights must be of type int")))
      weight_types;
459       (* make sure no nodes are initialized more than once *)
460       ignore(List.fold_left (fun m (n, _)  -> if StringMap.mem n m then
461                               raise(Failure("graph literal cannot initialize the
      same node more than once"))
462                             else StringMap.add n true m) StringMap.empty n_lst);
463       (* make sure no edge appears more than once (may happen with we(di)graphs) *)
464       ignore(List.fold_left (fun m (f,t,_)  -> if StringMap.mem (f ^ "+" ^ t) m
      then
465                               raise(Failure("graph literal cannot feature the
      same edge with different weights"))
466                             else StringMap.add (f ^ "+" ^ t) true m) StringMap.
      empty ed_lst);
467       match str_lst with
468         [] -> SGraph_Lit([], [], [], Graph(Void), Void), env
469       | _ ->
470         let is_declared env str = (StringMap.mem str env.env_flocals ||
471                                    StringMap.mem str env.env_fformals ||
472                                    StringMap.mem str env.env_globals) in
473         let get_declared_type env str =
474           let node_type =
475             if (StringMap.mem str env.env_flocals) then
476               StringMap.find str env.env_flocals
477             else if (StringMap.mem str env.env_fformals) then
478               StringMap.find str env.env_fformals
479             else
480               StringMap.find str env.env_globals
481           in match node_type with Node(t) -> t | _ -> node_type
482         in
483         let declared = List.filter (is_declared env) str_lst in
484         let declared_types = List.map (get_declared_type env) declared in
485         let rec check_consistent types = (match types with
486               [] -> true
487             | [_] -> true
488             | hd :: tl -> (hd = List.hd tl) && (check_consistent tl))
489         in
490         if (not (check_consistent declared_types)) then
491           raise(Failure("all nodes in graph literal must have the same data type"))
      ;
492         (match (declared_types, n_lst) with
```

```
493          [], [] -> raise(Failure("graph literal must contain at least one
    previously " ^
494                                "declared node or at least one initialized node
    "));
495        | _, [] -> let t = List.hd declared_types in
496          let newenv = List.fold_left (fun x y -> let (_, z) = check_vdecl (Node(t
    )) y Noexpr true x in z) env str_lst in
497          SGraph_Lit(str_lst, ed_lst_checked, [], (graph_type t), t), newenv
498        | [], _ -> let (s,_) = convert_expr (snd (List.hd n_lst)) env in
499          let t = get_sexpr_type s in
500          List.iter (fun n -> let (sn,_) = convert_expr (snd n) env in
501                      let tn = get_sexpr_type sn in
502                      if tn <> t then raise (Failure("node type mismatch of " ^
    string_of_typ t ^ " and " ^ string_of_typ tn))) n_lst;
503          let newenv = List.fold_left (fun x y -> let (_, z) = check_vdecl (Node(t
    )) y Noexpr true x in z) env str_lst in
504          let nodes = List.map (fun (x,y) -> let (s,_) = convert_expr y newenv in
    (x,s)) n_lst
505          in
506          (SGraph_Lit(str_lst, ed_lst_checked, nodes, (graph_type t), t), newenv)
507        | _, _ ->
508          let n_lst_types = List.map (fun n -> (get_sexpr_type (fst (convert_expr
    (snd n) env)))) n_lst in
509          if (check_consistent (declared_types @ n_lst_types)) then
510            let t = List.hd declared_types in
511            let newenv = List.fold_left (fun x y -> let (_, z) = check_vdecl (Node
    (t)) y Noexpr true x in z) env str_lst in
512            let nodes = List.map (fun (x,y) -> let (s,_) = convert_expr y newenv
    in (x,s)) n_lst
513            in (SGraph_Lit(str_lst, ed_lst_checked, nodes, (graph_type t), t),
    newenv)
514          else
515            raise(Failure("all nodes in graph literal must have the same data type
    "));)
516
517 and check_call str e_lst env =
518     (* can't call main *)
519     if str == "main" then raise (Failure ("cant make call to main"))
520     (* check if function can be found*)
521     else if not (StringMap.mem str env.env_fmap) then report_function_not_found
    str
522     else
523     (* semantically check all the arguments*)
524     let checked_args, env = convert_expr_list e_lst env in
525
526     (* get the types of the args*)
527     let arg_types = get_sexpr_lst_type (checked_args) in
528
529
530     if not (StringMap.mem str env.env_sfmap) then
531         let fdecl = StringMap.find str env.env_fmap
532         in
533         let nenv =
534         {
535             env_name = env.env_name;
536             env_return_type = env.env_return_type;
537             env_fmap = env.env_fmap;
```

```
538             env_sfmap = (convert_fdecl fdecl.f_name fdecl.f_formals env).
    env_sfmap;
539             env_globals = env.env_globals;
540             env_flocals = env.env_flocals;
541             env_fformals = env.env_fformals;
542             env_in_loop = env.env_in_loop;
543         }
544         in
545 
546         let sfdecl = StringMap.find str nenv.env_sfmap
547         in try
548               (* confirm types match *)
549               List.iter2 (fun t1 (t2, _) -> if t1 <> t2 then report_typ_args t2
    t1 else ()) arg_types sfdecl.sf_formals;
550               let sexpr_lst, env = convert_expr_list e_lst env in
551                   SCall(str, sexpr_lst, sfdecl.sf_typ), nenv
552           with
553           (* wrong number of arguments *)
554               Invalid_argument _ -> raise (Failure ("expected " ^ string_of_int (
    List.length sfdecl.sf_formals) ^ "arguments when "
555               ^ string_of_int (List.length e_lst) ^ " arguments were provided"))
556     else
557           let sfdecl = StringMap.find str env.env_sfmap in
558           try
559               (* confirm types match *)
560               List.iter2 (fun t1 (t2, _) -> if t1 <> t2 then report_typ_args t2
    t1 else ()) arg_types sfdecl.sf_formals;
561               let sexpr_lst, env = convert_expr_list e_lst env in
562                   SCall(str, sexpr_lst, sfdecl.sf_typ), env
563           with
564           (* wrong number of arguments *)
565               Invalid_argument _ -> raise (Failure ("expected " ^ string_of_int (
    List.length sfdecl.sf_formals) ^ "arguments when "
566               ^ string_of_int (List.length e_lst) ^ " arguments were provided"))
567 
568 and convert_stmt stmt env = match stmt with
569   Block(s_lst)            -> (check_block s_lst env)
570   | If(e, s1, s2)         -> (check_if e s1 s2 env)
571   | For(e1, e2, e3, s)    -> (check_for e1 e2 e3 s env)
572   | While(e, s)           -> (check_while e s env)
573   | For_Node(str, e, s)   -> (check_for_node str e s env)
574   | For_Edge(str, e, s)   -> (check_for_edge str e s env)
575   | Bfs(str, e1, e2, s)   -> (check_bfs str e1 e2 s env)
576   | Dfs(str, e1, e2, s)   -> (check_dfs str e1 e2 s env)
577   | Break                 -> (check_break env, env)
578   | Continue              -> (check_continue env, env)
579   | Expr(e)               -> (check_expr_stmt e env)
580   | Vdecl(t, str, e)      -> (check_vdecl t str e false env)
581   | Return(e)             -> (check_return e env)
582 
583 
584 and check_block s_lst env =
585     match s_lst with
586     []    -> SBlock([SExpr(SNoexpr, Void)]), env
587     | _   -> (*check every statement, and put those checked statements in a list
    *)
588               let rec add_sstmt acc stmt_lst env = match stmt_lst with
```

```
589                 [] -> acc, env
590                 | Return _ :: _ :: _  -> raise (Failure("nothing may follow a
    return"))
591                 | st :: st_lst ->
592                       let sstmt, new_env = convert_stmt st env in
593                       let new_acc = sstmt::acc in add_sstmt new_acc st_lst
    new_env
594               in
595               let sblock, nenv = add_sstmt [] s_lst env in
596               let new_env = {
597                 env_name = env.env_name;
598                 env_return_type = env.env_return_type;
599                 env_fmap = env.env_fmap;
600                 env_sfmap = nenv.env_sfmap;
601                 env_globals = env.env_globals;
602                 env_flocals = env.env_flocals;
603                 env_fformals = env.env_fformals;
604                 env_in_loop = env.env_in_loop;
605             }
606             in
607             (SBlock(List.rev sblock), new_env)
608
609 and check_if cond is es env =
610     (* semantically check the condition *)
611     let scond,env = convert_expr cond env in
612         (match get_sexpr_type scond with
613             Bool ->
614                 let (sis, env) = convert_stmt is env in
615                 let (ses, env) = convert_stmt es env in
616                 SIf(scond, sis, ses), env
617             | _ -> raise(Failure("Expected boolean expression")))
618
619
620 and check_for e1 e2 e3 s env =
621     let (se1, env1) = convert_expr e1 env in (* a new var may be added to locals
    there*)
622     let (se2, env2) = convert_expr e2 env1 in
623     let (se3, env3) = convert_expr e3 env2 in
624     let new_env =
625
626     {
627       env_name = env3.env_name;
628       env_return_type = env3.env_return_type;
629       env_fmap = env3.env_fmap;
630       env_sfmap = env3.env_sfmap;
631       env_globals = env3.env_globals;
632       env_flocals = env3.env_flocals;
633       env_fformals = env3.env_fformals;
634       env_in_loop = true;
635     }
636     in
637
638     let (for_body, nenv) = convert_stmt s new_env in
639
640     (* check if you have a return in a for *)
641     match for_body with
642       SBlock(slst) ->
```

```
643             let rets = List.filter (fun x -> match x with
644                                         SReturn(expr)-> true
645                                         | _ -> false ) slst
646             in
647             if List.length rets != 0 then raise(Failure("cannot return in for loop
      "));
648 
649 
650       let nnenv =
651       {
652         env_name = env.env_name;
653         env_return_type = env.env_return_type;
654         env_fmap = env.env_fmap;
655         env_sfmap = nenv.env_sfmap;
656         env_globals = env.env_globals;
657         env_flocals = env.env_flocals;
658         env_fformals = env.env_fformals;
659         env_in_loop = env.env_in_loop;
660       }
661       in
662       if (get_sexpr_type se2) = Bool then
663           SFor(se1, se2, se3, for_body), nnenv
664       else raise(Failure("Expected boolean expression"))
665 
666 
667 and check_while e s env =
668       let (se, nenv) = convert_expr e env in
669       let new_env =
670       {
671         env_name = nenv.env_name;
672         env_return_type = nenv.env_return_type;
673         env_fmap = nenv.env_fmap;
674         env_sfmap = nenv.env_sfmap;
675         env_globals = nenv.env_globals;
676         env_flocals = nenv.env_flocals;
677         env_fformals = nenv.env_fformals;
678         env_in_loop = true;
679       }
680       in
681       let (while_body, nenv) = convert_stmt s new_env in
682 
683       let nnenv =
684       {
685         env_name = env.env_name;
686         env_return_type = env.env_return_type;
687         env_fmap = env.env_fmap;
688         env_sfmap = nenv.env_sfmap;
689         env_globals = env.env_globals;
690         env_flocals = env.env_flocals;
691         env_fformals = env.env_fformals;
692         env_in_loop = env.env_in_loop;
693       }
694       in
695       if (get_sexpr_type se) = Bool then
696           SWhile(se, while_body), nnenv
697       else raise(Failure("Expected boolean expression"))
698 
```

```
699 and check_for_node str e s env =
700     (* This is jank as hell but it's the last day so whatever *)
701     let graph_type = get_sexpr_type (fst (convert_expr e env)) in
702     let node_type = match graph_type with
703         Wedigraph(dt) | Wegraph(dt) | Graph(dt) | Digraph(dt) -> Node(dt)
704       | _ -> raise(Failure("type " ^ string_of_typ graph_type ^
705                           " may not be iterated with for_node")); (* TODO: write
      a test for this *)
706     in
707     let flocals = StringMap.add str node_type env.env_flocals in
708     let new_env =
709     {
710       env_name = env.env_name;
711       env_return_type = env.env_return_type;
712       env_fmap = env.env_fmap;
713       env_sfmap = env.env_sfmap;
714       env_globals = env.env_globals;
715       env_flocals = flocals;
716       env_fformals = env.env_fformals;
717       env_in_loop = env.env_in_loop;
718     }
719     in
720     let (se, senv) = convert_expr e new_env in
721     let gname = match se with (* cannot call the following methods on a NAMED
     graph*)
722          (* unnamed graph is safe since it cannot modify the graph itself *)
723           SId(str, Graph(_)) -> str
724         | SId(str, Digraph(_)) -> str
725         | SId(str, Wedigraph(_)) -> str
726         | SId(str, Wegraph(_)) -> str
727         | _ -> "" (*not sure if empty string is problematic *)
728     in
729     let (for_body, senv) = convert_stmt s senv in
730     let chk = match for_body with
731        SBlock(lst) -> match lst with
732                 [] -> ();
733                 | _ -> (* non empty statement lst, check em*)
734                 let chkcall x = match x with
735                 SExpr(SMethod(g,fname,_,_),_) -> (match g,fname with
736                         SId(s,_),"add_node" -> if s = gname then
     report_concurrent_mod "for_node"
737                         | SId(s,_),"remove_node" -> if s = gname then
     report_concurrent_mod "for_node"
738                         | _ -> ();)
739                 | _ -> ();
740                 in
741                 List.iter chkcall lst
742        | _ -> ()
743     in
744     let nenv =
745     {
746       env_name = env.env_name;
747       env_return_type = env.env_return_type;
748       env_fmap = env.env_fmap;
749       env_sfmap = senv.env_sfmap;
750       env_globals = env.env_globals;
751       env_flocals = env.env_flocals;
```

```
752         env_fformals = env.env_fformals;
753         env_in_loop = env.env_in_loop;
754       }
755       in
756       SFor_Node(str, se, for_body),nenv
757 
758 and check_for_edge str e s env =
759       let graph_type = get_sexpr_type (fst (convert_expr e env)) in
760       let edge_type = match graph_type with
761           Wedigraph(dt) -> Diwedge(dt)
762         | Wegraph(dt) -> Wedge(dt)
763         | Graph(dt) | Digraph(dt) -> Edge(dt)
764         | _ -> raise(Failure("type " ^ string_of_typ graph_type ^
765                               " may not be iterated with for_edge")); (* TODO: write
        a test for this *)
766       in
767       let flocals = StringMap.add str edge_type env.env_flocals in
768       let new_env =
769         {
770           env_name = env.env_name;
771           env_return_type = env.env_return_type;
772           env_fmap = env.env_fmap;
773           env_sfmap = env.env_sfmap;
774           env_globals = env.env_globals;
775           env_flocals = flocals;
776           env_fformals = env.env_fformals;
777           env_in_loop = env.env_in_loop;
778         }
779         in
780         let (se, senv) = convert_expr e new_env in
781         let gname = match se with (* cannot call the following methods on a NAMED
      graph*)
782                (* unnamed graph is safe since it cannot modify the graph itself *)
783                 SId(str, Graph(_)) -> str
784               | SId(str, Digraph(_)) -> str
785               | SId(str, Wedigraph(_)) -> str
786               | SId(str, Wegraph(_)) -> str
787               | _ -> "" (* not sure if empty string is problematic *)
788         in
789         let (for_body, senv) = convert_stmt s senv in
790         let chk = match for_body with
791              SBlock(lst) -> match lst with
792                       [] -> ();
793                       | _ -> (* non empty statement lst, check em *)
794                       let chkcall x = match x with
795                       SExpr(SMethod(g,fname,_,_),_) -> (match g,fname with
796                               SId(s,_),"add_edge" -> if s = gname then
      report_concurrent_mod "for_edge"
797                               | SId(s,_),"remove_edge" -> if s = gname then
      report_concurrent_mod "for_edge"
798                               | SId(s,_),"add_node" -> if s = gname then
      report_concurrent_mod "for_edge"
799                               | SId(s,_),"remove_node" -> if s = gname then
      report_concurrent_mod "for_edge"
800                               | _ -> ();)
801                       | _ -> ();
802                       in
```

```
803                     List.iter chkcall lst
804           | _ -> ()
805       in
806       let nenv =
807       {
808         env_name = env.env_name;
809         env_return_type = env.env_return_type;
810         env_fmap = env.env_fmap;
811         env_sfmap = senv.env_sfmap;
812         env_globals = env.env_globals;
813         env_flocals = env.env_flocals;
814         env_fformals = env.env_fformals;
815         env_in_loop = env.env_in_loop;
816       }
817       in
818       SFor_Edge(str, se, for_body),nenv
819 
820 
821 and check_bfs str e1 e2 s env =
822     let graph_type = get_sexpr_type (fst (convert_expr e1 env)) in
823     let node_type = match graph_type with
824         Wedigraph(dt) | Wegraph(dt) | Graph(dt) | Digraph(dt) -> Node(dt)
825       | _ -> raise(Failure("type " ^ string_of_typ graph_type ^
826                            " may not be iterated with bfs"));
827     in
828     let flocals = StringMap.add str node_type env.env_flocals in
829     let new_env =
830     {
831       env_name = env.env_name;
832       env_return_type = env.env_return_type;
833       env_fmap = env.env_fmap;
834       env_sfmap = env.env_sfmap;
835       env_globals = env.env_globals;
836       env_flocals = flocals;
837       env_fformals = env.env_fformals;
838       env_in_loop = env.env_in_loop;
839     }
840     in
841     let (se1, senv1) = convert_expr e1 new_env in
842     let (se2, senv2) = convert_expr e2 senv1 in
843     let source_type = get_sexpr_type se2 in
844     (match source_type with
845       Node(_) -> ()
846     | _ -> raise(Failure("source expr in bfs must be of type node")));
847     let gname = match se1 with (* cannot call the following methods on a NAMED
    graph*)
848           SId(str, Graph(_)) -> str
849         | SId(str, Digraph(_)) -> str
850         | SId(str, Wedigraph(_)) -> str
851         | SId(str, Wegraph(_)) -> str
852         | _ -> "" (*not sure if empty string is problematic *)
853     in
854     let (bfs_body, senv2) = convert_stmt s senv2 in
855     let chk = match bfs_body with
856        SBlock(lst) -> match lst with
857                  [] -> ();
858                  | _ -> (* non empty statement lst, check em*)
```

```
859                 let chkcall x = match x with
860                 SExpr(SMethod(g,fname,_,_),_) -> (match g,fname with
861                         SId(s,_),"add_edge" -> if s = gname then
    report_concurrent_mod "bfs"
862                         | SId(s,_),"remove_edge" -> if s = gname then
    report_concurrent_mod "bfs"
863                         | SId(s,_),"add_node" -> if s = gname then
    report_concurrent_mod "bfs"
864                         | SId(s,_),"remove_node" -> if s = gname then
    report_concurrent_mod "bfs"
865                         | _ -> ();)
866                 | _ -> ();
867                 in
868                 List.iter chkcall lst
869         | _ -> ()
870     in
871     let nenv =
872     {
873       env_name = env.env_name;
874       env_return_type = env.env_return_type;
875       env_fmap = env.env_fmap;
876       env_sfmap = senv2.env_sfmap;
877       env_globals = env.env_globals;
878       env_flocals = env.env_flocals;
879       env_fformals = env.env_fformals;
880       env_in_loop = env.env_in_loop;
881     }
882     in
883     SBfs(str, se1, se2, bfs_body), nenv
884
885
886 and check_dfs str e1 e2 s env =
887     let graph_type = get_sexpr_type (fst (convert_expr e1 env)) in
888     let node_type = match graph_type with
889         Wedigraph(dt) | Wegraph(dt) | Graph(dt) | Digraph(dt) -> Node(dt)
890       | _ -> raise(Failure("type " ^ string_of_typ graph_type ^
891                            " may not be iterated with dfs"));
892     in
893     let flocals = StringMap.add str node_type env.env_flocals in
894     let new_env =
895     {
896       env_name = env.env_name;
897       env_return_type = env.env_return_type;
898       env_fmap = env.env_fmap;
899       env_sfmap = env.env_sfmap;
900       env_globals = env.env_globals;
901       env_flocals = flocals;
902       env_fformals = env.env_fformals;
903       env_in_loop = env.env_in_loop;
904     }
905     in
906     let (se1, senv1) = convert_expr e1 new_env in
907     let (se2, senv2) = convert_expr e2 senv1 in
908     let source_type = get_sexpr_type se2 in
909     (match source_type with
910       Node(_) -> ()
911     | _ -> raise(Failure("source expr in dfs must be of type node")));
```

```
912     let gname = match se1 with (* cannot call the following methods on a NAMED
    graph*)
913            SId(str, Graph(_)) -> str
914          | SId(str, Digraph(_)) -> str
915          | SId(str, Wedigraph(_)) -> str
916          | SId(str, Wegraph(_)) -> str
917          | _ -> "" (*not sure if empty string is problematic *)
918     in
919     let (dfs_body, senv2) = convert_stmt s senv2 in
920     let chk = match dfs_body with
921        SBlock(lst) -> match lst with
922                  [] -> ();
923                  | _ -> (* non empty statement lst, check em*)
924                  let chkcall x = match x with
925                  SExpr(SMethod(g,fname,_,_),_) -> (match g,fname with
926                          SId(s,_),"add_edge" -> if s = gname then
    report_concurrent_mod "dfs"
927                          | SId(s,_),"remove_edge" -> if s = gname then
    report_concurrent_mod "dfs"
928                          | SId(s,_),"add_node" -> if s = gname then
    report_concurrent_mod "dfs"
929                          | SId(s,_),"remove_node" -> if s = gname then
    report_concurrent_mod "dfs"
930                          | _ -> ();)
931                  | _ -> ();
932                  in
933                  List.iter chkcall lst
934     in
935     let nenv =
936     {
937       env_name = env.env_name;
938       env_return_type = env.env_return_type;
939       env_fmap = env.env_fmap;
940       env_sfmap = senv2.env_sfmap;
941       env_globals = env.env_globals;
942       env_flocals = env.env_flocals;
943       env_fformals = env.env_fformals;
944       env_in_loop = env.env_in_loop;
945     }
946     in
947     SDfs(str, se1, se2, dfs_body), nenv
948
949
950 and check_break env =
951     if env.env_in_loop then
952         SBreak
953     else raise(Failure("can't break outside of a loop"))
954
955
956 and check_continue env =
957     if env.env_in_loop then
958         SContinue
959     else raise(Failure("can't continue outside of a loop"))
960
961
962 and check_expr_stmt e env =
963     let (se, nenv) = convert_expr e env in
```

```
964      let typ = get_sexpr_type se in (SExpr(se, typ), nenv)
965
966
967  and convert_fdecl fname fformals env =
968      let fdecl = StringMap.find fname env.env_fmap
969      in
970
971      report_duplicate (fun n -> match n with
972                  _, str -> "duplicate fformal " ^ str) fformals;
973
974      let formals_to_map m formal =
975        match formal with
976            (t, str) -> match t with
977              Void -> raise(Failure("cannot declare " ^ str ^ " as type void"))
978            | Map(Void) | Graph(Void) | Digraph(Void) | Wegraph(Void) | Wedigraph(
     Void) ->
979              raise(Failure("cannot have formal with type " ^ string_of_typ t))
980            | _ -> StringMap.add str t m
981      in
982
983      let formals = List.fold_left formals_to_map StringMap.empty fformals
984      in
985
986      let _ = match fdecl.f_typ with
987          Map(Void) | Graph(Void)  | Digraph(Void) | Wegraph(Void) | Wedigraph(Void
     ) ->
988          raise(Failure("cannot return " ^ string_of_typ fdecl.f_typ))
989        | _ -> ()
990      in
991
992      let env = {
993          env_name = fname;
994          env_return_type = fdecl.f_typ;
995          env_fmap = env.env_fmap;
996          env_sfmap = env.env_sfmap;
997          env_globals = env.env_globals;
998          env_flocals = StringMap.empty; (* locals should be empty at start check*)
999          env_fformals = formals;
1000         env_in_loop = env.env_in_loop
1001     }
1002     in
1003
1004     (* Semantically check all statements of the body *)
1005     let (sstmts, nenv ) = convert_stmt (Block fdecl.f_body) env
1006     in
1007     let check_ret = match sstmts with
1008         SBlock(lst) -> match lst with
1009         [] -> raise(Failure("missing return in " ^ fdecl.f_name));
1010         | _ ->
1011                 let rets = List.filter (fun x -> match x with
1012                         SReturn(expr)-> true
1013                         | _ -> false ) lst in
1014                 if List.length rets > 1 then
1015                 raise(Failure("misplaced return in " ^ fdecl.f_name))
1016                 else
1017                 match (List.nth lst ((List.length lst)-1)) with
1018                 SReturn(sexpr) -> ()
```

