Graphiti

A graph language.

Sydney Lee
Michal Porubcin
Emily Hao
Andrew Quijano
Alice Thum

Table of Contents

Introduction

The graphing language *Graphiti* is designed to aid the user in representing and manipulating graphs, as well as implementing graph-related algorithms such as breadth-first search and Dijkstra's shortest algorithm. With a simple and easy to understand syntax, *Graphiti* allows the user to create representations of graphs by adding nodes and manipulating edges between them.

While *Graphiti* offers a more straightforward approach when expressing graph-related algorithms, the user will also be able to use graphs in a more qualitative way by adding user-defined, qualitative data to the edges (e.g. Node A is a "friend of" Node B). Users will be able to access relationships between specific nodes in the graph using a query.

Language Manual

Data Types

Data Types - Overview

Type	Description
int	4 bytes (32 bit) integer values
float	8 bytes (64 bit) values
bool	1 byte (8 bit) true/false
char	1 byte (8 bit) value
string	Finite sequence of characters
list	Generic lnked list with unordered data
map	Collection of string key-value pairs
graph	A set of nodes and edges that connect pairs of nodes

Data Types - Simple Usage

All variables and their type must be declared at the beginning of the function. Variables are assigned using the = operator. The type of the variable precedes the variable name in declarations, and the value to be assigned is on the right of the = operator.

```
int main() {
      bool b;
      int a;
      float f;
      string s;
      list<int><int> l;
      map m;
      graph g;

      b = true;
      a = 3;
      f = 3.14159;
      s = "hello ";
      l = [1,2,3];
      m = {["mom":"susan"]};
      g = {{}};
}
```

Data Types - Lists

Lists will behave much like double ended queues in Java. They will support the following functions. For the examples given below, we will assume there is a list <T> l that is generated.

function	Return Type	Description	Example
len()	int	Returns the number of elements in the list.	l.len();
at(i)	T	Returns the element at index i in the list.	l.at(2);
set(i, e)	T	Sets the element at index i in the list to e. The original element at index i is returned to the user.	l.set(3, e);
add_head(e)	int	Adds element e to the head of the list.Return 0 on fail and and 1 on success	l.add_head(e);

pop_head()	T	Removes the head of the list and returns it to the user.	l.remove_head();
add_tail(e)	int	Adds element e to the tail of the list.Returns 0 on fail and 1 on success	l.add_tail(e);

Below are more detailed examples on using lists as well as their expected output.

Lists - Examples

Declaring an empty list is **NOT** permitted:

```
list<int> l;
l = [];
Failure("invalid assign list<int> = list<void> []")
```

Getting length of a list:

```
list<int> l;
l = [1, 2, 3, 4];
int x = l.len();
x = 4
```

Getting certain element from the list:

```
list <int> x;
x = [1, 2, 3, 4];
int y = 2;
int z = x.at(y);
z = 3
```

Setting certain element from the list:

```
list <int> x;
int y;
int z;
x = [1, 2, 3, 4];
y = 2;
z = x.set(0, y);
z = 1
x = [2, 2, 3, 4]
```

Adding new head:

```
list <int> x;
```

```
int y;
x = [1, 2, 3, 4];
y = 2;
x.add_head(y);
x = [2, 1, 2, 3, 4]
```

Appending a list:

```
int z;
list <int> x;
z = 5;
x = [1, 2, 3, 4];
l.add_tail(z);
l.add_tail(6);
x = [1, 2, 3, 4, 5, 6]
```

Removing head:

```
list <int> x;
int z;
x = [1, 2, 3, 4];
z = x.remove_head();
x = [2, 3, 4]
z = 1
```

Data Types - Maps

The map type contains a list of string-to-string key-value pairs.

function	Return Type	Description	Example
put(*key*, *value*)	int	*Key* and *value* are strings. Puts key, value pair into map. Returns 1 if successful, 0 if not.	m.put("uni1", "esh2160")
get(*key*)	string	*Key* is a string. Returns *key*'s value in the map. Returns NULL if *key* does not exist.	m.get("uni1")
containsKey(*key*)	int	*Key* is a string. Returns 1 if the map contains the *key*, else 0.	m.containsKey("uni1")

containsValue(*value*)	int	*Value* is a string. Returns 1 if the map contains the *value*, else 0.	m.containsValue(“esh2160”)
removeNode(*key*)	int	*Key* is a string. Removes the node with *key* from the map. Returns 1 if successful, else 0.	m.removeNode(“uni1”)
isEqual(*map*)	int	Compares *map1* and *map2*. If contents are the same, then returns 1. Else, 0.	m.isEqual(map2);

Below are more examples of how to use maps:

Maps - Examples:

Declaring an empty map:

```
map m;
m = {[]};
```

Declaring map with two key-value pairs:

```
map m;
m = {["Michal":"Language Guru",”Alice”:”System Architect”]};
```

Add a key/value pairing into the map:

```
map m;
m = {[]};
m.put("Michal", "Language Guru");
x{["Michal", "Language Guru"]}
```

Remove an key/value pairing from the map from the key:

```
map m;
m = {[]};
m.put("Michal", "Language Guru");
m.put("Andrew", "Manager");
m.removeNode("Andrew");
x{["Michal", "Language Guru"]}
```

Retrieve a value from the map from the key:

```
map m;
string a;
```

```
m = {[]};
m.put("Michal", "Language Guru");
a = m.get("Michal");
print(a);
"Language Guru"
```

Check if a map contains a specific key:

```
map m;
m = {[]};
m.put("Michal", "Language Guru");
if(m.containsKey("Michal"))
{
      print("true");
}
true
```

Check if a map contains a value:

```
map m;
m = {[]};
m.put("Michal", "Language Guru");
if(m.containsValue("Language Guru"))
{
      print("true");
}
true
```

Check if two maps are equal (contain the same data):

```
map x;
map y;
x = {[]};
y = {[]};
x.put("Michal", "Language Guru");
y.put("Michal", "Language Guru");
if(x.isEqual(y) == 1)
{
      print("true");
}
x.put("Emily", "Tester");
if( !x.isEqual(y) == 1)
{
      print("true");
}
True
True
```

Printing a map:

```
map m;
m = {["Michal":"Language Guru","Alice":"System Architect"]};
printm(x);
{
      "Michal" : "Language Guru",
      "Alice" : "System Architect"
}
```

Data Types - Graphs

Graph Declaration

Declare an empty graph and assign it to a graph variable `g`:

```
graph g = {{}};
```

Declare a graph with existing maps `A`, `B`, and `C`:

```
graph g = {{A,B,C}};
```

Declare a graph with existing maps `A` and `B` with an edge between them

```
graph g = {{A -> B}};
```

An anonymous graph is intuitively a graph not assigned to a variable:

```
{{ A->B }}
```

This syntax does not return the graph, or the subgraph with the specified nodes. It is also distinct from an anonymous graph due to the graph variable prefixing the braces. Examples of this syntax are shown in the below sections.

Maps in Graphs

Nodes in graphs only hold map data, but users are not allowed direct access to the node itself. Maps are automatically wrapped into a graph node by Graphiti.

Declare a node outside of a graph (creates a node `A` with key: `"uni"` and data: `"mp3242"` and assigns it to variable `A`):

```
map A = {["uni" : "mp3242"]};
```

Declare a node inside of a graph:

```
graph g = {{ {["uni" : "mp3242"]} }};
```

Declare multiple nodes inside of a graph:

```
graph g = {{ {["uni" : "mp3242"]}, {["uni" : "aq0000"]} }};
```

Declare two new nodes and connect them with an edge inside of a graph:

```
graph g = {{ {["uni" : "mp3242"]} -> {["uni" : "aq0000"]} }};
```

The above creates a new node with key: `"uni"` and data: `"mp3242"` and another node with key: `"uni"` and data: `"aq0000"` with a unidirectional edge from `A` to `B`.

Add node `A` to graph `g`. The data of node `A` is copied into the graph. Mutating `A` outside the graph does not affect the data in the graph.

```
g{{ A }}
g = g + {{A}}; //equivalent
```

Add nodes `A` and `B` to graph `g`.

```
g{{ A, B }};
```

Delete a node or multiple nodes from graph `g`.

```
g{{ ~A }};
g{{ ~A, ~B }};
```

Edges in Graphs

There is no edge type in *Graphiti*. Nevertheless, it is simple to add, remove, and manipulate edges, as can be seen in the examples below.

Declare graph `g` with unidirectional edge from node `A` to node `B`

```
graph g = {{ A -> B }};
```

Declare graph `g` with undirected/bidirectional edge from node `A` to node `B`.

```
graph g = {{ A -> B; B -> A }};
graph g = {{ A -- B }}; //shortcut
```

An edge declared with `--` is a shortcut for declaring an edge in both directions with the same weight. In *Graphiti*, there are no true undirected edges; they are represented as bidirectional edges.

Add an edge in graph `g` between two vertices that hold data from maps `A` and `B`:

```
g{{ A->B }};
```

If the edge from `A` to `B` already exists, then nothing changes. If either node (or both) has not been added to the graph, then it will be added automatically.

Declare edges between different nodes in a graph:

```
g{{ A -> C, B-> A }};
```

The above will create edges between already

Delete an edge from node `A` to node `B` from graph `g`:

```
g{{ A ~> B }};
```

Declare weight of `"employee_of"` on an edge from node `A` to node in a graph.

```
g{{ A ["employee_of"]-> B }};
```

Declare a non weighted edge:

```
g{{ A [""]-> B }};
g{{ A -> B }}; //weight braces not necessary
```

All edges have a default weight of an empty string: `""`

Modify edge weight in graph g (the second option is a built-in function, see section 7):

```
g{{ A ["boss_of"]-> B }};
modify_edge(g, A, B, old_data, new_data);
```

If the user declares an edge which already exists, then it will replace the edge. If the edge does not exist, but the vertices are created, then a new edge will connect the two nodes. If either of the nodes does not exist, they will be automatically added to the graph.

Change edge direction:

```
g{{ A ~> B; B -> A}};
```

Operators

Operators - Arithmetic Operators

Type	Description
=	Assignment
+	Addition between ints, doubles, graphs; string concatenation; list appending
−	Subtraction between ints, doubles
*	Multiplication between ints, doubles

/	Division between ints, doubles
%	Remainder for ints (modulo)

Operators - Logical Operators

Type	Description
==	Compares two values and returns true if they are equal, else false.
!=	Compares two values and returns true if they are not equal, else false.
<	Returns true if the value on the left is less than the value on the right for ints and floats.
>	Returns true if the value on the left is greater than the value on the right for integers and floats.
<=	Returns true if the value on the left is less than or equal to the value on the right for integers and floats.
>=	Returns true if the value on the left is greater than or equal to the value on the right for integers and floats.
&&	Compares two boolean values and returns true if they are both true.
||	Compares two boolean values and returns true if at least one is true.

Operators - Graph Operators

Type	Description
|	Graph union : Returns a graph of all nodes (edges are not included)
&	Graph intersection : Returns a graph of shared nodes (edges are not included)
->	Directed edge
--	Undirected/Bidirectional edge

`{}--`	Weighted edge (weight in brackets)
`~`	Delete node
`~>`	Delete edge

Operators - The + Operator

Type	Description
`<list> + <*>`	Add element to the end of a list. This assumes the element is the same type as the list. E. g. list<int> x; x = [4, 5, 6] x + [7] x = [4, 5, 6, 7]
`<string> + <string>`	Concatenate strings
`<graph> + <graph>`	Add all nodes and edges of one graph to another graph

Operators - Comments

Type	Description
`/*...*/`	Comments content between `/*` and `*/`

Control Flow

Control Flow - Conditionals

If-else blocks follow a familiar C-style syntax:

```
if (condition) {
      statement;
} else if (condition) {
      statement;
} else {
      statement;
}
```

Control Flow - Loops

Graphiti uses C-style for- and while-loop syntax as well.

```
int i;
for (i = 0; i < n; i++) {
      statement;
}

while (true) {
      statement;
}
```

Functions

Function declaration syntax is also familiar:

```
int add(int a, int b) {
      // logic
}
```

Users must utilize a main function, where the program begins, and return 0:

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

Project Plan

Project Plan - Process

We used Git as our version control system, with our code stored in a private repository on GitHub with a master branch. During meetings, meeting minutes were taken in a shared Google Doc so anything we discussed would be kept track of in one place and members who missed would be informed of what happened. We also created a spreadsheet keeping track of tasks, assignees, estimated completion dates, and deadlines. This allowed us to keep track of bugs, task dependencies, and our overall progress.