```
1019                 | _ -> raise(Failure("missing return in " ^ fdecl.f_name))
1020      in
1021     let sfdecl = {
1022         sf_typ = env.env_return_type;
1023         sf_name = fdecl.f_name;
1024         sf_formals = fformals; (*skips check? *)
1025         sf_body = match sstmts with SBlock(sl) -> sl | _ -> [] ;
1026     }
1027     in
1028     let new_env = {
1029         env_name = fname;
1030         env_return_type = fdecl.f_typ;
1031         env_fmap = env.env_fmap;
1032         env_sfmap = StringMap.add fname sfdecl nenv.env_sfmap;
1033         env_globals = env.env_globals;
1034         env_flocals = env.env_flocals;
1035         env_fformals = formals;
1036         env_in_loop = env.env_in_loop
1037     }
1038   in new_env
1039
1040
1041 and check_vdecl t str e from_graph_lit env =
1042     let t_is_node = match t with Node(_) -> true | _ -> false in
1043     if (StringMap.mem str env.env_flocals || StringMap.mem str env.env_fformals
     || StringMap.mem str env.env_globals)
1044     then
1045       (* if this vdecl is from a graph literal and we've already declared str
1046          as this type of node, this is fine - otherwise, reject *)
1047       if (from_graph_lit && t_is_node) then
1048          (if (StringMap.mem str env.env_flocals) then
1049             (if (StringMap.find str env.env_flocals <> t) then
1050                raise(Failure("cannot reinitialize existing variable")))
1051           else if (StringMap.mem str env.env_fformals) then
1052             (if (StringMap.find str env.env_fformals <> t) then
1053                raise(Failure("cannot reinitialize existing variable")))
1054           else if (StringMap.mem str env.env_globals) then
1055             (if (StringMap.find str env.env_globals <> t) then
1056                raise(Failure("cannot reinitialize existing variable")))
1057           else raise(Failure("cannot reinitialize existing variable")))
1058         else raise(Failure("cannot reinitialize existing variable"));
1059
1060     let _ = match t with
1061         Void -> raise(Failure("cannot declare " ^ str ^ " as type void"))
1062       | Map(Void) | Graph(Void) | Digraph(Void) | Wegraph(Void) | Wedigraph(Void)
      ->
1063         raise(Failure("cannot declare variable " ^ str ^ " with type " ^
     string_of_typ t))
1064       | _ -> ()
1065     in
1066
1067     let flocals = StringMap.add str t env.env_flocals in
1068     let new_env =
1069       {
1070         env_name = env.env_name;
1071         env_return_type = env.env_return_type;
1072         env_fmap = env.env_fmap;
```

```
1073          env_sfmap = env.env_sfmap;
1074          env_globals = env.env_globals;
1075          env_flocals = flocals;
1076          env_fformals = env.env_fformals;
1077          env_in_loop = env.env_in_loop;
1078        }
1079      in
1080
1081      let (se, nenv) = convert_expr e new_env
1082      in
1083      let typ = get_sexpr_type se
1084      in
1085      if typ <> Void then
1086        (if t <> typ then (raise(Failure("expression type mismatch " ^
       string_of_typ t ^ " and " ^ string_of_typ typ)))
1087        else (SVdecl(t, str, se), nenv))
1088      else
1089         (SVdecl(t, str, se), nenv)
1090
1091
1092  and check_return e env =
1093      let se, new_env = convert_expr e env in
1094      let typ = get_sexpr_type se in
1095      if typ = env.env_return_type
1096      then
1097      let nenv =
1098      {
1099        env_name = env.env_name;
1100        env_return_type = env.env_return_type;
1101        env_fmap = env.env_fmap;
1102        env_sfmap = new_env.env_sfmap;
1103        env_globals = env.env_globals;
1104        env_flocals = env.env_flocals;
1105        env_fformals = env.env_fformals;
1106        env_in_loop = env.env_in_loop;
1107      }
1108      in
1109      SReturn(se), nenv
1110      else raise(Failure("expected return type " ^ string_of_typ env.
       env_return_type
1111        ^ " but got return type " ^ string_of_typ typ))
1112
1113  let convert_ast globals fdecls fmap =
1114      let _ = try StringMap.find "main" fmap with
1115          Not_found -> raise(Failure("missing main"))
1116      in
1117
1118      report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd globals);
1119
1120      let convert_globals m global =
1121        match global with
1122        (typ, str) -> if (typ <> Void) then StringMap.add str typ m
1123        else raise(Failure("global " ^ str ^ " cannot have a void type"))
1124      in
1125
1126      (* semantically checked globals in a map *)
1127      let globals_map = List.fold_left convert_globals StringMap.empty globals
```

```
1128     in
1129
1130     (* check for duplicate functions *)
1131     report_duplicate (fun f -> "duplicate function " ^ f.f_name) fdecls;
1132
1133     (*List.iter (fun x -> convert_fdecl x x.f_name env) fdecls; *)
1134
1135
1136     let env = {
1137         env_name = "main";
1138         env_return_type = Int;
1139         env_fmap = fmap;
1140         env_sfmap = StringMap.empty;
1141         env_globals = globals_map;
1142         env_flocals = StringMap.empty;
1143         env_fformals = StringMap.empty;
1144         env_in_loop = false;
1145     }
1146     in
1147
1148     (* this is the environment with all the sfdecls, stemming from main *)
1149     let sfdecl_env = convert_fdecl "main" [] env in
1150     let sfdecls = List.rev(List.fold_left (fun lst (_, sfdecl) -> sfdecl :: lst)
1151                   [] (StringMap.bindings sfdecl_env.env_sfmap))
1152     in (globals, sfdecls)
1153
1154
1155 let build_fmap fdecls =
1156     (* built in *)
1157     let built_in_decls =  StringMap.add "print"
1158      { f_typ = Void; f_name = "print"; f_formals = [(Int, "x")];
1159        f_body = [] } (StringMap.add "printb"
1160      { f_typ = Void; f_name = "printb"; f_formals = [(Bool, "x")];
1161        f_body = [] } (StringMap.singleton "prints"
1162      { f_typ = Void; f_name = "prints"; f_formals = [(String, "x")];
1163        f_body = [] } ))
1164     in
1165
1166     let check_fdecls map fdecl =
1167         if StringMap.mem fdecl.f_name map then
1168             raise (Failure ("duplicate function " ^ fdecl.f_name))
1169         else if StringMap.mem fdecl.f_name built_in_decls then
1170             raise (Failure ("reserved function name " ^ fdecl.f_name))
1171         else StringMap.add fdecl.f_name fdecl map
1172     in
1173     List.fold_left (fun map fdecl -> check_fdecls map fdecl) built_in_decls
        fdecls
1174
1175
1176 let check ast = match ast with
1177     (globals, fdecls) ->
1178       let fmap = build_fmap fdecls in
1179       let sast = convert_ast globals fdecls fmap
1180       in
1181       sast
```

Semantic Checking: semant.ml

```
(* Authors:
Jennifer Bi jb3495
Jessie Liu jll2219
*)
open Ast
open Sast

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

and report_undeclared_id_assign str =
    raise (Failure ("assign to undeclared identifier " ^ str))

(* prob will want to change these to actual exceptions *)
and report_bad_binop t1 op t2 =
    raise (Failure ("illegal binary operator " ^
                string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                string_of_typ t2))

and report_bad_unop uop t =
    raise (Failure ("illegal unary operator " ^
                string_of_uop uop ^ " " ^ string_of_typ t))

and report_bad_assign lt rt =
    raise (Failure ("illegal assignment " ^ string_of_typ lt ^
                " = " ^ string_of_typ rt))

and check_not_void exceptf = function
    (Void, n) -> raise (Failure (exceptf n))
    | _ -> ()

and report_function_not_found func =
    raise (Failure ("function " ^ "not found"))

and report_num_args args provided =
    raise (Failure ("expected " ^ string_of_int args ^ "arguments when "
        ^ string_of_int provided ^ " arguments were provided"))

and report_typ_args ot nt =
    raise (Failure ("expected " ^ string_of_typ ot ^ " when "
        ^ string_of_typ nt ^ " type was provided"))

and report_meth_not_found s2 =
    raise (Failure ("method " ^ s2 ^ " not found"))
and report_concurrent_mod s =
    raise(Failure("concurrent modification of graph in " ^ s))
```

Semantic Checking: exception.ml

8.4 Code Generation

```
(* Authors:
Daniel Benett deb2174
Seth Benjamin sjb2190
Jennifer Bi jb3495
Jessie Liu jll2219
*)
module L = Llvm
module A = Ast
module S = Sast

module StringMap = Map.Make(String)

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

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

  let ltype_of_typ = function
      A.Int -> i32_t
    | A.Bool -> i1_t
    | A.Float -> float_t
    | A.String -> str_t
    | A.Node(_) -> void_ptr_t
    | A.Graph(_) -> void_ptr_t
    | A.Digraph(_) -> void_ptr_t
    | A.Wegraph(_) -> void_ptr_t
    | A.Wedigraph(_) -> void_ptr_t
    | A.Edge(_) -> void_ptr_t
    | A.Wedge(_) -> void_ptr_t
    | A.Diwedge(_) -> void_ptr_t
    | A.Map(_) -> void_ptr_t
    | A.Void -> void_t
    (* TODO: add wedge, handle generics *)
  in

  let get_sexpr_type sexpr = match sexpr with
      S.SId(_, typ)                 -> typ
    | S.SBinop(_, _, _, typ)        -> typ
    | S.SUnop(_, _, typ)            -> typ
    | S.SAssign(_, _, typ)          -> typ
    | S.SCall(_, _, typ)            -> typ
    | S.SMethod(_,_,_, typ)         -> typ
    | S.SBool_Lit(_)                -> Bool
    | S.SInt_Lit(_)                 -> Int
    | S.SFloat_Lit(_)               -> Float
    | S.SString_Lit(_)              -> String
```

```
55      | S.SGraph_Lit(_,_,_,subtype,_) -> subtype
56      | S.SNoexpr                      -> Void
57    in
58
59    (* Declare each global variable; remember its value in a map *)
60    let global_vars =
61      let global_var m (t, n) =
62        let init = L.const_int (ltype_of_typ t) 0
63        in StringMap.add n (L.define_global n init the_module) m in
64      List.fold_left global_var StringMap.empty globals in
65
66    (* Declare printf(), which the print built-in function will call *)
67    let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
68    let printf_func = L.declare_function "printf" printf_t the_module in
69
70    (* Declare functions that will be called to construct graphs *)
71    let new_graph_t = L.function_type void_ptr_t [||] in
72    let new_graph_func = L.declare_function "new_graph" new_graph_t the_module in
73
74    let add_vertex_t = L.function_type void_ptr_t [| void_ptr_t ; void_ptr_t |] in
75    let add_vertex_func = L.declare_function "add_vertex" add_vertex_t the_module
        in
76
77    let add_edge_t = L.function_type void_t [| void_ptr_t ; void_ptr_t |] in
78    let add_edge_func = L.declare_function "add_edge" add_edge_t the_module in
79
80    let add_wedge_t = L.function_type void_t [| void_ptr_t ; void_ptr_t ; i32_t |]
        in
81    let add_wedge_func = L.declare_function "add_wedge" add_wedge_t the_module in
82
83    let new_data_t = L.function_type void_ptr_t [||] in
84    let new_data_func = L.declare_function "new_data" new_data_t the_module in
85
86    let set_data_int_t = L.function_type void_t [| void_ptr_t ; i32_t |] in
87    let set_data_int_func = L.declare_function "set_data_int" set_data_int_t
        the_module in
88
89    let set_data_float_t = L.function_type void_t [| void_ptr_t ; float_t |] in
90    let set_data_float_func = L.declare_function "set_data_float" set_data_float_t
        the_module in
91
92    let set_data_char_ptr_t = L.function_type void_t [| void_ptr_t ; str_t |] in
93    let set_data_char_ptr_func = L.declare_function "set_data_char_ptr"
        set_data_char_ptr_t the_module in
94
95    let set_data_void_ptr_t = L.function_type void_t [| void_ptr_t ; void_ptr_t |]
        in
96    let set_data_void_ptr_func = L.declare_function "set_data_void_ptr"
        set_data_void_ptr_t the_module in
97
98    let get_data_int_t = L.function_type i32_t [| void_ptr_t |] in
99    let get_data_int_func = L.declare_function "get_data_int" get_data_int_t
        the_module in
100
101   let get_data_float_t = L.function_type float_t [| void_ptr_t |] in
102   let get_data_float_func = L.declare_function "get_data_float" get_data_float_t
        the_module in
```

```
103
104     let get_data_char_ptr_t = L.function_type str_t [| void_ptr_t |] in
105     let get_data_char_ptr_func = L.declare_function "get_data_char_ptr"
          get_data_char_ptr_t the_module in
106
107     let get_data_void_ptr_t = L.function_type void_ptr_t [| void_ptr_t |] in
108     let get_data_void_ptr_func = L.declare_function "get_data_void_ptr"
          get_data_void_ptr_t the_module in
109
110     (* Declare functions that will be called for for_node *)
111     let num_vertices_t = L.function_type i32_t [| void_ptr_t |] in
112     let num_vertices_func = L.declare_function "num_vertices" num_vertices_t
          the_module in
113
114     let get_head_vertex_t = L.function_type void_ptr_t [| void_ptr_t |] in
115     let get_head_vertex_func = L.declare_function "get_head_vertex"
          get_head_vertex_t the_module in
116
117     let get_next_vertex_t = L.function_type void_ptr_t [| void_ptr_t |] in
118     let get_next_vertex_func = L.declare_function "get_next_vertex"
          get_next_vertex_t the_module in
119
120     let get_data_from_vertex_t = L.function_type void_ptr_t [| void_ptr_t |] in
121     let get_data_from_vertex_func = L.declare_function "get_data_from_vertex"
          get_data_from_vertex_t the_module in
122
123     (* Declare functions corresponding to graph methods *)
124     let add_vertex_if_not_t = L.function_type void_t [| void_ptr_t ; void_ptr_t |]
          in
125     let add_vertex_if_not_func = L.declare_function "add_vertex_if_not_present"
          add_vertex_if_not_t the_module in
126
127     let remove_vertex_t = L.function_type void_t [| void_ptr_t ; void_ptr_t |] in
128     let remove_vertex_func = L.declare_function "remove_vertex" remove_vertex_t
          the_module in
129
130     let has_vertex_t = L.function_type i32_t [| void_ptr_t ; void_ptr_t |] in
131     let has_vertex_func = L.declare_function "has_vertex" has_vertex_t the_module
          in
132
133     let add_edge_method_t = L.function_type void_t [| void_ptr_t ; void_ptr_t ;
          void_ptr_t |] in
134     let add_edge_method_func = L.declare_function "add_edge_method"
          add_edge_method_t the_module in
135
136     let add_wedge_method_t = L.function_type void_t [| void_ptr_t ; void_ptr_t ;
          void_ptr_t ; i32_t |] in
137     let add_wedge_method_func = L.declare_function "add_wedge_method"
          add_wedge_method_t the_module in
138
139     let remove_edge_t = L.function_type void_t [| void_ptr_t ; void_ptr_t ;
          void_ptr_t |] in
140     let remove_edge_func = L.declare_function "remove_edge" remove_edge_t
          the_module in
141
142     let has_edge_t = L.function_type i32_t [| void_ptr_t ; void_ptr_t ; void_ptr_t
          |] in
```

```
143    let has_edge_func = L.declare_function "has_edge" has_edge_t the_module in
144
145    let graph_neighbors_t = L.function_type void_ptr_t [| void_ptr_t ; void_ptr_t
         |] in
146    let graph_neighbors_func = L.declare_function "graph_neighbors"
         graph_neighbors_t the_module in
147
148    let get_edge_weight_t = L.function_type i32_t [| void_ptr_t ; void_ptr_t ;
         void_ptr_t |] in
149    let get_edge_weight_func = L.declare_function "graph_get_edge_weight"
         get_edge_weight_t the_module in
150
151    let set_edge_weight_t = L.function_type void_t [| void_ptr_t ; void_ptr_t ;
         void_ptr_t ; i32_t |] in
152    let set_edge_weight_func = L.declare_function "graph_set_edge_weight"
         set_edge_weight_t the_module in
153
154    let set_undirected_edge_weight_t = L.function_type void_t [| void_ptr_t ;
         void_ptr_t ; void_ptr_t ; i32_t |] in
155    let set_undirected_edge_weight_func = L.declare_function "
         graph_set_undirected_edge_weight" set_undirected_edge_weight_t the_module in
156
157    let print_int_t = L.function_type void_t [| void_ptr_t |] in
158    let print_int_func = L.declare_function "print_int" print_int_t the_module in
159
160    let print_float_t = L.function_type void_t [| void_ptr_t |] in
161    let print_float_func = L.declare_function "print_float" print_float_t
         the_module in
162
163    let print_bool_t = L.function_type void_t [| void_ptr_t |] in
164    let print_bool_func = L.declare_function "print_bool" print_bool_t the_module
         in
165
166    let print_char_ptr_t = L.function_type void_t [| void_ptr_t |] in
167    let print_char_ptr_func = L.declare_function "print_char_ptr" print_char_ptr_t
         the_module in
168
169    let print_unweighted_int_t = L.function_type void_t [| void_ptr_t |] in
170    let print_unweighted_int_func = L.declare_function "print_unweighted_int"
         print_unweighted_int_t the_module in
171
172    let print_unweighted_float_t = L.function_type void_t [| void_ptr_t |] in
173    let print_unweighted_float_func = L.declare_function "print_unweighted_float"
         print_unweighted_float_t the_module in
174
175    let print_unweighted_char_ptr_t = L.function_type void_t [| void_ptr_t |] in
176    let print_unweighted_char_ptr_func = L.declare_function "
         print_unweighted_char_ptr" print_unweighted_char_ptr_t the_module in
177
178    let print_unweighted_bool_t = L.function_type void_t [| void_ptr_t |] in
179    let print_unweighted_bool_func = L.declare_function "print_unweighted_bool"
         print_unweighted_bool_t the_module in
180
181    (* Declare functions that will be called for bfs and dfs on graphs*)
182    let find_vertex_t = L.function_type void_ptr_t [| void_ptr_t ; void_ptr_t |] in
183    let find_vertex_func = L.declare_function "find_vertex" find_vertex_t
         the_module in
```

```
184
185   let get_visited_array_t = L.function_type void_ptr_t [| void_ptr_t |] in
186   let get_visited_array_func = L.declare_function "get_visited_array"
        get_visited_array_t the_module in
187
188   let get_bfs_queue_t = L.function_type void_ptr_t [| void_ptr_t ; void_ptr_t |]
        in
189   let get_bfs_queue_func = L.declare_function "get_bfs_queue" get_bfs_queue_t
        the_module in
190
191   let get_dfs_stack_t = L.function_type void_ptr_t [| void_ptr_t ; void_ptr_t |]
        in
192   let get_dfs_stack_func = L.declare_function "get_dfs_stack" get_dfs_stack_t
        the_module in
193
194   let get_next_bfs_vertex_t = L.function_type void_ptr_t [| void_ptr_t ;
        void_ptr_t |] in
195   let get_next_bfs_vertex_func = L.declare_function "get_next_bfs_vertex"
        get_next_bfs_vertex_t the_module in
196
197   let get_next_dfs_vertex_t = L.function_type void_ptr_t [| void_ptr_t ;
        void_ptr_t |] in
198   let get_next_dfs_vertex_func = L.declare_function "get_next_dfs_vertex"
        get_next_dfs_vertex_t the_module in
199
200   let bfs_done_t = L.function_type i32_t [| void_ptr_t |] in
201   let bfs_done_func = L.declare_function "bfs_done" bfs_done_t the_module in
202
203   let dfs_done_t = L.function_type i32_t [| void_ptr_t |] in
204   let dfs_done_func = L.declare_function "dfs_done" dfs_done_t the_module in
205
206   (* Declare functions that will be used for edge creation and for_edge *)
207   let edge_from_t = L.function_type void_ptr_t [| void_ptr_t |] in
208   let edge_from_func = L.declare_function "edge_from" edge_from_t the_module in
209
210   let edge_to_t = L.function_type void_ptr_t [| void_ptr_t |] in
211   let edge_to_func = L.declare_function "edge_to" edge_to_t the_module in
212
213   let edge_weight_t = L.function_type i32_t [| void_ptr_t |] in
214   let edge_weight_func = L.declare_function "edge_weight" edge_weight_t
        the_module in
215
216   let edge_set_weight_t = L.function_type void_t [| void_ptr_t ; i32_t |] in
217   let edge_set_weight_func = L.declare_function "edge_set_weight"
        edge_set_weight_t the_module in
218
219   let undirected_edge_set_weight_t = L.function_type void_t [| void_ptr_t ; i32_t
          |] in
220   let undirected_edge_set_weight_func = L.declare_function "
        undirected_edge_set_weight" undirected_edge_set_weight_t the_module in
221
222   let construct_edge_list_t = L.function_type void_ptr_t [| void_ptr_t |] in
223   let construct_edge_list_func = L.declare_function "construct_edge_list"
        construct_edge_list_t the_module in
224
225   let construct_undirected_edge_list_t = L.function_type void_ptr_t [| void_ptr_t
          |] in
```

```
226   let construct_undirected_edge_list_func = L.declare_function "
        construct_undirected_edge_list" construct_undirected_edge_list_t the_module
        in
227
228   let num_edges_t = L.function_type i32_t [| void_ptr_t |] in
229   let num_edges_func = L.declare_function "num_edges" num_edges_t the_module in
230
231   let get_next_edge_t = L.function_type void_ptr_t [| void_ptr_t |] in
232   let get_next_edge_func = L.declare_function "get_next_edge" get_next_edge_t
        the_module in
233
234   (* Declare functions that will be called for maps *)
235   let make_map_t = L.function_type void_ptr_t [||] in
236   let make_map_func = L.declare_function "make_map" make_map_t the_module in
237
238   let map_contains_t = L.function_type i32_t [| void_ptr_t ; void_ptr_t |] in
239   let map_contains_func = L.declare_function "contains_key" map_contains_t
        the_module in
240
241   let map_put_void_ptr_t = L.function_type void_t [| void_ptr_t ; void_ptr_t ;
        void_ptr_t |] in
242   let map_put_void_ptr_func = L.declare_function "put" map_put_void_ptr_t
        the_module in
243
244   let map_put_int_t = L.function_type void_t [| void_ptr_t ; void_ptr_t ; i32_t
        |] in
245   let map_put_int_func = L.declare_function "put_int" map_put_int_t the_module in
246
247   (* TODO: remove when nodes become generic *)
248   (*let map_put_int_ptr_t = L.function_type void_t [| void_ptr_t ; void_ptr_t ;
        i32_ptr_t |] in
249   let map_put_int_ptr_func = L.declare_function "put_int_ptr" map_put_int_ptr_t
        the_module in*)
250
251   let map_put_char_ptr_t = L.function_type void_t [| void_ptr_t ; void_ptr_t ;
        str_t |] in
252   let map_put_char_ptr_func = L.declare_function "put_char_ptr"
        map_put_char_ptr_t the_module in
253
254   let map_put_float_t = L.function_type void_t [| void_ptr_t ; void_ptr_t ;
        float_t |] in
255   let map_put_float_func = L.declare_function "put_float" map_put_float_t
        the_module in
256
257   let map_get_void_ptr_t = L.function_type void_ptr_t [| void_ptr_t ; void_ptr_t
        |] in
258   let map_get_void_ptr_func = L.declare_function "get" map_get_void_ptr_t
        the_module in
259
260   let map_get_int_t = L.function_type i32_t [| void_ptr_t ; void_ptr_t |] in
261   let map_get_int_func = L.declare_function "get_int" map_get_int_t the_module in
262
263   (* TODO: remove when nodes become generic *)
264   (*let map_get_int_ptr_t = L.function_type i32_ptr_t [| void_ptr_t ; void_ptr_t
        |] in
265   let map_get_int_ptr_func = L.declare_function "get_int_ptr" map_get_int_ptr_t
        the_module in*)
```

```
266
267    let map_get_char_ptr_t = L.function_type str_t [| void_ptr_t ; void_ptr_t |] in
268    let map_get_char_ptr_func = L.declare_function "get_char_ptr"
         map_get_char_ptr_t the_module in
269
270    let map_get_float_t = L.function_type float_t [| void_ptr_t ; void_ptr_t |] in
271    let map_get_float_func = L.declare_function "get_float" map_get_float_t
         the_module in
272
273
274
275
276    let function_decls =
277      let function_decl m fdecl =
278        let name = fdecl.S.sf_name
279        and formal_types =
280          Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.S.sf_formals)
281        in let ftype = L.function_type (ltype_of_typ fdecl.S.sf_typ) formal_types
         in
282        StringMap.add name (L.define_function name ftype the_module, fdecl) m in
283      List.fold_left function_decl StringMap.empty functions in
284
285    (* Fill in the body of the given function *)
286    let build_function_body fdecl =
287      let (the_function, _) = StringMap.find fdecl.S.sf_name function_decls in
288      let builder = L.builder_at_end context (L.entry_block the_function) in
289
290      let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder and
291        string_format_str = L.build_global_stringptr "%s\n" "fmt" builder in
292
293      (* Allocate formal arguments on the stack, initialize their value, and
294         remember their values in a map also containing all global vars. *)
295      let globals_and_formals =
296        let add_formal m (t, n) p = L.set_value_name n p;
297          let local = L.build_alloca (ltype_of_typ t) n builder in
298          ignore (L.build_store p local builder);
299          StringMap.add n local m
300        in
301        List.fold_left2 add_formal global_vars fdecl.S.sf_formals
302          (Array.to_list (L.params the_function))
303      in
304
305      (* Construct the locally declared variables for a block. Allocate each on the
306         stack, initialize their value, if appropriate, and remember their values
         in
307         the map m, passed as an argument. This is called every time A.Block is
         processed,
308         allowing every bracketed block to have its own scope. *)
309      let add_local_vars m builder sl =
310        (* When initializing graphs with graph literals, nodes do not have to be
311          declared; e.g. "graph g = [A -- B]" implicitly declares nodes A and B (
         unless
312          they have been already declared explicitly or in a previous graph). Thus
         ,
313          whenever we encounter a graph literal, we have to construct all the new
         nodes
314          it uses as local variables. The following function handles this. *)
```