Project Plan - Timeline

- 9/15: Set up ssh-ing into virtual machine and environment
- 10/9: Add necessary tokens to scanner, parser, and AST
- 11/14: Hello World program compiled

- 12/2: Finished maps semantic checking and code generation
- 12/9: Finished graph semantic checking and code generation
- 12/9: Testing script and test cases added
- 12/12: Maps, lists, and graphs added to C library
- 12/17: Finished adding tests for maps and graphs
- 12/19: Lists functionality completed
- 12/19: Final Presentation

Project Plan - Roles and Responsibilities

Though we did have formally designated roles, our team members were fairly flexible in the tasks we took on, especially at first. When we had a more concrete outline of our language in place, we divided up and worked on specific features, e.g. maps, lists, and graphs. Each member or pair was then tasked with working on that feature from scanner/parser to codegen and C library. For example, Andrew worked on lists, Michal and Sydney worked on graphs, and Alice and Emily worked on maps.

Member	Role	Responsibilities
Andrew Quijano	Manager	Lists from scanner/parser to codegen
Michal Porubcin	Language Guru	Graphs through codegen and C library, standardizing syntax
Alice Thum	System Architect	Maps from scanner/parser to codegen, maps and lists in the C library
Sydney Lee	System Architect	Graphs through codegen and C library, demonstration code
Emily Hao	Tester	Maps from scanner/parser to codegen, testing suite

Project Plan - Software Development

- **Git**: version control system
- **OCaml**: used for scanning, parsing, semantic checking, and code generation
- **C**: used for building the graph.c library linked in codegen.ml
- **Makefile**: compiling and linking the code

- **Bash**: used for automated testing

Architectural Design

Architectural Design - Scanner, Parser, and AST

The scanner will read in tokens from the user's input and return an abstract syntax tree. The parser will choose to accept or reject the program based on whether it adheres to the specified grammar. Our parser implementation for maps and lists was fairly straightforward, while graphs required some more complicated rules due to the multiple ways a graph literal could be declared, the symbols for modifying and adding edges/vertices, and the general complexity of the syntax that we decided on.

Architectural Design - SAST and Semantic Checking

In semantic checking, we perform type checking to ensure that the correct argument and return types are provided to each function we implement. As we step through the AST, we bind each token to a semantically checked expression if it correct and raise a failure if the expression is not of the correct type. We then output the list of semantically checked expressions as an SAST.

Architectural Design - Code Generation

In the code generation step of the project, we go through the SAST produced by semantic checking and output an LLVM module. Starting from globals and function lists, we step through all the way down to the terminals. Globals are added to a symbol table and allocated registers, while function arguments are also processed and added to the symbol table. After the function bodies are also processed with their own symbol tables. We then move on to the statements in our code, converting them to LLVM instructions that are outputted in the module.

Architectural Design - C Library

We implemented a series of C structs to store graphs, maps, and lists. Maps and lists both made use of a simple linked-list format for simplicity's sake. The C functions were called from the code generation stage of the project. Our C functions included those used to add and modify entries in maps, lists, graphs, remove entries, and initialize new maps, lists, and vertices in graphs.

Test Plan

Testing Phases - Scanner, Parser, and AST

Throughout the semester, we tested the scanner, parser, and AST after incorporating each feature manually with *menhir*. Using *menhir,* we could manually input a stream of tokens and confirm if the parser would accept or reject.

Testing Phases - Through Codegen

The main indicator of a working language was compiling a Graphiti program that prints “Hello World”. As more features of our language, such as maps and graphs, were implemented through Codegen, more test programs testing basic functions, such as map and graph initializations were written.

Automation

test.sh is a script that goes through, compiles, and runs the *.gra* files in the *tests* directory. Each output is compared the corresponding expected output (*.err* or *.out*). If the output did not match the expected output, then the script would output a fail message, and a *.diff* file would be generated. More details about the process and result of each test file run is in *test.log*.

Test Files

There are two types of test files in the *tests* directory. Files that start with *fail* are files that are meant to generate an error. Files that start with *test* are meant to compile and generate the expected output. Expected errors and outputs are written in corresponding *.err* and *.out* files, respectively.

We focused on testing the following features:

- Type Declaration and Assignment - We tested basic declaration and assignment for integers, strings, floats, bools, lists, maps, and graphs. We also verified that Graphiti would catch incorrect type assignments.

- Type Operations - We tested basic operations on integers, floats, and booleans. For graphs, we tested their allowed operations, such as graph union and graph intersection.

- Control Flow - We checked if-else blocks whose bodies should or should not execute, and verified the number iterations of loops. We also checked while loops and basic for loops and verified the number of interactions.

- Global Recursion - We checked basic recursion functions that were allowed in Micro_C, like fibonacci and factorial.

- Methods - We tested graph, list, and map methods. Specifically, we verified that the return types and values for each method were correct. We also verified that the methods would throw errors if the argument types are incorrect.

- Syntax - We confirmed that mistakes in syntax, such as missing semicolons and missing brackets were detected.

Test Files

One example test file that tests maps:

```
int main() {
    map m;
    string a;
    m = {[]};
    m.put("Michal","Language Guru");
    a = m.get("Michal");
    print(a);
    return 0;
}
```

The expected output:

```
Language Guru
~
```

One example fail file that tests method argument type:

```
/* throw error if .put type is incorrect */
int main() {
        map m;
        m = {[]};
        m.put("Michal",100);
        return 0;
}
```

The expected error output:

```
Fatal error: exception Failure("Value must be a string type instead of int")
~
```

Lessons Learned

Lessons Learned - Sydney Lee

This class was a challenge, and there were a lot of lessons to learn from it. I thought that the language that we created was really interesting in concept. It was very exciting to see how graphs in a conventional syntax like Java could be transformed into a new and simple syntax that we designed. It was also very rewarding to see our work compiling and being able to write projects in our language. We, however, did have a couple of setbacks. I think time management and team communication could have been stronger, which would have made the the timeline of the project move more efficiently. We ended up having to sacrifice certain functionalities and re-implementing things because of the lack of communication, but if we had delegated roles earlier, a lot of extra work could have been avoided. There were instances where members who were assigned roles in the project would get them done well after their set deadlines, which set the project back significantly. Overall I enjoyed the course and enjoyed the given challenge of what we needed to do, and with more communication, this language would have been a better success.

Lessons Learned - Michal Porubcin

As language guru, I was very excited to work on the project at the beginning of the semester. I proposed the idea of a graph language, due in part to the suggestions of Alice and Andrew about a possible connection we could make to neo4j databases. While we all knew originality was secondary to learning and understanding for this project, I was initially discouraged by the fact that many people had done graph languages before. Nevertheless we resolved to put our own spin on it. We aimed to showcase easy graph manipulation, but featuring maps and lists as a core part of the language as well. I realized a lot of the early decisions I made about the language were unimportant syntactic shortcuts for the user, and became more cognizant of features that actually provided novel functionality. It was interesting to see how our language evolved as we understood more about the compiler, progressing from scanner through codegen. My ambitions grew with my understanding, and towards the end of the semester we were going to feature three more interesting demos with maps, lists, and graphs working in tandem (the first was a standard graph search algorithm, the second was a predator-prey simulation, and the third was a demonstration of json export to neo4j). Unfortunately we completely underestimated the amount of work required in the final stretch – to merge our codebases, smooth out little bugs, etc. – and necessarily cut many features to meet the deadline. It was an avoidable mistake, but I am still glad I walked out knowing much more than I did coming into the class.

Lessons Learned - Emily Hao

This project was definitely challenging, but I learned a lot about compilers, languages, teamwork, and communication. At the beginning of the semester, I found it difficult to understand the scope of the project and what completing it would entail, but as the semester went on, my understanding of OCAML and compiling improved, and I was eventually able to implement new features more easily and efficiently. It was encouraging to see each finish each step of the process with success, and it was exciting every time we overcame a barrier. In addition to the technical challenges that the project presented, I found that team organization and communication was, at a lot of moments, frustrating and difficult. At first, we were not sure which tasks needed to be done and which team roles needed to be fulfilled, but after we established those things, our team still struggled to maintain a proper timeline and efficient workflow. For example, we often miscommunicated about which tasks needed to be completed, as well as who was working on what. Because of that, we missed deadlines and found ourselves rushing to implement all of the features we had initially planned for Graphiti to have. Although we had all of the elements finished, we underestimated the time it would take to fully integrate every component, debug, and write the demos that we wished to show. I also learned a lot about working on the same code with multiple people, encouraging me to become more comfortable with github. In conclusion, I believe that creating Graphiti was truly rewarding, despite all of the difficulties, and I am proud of what Graphiti is and can be.

Lessons Learned - Andrew Quijano

I had learned how use OCaml because of this project. Ocaml had been an unusual language to get used to as I was used to Java, C and Python. However, once seeing how the compiler was built up from Ocaml, I can see why Ocaml is useful for compilers as it is efficient in adjusting to all the nuances of a Programming Language as being able to successfully tokenize, parse, semantically check and generate code with a language like Java seems to be much more prone to overcomplexity or bugs. Advice for future PLT classes: Start the project early. Ideally, start with a very basic component done at a time. Looking back, what may have been better was to start with completing lists or map as a first priority around mid-October. Seeing one advanced structure using the C library complete would have made completing the other structs like lists and graphs more easier as less time was spent understanding the MIcro-C environment and more time was spent into adding features.

Lessons Learned - Alice Thum

I think the biggest takeaway from this project was the importance of team organization. It was not until the project was well underway that we finally began keeping tracks of specific tasks

that needed to be implemented, for example a map.put() function, or bugs that needed to be fixed or test cases that weren't passing. Internally, we also had very few deadlines beyond the class deliverables, for example the parser or the "hello world" program. I think in the future, I would want to create a fixed team workflow manual that would include not just style guides for code, but proper Git workflow, a more thorough peer-review system so that janky commits could not make their way into the master branch, and a detailed feature-by-feature task board such as ones used in Zenhub. We did have problems with members forgetting to work on their own branches and pushing code that did not compile to master. In addition, though we allotted each member tasks to complete, at certain times a member would reimplement some feature that another member had already completed, or stray from their allotted task and do double work.

Appendix

Scanner

```
(* Ocamllex scanner for MicroC *)

{ open Parser }

let digit = ['0' - '9']
let digits = digit+

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '['      { LBRACK }
| ']'      { RBRACK }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI   }
| ':'      { COLON  }
| ','      { COMMA  }
| '+'      { PLUS   }
| '-'      { MINUS  }
| '*'      { TIMES  }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| '%'      { MOD    }
| "+="     { ADDASN }
| "-="     { MINASN }
| "*="     { TIMASN }
| "/="     { DIVASN }
| "=="     { EQ }
```

```
  | "!="     { NEQ }
  | '<'      { LT }
  | "<="     { LEQ }
  | ">"      { GT }
  | ">="     { GEQ }
  | "&&"     { AND }
  | "||"     { OR }
  | "!"      { NOT }
  | "|"      { UNION }
  | "&"      { INTERSECT }
  | "if"     { IF }
  | "else"   { ELSE }
  | "for"    { FOR }
  | "while"  { WHILE }
  | "return" { RETURN }
  | "true"   { BLIT(true)  }
  | "false"  { BLIT(false) }
  | "int"    { INT }
  | "char"   { CHAR }
  | "bool"   { BOOL }
  | "float"  { FLOAT }
  | "string" { STR }
  | "map"    { MAP }
  | "void"   { VOID }
  | "map"    { MAP }
  | "graph"  { GRAPH }
  | "list"   { LIST  }
  | "{{"     { LGRAPH }
  | "}}"     { RGRAPH }
  | "--"     { UNIARR }
  | "->"     { DIRARR }
  | "~>"     { DELEDGE }
  | "~"      { DELNODE }
  | ".get_edges"     { GRAPH_EDGES }
  | ".get_neighbors" { GRAPH_NODES }
  | ".get_all_nodes" { GRAPH_ALL_VERTICES }

  |"{["               { LMAP }
  |"]}"               { RMAP }
  |".put"             { MAP_PUT }
  |".get"             { MAP_GET }
  |".containsKey"     { MAP_CONTAINS_KEY }
  |".containsValue"   { MAP_CONTAINS_VALUE }
  |".removeNode"      { MAP_REMOVE_NODE }
  |".isEqual"         { MAP_IS_EQUAL }

  |".len"     { LIST_SIZE }
  |".at"         { LIST_GET  }
  |".set"        { LIST_SET  }
  |".add_head"   { LIST_ADD_H}
  |".remove_head"  { LIST_RM_H   }
  |".add_tail"    { LIST_ADD_T   }
```

```
  | digits as lxm { LITERAL(int_of_string lxm) }
  | digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*     as lxm { ID(lxm) }
  | '"' (([' '-'!' '#'-'[' ']'-'~'])* as s) '"' { STRLIT(s) }
  | eof { EOF }
  | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

Parser