```
315       let rec add_nodes_from_graph_lits m expr = match expr with
316           S.SGraph_Lit(nodes, edges, _, _, _) ->
317           let add_node m node =
318             if (StringMap.mem node m) then
319               m
320             else
321               let local_node_var = L.build_alloca (ltype_of_typ (A.Node(A.Int)))
    node builder in
322               let new_data_ptr = L.build_call new_data_func [||] "tmp_data"
    builder in
323               ignore(L.build_store new_data_ptr local_node_var builder);
324               StringMap.add node local_node_var m
325           in
326           List.fold_left add_node m nodes
327         (* for any sexpr containing sub-sexprs, recursively call on those *)
328         | S.SAssign(_, e, _) -> add_nodes_from_graph_lits m e
329         | S.SMethod(e, _, el, _) ->
330           List.fold_left (fun mp ex -> add_nodes_from_graph_lits mp ex) (
    add_nodes_from_graph_lits m e) el
331         | S.SCall(_, el, _) -> List.fold_left add_nodes_from_graph_lits m el
332         (* these shouldn't be able to have graph lits in them, but just for
    completion: *)
333         | S.SUnop(_, e, _) -> add_nodes_from_graph_lits m e
334         | S.SBinop(e1, _, e2, _) -> List.fold_left add_nodes_from_graph_lits m [
    e1; e2]
335         | _ -> m (* ignore expressions without subexpressions *)
336       in
337
338       (* find all local variables declared in block *)
339       (* TODO: does this need to be recursive?
340          see: if (true) printb([A].has_node(A) vs if (true) {printb([A].has_node(
    A))};
341          either make recursive, or make [A].has_node(A) not work - probably the
    second *)
342       let add_local m stmt = match stmt with
343           S.SVdecl(t, n, e) ->
344           (* if e contains a graph literal, adds new nodes to m; else m unchanged
     *)
345           let m = add_nodes_from_graph_lits m e in
346           let local_var = L.build_alloca (ltype_of_typ t) n builder in
347           (* if we're declaring a node (and not immediately initializing it to
    another node)
348              we need to call new_data() from C to get a unique data pointer and
    store it in
349              the allocated register *)
350           (match t with
351              A.Node(_) -> if e == S.SNoexpr then
352                let new_data_ptr = L.build_call new_data_func [||] "tmp_data"
    builder in
353                ignore(L.build_store new_data_ptr local_var builder);
354              else ()
355
356             (* Same thing if we declare a map: call make_map() to construct the
    map *)
357             | A.Map(_) -> if e == S.SNoexpr then
358                let map_ptr = L.build_call make_map_func [||] "tmp_map" builder in
359                ignore(L.build_store map_ptr local_var builder);
```

```
360             else ()
361
362           | _ -> ());
363           (* add new variable to m *)
363           StringMap.add n local_var m
365       (* non-vdecl stmts might have graph literals, so match on those to get
      them *)
366       | S.SExpr(e, _)
367       | S.SIf(e, _, _)
368       | S.SWhile(e, _)
369       | S.SFor_Node(_, e, _)
370       | S.SFor_Edge(_, e, _)
371       | S.SReturn(e) -> add_nodes_from_graph_lits m e
372       | S.SBfs(_, e1, e2, _)
373       | S.SDfs(_, e1, e2, _) -> List.fold_left add_nodes_from_graph_lits m [e1;
       e2]
374       | S.SFor(e1, e2, e3, _) -> List.fold_left add_nodes_from_graph_lits m [e1
      ; e2; e3]
375       | _ -> m
376     in
377     List.fold_left add_local m sl (* return value of add_local_vars *)
378   in
379
380   (* Given a symbol table "vars", return the value for a variable
381      or formal argument in the table *)
382   let lookup vars n = StringMap.find n vars in
383
384   (* Construct code for an expression; return its value *)
385   let rec expr vars builder = function
386       S.SInt_Lit i -> L.const_int i32_t i
387     | S.SBool_Lit b -> L.const_int i1_t (if b then 1 else 0)
388     | S.SNoexpr -> L.const_int i32_t 0
389     | S.SId (s,_) -> L.build_load (lookup vars s) s builder
390     | S.SString_Lit s -> L.build_global_stringptr s "str" builder
391     | S.SFloat_Lit f -> L.const_float float_t f
392     | S.SBinop (e1, op, e2, _) ->
393       let e1' = expr vars builder e1
394       and e2' = expr vars builder e2 in
395       let e_type = get_sexpr_type e1 in
396       (match e_type with
397        | A.Float -> (match op with
398              A.Add     -> L.build_fadd
399            | A.Sub     -> L.build_fsub
400            | A.Mult    -> L.build_fmul
401            | A.Div     -> L.build_fdiv
402            | A.Mod     -> L.build_frem (* TODO: look what this actually means
      *)
403            | A.And     -> L.build_and
404            | A.Or      -> L.build_or
405            | A.Eq      -> L.build_fcmp L.Fcmp.Oeq
406            | A.Neq     -> L.build_fcmp L.Fcmp.One
407            | A.Less    -> L.build_fcmp L.Fcmp.Olt
408            | A.Leq     -> L.build_fcmp L.Fcmp.Ole
409            | A.Greater -> L.build_fcmp L.Fcmp.Ogt
410            | A.Geq     -> L.build_fcmp L.Fcmp.Oge) e1' e2' "tmp" builder
411        | _ -> (match op with
412              A.Add     -> L.build_add
```

```
413               | A.Sub      -> L.build_sub
414               | A.Mult     -> L.build_mul
415               | A.Div      -> L.build_sdiv
416               | A.Mod      -> L.build_srem
417               | A.And      -> L.build_and
418               | A.Or       -> L.build_or
419               | A.Eq       -> L.build_icmp L.Icmp.Eq
420               | A.Neq      -> L.build_icmp L.Icmp.Ne
421               | A.Less     -> L.build_icmp L.Icmp.Slt
422               | A.Leq      -> L.build_icmp L.Icmp.Sle
423               | A.Greater -> L.build_icmp L.Icmp.Sgt
424               | A.Geq      -> L.build_icmp L.Icmp.Sge) e1' e2' "tmp" builder)
425        | S.SUnop(op, e, _) ->
426          let e' = expr vars builder e in
427          (match op with
428             A.Neg      -> if ((get_sexpr_type e) = A.Float) then L.build_fneg else
       L.build_neg
429           | A.Not      -> L.build_not) e' "tmp" builder
430        | S.SAssign(id, e, _) -> let e' = expr vars builder e in
431          ignore (L.build_store e' (lookup vars id) builder); e'
432        | S.SGraph_Lit (nodes, edges, nodes_init, graph_subtyp, _) -> (* TODO: use
       the last field for generics *)
433          (* create new graph struct, return pointer *)
434          let g = L.build_call new_graph_func [||] "tmp" builder in
435          (* map node names to vertex_list_node pointers created by calling
       add_vertex *)
436          let get_data_ptr node = L.build_load (lookup vars node) node builder in
437          let call_add_vertex node = L.build_call add_vertex_func [| g ; (
       get_data_ptr node) |] ("vertex_struct_" ^ node) builder in
438          let nodes_map = List.fold_left (fun map node -> StringMap.add node (
       call_add_vertex node) map) StringMap.empty nodes in
439          (* add edge *)
440          let add_edge n1 n2 =
441            L.build_call add_edge_func [| (StringMap.find n1 nodes_map) ; (
       StringMap.find n2 nodes_map) |] "" builder
442          and add_wedge n1 n2 w =
443            L.build_call add_wedge_func [| (StringMap.find n1 nodes_map) ; (
       StringMap.find n2 nodes_map) ; (expr vars builder w) |] "" builder
444          in ignore(match graph_subtyp with
445               A.Wegraph(_) | A.Wedigraph(_) -> List.map (fun (n1, n2, w) ->
       add_wedge n1 n2 w) edges
446             | _ (* unweighted graphs *) -> List.map (fun (n1, n2, _) -> add_edge
       n1 n2) edges);
447          (* initialize nodes with data *)
448          let data_type = (match graph_subtyp with A.Graph(t) | A.Digraph(t) | A.
       Wegraph(t) | A.Wedigraph(t) -> t
449                                               | _ -> A.Void) in
450          let set_data type_func (node, data)  =
451            L.build_call type_func [| (get_data_ptr node) ; (expr vars builder data
       ) |] "" builder in
452          let set_data_bool (node, data) =
453            let data_bool = L.build_intcast (expr vars builder data) i32_t "
       tmp_intcast" builder in
454            L.build_call set_data_int_func [| (get_data_ptr node) ; data_bool |] ""
        builder
455          in
456
```

```
457         let set_all_data = (match data_type with
458               A.Int -> List.map (set_data set_data_int_func) nodes_init
459             | A.Bool -> List.map set_data_bool nodes_init
460             | A.Float -> List.map (set_data set_data_float_func) nodes_init
461             | A.String -> List.map (set_data set_data_char_ptr_func) nodes_init
462             | _ -> List.map (set_data set_data_void_ptr_func) nodes_init)
463         in
464         if (data_type <> A.Void) then ignore(set_all_data);
465         (* return pointer to graph struct *)
466         g
467       | S.SCall ("print", [e], _) | S.SCall ("printb", [e], _) ->
468         L.build_call printf_func [| int_format_str ; (expr vars builder e) |]
469           "printf" builder
470       | S.SCall ("prints", [e], _) ->
471         L.build_call printf_func [| string_format_str ; (expr vars builder e) |]
472           "prints" builder
473       | S.SCall (f, act, _) ->
474         let (fdef, fdecl) = StringMap.find f function_decls in
475         let actuals = List.rev (List.map (expr vars builder) (List.rev act)) in
476         let result = (match fdecl.S.sf_typ with A.Void -> ""
477                                                | _ -> f ^ "_result") in
478         L.build_call fdef (Array.of_list actuals) result builder
479 
480       (* node methods *)
481       | S.SMethod (node_expr, "data", [], _) ->
482         let data_ptr = expr vars builder node_expr in
483         let data_type = (match (get_sexpr_type node_expr) with A.Node(t) -> t) in
484         let get_data_func = (match data_type with
485               A.Int | A.Bool -> get_data_int_func
486             | A.Float -> get_data_float_func
487             | A.String -> get_data_char_ptr_func
488             | _ -> get_data_void_ptr_func) in
489         let ret = L.build_call get_data_func [| data_ptr |] "tmp_data" builder in
490         if (data_type = A.Bool) then
491           (L.build_icmp L.Icmp.Ne ret (L.const_int i32_t 0) "tmp_booldata"
    builder)
492         else
493           ret
494       | S.SMethod (node_expr, "set_data", [data_expr], _) ->
495         let data_ptr = expr vars builder node_expr in
496         let data_type = (match (get_sexpr_type node_expr) with A.Node(t) -> t) in
497         let new_data = expr vars builder data_expr in
498         let new_data = if (data_type = A.Bool) then
499             L.build_intcast new_data i32_t "tmp_intcast" builder
500           else
501             new_data
502         in
503         let set_data_func = (match data_type with
504               A.Int | A.Bool -> set_data_int_func
505             | A.Float -> set_data_float_func
506             | A.String -> set_data_char_ptr_func
507             | _ -> set_data_void_ptr_func) in
508         L.build_call set_data_func [| data_ptr ; new_data |] "" builder
509 
510       (* edge methods *)
511       | S.SMethod (edge_expr, "from", [], _) ->
512         let data_ptr = expr vars builder edge_expr in
```

```
513         L.build_call edge_from_func [| data_ptr |] "tmp_edge_from" builder
514       | S.SMethod (edge_expr, "to", [], _) ->
515         let data_ptr = expr vars builder edge_expr in
516         L.build_call edge_to_func [| data_ptr |] "tmp_edge_to" builder
517       | S.SMethod (edge_expr, "weight", [], _) -> (* TODO: add sem. check so
    these can only be done on wedges *)
518         let data_ptr = expr vars builder edge_expr in
519         L.build_call edge_weight_func [| data_ptr |] "tmp_edge_weight" builder
520       | S.SMethod (edge_expr, "set_weight", [data], _) ->
521         let data_ptr = expr vars builder edge_expr in
522         let edge_type = get_sexpr_type edge_expr in
523         let which_func = match edge_type with Diwedge(_) -> edge_set_weight_func
524                                               | _ ->
    undirected_edge_set_weight_func in
525         L.build_call which_func [| data_ptr ; (expr vars builder data) |] ""
    builder
526 
527       (* graph methods *)
528       | S.SMethod (graph_expr, "print", [], _) ->
529         let graph_ptr = expr vars builder graph_expr in
530         let graph_type = get_sexpr_type graph_expr in
531         let print_func = (match graph_type with
532               A.Graph(A.Int) | A.Digraph(A.Int) -> print_unweighted_int_func
533             | A.Wegraph(A.Int) | A.Wedigraph(A.Int) -> print_int_func
534             | A.Graph(A.Float) | A.Digraph(A.Float) ->
    print_unweighted_float_func
535             | A.Wegraph(A.Float) | A.Wedigraph(A.Float) -> print_float_func
536             | A.Graph(A.String) | A.Digraph(A.String) ->
    print_unweighted_char_ptr_func
537             | A.Wegraph(A.String) | A.Wedigraph(A.String) -> print_char_ptr_func
538             | A.Graph(A.Bool) | A.Digraph(A.Bool) -> print_unweighted_bool_func
539             | A.Wegraph(A.Bool) | A.Wedigraph(A.Bool) -> print_bool_func
540           ) in
541         L.build_call print_func [| graph_ptr |] "" builder
542       | S.SMethod (graph_expr, "add_node", [node_expr], _) ->
543         let graph_ptr = expr vars builder graph_expr
544         and data_ptr = expr vars builder node_expr in
545         L.build_call add_vertex_if_not_func [| graph_ptr ; data_ptr |] "" builder
546       | S.SMethod (graph_expr, "remove_node", [node_expr], _) ->
547         let graph_ptr = expr vars builder graph_expr
548         and data_ptr = expr vars builder node_expr in
549         L.build_call remove_vertex_func [| graph_ptr ; data_ptr |] "" builder
550       | S.SMethod (graph_expr, "has_node", [node_expr], _) ->
551         let graph_ptr = expr vars builder graph_expr
552         and data_ptr = expr vars builder node_expr in
553         let ret = L.build_call has_vertex_func [| graph_ptr ; data_ptr |] ""
    builder in
554         L.build_icmp L.Icmp.Eq ret (L.const_int i32_t 1) "has_node" builder
555       | S.SMethod (graph_expr, "add_edge", [from_node_expr ; to_node_expr], _) ->
556         let graph_ptr = expr vars builder graph_expr
557         and from_data_ptr = expr vars builder from_node_expr
558         and to_data_ptr = expr vars builder to_node_expr in
559         (* if is an undirected graph, add reverse edge as well *)
560         let graph_type = get_sexpr_type graph_expr in
561         (match graph_type with
562            A.Graph(_) ->
```

```
563            ignore(L.build_call add_edge_method_func [| graph_ptr ; to_data_ptr ;
      from_data_ptr |] "" builder)
564           | _ -> ());
565          L.build_call add_edge_method_func [| graph_ptr ; from_data_ptr ;
      to_data_ptr |] "" builder
566        | S.SMethod (graph_expr, "add_edge", [from_node_expr ; to_node_expr ;
      weight_expr], _) ->
567          let graph_ptr = expr vars builder graph_expr
568          and from_data_ptr = expr vars builder from_node_expr
569          and to_data_ptr = expr vars builder to_node_expr
570          and weight = expr vars builder weight_expr in
571          (* if is an undirected graph, add reverse edge as well *)
572          let graph_type = get_sexpr_type graph_expr in
573          (match graph_type with
574             A.Wegraph(_) ->
575             ignore(L.build_call add_wedge_method_func [| graph_ptr ; to_data_ptr ;
       from_data_ptr ; weight |] "" builder)
576           | _ -> ());
577          L.build_call add_wedge_method_func [| graph_ptr ; from_data_ptr ;
      to_data_ptr ; weight |] "" builder
578        | S.SMethod (graph_expr, "remove_edge", [from_node_expr ; to_node_expr], _)
       ->
579          let graph_ptr = expr vars builder graph_expr
580          and from_data_ptr = expr vars builder from_node_expr
581          and to_data_ptr = expr vars builder to_node_expr in
582          (* if this is an undirected graph, remove reverse edge as well *)
583          let graph_type = get_sexpr_type graph_expr in
584          (match graph_type with
585             A.Graph(_) | A.Wegraph(_) ->
586             ignore(L.build_call remove_edge_func [| graph_ptr ; to_data_ptr ;
      from_data_ptr |] "" builder)
587           | _ -> ());
588          L.build_call remove_edge_func [| graph_ptr ; from_data_ptr ; to_data_ptr
      |] "" builder
589        | S.SMethod (graph_expr, "has_edge", [from_node_expr ; to_node_expr], _) ->
590          let graph_ptr = expr vars builder graph_expr
591          and from_data_ptr = expr vars builder from_node_expr
592          and to_data_ptr = expr vars builder to_node_expr in
593          let ret = L.build_call has_edge_func [| graph_ptr ; from_data_ptr ;
      to_data_ptr |] "" builder in
594          L.build_icmp L.Icmp.Eq ret (L.const_int i32_t 1) "has_edge" builder
595        | S.SMethod (graph_expr, "neighbors", [hub_node], _) ->
596          let graph_ptr = expr vars builder graph_expr
597          and hub_data_ptr = expr vars builder hub_node in
598          L.build_call graph_neighbors_func [| graph_ptr ; hub_data_ptr |] ""
      builder
599        | S.SMethod (graph_expr, "get_edge_weight", [from_node_expr ; to_node_expr
      ], _) ->
600          let graph_ptr = expr vars builder graph_expr
601          and from_data_ptr = expr vars builder from_node_expr
602          and to_data_ptr = expr vars builder to_node_expr in
603          L.build_call get_edge_weight_func [| graph_ptr ; from_data_ptr ;
      to_data_ptr |] "" builder
604        | S.SMethod (graph_expr, "set_edge_weight", [from_node_expr ; to_node_expr
      ; weight_expr], _) ->
605          let graph_ptr = expr vars builder graph_expr
606          and from_data_ptr = expr vars builder from_node_expr
```

```
607         and to_data_ptr = expr vars builder to_node_expr
608         and weight = expr vars builder weight_expr in
609         let graph_type = get_sexpr_type graph_expr in
610         let which_func = (match graph_type with
611               A.Wegraph(_) -> set_undirected_edge_weight_func
612             | _ -> set_edge_weight_func) in
613         L.build_call which_func [| graph_ptr ; from_data_ptr ; to_data_ptr ;
    weight |] "" builder
614
615       (* map methods*)
616       | S.SMethod (map_expr, "put", [node_expr ; value_expr], _) ->
617         let map_ptr = expr vars builder map_expr
618         and node_ptr = expr vars builder node_expr
619         and value = expr vars builder value_expr in
620         let map_type = get_sexpr_type map_expr in
621         let value_type = (match map_type with Map(t) -> t | _ -> A.Int (* never
    happens *)) in
622         let which_func = (match value_type with
623               A.Int -> map_put_int_func
624             | A.String -> map_put_char_ptr_func
625             | A.Bool -> map_put_int_func
626             | A.Float -> map_put_float_func
627             | _ -> map_put_void_ptr_func) in
628         let value = if (value_type = A.Bool) then
629             L.build_intcast value i32_t "tmp_intcast" builder
630           else
631             value
632         in
633         L.build_call which_func [| map_ptr ; node_ptr ; value |] "" builder
634       | S.SMethod (map_expr, "get", [node_expr], _) ->
635         let map_ptr = expr vars builder map_expr
636         and node_ptr = expr vars builder node_expr in
637         let map_type = get_sexpr_type map_expr in
638         let value_type = (match map_type with Map(t) -> t | _ -> A.Int) in
639         let which_func = (match value_type with
640               A.Int -> map_get_int_func
641             | A.String -> map_get_char_ptr_func
642             | A.Bool -> map_get_int_func
643             | A.Float -> map_get_float_func
644             | _ -> map_get_void_ptr_func) in
645         let ret = L.build_call which_func [| map_ptr ; node_ptr |] "tmp_get"
    builder in
646         if (value_type = A.Bool) then
647           (L.build_icmp L.Icmp.Ne ret (L.const_int i32_t 0) "tmp_get_bool"
    builder)
648         else
649           ret
650       | S.SMethod (map_expr, "contains", [node_expr], _) ->
651         let map_ptr = expr vars builder map_expr
652         and node_ptr = expr vars builder node_expr in
653         let ret =
654           L.build_call map_contains_func [| map_ptr ; node_ptr |] "tmp_contains"
    builder in
655         L.build_icmp L.Icmp.Eq ret (L.const_int i32_t 1) "contains" builder
656
657     in
658
```

```
659     (* Invoke "f builder" if the current block doesn't already
660        have a terminal (e.g., a branch). *)
661     let add_terminal builder f =
662       match L.block_terminator (L.insertion_block builder) with
663         Some _ -> ()
664       | None -> ignore (f builder) in
665 
666     (* Build the code for the given statement; return the builder for
667        the statement's successor *)
668     let rec stmt vars builder = function
669         S.SBlock sl -> let vars = add_local_vars vars builder sl in
670         List.fold_left (stmt vars) builder sl
671       | S.SExpr (e, _) -> ignore (expr vars builder e); builder
672       | S.SVdecl(t, n, e) -> ignore (expr vars builder e); builder
673       | S.SReturn e -> ignore (match fdecl.S.sf_typ with
674             A.Void -> L.build_ret_void builder
675           | _ -> L.build_ret (expr vars builder e) builder); builder
676       | S.SIf (predicate, then_stmt, else_stmt) ->
677         let bool_val = expr vars builder predicate in
678         let merge_bb = L.append_block context "merge" the_function in
679 
680         let then_bb = L.append_block context "then" the_function in
681         add_terminal (stmt vars (L.builder_at_end context then_bb) then_stmt)
682           (L.build_br merge_bb);
683 
684         let else_bb = L.append_block context "else" the_function in
685         add_terminal (stmt vars (L.builder_at_end context else_bb) else_stmt)
686           (L.build_br merge_bb);
687 
688         ignore (L.build_cond_br bool_val then_bb else_bb builder);
689         L.builder_at_end context merge_bb
690 
691       | S.SWhile (predicate, body) ->
692         let pred_bb = L.append_block context "while" the_function in
693         ignore (L.build_br pred_bb builder);
694 
695         let body_bb = L.append_block context "while_body" the_function in
696         add_terminal (stmt vars (L.builder_at_end context body_bb) body)
697           (L.build_br pred_bb);
698 
699         let pred_builder = L.builder_at_end context pred_bb in
700         let bool_val = expr vars pred_builder predicate in
701 
702         let merge_bb = L.append_block context "merge" the_function in
703         ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
704         L.builder_at_end context merge_bb
705       | S.SBreak -> builder (*not implemented *)
706       | S.SContinue -> builder (*not implemented *)
707       | S.SFor (e1, e2, e3, body) ->
708         stmt vars builder( S.SBlock [S.SExpr (e1, get_sexpr_type e1) ;
709                                     S.SWhile (e2, S.SBlock [body ; S.SExpr (e3,
    get_sexpr_type e3)]) ] );
710       | S.SFor_Node (n, g, body) ->
711         let graph_ptr = (expr vars builder g) in
712 
713         (* allocate counter variable - counts number of nodes seen so far *)
714         let counter = L.build_alloca i32_t "counter" builder in
```

```
715         ignore(L.build_store (L.const_int i32_t 0) counter builder);
716         (* get number of nodes in graph *)
717         let size = L.build_call num_vertices_func [| graph_ptr |] "size" builder
      in
718         (* allocate register for n; then, add to symbol table, so the body can
      access it *)
719         let node_var = L.build_alloca (ltype_of_typ (A.Node(A.Int))) n builder in
720         let vars = StringMap.add n node_var vars in
721 
722         (* allocate pointer to current vertex struct *)
723         let current_vertex_ptr = L.build_alloca void_ptr_t "current" builder in
724         (* get head of vertex list *)
725         let head_vertex = L.build_call get_head_vertex_func [| graph_ptr |] "head
      " builder in
726         ignore(L.build_store head_vertex current_vertex_ptr builder);
727 
728         let pred_bb = L.append_block context "while" the_function in
729         ignore (L.build_br pred_bb builder);
730 
731         let body_bb = L.append_block context "while_body" the_function in
732         let body_builder = L.builder_at_end context body_bb in
733         (* load value of current vertex *)
734         let current_vertex = L.build_load current_vertex_ptr "current_tmp"
      body_builder in
735         (* get node data pointer from current vertex struct *)
736         let data_ptr = L.build_call get_data_from_vertex_func [| current_vertex
      |] (n ^ "_tmp") body_builder in
737         ignore(L.build_store data_ptr node_var body_builder);
738 
739         (* change current_vertex to be pointer to next_vertex *)
740         let next_vertex = L.build_call get_next_vertex_func [| current_vertex |]
      "next" body_builder in
741         ignore(L.build_store next_vertex current_vertex_ptr body_builder);
742         (* increment counter *)
743         let counter_val = L.build_load counter "counter_tmp" body_builder in
744         let counter_incr = L.build_add (L.const_int i32_t 1) counter_val "
      counter_incr" body_builder in
745         ignore(L.build_store counter_incr counter body_builder);
746         (* build body of loop *)
747         add_terminal (stmt vars body_builder body) (L.build_br pred_bb);
748 
749         (* branch to while_body iff counter < size *)
750         let pred_builder = L.builder_at_end context pred_bb in
751         let counter_val = L.build_load counter "counter_tmp" pred_builder in
752         let done_bool_val = L.build_icmp L.Icmp.Slt counter_val size "done"
      pred_builder in
753 
754         let merge_bb = L.append_block context "merge" the_function in
755         ignore (L.build_cond_br done_bool_val body_bb merge_bb pred_builder);
756         L.builder_at_end context merge_bb
757 
758       | S.SFor_Edge (e, g, body) ->
759         let graph_ptr = (expr vars builder g) in
760 
761         (* allocate counter variable - counts number of edges seen so far *)
762         let counter = L.build_alloca i32_t "counter" builder in
763         ignore(L.build_store (L.const_int i32_t 0) counter builder);
```

```
764
765         (* allocate register for e; then, add to symbol table, so the body can
     access it *)
766         let edge_var = L.build_alloca (ltype_of_typ (A.Edge(A.Int))) e builder in
767         let vars = StringMap.add e edge_var vars in
768
769         (* allocate pointer to current edge struct *)
770         let current_edge_ptr = L.build_alloca void_ptr_t "current" builder in
771         (* construct edge list and get head *)
772         let graph_type = get_sexpr_type g in
773         let construct_func = (match graph_type with
774               A.Digraph(_) | A.Wedigraph(_) -> construct_edge_list_func
775             | _ -> construct_undirected_edge_list_func) in
776         let head_edge = L.build_call construct_func [| graph_ptr |] "head"
     builder in
777         ignore(L.build_store head_edge current_edge_ptr builder);
778
779         (* get number of edges *)
780         let size = L.build_call num_edges_func [| head_edge |] "size" builder in
781
782         let pred_bb = L.append_block context "while" the_function in
783         ignore (L.build_br pred_bb builder);
784
785         let body_bb = L.append_block context "while_body" the_function in
786         let body_builder = L.builder_at_end context body_bb in
787
788         let current_edge = L.build_load current_edge_ptr "current_tmp"
     body_builder in
789
790         (* load value of current edge into edge_var *)
791         ignore(L.build_store current_edge edge_var body_builder);
792
793         (* change edge_var to be pointer to next edge *)
794         let next_edge = L.build_call get_next_edge_func [| current_edge |] "next"
      body_builder in
795         ignore(L.build_store next_edge current_edge_ptr body_builder);
796         (* increment counter *)
797         let counter_val = L.build_load counter "counter_tmp" body_builder in
798         let counter_incr = L.build_add (L.const_int i32_t 1) counter_val "
     counter_incr" body_builder in
799         ignore(L.build_store counter_incr counter body_builder);
800         (* build body of loop *)
801         add_terminal (stmt vars body_builder body) (L.build_br pred_bb);
802
803         (* branch to while_body iff counter < size *)
804         let pred_builder = L.builder_at_end context pred_bb in
805         let counter_val = L.build_load counter "counter_tmp" pred_builder in
806         let done_bool_val = L.build_icmp L.Icmp.Slt counter_val size "done"
     pred_builder in
807
808         let merge_bb = L.append_block context "merge" the_function in
809         ignore (L.build_cond_br done_bool_val body_bb merge_bb pred_builder);
810         L.builder_at_end context merge_bb
811
812       | S.SBfs (n, g, r, body) ->
813         let graph_ptr = (expr vars builder g) in
814         let root_ptr = (expr vars builder r) in
```

```
815
816         (* allocate register for n; then, add to symbol table, so the body can
        access it *)
817         let node_var = L.build_alloca (ltype_of_typ (A.Node(A.Int))) n builder in
818         (* add the node data pointer to symbol table, so the body can access it
        *)
819         let vars = StringMap.add n node_var vars in
820         (* allocate pointer to current vertex struct *)
821         let current_vertex_ptr = L.build_alloca void_ptr_t "current" builder in
822         (* get root vertex_list_node for bfs search *)
823         let root_vertex = L.build_call find_vertex_func [| graph_ptr ; root_ptr
        |] "root" builder in
824         ignore(L.build_store root_vertex current_vertex_ptr builder);
825
826         let visited = L.build_call get_visited_array_func [| graph_ptr |] "
        visited" builder in
827
828         let queue = L.build_call get_bfs_queue_func [| root_vertex ; visited |] "
        queue" builder in
829         (* populate queue and visited on the root node, but do not need to save
        returned vertex
830            because current_vertex_ptr is already root_vertex during first
        iteration of the loop *)
831         ignore(L.build_call get_next_bfs_vertex_func [| visited ; queue |] "
        get_next" builder);
832
833         let pred_bb = L.append_block context "while" the_function in
834         ignore (L.build_br pred_bb builder);
835
836         let body_bb = L.append_block context "while_body" the_function in
837         let body_builder = L.builder_at_end context body_bb in
838         (* load value of current vertex *)
839         let current_vertex = L.build_load current_vertex_ptr "current_tmp"
        body_builder in
840         (* get node data pointer from current vertex struct *)
841         let data_ptr = L.build_call get_data_from_vertex_func [| current_vertex
        |] (n ^ "_tmp") body_builder in
842         ignore(L.build_store data_ptr node_var body_builder);
843
844         (* change current_vertex to be pointer to next_vertex *)
845         let next_vertex = L.build_call get_next_bfs_vertex_func [| visited ;
        queue |] "get_next" body_builder in
846         ignore(L.build_store next_vertex current_vertex_ptr body_builder);
847
848         (* build body of loop *)
849         add_terminal (stmt vars body_builder body) (L.build_br pred_bb);
850         let pred_builder = L.builder_at_end context pred_bb in
851         (* determine whether current_vertex_ptr is NULL using c bfs_done function
         *)
852         let pred_vertex = L.build_load current_vertex_ptr "pred_tmp" pred_builder
         in
853         let done_flag = L.build_call bfs_done_func [| pred_vertex |] "done"
        pred_builder in
854         (* branch to while_body iff done_flag is 0 (i.e. if current_vertex_ptr is
         not NULL) *)
855         let done_bool_val = L.build_icmp L.Icmp.Eq done_flag (L.const_int i32_t
        0) "done_pred" pred_builder in
```

```
856
857         let merge_bb = L.append_block context "merge" the_function in
858         ignore (L.build_cond_br done_bool_val body_bb merge_bb pred_builder);
859         L.builder_at_end context merge_bb
860       | S.SDfs (n, g, r, body) ->
861         let graph_ptr = (expr vars builder g) in
862         let root_ptr = (expr vars builder r) in
863
864         (* allocate register for n; then, add to symbol table, so the body can
    access it *)
865         let node_var = L.build_alloca (ltype_of_typ (A.Node(A.Int))) n builder in
866         (* add the node data pointer to symbol table, so the body can access it
    *)
867         let vars = StringMap.add n node_var vars in
868         (* allocate pointer to current vertex struct *)
869         let current_vertex_ptr = L.build_alloca void_ptr_t "current" builder in
870         (* get root vertex_list_node for bfs search *)
871         let root_vertex = L.build_call find_vertex_func [| graph_ptr ; root_ptr
    |] "root" builder in
872         ignore(L.build_store root_vertex current_vertex_ptr builder);
873
874         let visited = L.build_call get_visited_array_func [| graph_ptr |] "
    visited" builder in
875
876         let stack = L.build_call get_dfs_stack_func [| root_vertex ; visited |] "
    stack" builder in
877         (* populate stack and visited on the root node, but do not need to save
    returned vertex
878            because current_vertex_ptr is already root_vertex during first
    iteration of the loop *)
879         ignore(L.build_call get_next_dfs_vertex_func [| visited ; stack |] "
    get_next" builder);
880
881         let pred_bb = L.append_block context "while" the_function in
882         ignore (L.build_br pred_bb builder);
883
884         let body_bb = L.append_block context "while_body" the_function in
885         let body_builder = L.builder_at_end context body_bb in
886         (* load value of current vertex *)
887         let current_vertex = L.build_load current_vertex_ptr "current_tmp"
    body_builder in
888         (* get node data pointer from current vertex struct *)
889         let data_ptr = L.build_call get_data_from_vertex_func [| current_vertex
    |] (n ^ "_tmp") body_builder in
890         ignore(L.build_store data_ptr node_var body_builder);
891
892         (* change current_vertex to be pointer to next_vertex *)
893         let next_vertex = L.build_call get_next_dfs_vertex_func [| visited ;
    stack |] "get_next" body_builder in
894         ignore(L.build_store next_vertex current_vertex_ptr body_builder);
895
896         (* build body of loop *)
897         add_terminal (stmt vars body_builder body) (L.build_br pred_bb);
898         let pred_builder = L.builder_at_end context pred_bb in
899         (* determine whether current_vertex_ptr is NULL using c dfs_done function
     *)
```

```
900         let pred_vertex = L.build_load current_vertex_ptr "pred_tmp" pred_builder
     in
901         let done_flag = L.build_call dfs_done_func [| pred_vertex |] "done"
    pred_builder in
902         (* branch to while_body iff done_flag is 0 (i.e. if current_vertex_ptr is
     not NULL) *)
903         let done_bool_val = L.build_icmp L.Icmp.Eq done_flag (L.const_int i32_t
    0) "done_pred" pred_builder in
904
905         let merge_bb = L.append_block context "merge" the_function in
906         ignore (L.build_cond_br done_bool_val body_bb merge_bb pred_builder);
907         L.builder_at_end context merge_bb
908     in
909
910     (* Build the code for each statement in the function *)
911     let builder = stmt globals_and_formals builder (S.SBlock fdecl.S.sf_body) in
912
913     (* Add a return if the last block falls off the end *)
914     add_terminal builder (match fdecl.S.sf_typ with
915           A.Void -> L.build_ret_void
916         | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
917   in
918
919   List.iter build_function_body functions;
920   the_module
```

Code generation: codegen.ml

```
1  (* Authors:
2  Daniel Benett deb2174
3  Seth Benjamin sjb2190
4  Jennifer Bi jb3495
5  Jessie Liu jll2219
6  *)
7
8  (* Top-level of the Giraph compiler: scan & parse the input,
9     check the resulting AST and generate SAST, generate LLVM IR,
10    and dump the module *)
11
12 module StringMap = Map.Make(String)
13
14 type action = Ast | LLVM_IR | Compile
15
16 let _ =
17     let action = ref Compile in
18     let set_action a () = action := a in
19     let speclist = [
20       ("-a", Arg.Unit (set_action Ast), "Print the SAST");
21       ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
22       ("-c", Arg.Unit (set_action Compile),
23         "Check and print the generated LLVM IR (default)");
24     ] in
25     let usage_msg = "usage: ./giraph.native [-a|-l|-c] [file.gir]" in
26     let channel = ref stdin in
27     Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
28     let lexbuf = Lexing.from_channel !channel in
29     let ast = Parser.program Scanner.token lexbuf in
30     let sast = Semant.check ast in
```

```
31     match !action with
32         Ast -> (*print_string (Ast.string_of_program ast)*) print_string (Sast.
       string_of_sprogram sast)
33       | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast)
       )
34       | Compile -> let m = Codegen.translate sast in
35         Llvm_analysis.assert_valid_module m;
36         print_string (Llvm.string_of_llmodule m)
```

Code generation: giraph.ml

```
1
2
3  .PHONY : all
4  all : clean giraph.native graph.o
5
6  .PHONY : giraph.native
7  giraph.native :
8    ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \
9      giraph.native
10
11 .PHONY : clean
12 clean :
13   rm -f *.o
```

Makefile

8.5 C Libraries

```
1  /* Authors:
2  Daniel Benett deb2174
3  Seth Benjamin sjb2190
4  */
5
6  #include <stdio.h>
7  #include <string.h>
8  #include <stdlib.h>
9
10 /* for use with generic graph/node types */
11 union data_type {
12   int i;
13   float f;
14   char *s;
15   void *v;
16 };
17
18 /* so we can cast void *'s to floats/ints in maps without triggering undefined
19    behavior
20    shoulda done the whole map thing with a union but hindsight is 20/20 */
21 union extract_float {
22     float vf;
23     void *vp;
24 };
25 union extract_int {
26     int vi;
27     void *vp;
```

```
28 };
29
30 /* Terminology-wise, we've painted ourselves into a corner here.
31    Elsewhere in in this project, "node" refers to a single node in a graph.
32    That is NOT true in this file. In this file, "vertex" refers to a node in
33    a graph, and "node" refers to a single node of a linked list. */
34
35 /////////////////////// STRUCTS ///////////////////////
36
37 /* A single node of the adjacency list for a single vertex. */
38 struct adj_list_node {
39   struct vertex_list_node *vertex;
40   struct adj_list_node *next;
41   int weight;
42 };
43
44 /* a single node in the edge list containing relevant edge information */
45 struct edge_list_node {
46   struct vertex_list_node *from;
47   struct vertex_list_node *to;
48   struct edge_list_node *next;
49   int weight;
50 };
51
52 /* A single vertex in a graph. */
53 struct vertex_list_node {
54   void *data;
55   struct adj_list_node *adjacencies;
56   struct vertex_list_node *next;
57 };
58
59 /* A graph. */
60 struct graph {
61   struct vertex_list_node *head;
62 };
63
64 /* node for queue */
65 struct queue_list_node {
66   struct vertex_list_node *v;
67   struct queue_list_node *next;
68 };
69
70 /* node for stack */
71 struct stack_list_node {
72   struct vertex_list_node *v;
73   struct stack_list_node *next;
74 };
75
76 /* node for map */
77 struct map_node {
78   struct map_node *next;
79   unsigned int key;
80   void *value;
81 };
82
83 ///////////////////// END STRUCTS /////////////////////
84
```

```
85
86 ////////////////////////// MAP //////////////////////////
87 /* A single node of the adjacency list for a single vertex. */
88
89 struct map_node *get_node(unsigned int key, void *value) {
90   struct map_node *out = (struct map_node *) malloc(sizeof(struct map_node));
91   out->key = key;
92   out->value = value;
93   out->next = NULL;
94   return out;
95 }
96
97 /* returns pointer to map */
98 void *make_map() {
99   /* (size - 1) is hashable. 432 is the the 83rd prime number (431) plus 1. */
100  int default_size = 432;
101  struct map_node **map = (struct map_node **) malloc(sizeof(struct map_node *) *
       default_size);
102  memset(map, 0, sizeof(struct map_node *) * default_size);
103  int *size = (int *) malloc(sizeof(int));
104  *size = default_size;
105  map[0] = get_node(0, size);
106  return map;
107 }
108
109 void free_map(void *map_in) {
110  struct map_node **map = (struct map_node **) map_in;
111  int size = *((int *) map[0]->value);
112  for (int i = 0; i < size; i++) {
113    struct map_node *bucket = map[i];
114    if (bucket) {
115      struct map_node *next = bucket->next;
116      free(bucket);
117      while (next) {
118        bucket = next;
119        next = bucket->next;
120        free(bucket);
121      }
122    }
123  }
124  free(map);
125 }
126
127 /* hash function: hash 0 is reserved for table size */
128 int hash(unsigned int in, int size) {
129  return 1 + ((in * 997) % (size - 1));
130 }
131
132 /* if putting into a key already in map, replace value */
133 void put(void *map_in, void *key, void *value) {
134  struct map_node **map = (struct map_node **) map_in;
135  int size = *((int *) map[0]->value);
136  unsigned int for_hash = (unsigned int) key;
137  int hash_val = hash(for_hash, size);
138  struct map_node *bucket = map[hash_val];
139  if (!bucket) {
140    map[hash_val] = get_node((unsigned int) key, value);
```

```
141     return;
142   }
143   if (bucket->key == (unsigned int) key) {
144     bucket->value = value;
145     return;
146   }
147   while (bucket->next) {
148     if (bucket->key == (unsigned int) key) {
149       bucket->value = value;
150       return;
151     }
152     bucket = bucket->next;
153   }
154   bucket->next = get_node((unsigned int) key, value);
155 }
156 
157 /* returns NULL if not found */
158 void *get(void *map_in, void *key) {
159   struct map_node **map = (struct map_node **) map_in;
160   int size = *((int *) map[0]->value);
161   unsigned int for_hash = (unsigned int) key;
162   int hash_val = hash(for_hash, size);
163   struct map_node *bucket = map[hash_val];
164   if (!bucket) {
165     return NULL;
166   }
167   while (bucket) {
168     if (bucket->key == (unsigned int) key) {
169       return bucket->value;
170     }
171     bucket = bucket->next;
172   }
173   return NULL;
174 }
175 
176 int contains_key(void *map_in, void *key) {
177   struct map_node **map = (struct map_node **) map_in;
178   int size = *((int *) map[0]->value);
179   unsigned int for_hash = (unsigned int) key;
180   int hash_val = hash(for_hash, size);
181   struct map_node *bucket = map[hash_val];
182   if (!bucket) {
183     return 0;
184   }
185   while (bucket) {
186     if (bucket->key == (unsigned int) key) {
187       return 1;
188     }
189     bucket = bucket->next;
190   }
191   return 0;
192 }
193 
194 /* The following functions implement put() for the built-in types in giraph. */
195 
196 void put_int(void *map_in, void *key, int value) {
197   union extract_int ei;
```

```
198    ei.vi = value;
199    put(map_in, key, ei.vp);
200  }
201
202  void put_int_ptr(void *map_in, void *key, void *value) {
203    put(map_in, key, (void *) value);
204  }
205
206  void put_char_ptr(void *map_in, void *key, char *value) {
207    put(map_in, key, (void *) value);
208  }
209
210  void put_float(void *map_in, void *key, float value) {
211    union extract_float ef;
212    ef.vf = value;
213    put(map_in, key, ef.vp);
214  }
215
216  /* The following functions implement get() for the built-in types in giraph. */
217
218  int get_int(void *map_in, void *key) {
219    union extract_int ei;
220    ei.vp = get(map_in, key);
221    return ei.vi;
222  }
223
224  int *get_int_ptr(void *map_in, void *key) {
225    return (int *) get(map_in, key);
226  }
227
228  char *get_char_ptr(void *map_in, void *key) {
229    return (char *) get(map_in, key);
230  }
231
232  float get_float(void *map_in, void *key) {
233    union extract_float ef;
234    ef.vp = get(map_in, key);
235    return ef.vf;
236  }
237
238
239
240
241  //////////////////////// END MAP ///////////////////////
242
243
244  ////////////////////// EDGE METHODS ////////////////////
245
246  void *edge_from(void *e) {
247    return ((struct edge_list_node *) e)->from->data;
248  }
249
250  void *edge_to(void *e) {
251    return ((struct edge_list_node *) e)->to->data;
252  }
253
254  int edge_weight(void *e) {
```

```
255   return ((struct edge_list_node *) e)->weight;
256 }
257
258 void undirected_edge_set_weight(void *e_in, int new_weight) {
259   struct edge_list_node *e = (struct edge_list_node *) e_in;
260   struct adj_list_node *from_adj = e->from->adjacencies;
261   struct adj_list_node *to_adj = e->to->adjacencies;
262   while (from_adj) {
263     if (from_adj->vertex == e->to) {
264       from_adj->weight = new_weight;
265     }
266     from_adj = from_adj->next;
267   }
268   while (to_adj) {
269     if (to_adj->vertex == e->from) {
270       to_adj->weight = new_weight;
271     }
272     to_adj = to_adj->next;
273   }
274   ((struct edge_list_node *) e)->weight = new_weight;
275 }
276
277 void edge_set_weight(void *e_in, int new_weight) {
278   struct edge_list_node *e = (struct edge_list_node *) e_in;
279   struct adj_list_node *from_adj = e->from->adjacencies;
280   while (from_adj) {
281     if (from_adj->vertex == e->to) {
282       from_adj->weight = new_weight;
283     }
284     from_adj = from_adj->next;
285   }
286   ((struct edge_list_node *) e)->weight = new_weight;
287 }
288
289 /////////////////// END EDGE METHODS //////////////////
290
291
292 ///////////////////// NODE METHODS ////////////////////
293
294 /* Change the data stored in a node<int> data pointer. */
295 void set_data_int(void *data_ptr, int data_val) {
296   ((union data_type *) data_ptr)->i = data_val;
297 }
298
299 /* Change the data stored in a node<float> data pointer. */
300 void set_data_float(void *data_ptr, float data_val) {
301   ((union data_type *) data_ptr)->f = data_val;
302 }
303
304 /* Change the data stored in a node<string> data pointer. */
305 void set_data_char_ptr(void *data_ptr, char *data_val) {
306   ((union data_type *) data_ptr)->s = data_val;
307 }
308
309 /* Change the data stored in a data pointer for all other node types. */
310 void set_data_void_ptr(void *data_ptr, void *data_val) {
311   /* *((void **) data_ptr) = data_val; */
```

```
312   ((union data_type *) data_ptr)->v = data_val;
313 }
314
315 /* Get data stored in a node<int> data pointer. */
316 int get_data_int(void *data_ptr) {
317   return ((union data_type *) data_ptr)->i;
318 }
319
320 /* Get data stored in a node<float> data pointer. */
321 float get_data_float(void *data_ptr) {
322   return ((union data_type *) data_ptr)->f;
323 }
324
325 /* Get data stored in a node<string> data pointer. */
326 char *get_data_char_ptr(void *data_ptr) {
327   return ((union data_type *) data_ptr)->s;
328 }
329
330 /* Get data stored in a data pointer for all other node types. */
331 void *get_data_void_ptr(void *data_ptr) {
332   /* return *((void **) data_ptr); */
333   return ((union data_type *) data_ptr)->v;
334 }
335
336 //////////////////// END NODE METHODS //////////////////
337
338
339 ////////////////// GRAPH HELPER METHODS ////////////////
340 /* Find and return the vertex_list_node associated with a data pointer.
341    Returns null if there is none. */
342 void *find_vertex(void *g_in, void *data_ptr) {
343   struct graph *g = (struct graph *) g_in;
344   struct vertex_list_node *vertex = g->head;
345   while (vertex) {
346     if (vertex->data == data_ptr) {
347       return vertex;
348     }
349     vertex = vertex->next;
350   }
351   return NULL;
352 }
353
354 /* Returns a pointer to a new graph. */
355 void *new_graph() {
356   struct graph *g = malloc(sizeof(struct graph));
357   g->head = NULL;
358
359   return (void *) g;
360 }
361
362 /* Allocate a new unique data pointer. */
363 void *new_data() {
364   return (void *) malloc(sizeof(union data_type));
365 }
366
367 /* Add a new vertex to the end of the vertex list in a graph, and return a
368    pointer to the new vertex. */
```

```
369 void *add_vertex(void *graph_ptr, void *data_ptr) {
370   struct vertex_list_node *vertex = malloc(sizeof(struct vertex_list_node));
371   vertex->data = data_ptr;
372   vertex->adjacencies = NULL;
373   vertex->next = NULL;
374
375   struct graph *g = (struct graph *) graph_ptr;
376   if (g->head == NULL) {
377     g->head = vertex;
378   } else {
379     struct vertex_list_node *last_node = g->head;
380     while (last_node->next) {
381       last_node = last_node->next;
382     }
383     last_node->next = vertex;
384   }
385
386   return (void *) vertex;
387 }
388
389 /* Add a directed, weighted edge between two vertices */
390 void add_wedge(void *from_ptr, void *to_ptr, int w) {
391   struct vertex_list_node *from = (struct vertex_list_node *) from_ptr;
392   struct vertex_list_node *to = (struct vertex_list_node *) to_ptr;
393   if (from->adjacencies == NULL) {
394     from->adjacencies = malloc(sizeof(struct adj_list_node));
395     from->adjacencies->vertex = to;
396     from->adjacencies->weight = w;
397     from->adjacencies->next = NULL;
398   } else {
399     struct adj_list_node *last_node = from->adjacencies;
400     while (last_node->next) {
401       last_node = last_node->next;
402     }
403     last_node->next = malloc(sizeof(struct adj_list_node));
404     last_node->next->vertex = to;
405     last_node->next->weight = w;
406     last_node->next->next = NULL;
407   }
408 }
409
410 /* Add a (directed) edge between two vertices. Give default weight 0. */
411 void add_edge(void *from_ptr, void *to_ptr) {
412   add_wedge(from_ptr, to_ptr, 0);
413 }
414 /////////////// END GRAPH HELPER METHODS //////////////
415
416
417 //////////////////// GRAPH METHODS ////////////////////
418
419 /* Given a graph and a data pointer, checks if the graph has a vertex associated
420    with the data pointer, and if not, creates one and adds it to the graph.
421    Corresponds to add_node method in giraph. */
422 void add_vertex_if_not_present(void *g_in, void *data_ptr) {
423   struct graph *g = (struct graph *) g_in;
424   if (find_vertex(g_in, data_ptr) == NULL) {
425     add_vertex(g_in, data_ptr);
```

```
426     }
427   }
428
429   /* Given a graph and a data pointer, finds the vertex in the graph associated
430      with the data pointer and removes it from the vertex list and all adjacency
431      lists. If no such vertex exists, does nothing.
432      Corresponds to remove_node method in giraph. */
433   void remove_vertex(void *g_in, void *data_ptr) {
434     struct graph *g = (struct graph *) g_in;
435     struct vertex_list_node *remove = (struct vertex_list_node *) find_vertex(g_in,
          data_ptr);
436     if (remove == NULL) {
437       return;
438     }
439
440     /* Iterate through all vertices and remove this vertex from any adjacency
441        lists. */
442     struct vertex_list_node *vertex = g->head;
443     while (vertex) {
444       if (vertex->adjacencies) {
445         /* if we need to remove the first adjacency, set vertex's
446            "adjacencies" pointer to be the next adjacency */
447         struct adj_list_node *curr_e = vertex->adjacencies;
448         if (curr_e->vertex == remove) {
449           vertex->adjacencies = curr_e->next;
450           free(curr_e); /* woooaaahh */
451         } else {
452           /* else, just remove appropriate adj_list_node from list
453              by reconnecting surrounding nodes */
454           struct adj_list_node *prev_e = vertex->adjacencies;
455           curr_e = prev_e->next;
456
457           while (curr_e) {
458             if (curr_e->vertex == remove) {
459               prev_e->next = curr_e->next;
460               free(curr_e);
461               break;
462             }
463             prev_e = curr_e;
464             curr_e = curr_e->next;
465           }
466         }
467       }
468       vertex = vertex->next;
469     }
470
471     /* Remove vertex from vertex list. */
472     struct vertex_list_node *curr_v = g->head;
473     /* If it's the first vertex, connect g->head to next vertex. */
474     if (curr_v == remove) {
475       g->head = curr_v->next;
476       free(curr_v);
477       return;
478     }
479
480     /* Else, remove from vertex list by reconnecting surrounding nodes. */
481     struct vertex_list_node *prev_v = g->head;
```