```
/* Ocamlyacc parser for MicroC */

%{
open Ast
%}

/*precedence not assigned here*/
%token SEMI LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE COMMA
%token MAP_PUT MAP_GET MAP_CONTAINS_KEY MAP_CONTAINS_VALUE MAP_REMOVE_NODE MAP_IS_EQUAL
%token GRAPH_EDGES GRAPH_NODES GRAPH_ALL_VERTICES
%token NOT EQ NEQ LT LEQ GT GEQ AND OR UNION INTERSECT
%token MOD PLUS MINUS TIMES DIVIDE ASSIGN ADDASN MINASN TIMASN DIVASN
%token RETURN IF ELSE FOR WHILE INT CHAR BOOL FLOAT STR VOID GRAPH MAP
%token LGRAPH RGRAPH UNIARR DIRARR DELEDGE DELNODE
%token LIST LIST_SIZE LIST_GET LIST_SET LIST_ADD_H LIST_RM_H LIST_ADD_T /* LIST_RM_T */
%token COLON LMAP RMAP
/*
%token STR_SIZE
*/
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT
%token <char> CHARLIT
%token <string> STRLIT
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc COLON
%right ASSIGN ADDASN MINASN TIMASN DIVASN
/*
%left STRSIZE
*/
%left OR
```

```
%left AND
%right EQ NEQ
%left LT GT LEQ GEQ UNION INTERSECT
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT NEG
%right LIST_SIZE LIST_GET LIST_SET LIST_ADD_H LIST_RM_H LIST_ADD_T /*LIST_RM_T*/
%right  MAP_PUT MAP_GET MAP_CONTAINS_KEY MAP_CONTAINS_VALUE MAP_REMOVE_NODE
MAP_IS_EQUAL GRAPH_EDGES GRAPH_NODES
%right FOR
%%

program:
  decls EOF { $1 }

decls:
   /* nothing */ { ([], [])               }
 | decls vdecl { (($2 :: fst $1), snd $1) }
 | decls fdecl { (fst $1, ($2 :: snd $1)) }

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { typ = $1;
     fname = $2;
     formals = List.rev $4;
     locals = List.rev $7;
     body = List.rev $8 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { $1 }

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

typ:
    INT             { Int    }
  | CHAR            { Char   }
  | BOOL            { Bool   }
  | FLOAT           { Float  }
  | STR             { String }
  | VOID            { Void   }
  | MAP                         { Map    }
  | GRAPH                       { Graph  }
  | LIST LT typ GT              { List($3) }

vdecl_list:
    /* nothing */    { [(Int, "__i");
                        (List(Map), "__nodes");
                        (Int, "__l")]}
```

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

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

stmt:
    expr SEMI                               { Expr $1               }
  | RETURN expr_opt SEMI                    { Return $2             }
  | LBRACE stmt_list RBRACE                 { Block(List.rev $2)    }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)        }
  | WHILE LPAREN expr RPAREN stmt           { While($3, $5)         }

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

expr:
    LITERAL                   { Literal($1)            }
  | FLIT                      { Fliteral($1)           }
  | BLIT                      { BoolLit($1)            }
  | CHARLIT                   { CharLit($1)            }
  | ID                        { Id($1)                 }
  | STRLIT                    { StrLit($1)             }
  | LBRACK args_opt RBRACK    { ListLit($2)            }
  | LMAP map_lit_element RMAP { MapLit($2)             }
  | LGRAPH graph_i_opt RGRAPH   { GraphLit($2)         }
  | ID LGRAPH graph_m_opt RGRAPH  { GraphMod($1, $3)   }
  | expr PLUS   expr          { Binop($1, Add,   $3)   }
  | expr MINUS  expr          { Binop($1, Sub,   $3)   }
  | expr TIMES  expr          { Binop($1, Mult,  $3)   }
  | expr DIVIDE expr          { Binop($1, Div,   $3)   }
  | expr MOD    expr          { Binop($1, Mod,   $3)   }
  | expr EQ     expr          { Binop($1, Equal, $3)   }
  | expr NEQ    expr          { Binop($1, Neq,   $3)   }
  | expr LT     expr          { Binop($1, Less,  $3)   }
  | expr LEQ    expr          { Binop($1, Leq,   $3)   }
  | expr GT     expr          { Binop($1, Greater, $3) }
  | expr GEQ    expr          { Binop($1, Geq,   $3)   }
  | expr AND    expr          { Binop($1, And,   $3)   }
  | expr OR     expr          { Binop($1, Or,    $3)   }
  | expr UNION  expr          { Binop($1, Union, $3) }
  | expr INTERSECT expr       { Binop($1, Intersect, $3) }
  | MINUS expr %prec NOT      { Unop(Neg, $2)           }
  | NOT expr                  { Unop(Not, $2)          }
  | ID ASSIGN expr            { Assign($1, $3)         }
  | ID ADDASN expr            { OpAssign($1, Add, $3) }
  | ID MINASN expr            { OpAssign($1, Sub, $3) }
  | ID TIMASN expr            { OpAssign($1, Mult, $3) }
```

```
    | ID DIVASN expr             { OpAssign($1, Div, $3) }
    | ID LPAREN args_opt RPAREN  { Call($1, $3)           }
    | LPAREN expr RPAREN           { $2                   }

    | expr GRAPH_EDGES LPAREN expr RPAREN         { GraphEdges($1, $4) }
    | expr GRAPH_ALL_VERTICES LPAREN RPAREN       { GraphAll($1)  }
    | expr GRAPH_NODES LPAREN expr RPAREN         { GraphNodes($1, $4) }
    | expr MAP_PUT LPAREN expr COMMA expr RPAREN  { MapPut($1, $4, $6)        }
    | expr MAP_GET LPAREN expr RPAREN             { MapGet($1, $4)            }
    | expr MAP_CONTAINS_KEY LPAREN expr RPAREN    { MapContainsKey($1, $4)    }
    | expr MAP_CONTAINS_VALUE LPAREN expr RPAREN  { MapContainsValue ($1, $4) }
    | expr MAP_REMOVE_NODE LPAREN expr RPAREN     { MapRemoveNode ($1, $4)    }
    | expr MAP_IS_EQUAL LPAREN expr RPAREN        { MapIsEqual($1, $4)        }
    | expr LIST_SIZE LPAREN RPAREN                { ListSize($1)          }
    | expr LIST_GET LPAREN expr RPAREN            { ListGet($1, $4)       }
    | expr LIST_SET LPAREN expr COMMA expr RPAREN { ListSet($1, $4, $6)   }
    | expr LIST_ADD_H LPAREN expr RPAREN          { List_Add_Head($1, $4) }
    | expr LIST_RM_H LPAREN RPAREN                { List_Rm_Head($1)      }
    | expr LIST_ADD_T LPAREN expr RPAREN          { List_Add_Tail($1, $4) }

/* Used for Graphs */
graph_i_opt:
      /* nothing */    { [] }
    | graph_i_list     { $1 }

graph_i_list:
      graph_i_expr                        { $1 }
    | graph_i_list COMMA graph_i_expr     { $1 @ $3 }

graph_i_expr:
    /*(fromId, weight, toId)*/
      expr                                { [GraphAddVertex($1)] }
    | expr UNIARR expr                    { [GraphAddEdge($1, $3); GraphAddEdge($3,
$1)] }
    | expr DIRARR expr                    { [GraphAddEdge($1, $3)] }
    | expr LBRACK expr RBRACK UNIARR expr  { [GraphAddWedge($1, $3, $6);
GraphAddWedge($6, $3, $1)] }
    | expr LBRACK expr RBRACK DIRARR expr  { [GraphAddWedge($1, $3, $6)] }

graph_m_opt:
                    { [] }
    | graph_m_list   { $1 }

graph_m_list:
      graph_m_expr                    { $1 }
    | graph_m_list COMMA graph_m_expr  { $1 @ $3 }

graph_m_expr:
      expr                                  { [GraphAddVertex($1)] }
    | expr UNIARR expr                      { [GraphAddEdge($1, $3); GraphAddEdge($3,
$1)] }
```

```
      | expr DIRARR expr                        { [GraphAddEdge($1, $3)] }
      | expr LBRACK expr RBRACK UNIARR expr    { [GraphAddWedge($1, $3, $6);
GraphAddWedge($6, $3, $1)] }
      | expr LBRACK expr RBRACK DIRARR expr    { [GraphAddWedge($1, $3, $6)] }
      | DELNODE expr                           { [GraphDelVertex($2)] }
      | expr DELEDGE expr                      { [GraphDelEdge($1, $3)] }

/* Used for Maps */
map_lit_element:
    /* nothing */   { [] }
    |  map_list     { List.rev $1 }

map_list:
      map_entry                 { $1 }
    | map_list COMMA map_entry  { $1 @ $3 }

map_entry:
    expr COLON expr         { [($1, $3)] }

/* Used for Lists */
args_opt:
    /* nothing */ { [] }
  | args_list  { List.rev $1 }

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

AST

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

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

type uop = Neg | Not

type gop = Delnode | Deledge

type expr =
    Literal of int
  | Fliteral of string
  | BoolLit of bool
  | CharLit of char
  | StrLit of string
  | MapLit of (expr * expr) list
  | ListLit of expr list
  | Id of string
  | Binop of expr * op * expr
  | GraphLit of expr list
  | GraphMod of string * (expr list)
```

```
    | Unop of uop * expr
    | Assign of string * expr
    | OpAssign of string * op * expr
    | GraphEdges of expr * expr
    | GraphNodes of expr * expr
    | GraphAllNodes of expr
    | GraphAll of expr
    | GraphAddVertex of expr
    | GraphAddEdge of expr * expr
    | GraphAddWedge of expr * expr * expr
    | GraphDelVertex of expr
    | GraphDelEdge of expr * expr
    | MapPut of expr * expr * expr
    | MapGet of expr * expr
    | MapContainsKey of expr * expr
    | MapContainsValue of expr * expr
    | MapRemoveNode of expr * expr
    | MapIsEqual of expr * expr
    | ListSize of expr
    | ListGet of expr * expr
    | ListSet of expr * expr * expr
    | List_Add_Head of expr * expr
    | List_Rm_Head of expr
    | List_Add_Tail of expr * expr
    | Call of string * expr list
    | Noexpr

type typ =
    Int
  | Bool
  | Char
  | Float
  | String
  | Void
  | Graph
  | Map
  | List of typ

type bind = typ * string

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | While of expr * stmt

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
```

```
      body : stmt list;
    }

type program = bind list * func_decl list

(* Pretty-printing functions *)

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

let string_of_gop = function
    Delnode -> "~"
  | Deledge -> "~>"

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

let rec string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Char -> "char"
  | String -> "string"
  | Graph -> "graph"
  | Void -> "void"
  | Map -> "map"
  | List(l) -> "list" ^ "<" ^ string_of_typ l ^ ">"

let rec string_of_expr =
  function
    Literal(l) -> string_of_int l
  | Fliteral(l) -> l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | CharLit(l) -> Char.escaped l
  | StrLit(l) -> "\"" ^ l ^ "\""
```

```
    | ListLit(l) -> "[" ^ String.concat "," (List.map string_of_expr l) ^ "]"
      | Id(s) -> s
    | GraphLit(l) -> "{{" ^ String.concat "," (List.map (fun(e) -> string_of_expr e) l) ^
"}}"
    | GraphMod(id, l) -> id ^ "{{" ^ String.concat "," (List.map (fun(e) ->
string_of_expr e) l) ^ "}}"
    | GraphNodes(id, n) -> string_of_expr id ^ ".get_neighbors(" ^ string_of_expr n ^ ")"
    | GraphEdges(id, e) -> string_of_expr id ^ ".get_edges(" ^ string_of_expr e ^ ")"
    | GraphAllNodes(id) -> string_of_expr id ^ ".get_all_nodes()"
    | GraphAll(id) -> string_of_expr id ^ ".get_all_nodes()"
    | GraphAddVertex(_) -> "FAILED graphaddvertex not implemented"
    | GraphAddEdge(_,_) -> "FAILED graphaddedge not implemented"
    | GraphAddWedge(_,_,_) -> "FAILED graphaddwedge not implemented"
    | GraphDelVertex(_) -> "FAILED graphdelvertex not implemented"
    | GraphDelEdge(_,_) -> "FAILED graphdeledge not implemented"
    | MapLit(l) -> "{[" ^ String.concat "," (List.map (fun(k, v) -> string_of_expr k ^
":" ^ string_of_expr v) l) ^ "]}"
    | MapPut(m, key, value) -> string_of_expr m ^ ".put(" ^ string_of_expr key ^ "," ^
string_of_expr value ^ ")"
    | MapGet(m, key) -> string_of_expr m ^ ".get(" ^ string_of_expr key ^ ")"
    | MapContainsKey(m, key) -> string_of_expr m ^ ".containsKey(" ^ string_of_expr key ^
")"
    | MapContainsValue(m, value) -> string_of_expr m ^ ".containsValue(" ^ string_of_expr
value ^")"
    | MapRemoveNode(m, key) -> string_of_expr m ^ ".removeNode(" ^ string_of_expr key ^
")"
    | MapIsEqual(m1, m2) -> string_of_expr m1 ^ ".isEqual(" ^ string_of_expr m2 ^ ")"
    | ListSize(l) -> string_of_expr l ^ ".len"
    | ListGet(l, idx) -> string_of_expr l ^ ".at(" ^ string_of_expr idx ^ ")"
    | ListSet(l, idx, e) -> string_of_expr l ^ ".set(" ^ string_of_expr idx ^ "," ^
string_of_expr e ^ ")"
    | List_Add_Head(l, e) -> string_of_expr l ^ ".add_head(" ^ string_of_expr e^ ")"
    | List_Rm_Head(l) -> string_of_expr l ^ ".remove_head()"
    | List_Add_Tail(l, e) -> string_of_expr l ^ ".add_tail(" ^ string_of_expr e ^ ")"
    | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
    | Unop(o, e) -> string_of_uop o ^ string_of_expr e
    | Assign(v, e) -> v ^ " = " ^ string_of_expr e
    | OpAssign(v, o, e) -> v ^ string_of_op o ^ " " ^ string_of_expr e
    | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
    | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
    | Expr(expr) -> string_of_expr expr ^ ";\n";
    | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
    | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
    | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
    | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
```