```
482   curr_v = prev_v->next;
483   while (curr_v) {
484     if (curr_v == remove) {
485       prev_v->next = curr_v->next;
486       free(curr_v);
487       return;
488     }
489     prev_v = curr_v;
490     curr_v = curr_v->next;
491   }
492 }
493 
494 /* Given a graph and two data pointers, adds a directed, weighted edge between
       the
495    vertices corresponding to each data pointer. If either of such vertices
496    does not exist, they are created. If the edge already exists, does nothing.
497    Corresponds to add_edge method in giraph. */
498 void add_wedge_method(void *g_in, void *from_data_ptr, void *to_data_ptr, int w)
       {
499   struct graph *g = (struct graph *) g_in;
500   void *from = find_vertex(g_in, from_data_ptr);
501   if (from == NULL) {
502     from = add_vertex(g_in, from_data_ptr);
503   }
504   void *to = find_vertex(g_in, to_data_ptr);
505   if (to == NULL) {
506     to = add_vertex(g_in, to_data_ptr);
507   }
508   /* Check if from->to edge already exists - if so, return. */
509   struct adj_list_node *curr_adj = ((struct vertex_list_node *) from)->
       adjacencies;
510   while (curr_adj) {
511     if (curr_adj->vertex == to) {
512       return;
513     }
514     curr_adj = curr_adj->next;
515   }
516 
517   add_wedge(from, to, w);
518 }
519 
520 /* calls add_wedge_method with default weight of 0 */
521 void add_edge_method(void *g_in, void *from_data_ptr, void *to_data_ptr) {
522   add_wedge_method(g_in, from_data_ptr, to_data_ptr, 0);
523 }
524 
525 /* Given a graph and two data pointers, removes the directed edge between the
526    vertices corresponding to each data pointer. If either of such vertices
527    does not exist, or if the edge does not exist, does nothing.
528    Corresponds to remove_edge method in giraph. */
529 void remove_edge(void *g_in, void *from_data_ptr, void *to_data_ptr) {
530   struct graph *g = (struct graph *) g_in;
531   struct vertex_list_node *from = (struct vertex_list_node *) find_vertex(g_in,
       from_data_ptr);
532   struct vertex_list_node *to = (struct vertex_list_node *) find_vertex(g_in,
       to_data_ptr);
533   if (from == NULL || to == NULL) {
```

```
534       return;
535     }
536 
537     /* Remove adj_list_node for "to" from adjacency list of "from" */
538     if (from->adjacencies) {
539       /* if we need to remove the first adjacency, set from's
540          "adjacencies" pointer to be the next adjacency */
541       struct adj_list_node *curr = from->adjacencies;
542       if (curr->vertex == to) {
543         from->adjacencies = curr->next;
544         free(curr);
545       } else {
546         /* else, just remove appropriate adj_list_node from list
547            by reconnecting surrounding nodes */
548         struct adj_list_node *prev = from->adjacencies;
549         curr = prev->next;
550 
551         while (curr) {
552           if (curr->vertex == to) {
553             prev->next = curr->next;
554             free(curr);
555             break;
556           }
557           prev = curr;
558           curr = curr->next;
559         }
560       }
561     }
562   }
563 
564   /* Checks if a graph contains a vertex. Returns 1 if so, 0 otherwise.
565      Corresponds to graph.has_node() method in giraph. */
566   int has_vertex(void *g_in, void *data_ptr) {
567     return (find_vertex(g_in, data_ptr) != NULL);
568   }
569 
570   /* Checks if a graph contains an edge between two vertices.
571      Returns 1 if so, 0 otherwise.
572      Corresponds to graph.has_edge() method in giraph. */
573   int has_edge(void *g_in, void *from_data_ptr, void *to_data_ptr) {
574     struct graph *g = (struct graph *) g_in;
575     struct vertex_list_node *from = (struct vertex_list_node *) find_vertex(g_in,
        from_data_ptr);
576     struct vertex_list_node *to = (struct vertex_list_node *) find_vertex(g_in,
        to_data_ptr);
577 
578     /* If either of the vertices is not in the graph, neither is the edge. */
579     if (from == NULL || to == NULL) {
580       return 0;
581     }
582 
583     struct adj_list_node *curr_adj = ((struct vertex_list_node *) from)->
        adjacencies;
584     while (curr_adj) {
585       if (curr_adj->vertex == to) {
586         return 1;
587       }
```

```
588       curr_adj = curr_adj->next;
589     }
590     return 0;
591   }
592
593   /* return a graph pointer to a graph containing every neighboring vertex */
594   void *graph_neighbors(void *g_in, void *data_ptr) {
595     struct graph *g = (struct graph *) g_in;
596     struct vertex_list_node *v = find_vertex(g_in, data_ptr);
597
598     struct graph *g_out = (struct graph *) malloc(sizeof(struct graph));
599     /* if there are no adjs, or if data_ptr is not in g, return graph * with NULL
        head */
600     g_out->head = NULL;
601     if (v == NULL) {
602       return (void *)g_out;
603     }
604
605     struct adj_list_node *curr_adj = (struct adj_list_node *) v->adjacencies;
606
607     if (curr_adj) {
608       /* add first vertex in new graph with data in first adjacency */
609       g_out->head = (struct vertex_list_node *) malloc(sizeof(struct
        vertex_list_node));
610       struct vertex_list_node *curr_g_out = g_out->head;
611       curr_g_out->next = NULL;
612       curr_g_out->adjacencies = NULL;
613       curr_g_out->data = curr_adj->vertex->data;
614       curr_adj = curr_adj->next;
615
616       while (curr_adj) {
617         /* add all vertices in new graph with data in subsequent adjacencies */
618         curr_g_out->next = (struct vertex_list_node *) malloc(sizeof(struct
        vertex_list_node));
619         curr_g_out->next->data = curr_adj->vertex->data;
620         curr_g_out->next->next = NULL;
621         curr_g_out->next->adjacencies = NULL;
622         curr_adj = curr_adj->next;
623         curr_g_out = curr_g_out->next;
624       }
625     }
626
627     return (void *)g_out;
628   }
629
630   int graph_get_edge_weight(void *g_in, void *from_data_ptr, void *to_data_ptr) {
631     struct vertex_list_node *from = (struct vertex_list_node *) find_vertex(g_in,
        from_data_ptr);
632     if (from == NULL) {
633       return 0;
634     }
635     struct vertex_list_node *to = (struct vertex_list_node *) find_vertex(g_in,
        to_data_ptr);
636     if (to == NULL) {
637       return 0;
638     }
639     struct adj_list_node *curr_adj = from->adjacencies;
```

```
640   while (curr_adj) {
641     if (curr_adj->vertex == to) {
642       return curr_adj->weight;
643     }
644     curr_adj = curr_adj->next;
645   }
646   return 0;
647 }
648 
649 void graph_set_undirected_edge_weight(void *g_in, void *from_data_ptr, void *
        to_data_ptr, int new_weight) {
650   struct vertex_list_node *from = (struct vertex_list_node *) find_vertex(g_in,
        from_data_ptr);
651   if (from == NULL) {
652     return;
653   }
654   struct vertex_list_node *to = (struct vertex_list_node *) find_vertex(g_in,
        to_data_ptr);
655   if (to == NULL) {
656     return;
657   }
658   struct adj_list_node *curr_adj = from->adjacencies;
659   while (curr_adj) {
660     if (curr_adj->vertex == to) {
661       curr_adj->weight = new_weight;
662     }
663     curr_adj = curr_adj->next;
664   }
665   curr_adj = to->adjacencies;
666   while (curr_adj) {
667     if (curr_adj->vertex == from) {
668       curr_adj->weight = new_weight;
669     }
670     curr_adj = curr_adj->next;
671   }
672 }
673 
674 void graph_set_edge_weight(void *g_in, void *from_data_ptr, void *to_data_ptr,
        int new_weight) {
675   struct vertex_list_node *from = (struct vertex_list_node *) find_vertex(g_in,
        from_data_ptr);
676   if (from == NULL) {
677     return;
678   }
679   struct vertex_list_node *to = (struct vertex_list_node *) find_vertex(g_in,
        to_data_ptr);
680   if (to == NULL) {
681     return;
682   }
683   struct adj_list_node *curr_adj = from->adjacencies;
684   while (curr_adj) {
685     if (curr_adj->vertex == to) {
686       curr_adj->weight = new_weight;
687     }
688     curr_adj = curr_adj->next;
689   }
690 }
```

```
691
692  /////////////////// END GRAPH METHODS /////////////////
693
694
695  /////////////////////// FOR NODE /////////////////////
696
697  /* iterate through graph to get num vertices */
698  int num_vertices(void *g_in) {
699    struct graph *g = (struct graph *) g_in;
700    struct vertex_list_node *vertex = g->head;
701    int counter = 0;
702    while (vertex) {
703      counter++;
704      vertex = vertex->next;
705    }
706
707    return counter;
708  }
709
710  /* return head of graph */
711  void *get_head_vertex(void *g_in) {
712    struct graph *g = (struct graph *) g_in;
713    struct vertex_list_node *head = g->head;
714
715    return (void *) head;
716  }
717
718  /* given a vertex, returns next vertex from graph's list */
719  void *get_next_vertex(void *v_in) {
720    struct vertex_list_node *v = (struct vertex_list_node *) v_in;
721
722    return (void *) v->next;
723  }
724
725  /* return the data pointer stored in a vertex */
726  void *get_data_from_vertex(void *v_in) {
727    struct vertex_list_node *v = (struct vertex_list_node *) v_in;
728
729    return v->data;
730  }
731
732  ///////////////////// END FOR NODE ///////////////////
733
734
735  /////////////////////// FOR EDGE /////////////////////
736
737  /* construct a list of edge_list_node's and return head */
738  void *construct_edge_list(void *g_in) {
739    struct graph *g = (struct graph *) g_in;
740    struct vertex_list_node *v = g->head;
741    struct edge_list_node *head = NULL;
742    int first = 1;
743    struct edge_list_node *prev;
744    while (v) {
745      struct adj_list_node *adjacency = v->adjacencies;
746      while (adjacency) {
```

```
747         struct edge_list_node *e = (struct edge_list_node *) malloc(sizeof(struct
        edge_list_node));
748         e->from = v;
749         e->to = adjacency->vertex;
750         e->weight = adjacency->weight;
751         e->next = NULL;
752         if (first) {
753           head = e;
754           first = 0;
755         }
756         else {
757           prev->next = e;
758         }
759         prev = e;
760         adjacency = adjacency->next;
761       }
762       v = v->next;
763     }
764     if (!head) {
765       return NULL;
766     }
767     return head;
768   }
769
770   /* construct a list of edge_list_node's and return head */
771   void *construct_undirected_edge_list(void *g_in) {
772     struct graph *g = (struct graph *) g_in;
773     struct vertex_list_node *v = g->head;
774     struct edge_list_node *head = NULL;
775     int first = 1;
776     struct edge_list_node *prev;
777     void *map = make_map();
778     while (v) {
779       struct adj_list_node *adjacency = v->adjacencies;
780       while (adjacency) {
781         struct adj_list_node *to_adj_list = get(map, (void *) adjacency->vertex);
782         int opposite_edge_exists = 0;
783         while (to_adj_list) {
784           /*  if this v is not in an already edge-ified adj list */
785           if (to_adj_list->vertex == v) {
786             opposite_edge_exists = 1;
787           }
788           to_adj_list = to_adj_list->next;
789         }
790         if (!opposite_edge_exists) {
791           struct edge_list_node *e = (struct edge_list_node *) malloc(sizeof(struct
          edge_list_node));
792           e->from = v;
793           e->to = adjacency->vertex;
794           e->weight = adjacency->weight;
795           e->next = NULL;
796           if (first) {
797             head = e;
798             first = 0;
799           }
800           else {
801             prev->next = e;
```

```
802         }
803         prev = e;
804       }
805       adjacency = adjacency->next;
806     }
807     put(map, (void *) v, (void *) v->adjacencies);
808     v = v->next;
809   }
810   free_map(map);
811   if (!head) {
812     return NULL;
813   }
814   return head;
815 }
816
817 /* return size of list of edge_list_node's */
818 int num_edges(void *e_head) {
819   struct edge_list_node *e = (struct edge_list_node *) e_head;
820   int count = 0;
821   while (e) {
822     count++;
823     e = e->next;
824   }
825   return count;
826 }
827
828 /* get next edge_list_node in list */
829 void *get_next_edge(void *e_in) {
830   struct edge_list_node *e = (struct edge_list_node *)e_in;
831   return e->next;
832 }
833
834 ////////////////////// END FOR EDGE ////////////////////
835
836
837 ////////////////////// BFS and DFS /////////////////////
838
839 /* allocate an array of vertex pointers of size (num_vertices + 1), and
840 store num_vertices in a dummy vertex at the first index */
841 void *get_visited_array(void *g_in) {
842   int *size = malloc(sizeof(int));
843   *size = num_vertices(g_in);
844   struct vertex_list_node **visited =
845     (struct vertex_list_node **) malloc(sizeof(struct vertex_list_node *) * (*
      size + 1));
846   memset(visited, 0, sizeof(struct vertex_list_node *) * (*size + 1));
847   /* store num nodes in the graph in the first entry in the array */
848   struct vertex_list_node *dummy_size_node =
849     (struct vertex_list_node *) malloc(sizeof(struct vertex_list_node));
850   memset(visited, 0, sizeof(struct vertex_list_node));
851   dummy_size_node->data = size;
852   visited[0] = dummy_size_node;
853   return visited;
854 }
855
856 /* check if a vertex pointer is already in the visited array */
857 int unvisited(struct vertex_list_node *v, struct vertex_list_node **visited) {
```

```
858   int size = *(int *) visited[0]->data;
859   for (int i = 1; i <= size; i++) {
860     if (visited[i] == v) {
861       return 0;
862     }
863     if (!visited[i]) {
864       break;
865     }
866   }
867   return 1;
868 }
869
870 /* add a vertex pointer to the visited array  */
871 void add_visited(struct vertex_list_node **visited, struct vertex_list_node *v) {
872   int size = *(int *) visited[0]->data;
873   for (int i = 1; i <= size; i++) {
874     if (!visited[i]) {
875       visited[i] = v;
876       return;
877     }
878   }
879 }
880
881 /////////////////// BFS ///////////////////
882
883 /* create a bfs_queue, and push the first vertex pointer onto it */
884 void *get_bfs_queue(void *first_v, void *visited) {
885   struct queue_list_node *q = malloc(sizeof(struct queue_list_node));
886   memset(q, 0, sizeof(struct queue_list_node));
887   q->v = (struct vertex_list_node *) first_v;
888   q->next = NULL;
889   add_visited(visited, q->v);
890   return q;
891 }
892
893 /* create a queue_list_node with given vertex pointer and add it to the back of
       the queue */
894 void push_queue(struct vertex_list_node *vertex, struct queue_list_node *queue) {
895   /* if empty */
896   if (!queue->v) {
897     queue->v = vertex;
898     queue->next = NULL;
899     return;
900   }
901   /*else add to end */
902   while (queue->next) {
903     queue = queue->next;
904   }
905   struct queue_list_node *new_q = (struct queue_list_node *) malloc(sizeof(struct
        queue_list_node));
906   memset(new_q, 0, sizeof(struct queue_list_node));
907   new_q->next = NULL;
908   new_q->v = vertex;
909   queue->next = new_q;
910 }
911
912 /* pop a vertex pointer from the queue */
```

```
913 void *pop_queue(struct queue_list_node *queue) {
914   struct vertex_list_node *out = queue->v;
915   struct queue_list_node *tofree = queue->next;
916   if (queue->next) {
917     queue->v = queue->next->v;
918     queue->next = queue->next->next;
919     free(tofree);
920   }
921   else {
922     queue->v = NULL;
923   }
924   return out;
925 }
926
927 void cleanup_bfs(void *visited_in, void *queue_in) {
928   struct vertex_list_node **visited = (struct vertex_list_node **) visited_in;
929   struct queue_list_node *queue = (struct queue_list_node *) queue_in;
930   free(visited[0]->data);
931   free(visited[0]);
932   free(visited_in);
933   /* if empty */
934   if (!queue->v) {
935     free(queue);
936   }
937   /*else add to end */
938   while (queue->next) {
939     void *temp = queue;
940     queue = queue->next;
941     free(temp);
942   }
943 }
944
945 /* get the next graph vertex in bfs order, updating visited array and bfs queue
      */
946 void *get_next_bfs_vertex(void *visited_in, void *queue) {
947   struct vertex_list_node **visited = (struct vertex_list_node **) visited_in;
948   struct vertex_list_node *v = pop_queue(queue);
949   /* if queue empty we are done */
950   if (!v) {
951     cleanup_bfs(visited_in, queue);
952     return NULL;
953   }
954   struct adj_list_node *adjacency = v->adjacencies;
955   while (adjacency) {
956     if (unvisited(adjacency->vertex, visited)) {
957       push_queue(adjacency->vertex, queue);
958       add_visited(visited, adjacency->vertex);
959     }
960     adjacency = adjacency->next;
961   }
962   return v;
963 }
964
965 /* test if the vertex is null to determine if bfs has finished */
966 int bfs_done(void *curr_v) {
967   if (curr_v == NULL) {
968     return 1;
```

```
969     }
970     return 0;
971   }
972
973   //////////////////// DFS ////////////////////
974
975   /* create a stack_list_node with given vertex, connect to top of stack */
976   void push_stack(struct vertex_list_node *vertex, struct stack_list_node *s) {
977     struct stack_list_node *new_s = (struct stack_list_node *) malloc(sizeof(struct
          stack_list_node));
978     memset(new_s, 0, sizeof(struct stack_list_node));
979     new_s->v = vertex;
980     new_s->next = s->next;
981     s->next = new_s;
982   }
983
984   /* pop a vertex pointer from stack */
985   void *pop_stack(struct stack_list_node *s) {
986     if (s->next) {
987       struct stack_list_node *t = s->next;
988       s->next = t->next;
989       struct vertex_list_node *out = t->v;
990       free(t);
991       return out;
992     }
993     return NULL;
994   }
995
996   /* create a dfs_stack, and push the first vertex pointer onto it */
997   void *get_dfs_stack(void *first_v, void *visited) {
998     struct vertex_list_node *vertex = (struct vertex_list_node *) first_v;
999     struct stack_list_node *s_static_top = malloc(sizeof(struct stack_list_node));
1000    memset(s_static_top, 0, sizeof(struct stack_list_node));
1001    s_static_top->v = NULL;
1002    push_stack((struct vertex_list_node *) vertex, s_static_top);
1003    return s_static_top;
1004  }
1005
1006  void cleanup_dfs(void *visited_in, void *stack_in) {
1007    struct vertex_list_node **visited = (struct vertex_list_node **) visited_in;
1008    struct stack_list_node *stack = (struct stack_list_node *) stack_in;
1009    free(visited[0]->data);
1010    free(visited[0]);
1011    free(visited_in);
1012    while (stack->next) {
1013      void *temp = stack;
1014      stack = stack->next;
1015      free(temp);
1016    }
1017  }
1018
1019  /* get the next graph vertex in bfs order, updating visited array and bfs queue
         */
1020  void *get_next_dfs_vertex(void *visited_in, void *stack) {
1021    struct vertex_list_node **visited = (struct vertex_list_node **) visited_in;
1022    struct vertex_list_node *v = pop_stack(stack);
1023    while (v && unvisited(v, visited) == 0) {
```

```
1024      v = pop_stack(stack);
1025    }
1026    /* if stack empty we are done */
1027    if (!v) {
1028      cleanup_dfs(visited_in, stack);
1029      return NULL;
1030    }
1031    add_visited(visited, v);
1032    struct adj_list_node *adjacency = v->adjacencies;
1033    while (adjacency) {
1034      push_stack(adjacency->vertex, (struct stack_list_node *) stack);
1035      adjacency = adjacency->next;
1036    }
1037    return v;
1038  }
1039
1040  /* test if the vertex is null to determine if bfs has finished */
1041  int dfs_done(void *curr_v) {
1042    if (curr_v == NULL) {
1043      return 1;
1044    }
1045    return 0;
1046  }
1047
1048  ////////////////////// END BFS and DFS //////////////////////
1049
1050  //////////////////////// PRINT ///////////////////////
1051
1052  void print_int(void *graph_ptr) {
1053    struct graph *g = (struct graph *) graph_ptr;
1054    struct vertex_list_node *vertex = g->head;
1055    while (vertex) {
1056      printf("vertex: %d\n", ((union data_type *) vertex->data)->i);
1057      printf("adjacencies: ");
1058      struct adj_list_node *adjacency = vertex->adjacencies;
1059      while (adjacency) {
1060        printf("(%d, weight: %d) ",
1061             ((union data_type *) adjacency->vertex->data)->i,
1062             adjacency->weight);
1063        adjacency = adjacency->next;
1064      }
1065      printf("\n");
1066      vertex = vertex->next;
1067    }
1068    printf("\n");
1069  }
1070
1071  void print_float(void *graph_ptr) {
1072    struct graph *g = (struct graph *) graph_ptr;
1073    struct vertex_list_node *vertex = g->head;
1074    while (vertex) {
1075      printf("vertex: %f\n", ((union data_type *) vertex->data)->f);
1076      printf("adjacencies: ");
1077      struct adj_list_node *adjacency = vertex->adjacencies;
1078      while (adjacency) {
1079        printf("(%f, weight: %d) ",
1080             ((union data_type *) adjacency->vertex->data)->f,
```

```
            adjacency->weight);
        adjacency = adjacency->next;
      }
      printf("\n");
      vertex = vertex->next;
    }
    printf("\n");
}

void print_char_ptr(void *graph_ptr) {
  struct graph *g = (struct graph *) graph_ptr;
  struct vertex_list_node *vertex = g->head;
  while (vertex) {
    printf("vertex: %s\n", ((union data_type *) vertex->data)->s);
    printf("adjacencies: ");
    struct adj_list_node *adjacency = vertex->adjacencies;
    while (adjacency) {
      printf("(%s, weight: %d) ",
           ((union data_type *) adjacency->vertex->data)->s,
           adjacency->weight);
      adjacency = adjacency->next;
    }
    printf("\n");
    vertex = vertex->next;
  }
  printf("\n");
}

void print_unweighted_int(void *graph_ptr) {
  struct graph *g = (struct graph *) graph_ptr;
  struct vertex_list_node *vertex = g->head;
  while (vertex) {
    printf("vertex: %d\n", ((union data_type *) vertex->data)->i);
    printf("adjacencies: ");
    struct adj_list_node *adjacency = vertex->adjacencies;
    while (adjacency) {
      printf("%d ", ((union data_type *) adjacency->vertex->data)->i);
      adjacency = adjacency->next;
    }
    printf("\n");
    vertex = vertex->next;
  }
  printf("\n");
}

void print_unweighted_float(void *graph_ptr) {
  struct graph *g = (struct graph *) graph_ptr;
  struct vertex_list_node *vertex = g->head;
  while (vertex) {
    printf("vertex: %f\n", ((union data_type *) vertex->data)->f);
    printf("adjacencies: ");
    struct adj_list_node *adjacency = vertex->adjacencies;
    while (adjacency) {
      printf("%f ", ((union data_type *) adjacency->vertex->data)->f);
      adjacency = adjacency->next;
    }
    printf("\n");
```

```
1138      vertex = vertex->next;
1139    }
1140    printf("\n");
1141  }
1142
1143  void print_unweighted_char_ptr(void *graph_ptr) {
1144    struct graph *g = (struct graph *) graph_ptr;
1145    struct vertex_list_node *vertex = g->head;
1146    while (vertex) {
1147      printf("vertex: %s\n", ((union data_type *) vertex->data)->s);
1148      printf("adjacencies: ");
1149      struct adj_list_node *adjacency = vertex->adjacencies;
1150      while (adjacency) {
1151        printf("%s ", ((union data_type *) adjacency->vertex->data)->s);
1152        adjacency = adjacency->next;
1153      }
1154      printf("\n");
1155      vertex = vertex->next;
1156    }
1157    printf("\n");
1158  }
1159
1160  char *get_bool_str(int val) {
1161    if (val) {
1162      return "true";
1163    } else {
1164      return "false";
1165    }
1166  }
1167
1168  void print_bool(void *graph_ptr) {
1169    struct graph *g = (struct graph *) graph_ptr;
1170    struct vertex_list_node *vertex = g->head;
1171    while (vertex) {
1172      printf("vertex: %s\n", get_bool_str(((union data_type *) vertex->data)->i));
1173      printf("adjacencies: ");
1174      struct adj_list_node *adjacency = vertex->adjacencies;
1175      while (adjacency) {
1176        printf("(%s, weight: %d) ",
1177             get_bool_str(((union data_type *) adjacency->vertex->data)->i),
1178             adjacency->weight);
1179        adjacency = adjacency->next;
1180      }
1181      printf("\n");
1182      vertex = vertex->next;
1183    }
1184    printf("\n");
1185  }
1186
1187  void print_unweighted_bool(void *graph_ptr) {
1188    struct graph *g = (struct graph *) graph_ptr;
1189    struct vertex_list_node *vertex = g->head;
1190    while (vertex) {
1191      printf("vertex: %s\n", get_bool_str(((union data_type *) vertex->data)->i));
1192      printf("adjacencies: ");
1193      struct adj_list_node *adjacency = vertex->adjacencies;
1194      while (adjacency) {
```