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

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

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

SAST

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

open Ast

type sexpr = typ * sx
and sx =
    SLiteral of int
  | SFliteral of string
  | SBoolLit of bool
  | SCharLit of char
  | SStrLit of string
  | SListLit of sexpr list
  | SMapLit of (sexpr * sexpr) list
  | SGraphLit of sexpr  list
  | SGraphMod of string * (sexpr list)
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of string * sexpr
  | SGraphEdges of sexpr * sexpr
  | SGraphNodes of sexpr * sexpr
  | SGraphAllNodes of sexpr
  | SGraphAll of sexpr
  | SGraphAddVertex of sexpr
  | SGraphAddEdge of sexpr * sexpr
  | SGraphAddWedge of sexpr * sexpr * sexpr
  | SGraphDelVertex of sexpr
  | SGraphDelEdge of sexpr * sexpr
  | SMapPut of sexpr * sexpr * sexpr
  | SMapGet of sexpr * sexpr
  | SMapContainsKey of sexpr * sexpr
  | SMapContainsValue of sexpr * sexpr
  | SListSize of sexpr
  | SListGet of sexpr * sexpr
```

```
    | SListSet of sexpr * sexpr * sexpr
    | SList_Add_Head of sexpr * sexpr
    | SList_Rm_Head of sexpr
    | SList_Add_Tail of sexpr * sexpr
    | SOpAssign of string * op * sexpr
    | SMapRemoveNode of sexpr * sexpr
    | SMapIsEqual of sexpr * sexpr
    | SCall of string * sexpr list
    | SNoexpr

type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SWhile of sexpr * sstmt

type sfunc_decl = {
    styp : typ;
    sfname : string;
    sformals : bind list;
    slocals : bind list;
    sbody : sstmt list;
  }

type sprogram = bind list * sfunc_decl list

(* Pretty-printing functions *)
let rec string_of_sexpr (t, e) =

  "(" ^ string_of_typ t ^ " : " ^ (match e with
    SLiteral(l) -> string_of_int l
  | SBoolLit(true) -> "true"
  | SBoolLit(false) -> "false"
  | SFliteral(l) -> l
  | SCharLit(l) -> Char.escaped l
  | SMapLit(l) -> "{[" ^ String.concat "," (List.map (fun(k, v) -> string_of_sexpr k ^
":" ^ string_of_sexpr v) l ) ^ "]}"
  | SMapPut(m, k, v) -> string_of_sexpr m ^ ".put(" ^ string_of_sexpr k ^ "," ^
string_of_sexpr v ^ ")"
  | SMapGet(m, k) -> string_of_sexpr m ^ ".get(" ^ string_of_sexpr k ^ ")"
  | SMapContainsKey(m, k) -> string_of_sexpr m ^ ".containsKey(" ^ string_of_sexpr k ^
")"
  | SMapContainsValue(m, v) -> string_of_sexpr m ^ ".containsValue(" ^ string_of_sexpr
v ^ ")"
  | SMapRemoveNode(m, k) -> string_of_sexpr m ^ ".removeNode(" ^ string_of_sexpr k ^
")"
  | SMapIsEqual(m1, m2) -> string_of_sexpr m1 ^ ".isEqual(" ^ string_of_sexpr m2 ^ ")"
  | SListLit(l) -> "[" ^ String.concat "," (List.map string_of_sexpr l) ^ "]"
  | SListSize(l) -> string_of_sexpr l ^ ".len"
  | SListGet(l, idx) -> string_of_sexpr l ^ ".at(" ^ string_of_sexpr idx ^ ")"
```

```
    | SListSet(l, idx, e) -> string_of_sexpr l ^ ".set(" ^ string_of_sexpr idx ^ "," ^
  string_of_sexpr e ^ ")"
    | SList_Add_Head(l, e) -> string_of_sexpr l ^ ".add_head(" ^ string_of_sexpr e^ ")"
    | SList_Rm_Head(l) -> string_of_sexpr l ^ ".remove_head()"
    | SList_Add_Tail(l, e) -> string_of_sexpr l ^ ".add_tail(" ^ string_of_sexpr e ^ ")"
    | SGraphMod(g, l) -> g ^ "{{" ^ String.concat "," (List.map (fun(e) ->
  string_of_sexpr e) l) ^ "}}"
    | SGraphEdges(g, e) -> string_of_sexpr g ^ ".get_edges(" ^ string_of_sexpr e^ ")"
    | SGraphNodes(g, n) -> string_of_sexpr g ^ ".get_neighbors(" ^ string_of_sexpr n ^
  ")"
    | SGraphAllNodes(g) -> string_of_sexpr g ^ ".get_all_nodes()"
    | SGraphAll(g) -> string_of_sexpr g ^ ".get_all_nodes()"
    | SStrLit(l) -> l
    | SId(s) -> s
    | SBinop(e1, o, e2) ->
        string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
    | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
    | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
    | SOpAssign(v,o,e) -> v ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e
    | SCall(f, el) ->
        f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
    | SNoexpr -> ""
                  ) ^ ")"

let rec string_of_sstmt = function
    SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
    | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
    | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
    | SIf(e, s, SBlock([])) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
    | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
    | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s

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

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

Semant

```
(* Semantic checking for the Graphiti compiler *)
(* Authors:
    Lists: Andrew Quijano
```

```
    Graphs: Sydney Lee & Michal Porubcin
    Maps: Emily Hao & Alice Thum
*)


open Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of the AST. Returns an SAST if successful,
     throws an exception if something is wrong.

     Check each global variable, then check each function *)
let get_type(t, _) = t
let first_element (myList) = match myList with
 [] -> Void
| first_e1 :: _ -> get_type(first_e1)

(* Use this function to check if the sexpr is acceptable element in list*)
let valid_element_type = function
         (Void,_) -> raise(Failure("Invalid List<Void>!"))
    | _ -> ()
let check_type (ex, ty) =
    if ex = ty then ()
    else raise (Failure ("Mismatch desired expr and type!"))

let check (globals, functions) =

    (* Verify a list of bindings has no void types*)
    let check_binds (kind : string) (binds : bind list) =
        List.iter (function
    (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
            | _ -> ()) binds;

            (* Verify a list of bindings has no duplicate names*)
            let rec dups = function
                [] -> ()
            |   ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
          raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
                | _ :: t -> dups t
          in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
    in

  let check_list_binds (binds : sexpr list) =
      List.iter valid_element_type binds;

      let rec same_type = function
          [] -> ()
      | ((t1,_) :: (t2,_) :: _) when t1 != t2 ->
              raise (Failure ("List elements must be all same type!"))
```

```
        | _ :: t -> same_type t
        in same_type (List.sort (fun (a,_) (b,_) -> compare a b) binds);

        (*first_element(binds)*)
    in

      (**** Check global variables ****)

      check_binds "global" globals;

      (**** Check functions ****)

      (* Collect function declarations for built-in functions: no bodies *)
      let built_in_decls =
          let add_bind map (name, ty) = StringMap.add name {
              typ = Void;
              fname = name;
              formals = [(ty, "x")];
              locals = [];
              body = [] } map
          in List.fold_left add_bind StringMap.empty [("printi", Int);
                                                      ("printb", Bool);
                                                      ("printf", Float);
                                                      ("printbig", Int);
                                                      ("print", String);
                                                      ("printm", Map);
                                                      ("printl", List(String));
                                                      ("printl", List(Int));
                                                      ("printg", Graph)]
          in

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

      (* Collect all function names into one symbol table *)
      let function_decls = List.fold_left add_func built_in_decls functions
      in

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

```
    in

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

    let check_function func =
        (* Make sure no formals or locals are void or duplicates *)
        check_binds "formal" func.formals;
        check_binds "local" func.locals;

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

        (* Build local symbol table of variables for this function *)
        let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty
m)
                                     StringMap.empty (globals @ func.formals @
func.locals )
        in

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

        (* Return a semantically-checked expression, i.e., with a type *)

        let rec expr =
            let check_list m =
                let (t, _) = expr m in
                match t with
                 List(_) -> ()
                |_ -> raise (Failure ("Expected List<E> instead of " ^
string_of_typ t)) in
            let check_list_type m =
                let (t, _) = expr m in
                match t with
                 List(ty) -> ty
                |_ -> raise (Failure ("List must be of type list instead of " ^
string_of_typ t)) in
            let check_map m =
                let (t, mc) = expr m in
                match t with
                 Map -> (t, mc)
                |_ -> raise (Failure ("Map must be of type map instead of " ^
string_of_typ t)) in
            let check_key k =
                let (t, kc) = expr k in
                match t with
```

```
                    String -> (t, kc)
                  |_ -> raise (Failure ("Key must be type string instead of " ^
string_of_typ t)) in
              let check_value v =
                  let (t, vc) = expr v in
                      match t with
                   String -> (t, vc)
                  |_ -> raise (Failure ("Value must be a string type instead of "
^ string_of_typ t)) in
              let check_node n =
                  let (t, n') = expr n in
                  match t with
                   Map | Void -> (t, n')
                  | _ -> raise (Failure ("Node must be map type instead of " ^
string_of_typ t)) in
              let check_weight w =
                  let (t, n') = expr w in
                  match t with
                   String | Void -> (t, n')
                  | _ -> raise (Failure ("Weight must be a string type instead of
" ^ string_of_typ t)) in
              let check_graph g =
                      let (t, g') = expr g in
                      match t with
                      Graph -> (t, g')
                      | _ -> raise (Failure ("Graph must be graph type instead of
" ^ string_of_typ t)) in

              function
                Literal  l -> (Int, SLiteral l)
              | Fliteral l -> (Float, SFliteral l)
              | BoolLit l  -> (Bool, SBoolLit l)
              | CharLit l -> (Char, SCharLit l)
              | StrLit l -> (String, SStrLit l)
              | Noexpr     -> (Void, SNoexpr)
              | Id s       -> (type_of_identifier s, SId s)
              | ListLit l  -> check_list_binds (List.map expr l);
                  (List(first_element(List.map expr l)), SListLit (List.map expr
l))
              | MapLit l ->
                  let m = List.map (fun (k, v) ->
                  let k' = check_key k in
                  let v' = check_value v in
                  (k', v')) l in (Map, SMapLit m)
              | GraphLit l ->
                  let m = List.map (fun (e) -> expr e) l
                  in (Graph, SGraphLit (m))
              | GraphMod (g, l) ->
                  let m = List.map (fun (e) -> expr e) l
                  in (Graph, SGraphMod (g, m))
              | GraphEdges(g, n) ->
                  let g' = check_graph g
                  and n' = check_node n in
                  (Graph, SGraphEdges(g', n'))
```