```
1195        printf("%s ",
1196              get_bool_str(((union data_type *) adjacency->vertex->data)->i));
1197        adjacency = adjacency->next;
1198      }
1199      printf("\n");
1200      vertex = vertex->next;
1201    }
1202    printf("\n");
1203  }
1204
1205
1206
1207  /////////////////////// TESTING ///////////////////////
1208
1209  /*void print_graph(void *graph_ptr) {
1210    struct graph *g = (struct graph *) graph_ptr;
1211    struct vertex_list_node *vertex = g->head;
1212    while (vertex) {
1213      printf("vertex: %p\n", vertex);
1214      printf("data: %p\n", vertex->data);
1215      printf("adjacencies:");
1216      struct adj_list_node *adjacency = vertex->adjacencies;
1217      while (adjacency) {
1218        printf(" %p", adjacency->vertex);
1219        adjacency = adjacency->next;
1220      }
1221      printf("\n\n");
1222      vertex = vertex->next;
1223    }
1224    printf("\n");
1225  }
1226
1227  void print_queue(struct queue_list_node *queue) {
1228    fprintf(stderr, "printing queue: ");
1229    while (queue && queue->v) {
1230      fprintf(stderr, "%d ", *(int *) queue->v->data);
1231      queue = queue->next;
1232    }
1233    printf("\n");
1234  }
1235
1236  void print_visited(struct vertex_list_node **visited) {
1237    int size = *(int *) visited[0]->data;
1238    printf("printing visited (excluding dummy size node): [");
1239    for (int i = 1; i <= size; i++) {
1240      if (visited[i]) {
1241        printf("%d, ", *(int *) visited[i]->data);
1242      }
1243      else {
1244        printf("0x0, ");
1245      }
1246    }
1247    printf("]\n");
1248  }
1249
1250  void print_edges(struct edge_list_node *e) {
1251    while (e) {
```

```
1252      printf("from: %d  to: %d weight: %d\n", *(int *) e->from->data, *(int *) e->
         to->data, e->weight);
1253      e = e->next;
1254    }
1255  }
1256
1257  void add_bidirectional_edge(void *a, void *b) {
1258    add_edge(a, b);
1259    add_edge(b, a);
1260  }
1261
1262  int main() {
1263    struct graph *g = (struct graph *) new_graph();
1264    struct graph *g2 = (struct graph *) new_graph();
1265
1266    int *new_data = malloc(sizeof(int));
1267    add_vertex(g, new_data);
1268    struct vertex_list_node *head = (struct vertex_list_node *) get_head_vertex(g);
1269    *(int *) head->data = 0;
1270
1271    int vertices = 6;
1272    int save_vertex_num = 2;
1273    struct vertex_list_node *save;
1274    struct vertex_list_node *savedarray[vertices];
1275    savedarray[0] = head;
1276
1277    for (int i = 1; i < vertices; i++) {
1278      int *new_data = malloc(sizeof(int));
1279      struct vertex_list_node *vertex = (struct vertex_list_node *) add_vertex(g,
         new_data);
1280      *(int *) vertex->data = i;
1281
1282      if (i == save_vertex_num) {
1283        save = vertex;
1284      }
1285      savedarray[i] = vertex;
1286
1287    }
1288    add_bidirectional_edge(savedarray[0], savedarray[1]);
1289    add_bidirectional_edge(savedarray[1], savedarray[2]);
1290    add_bidirectional_edge(savedarray[2], savedarray[3]);
1291    add_bidirectional_edge(savedarray[3], savedarray[4]);
1292    add_bidirectional_edge(savedarray[0], savedarray[5]);
1293    add_bidirectional_edge(savedarray[1], savedarray[5]);
1294
1295    //printf("num vertices: %d\n", num_vertices(g));
1296
1297    //printf("before entering bfs land *save->data: %d \n", *(int *) save->data);
1298
1299    struct vertex_list_node **visited = get_bfs_visited_array(g2);
1300    void *queue = get_dfs_stack(savedarray[0], visited);
1301    struct vertex_list_node *curr = get_next_dfs_vertex(visited, queue);
1302    while (curr) {
1303      printf("asdfsdf %d\n", *curr->data);
1304      curr = get_next_dfs_vertex(visited, queue);
1305    }
1306
```

```
1307    struct graph *g_nei = (struct graph *) graph_neighbors(g, savedarray[0]->data);
1308
1309    graph_set_undirected_edge_weight(g, savedarray[3]->data, savedarray[4]->data,
          50);
1310
1311    graph_set_edge_weight(g, savedarray[4]->data, savedarray[0]->data, 80);
1312
1313    graph_set_edge_weight(g, savedarray[1]->data, savedarray[2]->data, 70);
1314
1315    graph_set_undirected_edge_weight(g, savedarray[0]->data, savedarray[3]->data,
          70);
1316
1317    fprintf(stderr, "%d \n", graph_get_edge_weight(g, savedarray[0]->data,
          savedarray[1]->data));
1318
1319    fprintf(stderr, "%d \n", graph_get_edge_weight(g, savedarray[3]->data,
          savedarray[4]->data));
1320
1321
1322    //print_edges(edge_list);
1323
1324    print_data((void *) g);
1325
1326    //printf("\n%d\n", num_edges(edge_list));
1327
1328  }*/
1329
1330  ///////////////////// END TESTING /////////////////////
```

C library: graph.c

8.6 Testing

```
1   #!/bin/sh
2
3   # Regression testing script for giraph, adapted from microC
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="/usr/local/opt/llvm/bin/lli"
10
11  # Path to the LLVM compiler
12  LLC="/usr/local/opt/llvm/bin/llc"
13
14  # Path to the C compiler
15  CC="cc"
16
17  # Path to the microc compiler.  Usually "./microc.native"
18  # Try "_build/microc.native" if ocamlbuild was unable to create a symbolic link.
19  GIRAPH="./giraph.native"
20
21  # Set time limit for all operations
22  ulimit -t 30
23
```

```
24  globallog=testall.log
25  rm -f $globallog
26  error=0
27  globalerror=0
28
29  keep=0
30
31  Usage() {
32      echo "Usage: testall.sh [options] [.mc files]"
33      echo "-k    Keep intermediate files"
34      echo "-h    Print this help"
35      exit 1
36  }
37
38  SignalError() {
39      if [ $error -eq 0 ] ; then
40      echo "FAILED"
41      error=1
42      fi
43      echo "  $1"
44  }
45
46  # Compare <outfile> <reffile> <difffile>
47  # Compares the outfile with reffile.  Differences, if any, written to difffile
48  Compare() {
49      generatedfiles="$generatedfiles $3"
50      echo diff -b $1 $2 ">" $3 1>&2
51      diff -b "$1" "$2" > "$3" 2>&1 || {
52      SignalError "$1 differs"
53      echo "FAILED $1 differs from $2" 1>&2
54      }
55  }
56
57  # Run <args>
58  # Report the command, run it, and report any errors
59  Run() {
60      echo $* 1>&2
61      eval $* || {
62      SignalError "$1 failed on $*"
63      return 1
64      }
65  }
66
67  # RunFail <args>
68  # Report the command, run it, and expect an error
69  RunFail() {
70      echo $* 1>&2
71      eval $* && {
72      SignalError "failed: $* did not report an error"
73      return 1
74      }
75      return 0
76  }
77
78  Check() {
79      error=0
80      basename=‘echo $1 | sed ’s/.*\\///
```

```
81                              s/.gir//'`
82      reffile=`echo $1 | sed 's/.gir$//'`
83      basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
84  
85      echo -n "$basename..."
86  
87      echo 1>&2
88      echo "###### Testing $basename" 1>&2
89  
90      generatedfiles=""
91  
92      generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe
        ${basename}.out" &&
93      Run "$GIRAPH" "$1" ">" "${basename}.ll" &&
94      Run "$LLC" "${basename}.ll" ">" "${basename}.s" &&
95      Run "$CC" "-o" "${basename}.exe" "${basename}.s" "graph.o" &&
96      Run "./${basename}.exe" > "${basename}.out" &&
97      Compare ${basename}.out ${reffile}.out ${basename}.diff
98  
99      # Report the status and clean up the generated files
100 
101     if [ $error -eq 0 ] ; then
102     if [ $keep -eq 0 ] ; then
103         rm -f $generatedfiles
104     fi
105     echo "OK"
106     echo "###### SUCCESS" 1>&2
107     else
108     echo "###### FAILED" 1>&2
109     globalerror=$error
110     fi
111 }
112 
113 CheckFail() {
114     error=0
115     basename=`echo $1 | sed 's/.*\\///
116                             s/.gir//'`
117     reffile=`echo $1 | sed 's/.gir$//'`
118     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
119 
120     echo -n "$basename..."
121 
122     echo 1>&2
123     echo "###### Testing $basename" 1>&2
124 
125     generatedfiles=""
126 
127     generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
128     RunFail "$GIRAPH" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
129     Compare ${basename}.err ${reffile}.err ${basename}.diff
130 
131     # Report the status and clean up the generated files
132 
133     if [ $error -eq 0 ] ; then
134     if [ $keep -eq 0 ] ; then
135         rm -f $generatedfiles
136     fi
```

```
137     echo "OK"
138     echo "###### SUCCESS" 1>&2
139     else
140     echo "###### FAILED" 1>&2
141     globalerror=$error
142     fi
143 }
144
145 while getopts kdpsh c; do
146     case $c in
147     k) # Keep intermediate files
148         keep=1
149         ;;
150     h) # Help
151         Usage
152         ;;
153     esac
154 done
155
156 shift `expr $OPTIND - 1`
157
158 LLIFail() {
159   echo "Could not find the LLVM interpreter \"$LLI\"."
160   echo "Check your LLVM installation and/or modify the LLI variable in testall.sh
      "
161   exit 1
162 }
163
164 which "$LLI" >> $globallog || LLIFail
165
166 if [ ! -f graph.o ]
167 then
168     echo "Could not find graph.o"
169     echo "Try \"make graph.o\""
170     exit 1
171 fi
172
173 if [ $# -ge 1 ]
174 then
175     files=$@
176 else
177     files="tests/test-*.gir tests/fail-*.gir"
178 fi
179
180 for file in $files
181 do
182     case $file in
183     *test-*)
184         Check $file 2>> $globallog
185         ;;
186     *fail-*)
187         CheckFail $file 2>> $globallog
188         ;;
189     *)
190         echo "unknown file type $file"
191         globalerror=1
192         ;;
```

```
193     esac
194 done
195
196 exit $globalerror
```

Testing script: testall.sh

8.7 Example code

```
1  !~ Author: Seth Benjamin sjb2190 ~!
2  void augment(wedigraph<string> flow, wedigraph<string> path) {
3    !~ Get bottleneck capacity of path. ~!
4    int min = -1;
5    for_edge (e : path) {
6      if (min > e.weight() || min == -1) {
7        min = e.weight();
8      }
9    }
10
11   !~ Augment flow. ~!
12   for_edge (e : path) {
13     if (flow.has_edge(e.from(), e.to())) {
14       int current_flow = flow.get_edge_weight(e.from(), e.to());
15       !~ Add bottleneck capacity to current flow. ~!
16       flow.set_edge_weight(e.from(), e.to(), current_flow + min);
17     } else {
18       int current_flow = flow.get_edge_weight(e.to(), e.from());
19       !~ Subtract bottleneck capacity from current flow. ~!
20       flow.set_edge_weight(e.to(), e.from(), current_flow - min);
21     }
22   }
23   return;
24 }
25
26 wedigraph<string> make_residual_graph(wedigraph<string> flow, wedigraph<string>
       network) {
27   wedigraph<string> residual_graph = [];
28   for_edge (e : flow) {
29     int forward = network.get_edge_weight(e.from(), e.to()) - e.weight();
30     int backward = e.weight();
31     if (forward > 0) {
32       residual_graph.add_edge(e.from(), e.to(), forward);
33     }
34     if (backward > 0) {
35       residual_graph.add_edge(e.to(), e.from(), backward);
36     }
37   }
38   return residual_graph;
39 }
40
41 wedigraph<string> edmonds_karp(wedigraph<string> network, node<string> source,
     node<string> sink) {
42   !~ The argument   network   contains the capacities as weights on edges.
43      Flow is represented with a graph exactly equivalent to network, but
44      with the flow on each edge as the weight instead of the capacity.
45      First, set up initial flow of 0 on every edge. ~!
```

```
46    wedigraph<string> flow = [];
47    for_edge (e : network) {
48      flow.add_edge(e.from(), e.to(), 0);
49    }
50    bool has_path = true;
51    while (has_path) {
52      wedigraph<string> residual = make_residual_graph(flow, network);
53      map<node<string>> parents;
54      !~ Find shortest s-t path with BFS. ~!
55      bfs (n : residual ; source) {
56        for_node (neighbor : residual.neighbors(n)) {
57          if (!parents.contains(neighbor)) {
58            parents.put(neighbor, n);
59          }
60        }
61      }
62      !~ If we didnt reach the sink, there is no s-t path in residual ~!
63      if (!parents.contains(sink)) {
64        has_path = false;
65      } else {
66        wedigraph<string> path = [];
67        node<string> i = sink;
68        while (i != source) {
69          path.add_edge(parents.get(i), i, residual.get_edge_weight(parents.get(i
     ), i));
70          i = parents.get(i);
71        }
72        augment(flow, path);
73      }
74    }
75    return flow;
76  }
77
78  int main() {
79    wedigraph<string> network = [s:"s" -{20}-> u:"u" -{10}-> t:"t" <-{20}- v:"v"
     <-{10}- s ;
80                                   u -{30}-> v];
81    prints("network");
82    network.print();
83
84    wedigraph<string> max_flow = edmonds_karp(network, s, t);
85    prints("max flow");
86    max_flow.print();
87
88    return 0;
89  }
```

example: edmonds-karp.gir

8.8 Project Log

```
1  commit 4c7ab67b2121ebc13488cba5a18f7e701022cfd5
2  Author: Seth Benjamin <sethjbenjamin@gmail.com>
3  Date:   Wed Dec 20 19:12:22 2017 -0500
4
5      add author comments
```

```
6
7  commit 68bd47968b3480abe0677d0cac2148c7cd4820d2
8  Author: Seth Benjamin <sethjbenjamin@gmail.com>
9  Date:   Wed Dec 20 17:14:36 2017 -0500
10
11     semant check to prevent void graphs
12
13 commit 6293a3c3ca4aa49b9db19a488aa63ef0fd89c383
14 Author: Seth Benjamin <sethjbenjamin@gmail.com>
15 Date:   Wed Dec 20 16:12:42 2017 -0500
16
17     implement bool graphs, print tests
18
19 commit 8d05802da2858066e4e4e028befa3d81650e5efb
20 Author: Seth Benjamin <sethjbenjamin@gmail.com>
21 Date:   Wed Dec 20 15:46:25 2017 -0500
22
23     add all print methods
24
25 commit dfdd8a8e842be964d422067b4bc288af5476bd5e
26 Author: Seth Benjamin <sethjbenjamin@gmail.com>
27 Date:   Wed Dec 20 06:24:58 2017 -0500
28
29     edmonds-karp prints with strings
30
31 commit a9235c68a59ee5940a26e9d232c599dcd98d17f9
32 Author: Seth Benjamin <sethjbenjamin@gmail.com>
33 Date:   Wed Dec 20 06:04:23 2017 -0500
34
35     prepare edmonds karp for use with generics
36
37 commit 8371fe9044f4cfffab66e102b2fa9d91cbd3c5cf
38 Author: Seth Benjamin <sethjbenjamin@gmail.com>
39 Date:   Wed Dec 20 05:59:02 2017 -0500
40
41     fix all tests after adding generics
42
43 commit c7508c5931142d6e564182cfe8a5ae60c286b0d8
44 Author: Seth Benjamin <sethjbenjamin@gmail.com>
45 Date:   Wed Dec 20 02:25:52 2017 -0500
46
47     wow generics work! need to fix all tests and add more semant checks
48
49 commit c001a844a795002f31b8050f6c111b75be26ac4c
50 Merge: ac67c82 977600d
51 Author: Seth Benjamin <sethjbenjamin@gmail.com>
52 Date:   Tue Dec 19 19:34:08 2017 -0500
53
54     Merge branch 'master' of https://github.com/jessieliu1/giraph
55
56 commit ac67c821603a47ef2fd6a401e294d70a56602108
57 Author: Seth Benjamin <sethjbenjamin@gmail.com>
58 Date:   Tue Dec 19 19:19:41 2017 -0500
59
60     fix float operation implementation, implement map<float>
61
62 commit 977600d9b18de223ae1c087e00c55ac13118456a
```

```
63  Author: jessieliu1 <jliu997@gmail.com>
64  Date:   Tue Dec 19 19:03:15 2017 -0500
65
66      semant checks for if map type is void
67
68  commit ecea45cad731d3f4d2ce62558ac083ae4b552b47
69  Author: Seth Benjamin <sethjbenjamin@gmail.com>
70  Date:   Tue Dec 19 18:09:33 2017 -0500
71
72      implement map<bool>
73
74  commit 191a01132b3bef3d0cc701575fadcb8fa7753e49
75  Merge: 4acaf9e 07aa887
76  Author: jessieliu1 <jliu997@gmail.com>
77  Date:   Tue Dec 19 17:49:53 2017 -0500
78
79      Merge branch 'master' of https://github.com/jessieliu1/giraph
80
81  commit 4acaf9e48c9b6039519052f37e815bef1bdde187
82  Author: jessieliu1 <jliu997@gmail.com>
83  Date:   Tue Dec 19 17:49:47 2017 -0500
84
85      semant indentation consistency
86
87  commit 7c2a5965d4f0697f1be8acf98a147a57f9504431
88  Author: Seth Benjamin <sethjbenjamin@gmail.com>
89  Date:   Tue Dec 19 17:43:43 2017 -0500
90
91      implement map<string>
92
93  commit 07aa88714c09dc49204d1b7de98289f315d8cd10
94  Author: Seth Benjamin <sethjbenjamin@gmail.com>
95  Date:   Tue Dec 19 05:15:28 2017 -0500
96
97      max. flow. fucking. works.
98
99  commit 6ef1625b21186a1ba8648c1303509778591f0ce9
100 Author: Seth Benjamin <sethjbenjamin@gmail.com>
101 Date:   Tue Dec 19 01:35:31 2017 -0500
102
103     add maps for nodes, graphs
104
105 commit 35676788f931ef70af8f68715e88a2c590ec7f7b
106 Author: Seth Benjamin <sethjbenjamin@gmail.com>
107 Date:   Tue Dec 19 00:43:52 2017 -0500
108
109     implement map.contains() in codegen
110
111 commit d3a654fa96989c28ea573d2045f43eee3409118f
112 Author: Seth Benjamin <sethjbenjamin@gmail.com>
113 Date:   Tue Dec 19 00:22:47 2017 -0500
114
115     int maps work w/ put, get
116
117 commit 7ac131c582fc0bd8b59be12df228cdc5f1d4bc61
118 Author: jessieliu1 <jliu997@gmail.com>
119 Date:   Mon Dec 18 23:51:37 2017 -0500
```

```
120
121         add tests for nested funcs in edge methods
122
123     commit 310b03360ebc912912cf35c6eba437d79a91e8c6
124     Merge: a527f26 5fb2048
125     Author: jessieliu1 <jliu997@gmail.com>
126     Date:   Mon Dec 18 23:49:50 2017 -0500
127
128         Merge branch 'master' of https://github.com/jessieliu1/giraph
129
130     commit a527f26231553ff4d823fd539abc75c1c6d89526
131     Author: jessieliu1 <jliu997@gmail.com>
132     Date:   Mon Dec 18 23:49:45 2017 -0500
133
134         add env propogation to edge methods
135
136     commit 5fb2048967184d7ef0d41a28dba455297e608145
137     Author: Jennifer Bi <jb3495@columbia.edu>
138     Date:   Mon Dec 18 22:40:02 2017 -0500
139
140         fixed issue with skipping nested func calls
141
142     commit 4556ff32039d0ca26f25a20e98bcd3f74d5efd47
143     Author: Seth Benjamin <sethjbenjamin@gmail.com>
144     Date:   Mon Dec 18 21:03:20 2017 -0500
145
146         kill off printbig
147
148     commit 6975e3454d660e26bdb2bdc631dde26cceabbf8d
149     Author: Seth Benjamin <sethjbenjamin@gmail.com>
150     Date:   Mon Dec 18 20:53:04 2017 -0500
151
152         remove old test file
153
154     commit 505afcce25f233cd376894e3a9e33f56109a8269
155     Author: Seth Benjamin <sethjbenjamin@gmail.com>
156     Date:   Mon Dec 18 20:50:08 2017 -0500
157
158         add binary and, or operators to parser, scanner
159
160     commit ede59bdbc33b774c392f5eca4a5d66815f5534cf
161     Author: Seth Benjamin <sethjbenjamin@gmail.com>
162     Date:   Mon Dec 18 19:41:54 2017 -0500
163
164         actually fix fail-func error msg, oops
165
166     commit d49bb7a1cc37aff1abaa25fd7dd9b5bf06df7244
167     Author: Seth Benjamin <sethjbenjamin@gmail.com>
168     Date:   Mon Dec 18 19:38:47 2017 -0500
169
170         fix fail-func error
171
172     commit bd8d5455c3008e50077f1df3f7036f45b20902f5
173     Author: Seth Benjamin <sethjbenjamin@gmail.com>
174     Date:   Mon Dec 18 19:35:25 2017 -0500
175
176         update test-funcallgraphlit
```

```
177
178 commit 6f22ec4087819d1b13b43e21a5614f16f3070508
179 Author: Seth Benjamin <sethjbenjamin@gmail.com>
180 Date:   Mon Dec 18 19:30:09 2017 -0500
181
182     recursively handle graph lits in any expr, not just assigns
183
184 commit 27945f0d9d0a5ceb3fdbb31c20fd1c2d1b5a9519
185 Author: Seth Benjamin <sethjbenjamin@gmail.com>
186 Date:   Mon Dec 18 19:27:56 2017 -0500
187
188     add bool_lit to semant convert_expr
189
190 commit aec6973408b41728a0e21f45fbd74c8c2d0051c4
191 Author: Seth Benjamin <sethjbenjamin@gmail.com>
192 Date:   Mon Dec 18 19:08:17 2017 -0500
193
194     ret_in_loop test returns 0
195
196 commit 8827a6bba6fbe2fe3c298a75da1eca94b9f4d4b8
197 Author: jessieliu1 <jliu997@gmail.com>
198 Date:   Mon Dec 18 16:13:45 2017 -0500
199
200     test return in loop
201
202 commit 482a673918fc72c74e8dca950d9b0cdba99fed0e
203 Author: jessieliu1 <jliu997@gmail.com>
204 Date:   Mon Dec 18 16:00:44 2017 -0500
205
206     add check for returns in for loops
207
208 commit c77253927644c3a5b365529ea41bd7f2a74d395f
209 Author: jessieliu1 <jliu997@gmail.com>
210 Date:   Mon Dec 18 15:39:21 2017 -0500
211
212     add err files for test cases, tests for calls/returns in loops
213
214 commit bc620ce51f6d0e06e38ccddfc1322a24ba53fabe
215 Merge: 5a1db22 a0210f7
216 Author: Jennifer Bi <jb3495@columbia.edu>
217 Date:   Mon Dec 18 13:00:32 2017 -0500
218
219     Merge branch 'master' of https://github.com/jessieliu1/giraph
220
221 commit a0210f780fb23469cf9be9b8d66a366071135d60
222 Author: Daniel Benett <deb2174@columbia.edu>
223 Date:   Mon Dec 18 12:47:54 2017 -0500
224
225     updated undirected for edge for self loops
226
227 commit 5a1db22ec3719eeb45cfe5b6271a36b911b5be5f
228 Merge: 5408256 4e34883
229 Author: Jennifer Bi <jb3495@columbia.edu>
230 Date:   Mon Dec 18 11:24:37 2017 -0500
231
232     Merge branch 'master' of https://github.com/jessieliu1/giraph
233
```

```
234 commit 540825646b52c3a1bd87a83797eef39ae0612fac
235 Author: Jennifer Bi <jb3495@columbia.edu>
236 Date:   Mon Dec 18 11:24:28 2017 -0500
237
238     err file for for_node fail test
239
240 commit a31c62a1a2815b4d8e7e3e3587b352e483d19f14
241 Author: Jennifer Bi <jb3495@columbia.edu>
242 Date:   Mon Dec 18 11:22:33 2017 -0500
243
244     added rest of concurrent modification checks+tests
245
246 commit 7fb51c20d8ba3db71d7742ad38e7398e430776c1
247 Author: Jennifer Bi <jb3495@columbia.edu>
248 Date:   Mon Dec 18 10:48:20 2017 -0500
249
250      fixed semant match error
251
252 commit 4b86f698937df832cf0ddb033eb765afe18b1119
253 Author: Jennifer Bi <jb3495@columbia.edu>
254 Date:   Mon Dec 18 10:42:47 2017 -0500
255
256     added for_node concurrent modification check
257
258 commit 4e34883e54343dcabc425cf1a1430ce09a77a655
259 Author: Seth Benjamin <sethjbenjamin@gmail.com>
260 Date:   Mon Dec 18 07:58:58 2017 -0500
261
262     add remove edge tests
263
264 commit a11c007064296e35a5b7d92a4869a7eed025a530
265 Author: Seth Benjamin <sethjbenjamin@gmail.com>
266 Date:   Mon Dec 18 07:52:17 2017 -0500
267
268     no reinitializing nodes or reweighting edges in graph literals
269
270 commit 62eb741682ca465180bca1ca6b2df08dc1e921d6
271 Author: Seth Benjamin <sethjbenjamin@gmail.com>
272 Date:   Mon Dec 18 07:17:26 2017 -0500
273
274     print returns int
275
276 commit 622bf89c9e5133b37ec5eaad7833bdc502c67995
277 Author: Seth Benjamin <sethjbenjamin@gmail.com>
278 Date:   Mon Dec 18 07:12:07 2017 -0500
279
280     edgeless graph literals can be any graph type
281
282 commit 1a7a16d12b3892f8d211c267bda55bc3cddc5dc5
283 Author: Seth Benjamin <sethjbenjamin@gmail.com>
284 Date:   Mon Dec 18 06:47:26 2017 -0500
285
286     edge.set_weight() acts appropriately given wegraph or wedigraph
287
288 commit 86d5abdb5e8c4093ba7c95d1e28b6251b8217581
289 Author: Seth Benjamin <sethjbenjamin@gmail.com>
290 Date:   Mon Dec 18 06:12:46 2017 -0500
```