```
              | GraphNodes(g, n) ->
                  let g' = check_graph g
                  and n' = check_node n in
                  (Graph, SGraphNodes(g', n'))
              | GraphAllNodes(g) ->
                  let g' = check_graph g in
                  (Graph, SGraphAllNodes(g'))
              | GraphAll(g) ->
                  let g' = check_graph g in
                  (Graph, SGraphAll(g'))
              | GraphAddVertex(n) ->
                  let n' = check_node n in
                  (Graph, SGraphAddVertex(n'))
              | GraphAddEdge(n1, n2) ->
                  let n1' = check_node n1
                  and n2' = check_node n2 in
                  (Graph, SGraphAddEdge(n1', n2'))
              | GraphAddWedge(n1, w, n2) ->
                  let n1' = check_node n1
                  and w' = check_weight w
                  and n2' = check_node n2 in
                  (Graph, SGraphAddWedge(n1', w', n2'))
              | GraphDelVertex(n) ->
                  let n' = check_node n in
                  (Graph, SGraphDelVertex(n'))
              | GraphDelEdge(n1, n2) ->
                  let n1' = check_node n1
                  and n2' = check_node n2 in
                  (Graph, SGraphDelEdge(n1', n2'))
              | Assign(var, e) as ex ->
                      let lt = type_of_identifier var
                      and (rt, e') = expr e in (* recursive *)
                      let err = "illegal assignment " ^ string_of_typ lt ^ " = "
  ^
                          string_of_typ rt ^ " in " ^ string_of_expr ex
                      in (check_assign lt rt err, SAssign(var, (rt, e')))
              (* Add all our operators BELOW *)
              | Unop(op, e) as ex ->
                      let (t, e') = expr e in
                      let ty = match op with
                          Neg when t = Int || t = Float -> t
                      | Not when t = Bool -> Bool
                      | _ -> raise (Failure ("illegal unary operator " ^
  string_of_uop op ^ string_of_typ t ^ " in " ^ string_of_expr ex))
                      in (ty, SUnop(op, (t, e')))
              | Binop(e1, op, e2) as e ->
                      let (t1, e1') = expr e1
                      and (t2, e2') = expr e2 in
                      (* All binary operators require operands of the same type
  *)
                      let same = t1 = t2 in
                      (* Determine expression type based on operator and operand
  types *)
                      let ty = match op with
```

```
                          Add | Sub | Mult | Div | Mod when same && t1 = Int   ->
Int
                      | Add when same && t1 = List(Int) -> List(Int)
                                          | Add when same && t1 = Graph -> Graph
                      | Add when same && t1 = List(String) -> List(String)
                      | Add when same && t1 = List(Map) -> List(Map)
                      | Add when same && t1 = List(Float) -> List(Float)
                                          | Add | Sub | Mult | Div when same &&
t1 = Float -> Float
                                          | Add                     when same &&
t1 = String-> String
                                          | Equal | Neq             when same
-> Bool
                      | Less | Leq | Greater | Geq
                      when same && (t1 = Int || t1 = Float) -> Bool
                      | And | Or when same && t1 = Bool -> Bool
                                          | Union | Intersect when same && t1 =
Graph -> Graph
                      | _ -> raise (
                  Failure ("illegal binary operator " ^ string_of_typ t1 ^ " " ^
string_of_op op ^ " " ^
                                  string_of_typ t2 ^ " in " ^ string_of_expr e))
                      in (ty, SBinop((t1, e1'), op, (t2, e2')))
              | OpAssign(var, op, e) as ex ->
                      let lt = type_of_identifier var
                      and (rt, e2') = expr e in
                      let err = "illegal op-assignment " ^ string_of_typ lt ^ " "
^ string_of_op op ^ " " ^
                          string_of_typ rt ^ " in " ^ string_of_expr ex
                      in (check_assign lt rt err, SOpAssign(var, op, (rt, e2')))

              | Call(fname, args) as call ->
                      let fd = find_func fname in
                      let param_length = List.length fd.formals in
                      if List.length args != param_length then
                          raise (Failure ("expecting " ^ string_of_int
param_length ^
                                                          " arguments in " ^
string_of_expr call))
                      else let check_call (ft, _) e =
                          let (et, e') = expr e in
                          let err = "illegal argument found " ^ string_of_typ et
^
                              " expected " ^ string_of_typ ft ^ " in " ^
string_of_expr e
                          in (check_assign ft et err, e')
                      in
                      let args' = List.map2 check_call fd.formals args
                      in (fd.typ, SCall(fname, args'))

              (* Map Methods *)
              | MapPut(m, k, v) -> let (a, b, c) = (fun (m, k, v) ->
                  let m' = check_map m in
                  let k' = check_key k in
```

```
                let v' = check_value v in
                (m', k', v')) (m, k, v)  in (Int, SMapPut (a, b, c))
            | MapContainsKey(m, k) -> let (a, b) = (fun (m, k) ->
                let m' = check_map m in
                let k' = check_key k in
                (m', k')) (m, k)  in (Int, SMapContainsKey (a, b))
            | MapContainsValue (m, v) -> let (a, b) = (fun (m, v) ->
                let m' = check_map m in
                let v' = check_value v in
                (m', v')) (m, v)  in (Int, SMapContainsValue (a, b))
            | MapGet(m, k) -> let (a, b) = (fun (m, k) ->
                let m' = check_map m in
                let k' = check_key k in
                (m', k')) (m, k)  in (String, SMapGet(a, b))
            | MapRemoveNode(m, k) -> let (a, b) = (fun (m, k) ->
                let m' = check_map m in
                let k' = check_key k in
                (m', k')) (m, k)  in (Int, SMapRemoveNode (a, b))
                | MapIsEqual(m1, m2) -> let (a, b) = (fun (m1, m2) ->
                let m1' = check_map m1 in
                let m2' = check_map m2 in
                (m1', m2')) (m1, m2)  in (Int, SMapIsEqual(a, b))

            (* List Methods *)
            | ListSize l -> check_list(l);
                (Int, SListSize (expr l))
            | ListGet(l, i) -> check_list(l);
                check_type(get_type(expr i), Int);
                (check_list_type(l), SListGet (expr l, expr i))
            | ListSet(l, i, e) -> check_list(l);
                check_type(get_type(expr i), Int);
                valid_element_type(expr e);
                (check_list_type(l), SListSet (expr l, expr i, expr e))
            | List_Add_Head(l, e) -> check_list(l);
                valid_element_type(expr e);
                (Void, SList_Add_Head (expr l, expr e))
            | List_Rm_Head(l) -> check_list(l);
                (check_list_type(l), SList_Rm_Head (expr l))
            | List_Add_Tail(l, e) -> check_list(l);
                valid_element_type(expr e);
                (Void, SList_Add_Tail (expr l, expr e))

            in

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

        (* Return a semantically-checked statement i.e. containing sexprs *)
        let rec check_stmt = function
                Expr e -> SExpr (expr e)
```

```
              | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt
b2)
              | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
              | Return e -> let (t, e') = expr e in
                  if t = func.typ then SReturn (t, e')
                  else raise (
        Failure ("return gives " ^ string_of_typ t ^ " expected " ^
             string_of_typ func.typ ^ " in " ^ string_of_expr e))

              (* A block is correct if each statement is correct and nothing
                   follows any Return statement.  Nested blocks are flattened. *)
              | Block sl ->
                      let rec check_stmt_list = function
                              [Return _ as s] -> [check_stmt s]
                          | Return _ :: _   -> raise (Failure "nothing may follow
a return")
                          | Block sl :: ss  -> check_stmt_list (sl @ ss) (*
Flatten blocks *)
                          | s :: ss         -> check_stmt s :: check_stmt_list ss
                          | []              -> []
                      in SBlock(check_stmt_list sl)

        in (* body of check_function *)
        { styp = func.typ;
            sfname = func.fname;
            sformals = func.formals;
            slocals  = func.locals;
            sbody = match check_stmt (Block func.body) with
    SBlock(sl) -> sl
              | _ -> raise (Failure ("internal error: block didn't become a
block?"))
        }
    in (globals, List.map check_function functions)
```

CodeGen

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

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

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

Detailed documentation on the OCaml LLVM library:

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

Authors:
    Lists: Andrew Quijano
```

```
      Graphs: Sydney Lee & Michal Porubcin
      Maps: Emily Hao & Alice Thum

  *)
  module C = Char
  module L = Llvm
  module A = Ast
  open Ast
  open Sast

  module StringMap = Map.Make(String)

  let get_type(t, _) = t
  let first_element (myList) = match myList with
   [] -> Void
  | first_e1 :: _ -> get_type(first_e1)

  let check_list_type m =
      let (t, _) = m in
      match t with
       List(ty) -> ty
      |_ -> raise (Failure ("List must be of type list in ListLit (CODEGEN): " ^
  string_of_typ t))

  (* translate : Sast.program -> Llvm.module *)
  let translate (globals, functions) =
    let context    = L.global_context () in

    (* Black Magic*)
    let llmem_graph = L.MemoryBuffer.of_file "graph.bc" in
    let llm_graph = Llvm_bitreader.parse_bitcode context llmem_graph in

    (* Create the LLVM compilation module into which
       we will generate code *)
    let the_module = L.create_module context "Graphiti" in

    (* Get types from the context *)
    let i32_t      = L.i32_type    context
    and i8_t       = L.i8_type     context
    and i1_t       = L.i1_type     context
    and float_t    = L.double_type context
    and void_t     = L.void_type   context
    and str_t      = L.pointer_type (L.i8_type context)
    and void_ptr_t = L.pointer_type (L.i8_type context)
    and lst_t      = L.pointer_type (match L.type_by_name llm_graph "struct.list"
  with
        None -> raise (Failure "Missing implementation for struct list")
      | Some t -> t)
    and map_t    = L.pointer_type (match L.type_by_name llm_graph "struct.map"
  with
        None -> raise (Failure "Missing implementation for struct map")
      | Some t -> t)
```

```
    and graph_t = L.pointer_type (match L.type_by_name llm_graph "struct.graph"
  with
        None -> raise (Failure "Missing implementation for struct graph")
      | Some t -> t)
    in

    (* Return the LLVM type for a MicroC type *)
    let ltype_of_typ = function
        A.Int   -> i32_t
      | A.Char  -> i8_t
      | A.Bool  -> i1_t
      | A.Float -> float_t
      | A.Void  -> void_t
      | A.String -> str_t
      | A.Map   -> map_t
      | A.List _ -> lst_t
      | A.Graph  ->  graph_t
    in

    (* Create a map of global variables after creating each *)
    let global_vars : L.llvalue StringMap.t =
      let global_var m (t, n) =
        let init = match t with
            A.Float -> L.const_float (ltype_of_typ t) 0.0
          | _ -> L.const_int (ltype_of_typ t) 0
        in StringMap.add n (L.define_global n init the_module) m in
      List.fold_left global_var StringMap.empty globals in

    (*print functions which built-in prints will call *)
    let printf_t : L.lltype =
        L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
    let printf_func : L.llvalue =
        L.declare_function "printf" printf_t the_module in

    let printbig_t : L.lltype =
        L.function_type i32_t [| i32_t |] in
    let printbig_func : L.llvalue =
        L.declare_function "printbig" printbig_t the_module in

    let printm_t : L.lltype =
        L.function_type map_t [| map_t |] in
    let printm_func : L.llvalue =
        L.declare_function "printm" printm_t the_module in

    let printl_t : L.lltype =
        L.function_type lst_t [| lst_t |] in
    let printl_func : L.llvalue =
        L.declare_function "printl" printl_t the_module in

    let printg_t : L.lltype =
        L.function_type graph_t [| graph_t |] in
    let printg_func : L.llvalue =
```

```
        L.declare_function "printg" printg_t the_module in


    (* Functions for Lists. It is generic so elements are void *! *)
    let make_list_t = L.function_type lst_t [||] in
    let make_list_func = L.declare_function "make_list" make_list_t the_module in

    let list_size_t = L.function_type i32_t [| lst_t |] in
    let list_size_func = L.declare_function "size" list_size_t the_module in

    let list_get_t = L.function_type void_ptr_t [| lst_t; i32_t |] in
    let list_get_func = L.declare_function "list_get" list_get_t the_module in

    let list_add_tail_t = L.function_type i32_t [| lst_t; void_ptr_t |] in
    let list_add_tail_func = L.declare_function "add_tail" list_add_tail_t
  the_module in

    (* Testing casting void inside the C code. Test <int>*)

    let list_set_int_t = L.function_type i32_t [| lst_t; i32_t; i32_t |] in
    let list_set_int_func = L.declare_function "list_set_int" list_set_int_t
  the_module in

    let list_add_head_int_t = L.function_type i32_t [| lst_t; i32_t |] in
    let list_add_head_int_func = L.declare_function "add_head_int"
  list_add_head_int_t the_module in

    let list_rm_head_int_t = L.function_type i32_t [| lst_t |] in
    let list_rm_head_int_func = L.declare_function "remove_head_int"
  list_rm_head_int_t the_module in

    let list_add_tail_int_t = L.function_type i32_t [| lst_t; i32_t |] in
    let list_add_tail_int_func = L.declare_function "add_tail_int"
  list_add_tail_int_t the_module in

    (* Convert Maps to Void * for usage for List<map> *)
    let list_set_map_t = L.function_type map_t [| lst_t; i32_t; map_t |] in
    let list_set_map_func = L.declare_function "list_set_map" list_set_map_t
  the_module in

    let list_add_head_map_t = L.function_type i32_t [| lst_t; map_t |] in
    let list_add_head_map_func = L.declare_function "add_head_map"
  list_add_head_map_t the_module in

    let list_rm_head_map_t = L.function_type map_t [| lst_t |] in
    let list_rm_head_map_func = L.declare_function "remove_head_map"
  list_rm_head_map_t the_module in

    let list_add_tail_map_t = L.function_type i32_t [| lst_t; map_t |] in
    let list_add_tail_map_func = L.declare_function "add_tail_map"
  list_add_tail_map_t the_module in
```

```
•
•     (* Convert strings to Void * for usage for List<string> *)
•     let list_set_str_t = L.function_type str_t [| lst_t; i32_t; str_t |] in
•     let list_set_str_func = L.declare_function "list_set_str" list_set_str_t
    the_module in
•
•     let list_add_head_str_t = L.function_type i32_t [| lst_t; str_t |] in
•     let list_add_head_str_func = L.declare_function "add_head_str"
    list_add_head_str_t the_module in
•
•     let list_rm_head_str_t = L.function_type str_t [| lst_t |] in
•     let list_rm_head_str_func = L.declare_function "remove_head_str"
    list_rm_head_str_t the_module in
•
•     let list_add_tail_str_t = L.function_type i32_t [| lst_t; str_t |] in
•     let list_add_tail_str_func = L.declare_function "add_tail_str"
    list_add_tail_str_t the_module in
•
•     (* Convert float/decimal to Void * for usage for List<float> *)
•     let list_set_dec_t = L.function_type float_t [| lst_t; i32_t; float_t |] in
•     let list_set_dec_func = L.declare_function "list_set_dec" list_set_dec_t
    the_module in
•
•     let list_add_head_dec_t = L.function_type i32_t [| lst_t; float_t  |] in
•     let list_add_head_dec_func = L.declare_function "add_head_dec"
    list_add_head_dec_t the_module in
•
•     let list_rm_head_dec_t = L.function_type float_t [| lst_t |] in
•     let list_rm_head_dec_func = L.declare_function "remove_head_dec"
    list_rm_head_dec_t the_module in
•
•     let list_add_tail_dec_t = L.function_type i32_t [| lst_t; float_t |] in
•     let list_add_tail_dec_func = L.declare_function "add_tail_dec"
    list_add_tail_dec_t the_module in
•
•     (* Functions for maps Map *)
•     let make_map_t = L.function_type map_t [||] in
•     let make_map_func = L.declare_function "make_map" make_map_t the_module in
•
•     let map_contains_key_t = L.function_type i32_t [| map_t; str_t |] in
•     let map_contains_key_func = L.declare_function "contains_key"
    map_contains_key_t the_module in
•
•     let map_contains_value_t = L.function_type i32_t [| map_t; str_t |] in
•     let map_contains_value_func = L.declare_function "contains_value"
    map_contains_value_t the_module in
•
•     let map_put_t = L.function_type i32_t [| map_t; str_t; str_t |] in
•     let map_put_func = L.declare_function "put" map_put_t the_module in
•
•     let map_get_t = L.function_type str_t [|map_t; str_t|] in
```

```
    let map_get_func = L.declare_function "map_get" map_get_t the_module in

    let map_remove_node_t = L.function_type i32_t [| map_t; str_t|] in
    let map_remove_node_func = L.declare_function "remove_node" map_remove_node_t
  the_module in

    let map_is_equal_t = L.function_type i32_t [| map_t; map_t|] in
    let map_is_equal_func = L.declare_function "is_equal" map_is_equal_t
  the_module in

    (* Function for graphs *)
    let graph_constructor_t = L.function_type graph_t [||] in
    let graph_constructor_f = L.declare_function "new_graph" graph_constructor_t
  the_module in

    let graph_add_vertex_t = L.function_type graph_t [|graph_t; map_t|] in
    let graph_add_vertex_f = L.declare_function "add_vertex" graph_add_vertex_t
  the_module in

    let graph_add_edge_t = L.function_type graph_t [|graph_t; map_t; map_t|] in
    let graph_add_edge_f = L.declare_function "add_edge" graph_add_edge_t
  the_module in

    let graph_add_wedge_t = L.function_type graph_t [|graph_t; map_t; str_t;
  map_t|] in
    let graph_add_wedge_f = L.declare_function "add_wedge" graph_add_wedge_t
  the_module in

    let graph_del_edge_t = L.function_type graph_t [|graph_t; map_t; map_t|] in
    let graph_del_edge_f = L.declare_function "delete_edge" graph_del_edge_t
  the_module in

    let graph_del_vertex_t = L.function_type graph_t [|graph_t; map_t|] in
    let graph_del_vertex_f = L.declare_function "delete_vertex"
  graph_del_vertex_t the_module in

    let graph_union_t = L.function_type graph_t [| graph_t; graph_t |] in
    let graph_union_f = L.declare_function "union_graph" graph_union_t the_module
  in

    let graph_intersection_t = L.function_type graph_t [| graph_t; graph_t |] in
    let graph_intersection_f = L.declare_function "intersection_graph"
  graph_intersection_t the_module in

    let graph_add_t = L.function_type graph_t [| graph_t; graph_t |] in
    let graph_add_f = L.declare_function "add" graph_add_t the_module in

    let graph_get_edges_t = L.function_type lst_t [|graph_t; map_t|] in
    let graph_get_edges_f = L.declare_function "_get_edges" graph_get_edges_t
  the_module in

```

```
    let graph_get_nodes_t = L.function_type lst_t [|graph_t; map_t|] in
    let graph_get_nodes_f = L.declare_function "get_edge_neighbors"
  graph_get_nodes_t the_module in

    let graph_get_all_nodes_t = L.function_type lst_t [|graph_t|] in
    let graph_get_all_nodes_f = L.declare_function "get_all_vertices"
  graph_get_all_nodes_t the_module in

    (* Miscellanous functions, string ops, list concat, etc.*)
    let concat_string_t = L.function_type str_t [| str_t; str_t |] in
    let concat_string_func = L.declare_function "concat_string" concat_string_t
  the_module in

    let length_t = L.function_type i32_t [| str_t |] in
    let length_func = L.declare_function "length" length_t the_module in

    let get_char_t = L.function_type str_t [| str_t; i32_t |] in
    let get_char_func = L.declare_function "get_char" get_char_t the_module in

    let string_equals_t = L.function_type i32_t [| str_t; str_t |] in
    let string_equals_func = L.declare_function "str_comp" string_equals_t
  the_module in

    let concat_list_t = L.function_type lst_t [| lst_t; lst_t |] in
    let concat_list_func = L.declare_function "concat" concat_list_t the_module
  in

    (* Define each function (arguments and return type) so we can
       call it even before we've created its body *)
    let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
      let function_decl m fdecl =
        let name = fdecl.sfname
        and formal_types =
          Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
        in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types in
        StringMap.add name (L.define_function name ftype the_module, fdecl) m in
      List.fold_left function_decl StringMap.empty functions in

    (* Fill in the body of the given function *)
    let build_function_body fdecl =
      let (the_function, _) = StringMap.find fdecl.sfname function_decls in
      let builder = L.builder_at_end context (L.entry_block the_function) in

      let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
      and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
      and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder
      in

      (* Construct the function's "locals": formal arguments and locally
         declared variables.  Allocate each on the stack, initialize their
         value, if appropriate, and remember their values in the "locals" map *)
      let local_vars =
```

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

        (* Allocate space for any locally declared variables and add the
         * resulting registers to our map *)
        and add_local m (t, n) =
          let local_var = L.build_alloca (ltype_of_typ t) n builder
          in StringMap.add n local_var m
        in

        (*let sformals = (List.map fst_snd_triple fdecl.sformals) in*)
        let formals = List.fold_left2 add_formal StringMap.empty fdecl.sformals
            (Array.to_list (L.params the_function)) in
        List.fold_left add_local formals fdecl.slocals
      in

      (* Return the value for a variable or formal argument.
         Check local names first, then global names *)
      let lookup n = try StringMap.find n local_vars
                     with Not_found -> StringMap.find n global_vars
      in

      (* Construct code for an expression; return its value *)
      let rec expr builder ((styp, e) : sexpr) =

      match e with
          SLiteral i  -> L.const_int i32_t i
        | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
        | SFliteral l -> L.const_float_of_string float_t l
        | SStrLit l    -> L.build_global_stringptr l "str" builder
        | SNoexpr     -> L.const_int i32_t 0
        | SId s       -> L.build_load (lookup s) s builder
        | SCharLit l -> L.const_int i8_t (C.code l)
        | SListLit l -> let rec list_fill lst = (function
          [] -> lst
          | sx :: rest ->
          let (t, _) = sx in
          let data = (match t with
                A.Map |A.Graph |A.List _ | A.String -> expr builder sx
              | _ -> let data = L.build_malloc (ltype_of_typ t) "data" builder in
                  let llvm =  expr builder sx
                  in ignore(L.build_store llvm data builder); data)
          in let data = L.build_bitcast data void_ptr_t "data" builder in
              ignore(L.build_call list_add_tail_func [| lst; data |]
    "list_add_tail" builder); list_fill lst rest) in
          let m = L.build_call make_list_func [||] "make_list" builder in
          list_fill m l
        | SListSize(l) -> let l' = expr builder l in
              L.build_call list_size_func [|l'|] "size" builder;
        | SListGet(l, idx) ->
```

```
          let ltype = ltype_of_typ styp in
          let lst = expr builder l in
          let index = expr builder idx in
          let data = L.build_call list_get_func [| lst; index |] "index" builder
in
          (match styp with
              A.List _ | A.Graph | A.String | A.Map -> L.build_bitcast data ltype
"data" builder
            | _ -> let data = L.build_bitcast data (L.pointer_type ltype)
"data" builder in
                L.build_load data "data" builder)
      | SListSet(l, idx, e) ->
            let r = (match check_list_type(l) with
             A.Int -> let l' = expr builder l and idx' = expr builder idx and
e' = expr builder e in
                L.build_call list_set_int_func [|l'; idx'; e'|] "list_set_int"
builder;
            | A.Map -> let l' = expr builder l and idx' = expr builder idx and
e' = expr builder e in
                L.build_call list_set_map_func [|l'; idx'; e'|] "list_set_map"
builder;
            | A.String -> let l' = expr builder l and idx' = expr builder idx
and e' = expr builder e in
                L.build_call list_set_str_func [|l'; idx'; e'|] "list_set_str"
builder;
            | A.Float -> let l' = expr builder l and idx' = expr builder idx
and e' = expr builder e in
                L.build_call list_set_dec_func [|l'; idx'; e'|] "list_set_dec"
builder;
            | _ -> raise(Failure("Not Valid List Lit Type!"))) in
            r
      | SList_Add_Head(l, e) ->
            let r = (match get_type(e) with
             A.Int -> let l' = expr builder l and e' = expr builder e in
                 L.build_call list_add_head_int_func [|l'; e'|] "add_head_int"
builder;
            | A.Map -> let l' = expr builder l and e' = expr builder e in
                 L.build_call list_add_head_map_func [|l'; e'|] "add_head_map"
builder;
            | A.String -> let l' = expr builder l and e' = expr builder e in
                 L.build_call list_add_head_str_func [|l'; e'|] "add_head_str"
builder;
            | A.Float -> let l' = expr builder l and e' = expr builder e in
                 L.build_call list_add_head_dec_func [|l'; e'|] "add_head_dec"
builder;
            | _ -> raise(Failure("Not Valid List Lit Type!"))) in
            r
      | SList_Rm_Head(l) ->
            let r = (match check_list_type(l) with
             A.Int -> let l' = expr builder l in
                L.build_call list_rm_head_int_func [|l'|] "remove_head_int"
builder;
            | A.Map -> let l' = expr builder l in
```