```
291
292     test dfs in 3rd self-loop test
293
294 commit ab09a76f7bc8884db939fba4b3779239cfd9ff1f
295 Author: Seth Benjamin <sethjbenjamin@gmail.com>
296 Date:   Mon Dec 18 06:10:19 2017 -0500
297
298     add tests for self-loops - two break
299
300 commit 71ad5a39fcb2f4e1338b24abbb0ae2777e1f6419
301 Author: Seth Benjamin <sethjbenjamin@gmail.com>
302 Date:   Mon Dec 18 05:47:27 2017 -0500
303
304     add has_edge, has_node and tests; fix semant bug to allow nodes to occur in
        multiple graph lits
305
306 commit 14a9f2ad7324e81be86e42fa283a766df65068bb
307 Author: Seth Benjamin <sethjbenjamin@gmail.com>
308 Date:   Mon Dec 18 04:43:34 2017 -0500
309
310     fix neighbors() segfaulting when given vertex not in graph
311
312 commit 8a5aeb5f1cc2a6f0e4c303ef9c1b542ea73d9dda
313 Author: Seth Benjamin <sethjbenjamin@gmail.com>
314 Date:   Mon Dec 18 04:32:54 2017 -0500
315
316     add_wedge_method checks if edge is present before adding
317
318 commit 1bb318c6960a4233f8baf51d96982cdfd25206ce
319 Author: Seth Benjamin <sethjbenjamin@gmail.com>
320 Date:   Mon Dec 18 04:26:21 2017 -0500
321
322     wedigraph test for get/set_edge_weight
323
324 commit c9d48ac8ebd98f9948282de4146ddb575e6c9499
325 Author: Seth Benjamin <sethjbenjamin@gmail.com>
326 Date:   Mon Dec 18 04:25:18 2017 -0500
327
328     get/setweight in codegen, tests
329
330 commit 42832aa7b36475bc2317a4facda0424b20d92b2d
331 Author: Seth Benjamin <sethjbenjamin@gmail.com>
332 Date:   Mon Dec 18 02:38:12 2017 -0500
333
334     add_edge method in codegen for weighted graphs; fix semant problems of non-
        initialized nodes and checking graph methods
335
336 commit 330e633da16d66375cd888a0a390b21e20f724b8
337 Merge: 3b66205 cd7ebf7
338 Author: Jennifer Bi <jb3495@columbia.edu>
339 Date:   Mon Dec 18 01:56:41 2017 -0500
340
341     Merge branch 'master' of https://github.com/jessieliu1/giraph
342
343 commit 3b6620557ed4b3cb88a481ab17159db947b925a5
344 Author: Jennifer Bi <jb3495@columbia.edu>
345 Date:   Mon Dec 18 01:56:31 2017 -0500
```

```
346
347     fixed last env nesting problems, + test
348
349 commit cd7ebf73a12aed19ed2b4e18271f7452ca8413ed
350 Author: Seth Benjamin <sethjbenjamin@gmail.com>
351 Date:   Mon Dec 18 01:51:54 2017 -0500
352
353     fix edge/graph method checking, add tests - all pushed tests pass at this
        point
354
355 commit c7097ed6a639f7febb40348c584c82578657e048
356 Author: Daniel Benett <deb2174@columbia.edu>
357 Date:   Mon Dec 18 01:16:20 2017 -0500
358
359     memset after malloc
360
361 commit 8d5cb9d7fba0c457a4ce0f7206c3ab441156f9a0
362 Author: Seth Benjamin <sethjbenjamin@gmail.com>
363 Date:   Mon Dec 18 01:13:43 2017 -0500
364
365     fix return type of check_edgemtd
366
367 commit a539b578043a7dad5c962d138fbaca4ad0f1753b
368 Author: Seth Benjamin <sethjbenjamin@gmail.com>
369 Date:   Mon Dec 18 01:10:39 2017 -0500
370
371     add wedge, diwedge types; semantic checking ensures weight() only called on
        those (with tests)
372
373 commit e65bc83f8cb7d21b5b7109c20074d055b54f7006
374 Author: Daniel Benett <deb2174@columbia.edu>
375 Date:   Mon Dec 18 01:02:33 2017 -0500
376
377     dfs with tests
378
379 commit 28c941c57e2949edba046d2bb1ec682f6fe9a038
380 Merge: ca0c4de 6b9e643
381 Author: Jennifer Bi <jb3495@columbia.edu>
382 Date:   Sun Dec 17 23:57:25 2017 -0500
383
384     Merge branch 'master' of https://github.com/jessieliu1/giraph
385
386 commit ca0c4de131329b608f090eeac31a59577fa2fcf4
387 Author: Jennifer Bi <jb3495@columbia.edu>
388 Date:   Sun Dec 17 23:57:06 2017 -0500
389
390     added return check and return tests
391
392 commit 6b9e6430431747f0162819448d44cd17cbd59374
393 Author: Seth Benjamin <sethjbenjamin@gmail.com>
394 Date:   Sun Dec 17 23:50:19 2017 -0500
395
396     check neighbors, get/set_edge_weight in semant
397
398 commit 53b60210b1881cf9285342198524ef82127f944e
399 Author: Jennifer Bi <jb3495@columbia.edu>
400 Date:   Sun Dec 17 23:46:13 2017 -0500
```

```
401
402     changed env propagation in checks
403
404 commit 95af9bac43774f55caa3cd4ae8ed290e2642fe40
405 Author: Seth Benjamin <sethjbenjamin@gmail.com>
406 Date:   Sun Dec 17 23:21:48 2017 -0500
407
408     fix for loops
409
410 commit 2e822b95ea28afa2dc925f9c5ee762a3d4a4268f
411 Author: Seth Benjamin <sethjbenjamin@gmail.com>
412 Date:   Sun Dec 17 23:05:55 2017 -0500
413
414     finally add gitignore
415
416 commit 2809b8fd0568e639a0dee7bf0c6250ba6d360b8c
417 Author: Seth Benjamin <sethjbenjamin@gmail.com>
418 Date:   Sun Dec 17 23:05:40 2017 -0500
419
420     for_edge works correctly for undirected graphs
421
422 commit 0b813e39256beb143190eac349d981435f0e71a8
423 Author: Daniel Benett <deb2174@columbia.edu>
424 Date:   Sun Dec 17 22:50:31 2017 -0500
425
426     add graph's edge_weight funcs
427
428 commit 4268bf1eb4d9042cb1aa02b46d6f713cc5b7dde5
429 Author: jessieliu1 <jliu997@gmail.com>
430 Date:   Sun Dec 17 22:40:30 2017 -0500
431
432     all the functions\! in the sast\!
433
434 commit 543233d3399043fcc102efb525878fa706c948e9
435 Author: Daniel Benett <deb2174@columbia.edu>
436 Date:   Sun Dec 17 22:25:40 2017 -0500
437
438     diff edge set_weight for undirected/directed
439
440 commit 71415f1d53b40497757b741d9496df6a53824ff3
441 Merge: 1acc4f8 fbd631e
442 Author: Seth Benjamin <sethjbenjamin@gmail.com>
443 Date:   Sun Dec 17 20:19:14 2017 -0500
444
445     Merge branch 'master' of https://github.com/jessieliu1/giraph
446
447 commit 1acc4f82e5852de422f334af69abeb9ace560bbd
448 Author: Seth Benjamin <sethjbenjamin@gmail.com>
449 Date:   Sun Dec 17 20:12:48 2017 -0500
450
451     codegen uses sast instead of ast
452
453 commit fbd631ef4104154f5bda4955905b1011ade34010
454 Author: Daniel Benett <deb2174@columbia.edu>
455 Date:   Sun Dec 17 19:37:24 2017 -0500
456
457     organize graph.c
```

```
458
459 commit b6a77128276a182a280a223d72b2827cf1aa50ec
460 Author: Daniel Benett <deb2174@columbia.edu>
461 Date:   Sun Dec 17 19:19:25 2017 -0500
462
463     delete map.c (moved into graph.c)
464
465 commit 171e137a780488a0aef230facbd0692ae04662c9
466 Author: Daniel Benett <deb2174@columbia.edu>
467 Date:   Sun Dec 17 19:18:37 2017 -0500
468
469     add graph neighbors method
470
471 commit 85efe80aee55d564e644f014eddc087f242e4dcf
472 Author: Seth Benjamin <sethjbenjamin@gmail.com>
473 Date:   Sun Dec 17 18:24:21 2017 -0500
474
475     fix graph method return types in semant
476
477 commit 57671da1b6a235fe48b3b4b980cc5e003b037f0a
478 Author: Seth Benjamin <sethjbenjamin@gmail.com>
479 Date:   Sun Dec 17 18:04:55 2017 -0500
480
481     printing sast
482
483 commit 22380987b0316f8a515020684888e6381ab5cfd5
484 Author: Daniel Benett <deb2174@columbia.edu>
485 Date:   Sun Dec 17 17:49:22 2017 -0500
486
487     put map into graph.c, create undirected for_edge, handle map put key
        collisions
488
489 commit e4e93af874e548cc996bd3f4fd721f56a7fd7d32
490 Merge: 19d1aaa 541d7f4
491 Author: Jennifer Bi <jb3495@columbia.edu>
492 Date:   Sun Dec 17 17:18:46 2017 -0500
493
494     Merge branch 'master' of https://github.com/jessieliu1/giraph
495
496 commit 19d1aaa76d8019834ab8057bd717ca937e125a7d
497 Author: Jennifer Bi <jb3495@columbia.edu>
498 Date:   Sun Dec 17 17:18:43 2017 -0500
499
500     change id matching to expr type check, for multiple method calls
501
502 commit 541d7f4c15a7b87a384ecb657651aa0391c597ee
503 Merge: 5608b02 eba5994
504 Author: jessieliu1 <jliu997@gmail.com>
505 Date:   Sun Dec 17 16:14:50 2017 -0500
506
507     resolve failures with list.hd in foredge tests
508
509 commit 5608b02d83b3399314bc0a2ea491427570d3d03c
510 Author: jessieliu1 <jliu997@gmail.com>
511 Date:   Sun Dec 17 16:04:02 2017 -0500
512
513     allow two method calls like .from().data()
```

```
514
515 commit eba5994076b2950a3a2e94a8c8a1b3e96581d33b
516 Merge: 0bf0725 b464054
517 Author: Jennifer Bi <jb3495@columbia.edu>
518 Date:   Sun Dec 17 15:18:42 2017 -0500
519
520     Merge branch 'master' of https://github.com/jessieliu1/giraph
521
522 commit 0bf07254c3f1bdf34730a24ed9173e3753f46287
523 Author: Jennifer Bi <jb3495@columbia.edu>
524 Date:   Sun Dec 17 15:18:31 2017 -0500
525
526     correct return types in SMethod
527
528 commit b46405422a3a15972e511c16399509511e9021af
529 Author: Daniel Benett <deb2174@columbia.edu>
530 Date:   Sun Dec 17 11:55:42 2017 -0500
531
532     add map contains_key method
533
534 commit 9c082d4d65ed4e7ab8fb016b0d8bb2a0a38c76ce
535 Author: Jennifer Bi <jb3495@columbia.edu>
536 Date:   Sun Dec 17 11:51:52 2017 -0500
537
538     added graph,edge method checks
539
540 commit 93f25186dd5ebd37e5e0b4f76cf8831a532b8f03
541 Author: Seth Benjamin <sethjbenjamin@gmail.com>
542 Date:   Sun Dec 17 08:37:30 2017 -0500
543
544     add add_node semantic check
545
546 commit c1817a9d83b27f041a11ace718b00ce98fd7a69d
547 Author: Seth Benjamin <sethjbenjamin@gmail.com>
548 Date:   Sun Dec 17 08:07:20 2017 -0500
549
550     wegraphs, wedigraphs work end-to-end
551
552 commit e50812499032d260f8b769625f28f7fee7d9f54f
553 Author: Seth Benjamin <sethjbenjamin@gmail.com>
554 Date:   Sun Dec 17 06:10:16 2017 -0500
555
556     update edmonds karp
557
558 commit 776e82dcafdf86e41aa67b7a910a956f258e0cd8
559 Author: jessieliu1 <jliu997@gmail.com>
560 Date:   Sun Dec 17 03:39:06 2017 -0500
561
562     init nodes in graph literal in semant
563
564 commit 534c14f5a459225a64423380f8583c87ab946c59
565 Author: Daniel Benett <deb2174@columbia.edu>
566 Date:   Sun Dec 17 03:25:56 2017 -0500
567
568     preparing graphs for generics
569
570 commit 35b0725fe8684751c8a778df4c6378a516d8ef71
```

```
571 Author: Daniel Benett <deb2174@columbia.edu>
572 Date:   Sun Dec 17 03:14:41 2017 -0500
573
574     reserve map location 0
575
576 commit b26e8080f681925904206e0cd79c9d44d8ec74b3
577 Author: Daniel Benett <deb2174@columbia.edu>
578 Date:   Sun Dec 17 03:00:16 2017 -0500
579
580     simple map
581
582 commit 16fa0d8a3018febca6040f124430d308b4ef1031
583 Author: Daniel Benett <deb2174@columbia.edu>
584 Date:   Sun Dec 17 02:29:14 2017 -0500
585
586     merge graph.c
587
588 commit fc0497bcbb0c96192ed1e3c22f4ecc233adc6440
589 Merge: efcfae7 c0adb84
590 Author: Daniel Benett <deb2174@columbia.edu>
591 Date:   Sun Dec 17 02:19:43 2017 -0500
592
593     merge
594
595 commit efcfae7b64637ea4d4b2a2f82a32eb15586c7eba
596 Author: Daniel Benett <deb2174@columbia.edu>
597 Date:   Sun Dec 17 02:16:42 2017 -0500
598
599     edge weights graph.c and codegen
600
601 commit c0adb842ab3f292d5a4148bef870d75240125581
602 Author: Seth Benjamin <sethjbenjamin@gmail.com>
603 Date:   Sun Dec 17 00:40:46 2017 -0500
604
605     only allocate new node if not immediately initialized to another
606
607 commit 00494c9872e9f6f72bcf246f9724fef71627e888
608 Author: Seth Benjamin <sethjbenjamin@gmail.com>
609 Date:   Sun Dec 17 00:32:21 2017 -0500
610
611     remove unnecessary edge functions
612
613 commit c48fb7dfa52e95dbf6bb57cf11276c04763757ac
614 Author: Daniel Benett <deb2174@columbia.edu>
615 Date:   Sat Dec 16 23:28:49 2017 -0500
616
617     working for_edge iteration
618
619 commit 1661ff7adc92cba6299e81b3c5f912b4494feca6
620 Merge: 1b55381 9ae827b
621 Author: Seth Benjamin <sethjbenjamin@gmail.com>
622 Date:   Sat Dec 16 23:07:56 2017 -0500
623
624     resolve merge conflicts
625
626 commit 1b553813d2ea2d7d26b2eb0462b1069dc2248f4f
627 Author: Seth Benjamin <sethjbenjamin@gmail.com>
```

```
628 Date:   Sat Dec 16 23:01:38 2017 -0500
629
630     digraphs work end to end
631
632 commit 8bd2620dc2922c6bf00b19b8d6211bffb1e40093
633 Author: Seth Benjamin <sethjbenjamin@gmail.com>
634 Date:   Sat Dec 16 22:03:06 2017 -0500
635
636     fix digraphs in parser
637
638 commit 9ae827bf0a3f6eb026cf2fe84b2123849811880e
639 Merge: ec03c15 bc87447
640 Author: jessieliu1 <jliu997@gmail.com>
641 Date:   Sat Dec 16 20:35:53 2017 -0500
642
643     Merge branch 'master' of https://github.com/jessieliu1/giraph
644
645 commit ec03c15e0abed6eee0c0aa1c06baadb2f7d9fa2c
646 Author: jessieliu1 <jliu997@gmail.com>
647 Date:   Sat Dec 16 20:35:45 2017 -0500
648
649     fix error with reinit
650
651 commit bc874473a2538da70e1e075cc3fffde06c08aa29
652 Author: Seth Benjamin <sethjbenjamin@gmail.com>
653 Date:   Sat Dec 16 19:58:42 2017 -0500
654
655     make testall check failing tests, add fail-vassign test
656
657 commit 8c640e1c4a71455e2c675b464fc61007243f11c2
658 Merge: 0df4e9c 3f0e888
659 Author: jessieliu1 <jliu997@gmail.com>
660 Date:   Sat Dec 16 19:17:27 2017 -0500
661
662     pulled?
663
664 commit 0df4e9cd9081991de187eb9dde243c5abe144da6
665 Merge: 744e5a7 0537793
666 Author: jessieliu1 <jliu997@gmail.com>
667 Date:   Sat Dec 16 19:15:31 2017 -0500
668
669     merge on my end
670
671 commit 3f0e88813c3a0c20db5a833fadc0106052945847
672 Merge: 0537793 a318a3a
673 Author: Jennifer Bi <jb3495@columbia.edu>
674 Date:   Sat Dec 16 19:05:52 2017 -0500
675
676     merge for real\!
677
678 commit 0537793119d423e915cbae1cee1421efee665a56
679 Author: Daniel Benett <deb2174@columbia.edu>
680 Date:   Sat Dec 16 16:57:27 2017 -0500
681
682     added some comments
683
684 commit a318a3aac041d8bc5091951bdc394818c36e627a
```

```
685 Author: jessieliu1 <jliu997@gmail.com>
686 Date:   Sat Dec 16 16:39:18 2017 -0500
687
688     fix to get sexpr for graphs
689
690 commit 1db15254635270b5879e338e71cbde09d057478f
691 Author: Daniel Benett <deb2174@columbia.edu>
692 Date:   Sat Dec 16 16:36:38 2017 -0500
693
694     working bfs with tests
695
696 commit b176f1f6855b06197f5da939cad54a6cc551a96b
697 Author: Jennifer Bi <jb3495@columbia.edu>
698 Date:   Sat Dec 16 14:41:33 2017 -0500
699
700     parsing for undirected graphs
701
702 commit a47b94ed9ba3fa2c51726801c026e063984c2336
703 Author: Jennifer Bi <jb3495@columbia.edu>
704 Date:   Sat Dec 16 13:58:36 2017 -0500
705
706     added parser helper fns and directed graphs
707
708 commit 389fd97b665a827963aebfb4e3ce8b2728b77a47
709 Merge: 4ba534a 9d03399
710 Author: Jennifer Bi <jb3495@columbia.edu>
711 Date:   Sat Dec 16 10:52:13 2017 -0500
712
713     Merge branch 'working' of https://github.com/jessieliu1/giraph into working
714
715 commit 4ba534a2ba18252ba1422d36142b95c23958cec6
716 Author: Jennifer Bi <jb3495@columbia.edu>
717 Date:   Sat Dec 16 10:51:33 2017 -0500
718
719     type inference/node decl checking for graphs
720
721 commit 1b18aaffc70d3c729cb208790230125f99407d65
722 Author: Seth Benjamin <sethjbenjamin@gmail.com>
723 Date:   Sat Dec 16 06:15:24 2017 -0500
724
725     add graph.remove_edge() method
726
727 commit ffc9b985627c1a8172cb4bf123938fec69ccf54f
728 Author: Seth Benjamin <sethjbenjamin@gmail.com>
729 Date:   Sat Dec 16 05:44:33 2017 -0500
730
731     add graph.add_edge() method
732
733 commit 976afbc1fec5cc41f24d5f8f66ea55dd85abd114
734 Author: Seth Benjamin <sethjbenjamin@gmail.com>
735 Date:   Sat Dec 16 05:11:53 2017 -0500
736
737     add graph.remove_node() method with tests
738
739 commit 5dbac7e0faba2ce044500ddab82306f2ab863a60
740 Author: Seth Benjamin <sethjbenjamin@gmail.com>
741 Date:   Sat Dec 16 05:11:23 2017 -0500
```

```
742
743     test for add_node when adding duplicate node
744
745 commit 1c917fbeebf322eb6c827241b48870abf9bdf3f8
746 Author: Seth Benjamin <sethjbenjamin@gmail.com>
747 Date:   Sat Dec 16 03:53:27 2017 -0500
748
749     add_node only adds if node is not already present
750
751 commit a10792e6922439b758ce178f53278e73e1ce79a7
752 Author: Seth Benjamin <sethjbenjamin@gmail.com>
753 Date:   Sat Dec 16 02:33:05 2017 -0500
754
755     graph.add_node() works
756
757 commit 53c92d2490ea7ef6ba72b7cc6dd5c61ba83b96e3
758 Author: Seth Benjamin <sethjbenjamin@gmail.com>
759 Date:   Sat Dec 16 02:12:44 2017 -0500
760
761     methods are called on expressions not strings
762
763 commit f4d18e9ffd08f3988d68b66e61688b5ec20911f3
764 Author: Seth Benjamin <sethjbenjamin@gmail.com>
765 Date:   Sat Dec 16 01:23:20 2017 -0500
766
767     all data is malloc'd c-side so it doesn't die when node names go out of scope
768
769 commit 9d03399c6d5273b04583c8be858be80216653181
770 Author: jessieliu1 <jliu997@gmail.com>
771 Date:   Sat Dec 16 00:29:17 2017 -0500
772
773     added: can't do reinitialization of vars
774
775 commit 3d624667dcc30c2414486cf1f41b8aad3c97c83f
776 Author: jessieliu1 <jliu997@gmail.com>
777 Date:   Sat Dec 16 00:07:30 2017 -0500
778
779     add lt gt neq leq to parser
780
781 commit 4e5087aa0e4c86ec2170f88c12ea2f949775080a
782 Author: Jennifer Bi <jb3495@columbia.edu>
783 Date:   Fri Dec 15 23:58:33 2017 -0500
784
785     merge master into working
786
787 commit 8d5944b180aca52e51738671eb863409555e3e60
788 Author: Seth Benjamin <sethjbenjamin@gmail.com>
789 Date:   Fri Dec 15 18:13:50 2017 -0500
790
791     for_node, node data tests
792
793 commit 0e97eb520b4dcf93cb01fde23940514cf6bbc07f
794 Author: Seth Benjamin <sethjbenjamin@gmail.com>
795 Date:   Fri Dec 15 17:33:34 2017 -0500
796
797     for_node works
798
```

```
799 commit c00aa4592b4e848709cedc3571a1161aa2d16075
800 Merge: 345ea8a c2f92a4
801 Author: Daniel Benett <deb2174@columbia.edu>
802 Date:   Fri Dec 15 16:22:47 2017 -0500
803
804     Merge branch 'master' of https://github.com/jessieliu1/giraph
805
806 commit 345ea8ae749d3e11716c4d8484d8e89ef72d8a75
807 Author: Daniel Benett <deb2174@columbia.edu>
808 Date:   Fri Dec 15 16:21:22 2017 -0500
809
810     new methods in graph.c
811
812 commit c2f92a4464494c4c0f031452735e0e1a247262c3
813 Author: Seth Benjamin <sethjbenjamin@gmail.com>
814 Date:   Fri Dec 15 04:16:23 2017 -0500
815
816     update testall to use graph.o, add test for scoped blocks
817
818 commit 16bc7d7a70ab51ac0b99c5d53fb9808df388a272
819 Author: Seth Benjamin <sethjbenjamin@gmail.com>
820 Date:   Fri Dec 15 03:31:32 2017 -0500
821
822     all bracketed blocks get their own scope in codegen
823
824 commit ae472b1cd4ab725fd5aad3534cf3e37012e6714e
825 Author: Seth Benjamin <sethjbenjamin@gmail.com>
826 Date:   Fri Dec 15 03:30:36 2017 -0500
827
828     add neq token
829
830 commit 92a0fa9f3265b494eefa09a4e596329f060f883e
831 Author: Seth Benjamin <sethjbenjamin@gmail.com>
832 Date:   Fri Dec 15 03:28:45 2017 -0500
833
834     remove check for printbig.o in testall.sh
835
836 commit 64e10ab944de10ccad518af8bf1ef0b4d632cb6c
837 Merge: ab22773 5edbb70
838 Author: jessieliu1 <jliu997@gmail.com>
839 Date:   Fri Dec 15 00:37:19 2017 -0500
840
841     Merge branch 'working' of https://github.com/jessieliu1/giraph into working
842
843 commit ab2277399b190e7ec1eb7c7829aaea87d3d9e387
844 Author: jessieliu1 <jliu997@gmail.com>
845 Date:   Fri Dec 15 00:37:10 2017 -0500
846
847     not quite good with return statements
848
849 commit 86e1d299ebf1134221518792b504313d81014e83
850 Author: Seth Benjamin <sethjbenjamin@gmail.com>
851 Date:   Thu Dec 14 23:29:28 2017 -0500
852
853     testing setting, accessing node data
854
855 commit 5451ef69e0ef91442f0aa94587649726f2cb2725
```

```
856 Author: Seth Benjamin <sethjbenjamin@gmail.com>
857 Date:   Thu Dec 14 23:29:05 2017 -0500
858
859     node methods data(), set_data() work
860
861 commit 6f9bc5bb926805d399acb429f3afcd80f10562e5
862 Merge: 3f011b8 4d569e6
863 Author: Jennifer Bi <jb3495@columbia.edu>
864 Date:   Thu Dec 14 23:17:10 2017 -0500
865
866     Merge branch 'master' of https://github.com/jessieliu1/giraph
867
868 commit 3f011b85310b4c8bc3899033fcffd0fdf99225a1
869 Author: Jennifer Bi <jb3495@columbia.edu>
870 Date:   Thu Dec 14 23:13:58 2017 -0500
871
872     Revert "merge branches, changed Graph to Graph_Lit in sast"
873
874     This reverts commit c7aa0534f6c2de3a9ec1f31dfb987425eefe18e9, reversing
875     changes made to 1b9a18b43ca6b9e0590ce10e4151dfa52d2c9e6d.
876
877 commit 5edbb7005409aed80733f8c52668a439b56c5cca
878 Merge: 639c2d3 33e6a13
879 Author: Jennifer Bi <jb3495@columbia.edu>
880 Date:   Thu Dec 14 23:07:08 2017 -0500
881
882     Merge branch 'master' into working
883
884 commit 33e6a134e83a112406843420a39cdeb2d1de0c8c
885 Author: Jennifer Bi <jb3495@columbia.edu>
886 Date:   Thu Dec 14 22:39:30 2017 -0500
887
888     fixed graph_lit sexpr/expr problem
889
890 commit 4d569e6930a5e1fa0de0a169fa8bb9aa28860fb0
891 Author: Daniel Benett <deb2174@columbia.edu>
892 Date:   Thu Dec 14 22:24:36 2017 -0500
893
894     merge fixing (remove NodeTyp/EdgeTyp)
895
896 commit 639c2d3bd154ad890b65bcdc0d370b27cb25befc
897 Author: jessieliu1 <jliu997@gmail.com>
898 Date:   Thu Dec 14 21:28:46 2017 -0500
899
900     fixed block failure, weird vassign failure though
901
902 commit c7aa0534f6c2de3a9ec1f31dfb987425eefe18e9
903 Merge: 1b9a18b 4e38f6a
904 Author: Jennifer Bi <jb3495@columbia.edu>
905 Date:   Thu Dec 14 20:01:22 2017 -0500
906
907     merge branches, changed Graph to Graph_Lit in sast
908
909 commit 4e38f6a78e0739e7970b9019dd46552920e21772
910 Author: jessieliu1 <jliu997@gmail.com>
911 Date:   Thu Dec 14 16:10:25 2017 -0500
912
```

```
913      remove newsemant
914
915  commit 800580fc92eb0df17b5ffaf8726c9685c16c0c19
916  Author: jessieliu1 <jliu997@gmail.com>
917  Date:   Thu Dec 14 16:07:55 2017 -0500
918
919      everything makes
920
921  commit cf65f92b3fefdb75ba049bb435c1a3583df07e64
922  Author: jessieliu1 <jliu997@gmail.com>
923  Date:   Thu Dec 14 15:47:27 2017 -0500
924
925      this semant compiles
926
927  commit 1b9a18b43ca6b9e0590ce10e4151dfa52d2c9e6d
928  Author: Seth Benjamin <sethjbenjamin@gmail.com>
929  Date:   Thu Dec 14 03:53:48 2017 -0500
930
931      when parsing graph literals, only add edge if not already present
932
933  commit 74aea20187d6cfe16826c1a7615876360d2f5c07
934  Author: Seth Benjamin <sethjbenjamin@gmail.com>
935  Date:   Thu Dec 14 03:38:46 2017 -0500
936
937      supports multiple components in a graph literal, separated by semicolons
938
939  commit ea7cf2bc220660aa96daa49de7df7fcdb2dcc5bd
940  Author: Seth Benjamin <sethjbenjamin@gmail.com>
941  Date:   Thu Dec 14 02:07:14 2017 -0500
942
943      remove 'function' from scanner/parser
944
945  commit 9559d79ef96abababaa7b677c3f50a07c4d03a43f9
946  Author: jessieliu1 <jliu997@gmail.com>
947  Date:   Thu Dec 14 01:15:41 2017 -0500
948
949      all the statement conversions
950
951  commit 4b2424d9624a122c99677b1c9fe1d649030da612
952  Author: jessieliu1 <jliu997@gmail.com>
953  Date:   Thu Dec 14 01:13:38 2017 -0500
954
955      add block checking
956
957  commit 6ca882e2e27719c6823af85619bd52496f64f634
958  Author: jessieliu1 <jliu997@gmail.com>
959  Date:   Thu Dec 14 01:08:44 2017 -0500
960
961      this semant compiles
962
963  commit af61a0c274867860bd92d49128a166f731b6492f
964  Author: Seth Benjamin <sethjbenjamin@gmail.com>
965  Date:   Thu Dec 14 00:51:32 2017 -0500
966
967      remove node and edge expr from ast, change graph expr to graph_lit
968
969  commit f0b65c2cda3d8ab43000604336ff40b568971dc1
```

```
970 Author: jessieliu1 <jliu997@gmail.com>
971 Date:   Thu Dec 14 00:32:27 2017 -0500
972
973     resolved formals type error with fdecls
974
975 commit 98906658506a332647450c478a4c7359417afb06
976 Author: jessieliu1 <jliu997@gmail.com>
977 Date:   Wed Dec 13 14:00:51 2017 -0500
978
979     need to fix weird sfdecl formal thing
980
981 commit ac42ad96b3a3fe2942384dedd822bd69e9107544
982 Merge: dee4742 b98e603
983 Author: jessieliu1 <jliu997@gmail.com>
984 Date:   Wed Dec 13 13:29:43 2017 -0500
985
986     debugging node type errors
987
988 commit dee4742a9c94a2b5b9590bd7606b8da7cac1df0d
989 Author: jessieliu1 <jliu997@gmail.com>
990 Date:   Wed Dec 13 13:11:54 2017 -0500
991
992     add exception
993
994 commit 0e84dc38b54c741617274d0b7636a7dbec74b4b1
995 Author: Seth Benjamin <sethjbenjamin@gmail.com>
996 Date:   Sun Dec 10 06:59:36 2017 -0500
997
998     initializing node data in graph literals works in codegen
999
1000 commit b98e6033171b6e020868e71b90bf0beb2d74f792
1001 Author: Jennifer Bi <jb3495@columbia.edu>
1002 Date:   Sun Dec 10 02:26:13 2017 -0500
1003
1004     added raise Failure functions
1005
1006 commit 5705c9f2b1551fe48bc2c7ad2decbbee641fbd64
1007 Merge: 21db32b f7415f6
1008 Author: Jennifer Bi <jb3495@columbia.edu>
1009 Date:   Sun Dec 10 02:11:49 2017 -0500
1010
1011     more sast print fixes
1012
1013 commit 21db32b422607f90740c09fee1d89f53737927cc
1014 Author: Jennifer Bi <jb3495@columbia.edu>
1015 Date:   Sun Dec 10 02:10:12 2017 -0500
1016
1017     more sast print fixes
1018
1019 commit f7415f6f0852b827e212ed1f1156710d630620a5
1020 Author: jessieliu1 <jliu997@gmail.com>
1021 Date:   Sun Dec 10 01:00:31 2017 -0500
1022
1023     sast print fixes
1024
1025 commit 9d80a930db2bdbde0ffbb733073243012f72f4f0
1026 Merge: 83552bf e582101
```

```
1027 Author: Jennifer Bi <jb3495@columbia.edu>
1028 Date:   Sun Dec 10 00:39:52 2017 -0500
1029
1030     added convert_ast function; resolve merge conflicts
1031
1032 commit 83552bf5f5b760652b823854f7b9a86d75c835af
1033 Author: Jennifer Bi <jb3495@columbia.edu>
1034 Date:   Sun Dec 10 00:36:52 2017 -0500
1035
1036     added convert_ast function; updated pretty-print
1037
1038 commit e582101e97ca5ffd26bbe20ef69efdab8b077b55
1039 Author: jessieliu1 <jliu997@gmail.com>
1040 Date:   Sun Dec 10 00:20:52 2017 -0500
1041
1042     finishing statement checks
1043
1044 commit e57a3d6665b4ac0cc98a0bb39a746fe93ab8229c
1045 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1046 Date:   Sat Dec 9 22:13:51 2017 -0500
1047
1048     scanning and parsing graphs with data
1049
1050 commit e2709e577de39698b128a82ae75af21ee7cbc569
1051 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1052 Date:   Sat Dec 9 21:44:45 2017 -0500
1053
1054     graph literals now surrounded by square brackets; single-node graphs work
1055
1056 commit e43fab55708620367439d1a4f58ed892dc7689f2
1057 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1058 Date:   Sat Dec 9 19:09:48 2017 -0500
1059
1060     small cleanup of graph codegen, fix node iterators in ast and parser
1061
1062 commit fe0aa9bb3f83613bb5de7c80015f8b152c297347
1063 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1064 Date:   Sat Dec 9 18:49:45 2017 -0500
1065
1066     graphs with cycles now work (still need to do parentheses and comma syntax)
1067
1068 commit e7a3323ceef1cbe7ad5bb845ceab43af30bebfb9
1069 Author: jessieliu1 <jliu997@gmail.com>
1070 Date:   Sat Dec 9 03:36:31 2017 -0500
1071
1072     round out the stmts
1073
1074 commit 18fe414983a610417cdb5c3ae45af396fc3005b4
1075 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1076 Date:   Sat Dec 9 02:24:18 2017 -0500
1077
1078     process and allocate nodes of graph literals as local variables
1079
1080 commit 82c8e6a9f914b9457e576f1f4e2bebdafb7922a4
1081 Author: jessieliu1 <jliu997@gmail.com>
1082 Date:   Thu Dec 7 03:12:07 2017 -0500
1083
```

```
1084         not quite compiled, but building semant.ml
1085
1086     commit a54bfac30c3a98fb0029a874d7e628e8c98f7d9b
1087     Author: Seth Benjamin <sethjbenjamin@gmail.com>
1088     Date:   Sat Dec 2 07:49:53 2017 -0500
1089
1090         graph codegen works for simple undirected graphs without data
1091
1092     commit ccd38f628a15f1e03624da4f8aa767b9426af5d9
1093     Author: Seth Benjamin <sethjbenjamin@gmail.com>
1094     Date:   Sat Dec 2 05:07:13 2017 -0500
1095
1096         begin C api for graph codegen
1097
1098     commit 4fb044aabb0d21031445adb4a9d1f6da53f8886d
1099     Author: Seth Benjamin <sethjbenjamin@gmail.com>
1100     Date:   Sat Dec 2 05:05:09 2017 -0500
1101
1102         check command line args of toexe.sh
1103
1104     commit 5557a6f5cb5d8e37bac7e2d641c475758060ac65
1105     Author: Seth Benjamin <sethjbenjamin@gmail.com>
1106     Date:   Sat Dec 2 03:35:57 2017 -0500
1107
1108         fix whitespace and pretty-printing of vdecl in ast
1109
1110     commit 321e3753d340f47c912fb6fe45f4847142d95b71
1111     Author: jessieliu1 <jliu997@gmail.com>
1112     Date:   Thu Nov 30 01:48:25 2017 -0500
1113
1114         add a lil more to env
1115
1116     commit e8316aabd6f9606e584cab52c5749f8487354df6
1117     Author: jessieliu1 <jliu997@gmail.com>
1118     Date:   Wed Nov 29 03:06:13 2017 -0500
1119
1120         add env into semant, changes to sast and formatting for codegen/parser
1121
1122     commit 404cababc18221189b9a4c20cb9689a47bf1f2f5
1123     Author: jessieliu1 <jliu997@gmail.com>
1124     Date:   Tue Nov 28 16:28:57 2017 -0500
1125
1126         more formatting
1127
1128     commit f785077d5a09e14e1e8d552f3bb1db6243748108
1129     Author: jessieliu1 <jliu997@gmail.com>
1130     Date:   Tue Nov 28 16:26:42 2017 -0500
1131
1132         some formatting
1133
1134     commit 13e8ba3354f45710580ed81ea601b06381cfaa7a
1135     Author: jenniferbi <jb3495@columbia.edu>
1136     Date:   Mon Nov 27 20:23:14 2017 -0500
1137
1138         skeleton for sast
1139
1140     commit 95ac6cf360d5919cbd5a496b240b698320bd10d0
```

```
1141 Merge: a359d0a 18e7059
1142 Author: jessieliu1 <jliu997@gmail.com>
1143 Date:   Mon Nov 27 15:26:56 2017 -0500
1144
1145     Merge branch 'working' of https://github.com/jessieliu1/giraph into working
1146
1147 commit a359d0a17c0e6e2a0f003b2c27fdd6df895a8155
1148 Author: jessieliu1 <jliu997@gmail.com>
1149 Date:   Mon Nov 27 15:26:43 2017 -0500
1150
1151     add failing tests with vassign
1152
1153 commit 18e70591df0b39c037a271fbfe0f36d7a3595813
1154 Author: Jennifer Bi <jb3495@columbia.edu>
1155 Date:   Sun Nov 26 14:27:05 2017 -0500
1156
1157     declare prints fn to fix unrecognized fn exception
1158
1159 commit 744e5a766c58cc52031cedd011f436f24c5a1ff0
1160 Author: jessieliu1 <jliu997@gmail.com>
1161 Date:   Sat Nov 25 19:29:14 2017 -0500
1162
1163     some formatting
1164
1165 commit 2917af2264c967f6d4fb2d15ab5458f82decfd7b
1166 Author: Jennifer Bi <jb3495@columbia.edu>
1167 Date:   Sat Nov 25 19:25:33 2017 -0500
1168
1169     minor changes, syntax
1170
1171 commit 3bef1a8d9f7681cd3e740e76255612e8a48f31ee
1172 Author: Jennifer Bi <jb3495@columbia.edu>
1173 Date:   Sat Nov 25 19:20:05 2017 -0500
1174
1175     semantic checking adapted from microc
1176
1177 commit 3dc67082a2ea9eb896d4fbacaab7a3e31d342437
1178 Author: jessieliu1 <jliu997@gmail.com>
1179 Date:   Sat Nov 25 11:42:18 2017 -0500
1180
1181     actually resolve all merge conflicts
1182
1183 commit b8c1e04fb38520d3138207cbcc37fb51cee2a84b
1184 Merge: ec08127 d281b53
1185 Author: jessieliu1 <jliu997@gmail.com>
1186 Date:   Sat Nov 25 11:00:29 2017 -0500
1187
1188     resolve merge conflicts in parse and codegen
1189
1190 commit d281b537a229ea2c17033db2ea83ac02b7ab911f
1191 Author: Jennifer Bi <jb3495@columbia.edu>
1192 Date:   Sat Nov 25 01:16:56 2017 -0500
1193
1194     one line var declaration and assignment
1195
1196 commit a74ebc760f6673e4b38d88d1a92bb3d758fd0b6e
1197 Author: Seth Benjamin <sethjbenjamin@gmail.com>
```

```
1198 Date:   Mon Nov 20 03:49:40 2017 -0500
1199
1200     correctly parsing and printing simple one-line undirected graphs
1201
1202 commit ec08127698f4d0ff3067072574f70d7fb1cbfaa9
1203 Merge: 7873670 0350777
1204 Author: jessieliu1 <jliu997@gmail.com>
1205 Date:   Sat Nov 18 01:14:36 2017 -0500
1206
1207     merging conflicts in parser
1208
1209 commit 03507779f0f3b9459bbbdcb33061de92add33500
1210 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1211 Date:   Sat Nov 18 01:10:52 2017 -0500
1212
1213     remove vdecls from fdecl; instead, local vars declared and parsed as
         statements in fnct body
1214
1215 commit 795d7f42d355e03d98dbcbd917f3a2d5060d7179
1216 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1217 Date:   Fri Nov 17 21:57:43 2017 -0500
1218
1219     begin parsing graphs
1220
1221 commit eefc0afbd222210ff6c030789c528eb90e4c6795
1222 Author: Daniel Benett <deb2174@columbia.edu>
1223 Date:   Sat Nov 11 15:53:28 2017 -0500
1224
1225     rm toexe.sh in tests dir
1226
1227 commit 7435bfed97c06ba73ae4fe16a9876143e9d6f51f
1228 Author: Daniel Benett <deb2174@columbia.edu>
1229 Date:   Sat Nov 11 15:47:00 2017 -0500
1230
1231     remove a comment
1232
1233 commit 7873670f5d11b0aeb91267b17c52f27e14bd69c5
1234 Merge: 4748fb1 a4cd358
1235 Author: jessieliu1 <jliu997@gmail.com>
1236 Date:   Sat Nov 11 15:45:22 2017 -0500
1237
1238     Merge branch 'master' of https://github.com/jessieliu1/giraph
1239
1240 commit 4748fb1f83493e38c661a322df3430367192038f
1241 Author: jessieliu1 <jliu997@gmail.com>
1242 Date:   Sat Nov 11 15:45:19 2017 -0500
1243
1244     some format cleanup
1245
1246 commit a4cd358c35e60907dbe3ad72a0cb0f817bcbe255
1247 Author: Daniel Benett <deb2174@columbia.edu>
1248 Date:   Sat Nov 11 15:21:58 2017 -0500
1249
1250     toexe.sh script, run with './toexe.sh tests/test-name.gir'
1251
1252 commit 124a0188b42c97fee788aeb6d4c67ac1cae8e4d3
1253 Author: Jennifer Bi <jb3495@columbia.edu>
```

```
1254 Date:   Sat Nov 11 14:48:49 2017 -0500
1255
1256     added regression test script and test files
1257
1258 commit 2b5bae2f026cb379821bd3ca37d0f1925a33f91c
1259 Author: Daniel Benett <deb2174@columbia.edu>
1260 Date:   Mon Oct 30 22:45:07 2017 -0400
1261
1262     add printbig.c
1263
1264 commit efb5712c99bf42f7fd91c9dbdec7773b45453df1
1265 Author: Daniel Benett <deb2174@columbia.edu>
1266 Date:   Mon Oct 30 22:40:41 2017 -0400
1267
1268     Hello, World
1269
1270 commit a85064b6a33c5858000b976eaf650ed48e40e85b
1271 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1272 Date:   Mon Oct 30 22:05:46 2017 -0400
1273
1274     add int,string,float,bool literals
1275
1276 commit b80e3045920a8934a3dcc5e0d528da1978d97597
1277 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1278 Date:   Mon Oct 30 04:20:45 2017 -0400
1279
1280     add hello_world test
1281
1282 commit bfb472e363fcf14d9e6288594601d08fade396cc
1283 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1284 Date:   Mon Oct 30 04:16:54 2017 -0400
1285
1286     add print0 test source
1287
1288 commit 2295ddff6981d562d3fee83f12bdd2ed375ddf5d
1289 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1290 Date:   Mon Oct 30 02:44:23 2017 -0400
1291
1292     fix order of token rule in scanner - codegen for return.gir works
1293
1294 commit dc047a76ffcbf7339d31200c5f9ad7714b9c816d
1295 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1296 Date:   Mon Oct 30 02:36:26 2017 -0400
1297
1298     remove char type
1299
1300 commit ce5ea36d181bd50522450d479ef145ede6783fc7
1301 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1302 Date:   Mon Oct 30 02:09:26 2017 -0400
1303
1304     fix empty token error by parsing IDs in scanner
1305
1306 commit a7be6a8e8c347d8209b520b2cab381d932fc44b0
1307 Merge: a88fc3b 24bb10b
1308 Author: Jennifer Bi <jb3495@columbia.edu>
1309 Date:   Sun Oct 29 12:07:04 2017 -0400
1310
```

```
1311      Merge branch 'master' of https://github.com/jessieliu1/giraph
1312
1313  commit a88fc3ba1e30e033a07b3756efaf0a0606b062d0
1314  Author: Jennifer Bi <jb3495@columbia.edu>
1315  Date:   Sun Oct 29 10:43:51 2017 -0400
1316
1317      updated giraph.ml skeleton
1318
1319  commit 24bb10be158e8f20ab5339be38b6918e9d7e87e4
1320  Author: Jennifer Bi <jb3495@columbia.edu>
1321  Date:   Sun Oct 29 10:43:51 2017 -0400
1322
1323      updated codegen
1324
1325  commit 71f65d4a54f0ff32b90657eeab8a729ca7f807c5
1326  Author: Daniel Benett <deb2174@columbia.edu>
1327  Date:   Sat Oct 28 18:09:53 2017 -0400
1328
1329      skeleton for end-to-end codegen
1330
1331  commit 42c1040a8167db0686965649a33beb68ebc9bd2d
1332  Merge: 3786ce7 e434213
1333  Author: jessieliu1 <jliu997@gmail.com>
1334  Date:   Sat Oct 28 15:42:04 2017 -0400
1335
1336      Merge branch 'master' of https://github.com/jessieliu1/giraph
1337
1338  commit 3786ce75707996b1899a847b463c5aa0d61b8a2b
1339  Author: jessieliu1 <jliu997@gmail.com>
1340  Date:   Sat Oct 28 15:41:46 2017 -0400
1341
1342      remove Int_Lit from ast
1343
1344  commit e43421375d5cfe968d6e48387fe8d240927c540a
1345  Author: Daniel Benett <deb2174@columbia.edu>
1346  Date:   Sat Oct 21 20:41:36 2017 -0400
1347
1348      remove functions and fn_list
1349
1350  commit 597a87b8a72e2b7f26843ca97bf34a441c5bdcf0
1351  Author: jessieliu1 <jliu997@gmail.com>
1352  Date:   Sat Oct 21 20:31:16 2017 -0400
1353
1354      resolve vdecl error in parser
1355
1356  commit 55c5a4367da69c45dbd8c3cff504c79ad18b7037
1357  Author: jessieliu1 <jliu997@gmail.com>
1358  Date:   Sat Oct 21 20:11:00 2017 -0400
1359
1360      change the parser program rule
1361
1362  commit 2772e1989e7830da24dd7732c085ae8f545bc0b8
1363  Author: Seth Benjamin <sethjbenjamin@gmail.com>
1364  Date:   Sun Oct 15 23:53:15 2017 -0400
1365
1366      add break, continue
1367
```

```
1368 commit c58993f60d55e6ea7e1366cb19ac6aa2ff6c35a4
1369 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1370 Date:   Sun Oct 15 22:00:12 2017 -0400
1371
1372     update edmonds_karp with for_edge, for_node
1373
1374 commit f1127d245b3a96227a9a5974fd53c87118b3dd0e
1375 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1376 Date:   Sun Oct 15 21:24:11 2017 -0400
1377
1378     add function calls
1379
1380 commit bb51009cb5bdabf82c6a48560deee935cc347314
1381 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1382 Date:   Sat Oct 14 22:27:47 2017 -0400
1383
1384     add for_node, for_edge, bfs, dfs
1385
1386 commit c31c8b7db1d9404ea2153b36f31e3df9e729749a
1387 Author: jessieliu1 <jliu997@gmail.com>
1388 Date:   Sat Oct 14 14:06:32 2017 -0400
1389
1390     Added node and edge to scanner/ast
1391
1392 commit d4fbff1bc5cee1b8a880db8406794a1089d92325
1393 Author: Jennifer Bi <jenniferbi@dyn-160-39-151-135.dyn.columbia.edu>
1394 Date:   Fri Oct 13 12:53:42 2017 -0400
1395
1396     fixed shift/reduce conflict, added missing token defs in parser
1397
1398 commit 8c713b642d03d9ab48e2c1c8c5aff330802ec9d6
1399 Author: Jennifer Bi <jenniferbi@dyn-160-39-183-163.dyn.columbia.edu>
1400 Date:   Fri Oct 13 10:28:36 2017 -0400
1401
1402     added Makefile
1403
1404 commit 70b6d83d38a1a55a602d7cc50ccd2eef73d79dd9
1405 Author: jenniferbi <jb3495@columbia.edu>
1406 Date:   Thu Oct 12 11:47:37 2017 -0400
1407
1408     basic parser and ast
1409
1410     parser that accepts skeleton code, i.e. empty vdecl_list and no stmt_lists
1411
1412 commit b59ffe954d15d7f5be7f7512fbbc2a629840c1dd
1413 Author: Seth Benjamin <sethjbenjamin@gmail.com>
1414 Date:   Wed Oct 11 17:45:12 2017 -0400
1415
1416     add hello world, edmonds-karp example programs
1417
1418 commit df4d4fcef83f56581eb959fd395b712ce03e0d67
1419 Author: jessieliu1 <jliu997@gmail.com>
1420 Date:   Tue Oct 10 15:06:27 2017 -0400
1421
1422     Formatting for scanner
1423
1424 commit b1065bf5916569181e148ef52ebed32df0c0d622
```

```
1425 Author: Daniel Benett <deb2174@columbia.edu>
1426 Date:   Mon Oct 9 22:16:30 2017 -0400
1427
1428     fix digits and letters in scanner
1429
1430 commit 5e788d65042feef84b9f790db0fee2038d51d598
1431 Author: Daniel Benett <deb2174@columbia.edu>
1432 Date:   Mon Oct 9 22:10:34 2017 -0400
1433
1434     forward slash
1435
1436 commit f9ccf5d0ffe9dfd8d515e8f680af8bed6ac16347
1437 Author: Daniel Benett <deb2174@columbia.edu>
1438 Date:   Mon Oct 9 22:09:34 2017 -0400
1439
1440     updated scanner
1441
1442 commit 079f9dfd93305ee59bec57d3e0f542e2446c5dbe
1443 Author: jenniferbi <jb3495@columbia.edu>
1444 Date:   Mon Oct 9 21:19:51 2017 -0400
1445
1446     Create scanner.mll
1447
1448 commit 700fe1651b846ed7a35b3c3567c0c734fddb4f75
1449 Author: jenniferbi <jb3495@columbia.edu>
1450 Date:   Mon Oct 9 21:19:29 2017 -0400
1451
1452     Delete scanner
1453
1454 commit 625f1c173db6739e548808c9f95c98e2bbddfe56
1455 Author: jenniferbi <jb3495@columbia.edu>
1456 Date:   Mon Oct 9 21:18:24 2017 -0400
1457
1458     Scanner skeleton
1459
1460 commit 98c73f215ab831035f847af8710eb3670e39fd41
1461 Author: Jessie Liu <jliu997@gmail.com>
1462 Date:   Mon Oct 9 21:17:55 2017 -0400
1463
1464     Create README.md
```

git repository can be found at: https://github.com/jessieliu1/giraph