```
                L.build_call list_rm_head_map_func [|l'|] "remove_head_map"
builder;
            | A.String -> let l' = expr builder l in
                L.build_call list_rm_head_str_func [|l'|] "remove_head_str"
builder;
            | A.Float -> let l' = expr builder l in
                L.build_call list_rm_head_dec_func [|l'|] "remove_head_dec"
builder;
            | _ -> raise(Failure("Not Valid List Lit Type!"))) in
            r
      | SList_Add_Tail(l, e) -> let r = (match get_type(e) with
             A.Int -> let l' = expr builder l and e' = expr builder e in
                L.build_call list_add_tail_int_func [|l'; e'|] "add_tail_int"
builder;
            | A.Map -> let l' = expr builder l and e' = expr builder e in
                L.build_call list_add_tail_map_func [|l'; e'|] "add_tail_map"
builder;
            | A.String -> let l' = expr builder l and e' = expr builder e in
                L.build_call list_add_tail_str_func [|l'; e'|] "add_tail_str"
builder;
            | A.Float -> let l' = expr builder l and e' = expr builder e in
                L.build_call list_add_tail_dec_func [|l'; e'|] "add_tail_dec"
builder;
            | _ -> raise(Failure("Not Valid List Lit Type!"))) in
            r
      | SMapLit l ->
          let m = L.build_call make_map_func [||] "make_map" builder in
          List.iter (fun (k, v) -> ignore(
              let k' = expr builder k
              and v' = expr builder v in
              L.build_call map_put_func [| m; k'; v'|] "put" builder)) l;
          m
      | SGraphLit l ->
          let g = L.build_call graph_constructor_f [||] "new_graph" builder in
          List.iter (fun (_, e) -> ignore(
              match e with
              | SGraphAddVertex(n) ->
                  let n' = expr builder n in
                  L.build_call graph_add_vertex_f [|g; n'|] "add_vertex"
builder
              | SGraphAddEdge (n1, n2) ->
                  let n1' = expr builder n1
                  and n2' = expr builder n2 in
                  L.build_call graph_add_edge_f [|g; n1'; n2'|] "add_edge"
builder
              | SGraphAddWedge (n1, w, n2) ->
                  let n1' = expr builder n1
                  and w' = expr builder w
                  and n2' = expr builder n2 in
                  L.build_call graph_add_wedge_f [|g; n1'; w'; n2'|]
"add_wedge" builder
              | _ -> raise(Failure("Unsupported operation.")))) l;
          g

```

```
        | SGraphMod (g, l) ->
            let g' = L.build_load (lookup g) g builder in
            List.iter (fun (_,e) -> ignore(
                match e with
                  SGraphAddVertex(n) ->
                    let n' = expr builder n in
                    L.build_call graph_add_vertex_f [|g'; n'|] "graph_add_vertex"
    builder
                | SGraphAddEdge (n1, n2) ->
                    let n1' = expr builder n1
                    and n2' = expr builder n2 in
                    L.build_call graph_add_edge_f [|g'; n1'; n2'|]
    "graph_add_edge" builder
                | SGraphAddWedge (n1, w, n2) ->
                    let n1' = expr builder n1
                    and w' = expr builder w
                    and n2' = expr builder n2 in
                    L.build_call graph_add_wedge_f [|g'; n1'; w'; n2'|]
    "graph_add_wedge" builder
                | SGraphDelVertex (n) ->
                    let n' = expr builder n in
                    L.build_call graph_del_vertex_f [|g'; n'|] "graph_del_vertex"
    builder
                | SGraphDelEdge (n1, n2) ->
                    let n1' = expr builder n1
                    and n2' = expr builder n2 in
                    L.build_call graph_del_edge_f [|g'; n1'; n2'|]
    "graph_del_edge" builder
                | _ -> raise(Failure("Unsupported operation.")))) l;
            g'
        | SAssign (s, e) -> let e' = expr builder e in
            ignore(L.build_store e' (lookup s) builder); e'

        | SBinop ((A.List(_), _) as e1, op, e2) ->
          let e1' = expr builder e1
          and e2' = expr builder e2 in
          (match op with
          A.Add -> L.build_call concat_list_func [| e1'; e2' |] "concat" builder
                  | _ -> raise(Failure("Invalid List operator " ^ A.string_of_op
    op)))

        | SBinop ((A.String, _) as e1, op, e2) ->
          let e1' = expr builder e1
          and e2' = expr builder e2 in
          (match op with
          A.Add -> L.build_call concat_string_func [| e1'; e2' |] "concat_string"
    builder
              | A.Equal -> (L.build_icmp L.Icmp.Ne) (L.const_int i32_t 0)
              (L.build_call string_equals_func [| e1'; e2' |] "str_comp" builder)
    "tmp" builder
              | A.Neq -> (L.build_icmp L.Icmp.Eq) (L.const_int i32_t 0)
              (L.build_call string_equals_func [| e1'; e2' |] "str_comp" builder)
    "tmp" builder
          | _ -> raise(Failure("Invalid string operator " ^ A.string_of_op op)))
```

```
•
•       | SBinop ((A.Float,_ ) as e1, op, e2) ->
•           let e1' = expr builder e1
•           and e2' = expr builder e2 in
•           (match op with
•             A.Add     -> L.build_fadd
•           | A.Sub     -> L.build_fsub
•           | A.Mult    -> L.build_fmul
•           | A.Div     -> L.build_fdiv
•           | A.Mod     -> L.build_frem
•           | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
•           | A.Neq     -> L.build_fcmp L.Fcmp.One
•           | A.Less    -> L.build_fcmp L.Fcmp.Olt
•           | A.Leq     -> L.build_fcmp L.Fcmp.Ole
•           | A.Greater -> L.build_fcmp L.Fcmp.Ogt
•           | A.Geq     -> L.build_fcmp L.Fcmp.Oge
•           | A.And
•           | A.Or ->
•               raise (Failure "internal error: semant should have rejected
  and/or on float")
•           | A.Union | A.Intersect  -> raise (Failure "only graph can be
  (union/intersect)ed")
•           ) e1' e2' "tmp" builder
•       | SBinop ((A.Graph, _) as e1, op, e2) ->
•           let e1' = expr builder e1
•           and e2' = expr builder e2 in
•           (match op with
•             A.Add -> L.build_call graph_add_f [|e1'; e2'|] "graph_add" builder
•           | A.Union     -> L.build_call graph_union_f [| e1'; e2' |]
  "graph_union" builder
•           | A.Intersect -> L.build_call graph_intersection_f [| e1'; e2' |]
  "graph_intersection" builder
•           | _ -> raise (Failure "graph can only be (union/intersect)ed"))
•       | SBinop (e1, op, e2) ->
•           let e1' = expr builder e1
•           and e2' = expr builder e2 in
•           (match op with
•             A.Add     -> L.build_add
•           | A.Sub     -> L.build_sub
•           | A.Mult    -> L.build_mul
•           | A.Div     -> L.build_sdiv
•           | A.Mod     -> L.build_srem
•           | A.And     -> L.build_and
•           | A.Or      -> L.build_or
•           | A.Equal   -> L.build_icmp L.Icmp.Eq
•           | A.Neq     -> L.build_icmp L.Icmp.Ne
•           | A.Less    -> L.build_icmp L.Icmp.Slt
•           | A.Leq     -> L.build_icmp L.Icmp.Sle
•           | A.Greater -> L.build_icmp L.Icmp.Sgt
•           | A.Geq     -> L.build_icmp L.Icmp.Sge
•           | A.Union | A.Intersect -> raise (Failure "only graph can be
  (union/intersect)ed")
•       ) e1' e2' "tmp" builder
•       | SUnop(op, ((t, _) as e)) ->
```

```
            let e' = expr builder e in
            (match op with
              A.Neg when t = A.Float -> L.build_fneg
            | A.Neg                  -> L.build_neg
            | A.Not                  -> L.build_not) e' "tmp" builder
        | SCall ("length", [e]) ->
          L.build_call length_func [| (expr builder e) |] "length" builder
        | SCall ("get_char", [str;index]) ->
                 let index = expr builder index and
                 str = expr builder str in
                 L.build_call get_char_func [| str; index |] "get_char" builder
        | SCall ("printl", [e]) ->
                 L.build_call printl_func [| (expr builder e) |] "printl"
    builder
        | SCall ("printi", [e]) | SCall ("printb", [e]) ->
            L.build_call printf_func [| int_format_str ; (expr builder e) |]
              "printf" builder
        | SCall ("printbig", [e]) ->
            L.build_call printbig_func [| (expr builder e) |] "printbig" builder
        | SCall ("printf", [e]) ->
            L.build_call printf_func [| float_format_str ; (expr builder e) |]
              "printf" builder
        | SCall ("print", [e]) ->
            L.build_call printf_func [| string_format_str ; (expr builder e) |]
    "printf" builder
        (* Built-in print functions for maps and graphs *)
        | SCall ("printm", [e]) ->
            L.build_call printm_func [| (expr builder e) |] "printm" builder
        | SCall ("printg", [e]) ->
            L.build_call printg_func [| (expr builder e) |] "printg" builder
        | SCall (f, args) ->
           let (fdef, fdecl) = StringMap.find f function_decls in
           let llargs = List.rev (List.map (expr builder) (List.rev args)) in
           let result = (match fdecl.styp with
                           A.Void -> ""
                         | _ -> f ^ "_result") in
           L.build_call fdef (Array.of_list llargs) result builder
        | SGraphEdges (g, n) ->
          let graph = expr builder g
          and node = expr builder n in
          L.build_call graph_get_edges_f [|graph; node|] "_get_edges" builder
        | SGraphNodes (g, n) ->
          let graph = expr builder g
          and node = expr builder n in
          L.build_call graph_get_nodes_f [|graph; node|] "get_edge_neighbors"
    builder
        | SGraphAllNodes (g) ->
          let graph = expr builder g in
          L.build_call graph_get_all_nodes_f [|graph|] "get_all_nodes" builder
       | SGraphAll (g) ->
          let graph = expr builder g in
          L.build_call graph_get_all_nodes_f [|graph|] "get_all_vertices" builder

        (* Map Methods*)
```

```
        | SMapPut (m, k, v) ->
          let map = expr builder m
          and key = expr builder k
          and value = expr builder v in
          L.build_call map_put_func [|map; key; value|] "put" builder;
        | SMapGet (m, k) ->
          let map = expr builder m
          and key = expr builder k in
          L.build_call map_get_func [|map;key|] "map_get" builder;
        | SMapContainsKey (m, k) ->
          let map = expr builder m
          and key = expr builder k in
          L.build_call map_contains_key_func [|map; key|] "contains_key" builder;
        | SMapContainsValue (m, v) ->
          let map = expr builder m
          and value = expr builder v in
          L.build_call map_contains_value_func [|map; value|] "contains_value"
builder;
        | SMapRemoveNode(m, k) ->
          let map = expr builder m
          and key = expr builder k in
          L.build_call map_remove_node_func [|map; key|] "remove_node" builder;
        | SMapIsEqual(m1, m2) ->
                  let map1 = expr builder m1
                  and map2 = expr builder m2 in
                  L.build_call map_is_equal_func [|map1; map2|] "is_equal"
builder;

        | _ -> raise(Failure("Unsupported operation."))
        in

      (* LLVM insists each basic block end with exactly one "terminator"
         instruction that transfers control.  This function runs "instr builder"
         if the current block does not already have a terminator.  Used,
         e.g., to handle the "fall off the end of the function" case. *)
      let add_terminal builder instr =
        match L.block_terminator (L.insertion_block builder) with
          Some _ -> ()
        | None -> ignore (instr builder) in

      (* Build the code for the given statement; return the builder for
         the statement's successor (i.e., the next instruction will be built
         after the one generated by this call) *)

      let rec stmt builder = function
          SBlock sl -> List.fold_left stmt builder sl
        | SExpr e -> ignore(expr builder e); builder
        | SReturn e -> ignore(match fdecl.styp with
                                (* Special "return nothing" instr *)
                               A.Void -> L.build_ret_void builder
                               (* Build return statement *)
                             | _ -> L.build_ret (expr builder e) builder );
                       builder
```

```
        | SIf (predicate, then_stmt, else_stmt) ->
           let bool_val = expr builder predicate in
           let merge_bb = L.append_block context "merge" the_function in
           let build_br_merge = L.build_br merge_bb in (* partial function *)

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

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

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

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

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

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

            let merge_bb = L.append_block context "merge" the_function in
            ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
            L.builder_at_end context merge_bb
          in
      (* Build the code for each statement in the function *)
      let builder = stmt builder (SBlock fdecl.sbody) in

      (* Add a return if the last block falls off the end *)
      add_terminal builder (match fdecl.styp with
          A.Void -> L.build_ret_void
        | A.Float -> L.build_ret (L.const_float float_t 0.0)
        | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
    in

    List.iter build_function_body functions;
    the_module
```

C Library

graph.h

```
#ifndef GRAPH_H
#define GRAPH_H
```

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

/*
 * MAP METHODS
 */

/*
 * Map node declaration.
 */
struct map_node {
    char *key;
    char *value;
    struct map_node *next;
};

/*
 * Map declaration.
 */
struct map {
    struct map_node *node_head;
    int size;
};

/*
 * Initializes a map.
 */
struct map * make_map();

/*
 * Puts a key-value pair into a map.
 * Returns a 1 if successful and 0 otherwise.
 */
int put(struct map *m, char *key, char *value);

/*
 * Gets a value from a map given a key.
 */
char * map_get(struct map *m, char *key);

/*
 * Returns 1 if a key is found in a map
 * and 0 otherwise.
 */
int contains_key(struct map *m, char *key);

/*
 * Returns 1 if a value is found in a map
 * and 0 otherwise.
```

```
 */
int contains_value(struct map *m, char *value);

/*
 * Removes a node from a map.
 * Returns a 1 if successful and 0 otherwise.
 * (e.g. empty list, list does not contain key)
 */
int remove_node(struct map *m, char *key);

/*
 * Compares two maps for equality.
 */
int is_equal(struct map *m1, struct map *m2);

/*
 * Frees allocated memory for a map.
 */
void free_map(struct map *m);

/*
 * Prints out map in specified format:
 * {
 *  "michal" : "language guru",
 *  "emily" : "tester"
 * }
 */
void printm(struct map *m);


/*
 * GRAPH METHODS
 */

struct edge {

    struct vertex *from;
    struct vertex *to;
    struct edge *next;
    char *data;

};

struct vertex {

    struct edge *connected_edges;
    struct vertex *next_vertex;
    struct map *data; /*should this be a void pointer or a struct map pointer
*/

};
```

```
struct graph {

    int vertex_count;
    int edge_count;
    struct vertex *vertex_head;

};


/*
 * creates a new graph
 */
struct graph * new_graph();

/*
 * makes and returns new vertex with data passed in
 */
struct vertex * new_vertex(struct map *data);

/*
 * creates new struct from a to b with the input data
 */
struct edge * new_edge(struct vertex *a, struct vertex *b, char *data);

/*
 * decides whether to call modify vertext or delete vertex *
 */
void modify_graph(struct graph *g, struct map *from, char *data, struct map
*to, int dec);

/*
 * create a new vertex with map data and adds it to g, returns vertex created
 */
void add_vertex(struct graph *g, struct map *data);

/*
 * removes the vertex from graph g and returns it
 */
void delete_vertex(struct graph *g, struct map *data);

/*
 * finds the vertex given the map and returns it, returns null if nothing found
 */
struct vertex * get_vertex(struct graph *g, struct map *data);

/*
 * modifies the vertx given the new data -
 * deletes the old node from graph g and adds new graph into g
 */
void modify_vertex(struct graph *g, struct map *old, struct map *new);
```

```
•
•
•  /*
•   * adds new edge of given data between two given nodes
•   */
•  void _add_edge(struct graph *g, struct map *a, struct map *b, char *data);
•
•  /*
•   * adds new edge with empty data
•   */
•  void add_edge(struct graph *g, struct map *a, struct map *b);
•
•  /*
•   * adds new edge with given data
•   */
•  void add_wedge(struct graph *g, struct map *a, char *data, struct map *b);
•
•  /*
•   * deletes an edge given the two nodes the edge is between
•   */
•  void delete_edge(struct graph *g, struct map *from, struct map *to);
•
•  /*
•   * checks if the edge between two given verticies in graph g exists
•   * returns a boolean - 1 if yes, 0 if no
•   */
•  int _find_edge(struct graph *g, struct map *from, struct map *to);
•
•  /*
•   * modifies edge between two verticies in graph g by deleting the edge
•   * and creating a new edge for it to be in between
•   */
•  void _modify_edge(struct graph *g, struct map *v, struct map *f, char *data);
•
•  /*
•   * intersection
•   */
•  struct graph * intersection_graph(struct graph *g, struct graph *h);
•
•  /*
•   * union
•   * */
•  struct graph * union_graph(struct graph *g, struct graph *h);
•
•  /*
•   * adds the two given graphs together and returns resulting graph
•   */
•  struct graph * add (struct graph *g, struct graph *h);
•
•  /*
•   * given a graph and a node, return all the edges given that node
```

```
 */
struct list * _get_edges(struct graph *g, struct map *data);

/*
 * get all the nodes of a graph
 */
struct list * get_all_vertices(struct graph *g);

struct list * get_edge_neighbors(struct graph *g, struct map *data);

/*
 * print functions
 */
void printg(struct graph *g);
void _print_edge(struct edge *e);
void print_vertex(struct map *m);


/* Clean up methods */
void _clean_graph(struct graph *G);
void _free_adjacency_row(struct vertex *V);
void _free_all_vertex(struct graph *g);
void _free_vertex(struct vertex *v);
void _free_edge(struct edge *e);

/*
char * to_string(int i) {
    char buffer[20];
    sprintf(buffer,"%d",i);
    return buffer;
}

int to_int(char *s) {
    return atoi(s);
}

int randint(int low, int high) {
    return rand() % (high + 1 - low) + low
}
*/

/*
 * LIST METHODS
 */

/*
 * A node in a linked list.
 */

union data_type {
    int i;
```

```
    float f;
    char * s;
    struct map * m;
};

struct list_node {
    void * data;
    //union data_type d;
    struct list_node * next;
};

/*
 * A linked list.
 * 'head' points to the first node in the list.
 *
 * This will be called generically as in
 * list<int> data;
 * list<map> data;
 */
struct list {
    int size;
    struct list_node * head;
};

/*
 * Initializes an empty list.
 */
struct list * make_list();

/*
 * Returns the size of a list.
 */
int size(struct list *l);

/*
 * Returns the data of element at index i of the list.
 */
void * list_get(struct list *l, int i);

/*
 * Sets the element at index i to data.
 * Will return 1 if successful (valid i)
 * and 0 otherwise.
 */
int set(struct list *l, int i, void *data);

/*
 * Adds to the front of the list.
 * Returns a 1 if successful and 0 otherwise.
 */
int add_head(struct list *l, void *data);

```

```
/*
 * Adds to the end of the list.
 * Returns a 1 if successful and 0 otherwise.
 */
int add_tail(struct list *l, void *data);

/*
 * Returns data from the head of the list and removes it.
 */
void * remove_head(struct list *l);

/*
 * Returns data from the tail of a list and removes it.
 */
void * remove_tail(struct list *l);

/*
 * Frees allocated memory for a list.
 */
void free_list(struct list *l);

/*
 * Prints out list of elements in a list.
 */
void printl(struct list *l);

/*
 * Build a new list that is concatenating two lists
 */
struct list * concat(struct list * a, struct list * b);

/* Testing the int casting to void!*/
int list_get_int (struct list * l, int index);
int list_set_int (struct list * l, int index, int E);
int add_head_int (struct list * l, int data);
int remove_head_int (struct list * l);
int add_tail_int (struct list * l, int data);
int remove_tail_int (struct list * l);

/* Use for double/decimals */
double list_get_dec (struct list * l, int index);
double list_set_dec (struct list * l, int index, double E);
int add_head_dec (struct list * l, double data);
double remove_head_dec (struct list * l);
int add_tail_dec (struct list * l, double data);
double remove_tail_dec (struct list * l);

/* Use for strings */
char * list_get_str (struct list * l, int index);
char * list_set_str (struct list * l, int index, char * E);
int add_head_str (struct list * l, char * data);
char * remove_head_str (struct list * l);
```

```
int add_tail_str (struct list * l, char * data);
char * remove_tail_str (struct list * l);

/* Use for maps */
struct map * list_get_map (struct list * l, int index);
struct map * list_set_map (struct list * l, int index, struct map * E);
int add_head_map (struct list * l, struct map * data);
struct map * remove_head_map (struct list * l);
int add_tail_map (struct list * l, struct map * data);
struct map * remove_tail_map (struct list * l);

/* Misc Methods. Strops, atoi, etc. */
int * random_int(int minimum_number, int max_number);
char * myItoa(int num);
int * myAtoi(char * str);
char * concat_string(char * a, char * b);
int length(char * s);
int str_comp(char * a, char * b);

// Ocaml views chars as ints. Ocaml should have a convert back! C.decode()?
// See Ocaml Char module!
char * get_char(char * s, int idx);

#endif
```

graph.c

```
#include "graph.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/*
 * MAP METHODS
 */

/*
 * Initializes an empty map.
 */
struct map * make_map() {

    struct map *m;
    m = malloc(sizeof(struct map));
    if (m == NULL)
        return NULL;

    m->node_head = NULL;
    m->size = 0;
    return m;
}

/*
```

```
 * Puts a key-value pair into a map.
 * Returns a 1 if successful and 0 otherwise.
 */
int put(struct map *m, char *key, char *value) {

    /* no duplicate keys allowed */
    if (contains_key(m, key))
        return 0;

    struct map_node *node;
    node = malloc(sizeof(struct map_node));
    if (node == NULL)
        return 0;

    node->key = key;
    node->value = value;

    if (m->node_head == NULL) {
        node->next = NULL;
        m->node_head = node;
    } else {
        node->next = m->node_head;
        m->node_head = node;
    }

    m->size += 1;
    return 1;
}

/*
 * Gets a value from a map given a key.
 */
char * map_get(struct map *m, char *key) {

    if (m->node_head == NULL)
        return NULL;

    struct map_node *current = m->node_head;
    while (current != NULL) {
        if (strcmp(current->key, key) == 0)
            return current->value;
        current = current->next;
    }
    return NULL;
}

/*
 * Returns 1 if a key is found in a map
 * and 0 otherwise.
 */
int contains_key(struct map *m, char *key) {

```

```
    if (m->node_head == NULL)
        return 0;

    struct map_node *current = m->node_head;
    while (current != NULL) {
        if (strcmp(current->key, key) == 0)
            return 1;
        current = current->next;
    }
    return 0;
}

/*
 * Returns 1 if a value is found in a map
 * and 0 otherwise.
 */
int contains_value(struct map *m, char *value) {

    if (m->node_head == NULL)
        return 0;

    struct map_node *current = m->node_head;
    while (current != NULL) {
        if (strcmp(current->value, value) == 0)
            return 1;
        current = current->next;
    }
    return 0;
}

/*
 * Removes a node from a map.
 * Returns a 1 if successful and a 0 otherwise.
 * (e.g. empty list, list does not contain key)
 */
int remove_node(struct map *m, char *key) {

    if (m->node_head == NULL)
        return 0;

    if (contains_key(m, key) == 0)
        return 0;

    struct map_node *current = m->node_head;
    struct map_node *prev = NULL;

    while (current != NULL) {

        if (strcmp(current->key, key) == 0) {
            /* removing the head of the list */
            if (current == m->node_head) {
                m->node_head = current->next;
```

```
                m->size -= 1;
                return 1;
            }
            prev->next = current->next;
            m->size -= 1;
            return 1;
        }

        if (prev == NULL) prev = m->node_head;
        else prev = prev->next;
        current = current->next;
    }
    return 1;
}

/*
 * Compares two maps for equality.
 */
int is_equal(struct map *m1, struct map *m2) {

    if (m1->size != m2->size)
        return 0;

    struct map_node *c1 = m1->node_head;
    while (c1 != NULL) {
        if (contains_key(m2, c1->key) == 0)
            return 0;
        if (map_get(m2, c1->key) != c1-> value)
            return 0;
        c1 = c1->next;
    }
    return 1;
}

/*
 * Frees allocated memory for a map.
 */
void free_map(struct map *m) {

    if (m->node_head != NULL) {
        /* free all individual nodes */
        struct map_node *current = m->node_head;
        struct map_node *after;

        while (current != NULL) {
            after = current->next;
            free(current);
            current = after;
        }
    }
    free(m);
}
```

```
 
 /*
  * Prints out map in specified format:
  * {
  *  "michal" : "language guru",
  *  "emily" : "tester"
  * }
  */
  void printm(struct map *m) {
     printf("{\n");

     struct map_node *current = m->node_head;
     while (current != NULL) {

         printf("\t\"%s\" : \"%s\"", current->key, current->value);
         if (current->next != NULL)
             printf(",\n");
         else
             printf("\n");
         current = current->next;
     }

     printf("}\n");
  }

 /*
  * GRAPH METHODS
  */

 /*
  * decides whether a graph should be modified or a node/edge should be deleted
  */

 struct graph *new_graph()
 {

     // Make sure you get correct parameters...
     struct graph * n = malloc(sizeof(struct graph));
     if(n == NULL)
     {
         printf("malloc failed!\n");
         return NULL;
     }
     n -> vertex_count = 0;
     n -> edge_count = 0;
     n -> vertex_head = NULL;
     return n;

 }


```

```
struct vertex * _new_vertex(struct map *data)
{

    struct vertex * new_vertex = malloc(sizeof(struct vertex));
    if(new_vertex == NULL){
        printf("malloc failed at build new node\n");
        return NULL;
    }

    //do we need a void star cast??
    new_vertex -> connected_edges = NULL;
    new_vertex -> next_vertex = NULL;
    new_vertex -> data = data;

    return new_vertex;

}

struct edge * _new_edge(struct vertex *a, struct vertex *b, char *data)
{

    struct edge *current = malloc(sizeof(struct edge));

    if(current == NULL){
        printf("malloc failed at build new node\n");
        return NULL;
    }

    current -> from = a;
    current -> to = b;
    current -> next = 0;
    current -> data = data;

    return current;

}

void modify_graph(struct graph *g, struct map * a, char *w, struct map *b, int
d) {
    if(d == 0){
    /*checks to see if B has data in it, if it does:*/
        if(b){
        //add vertices into graph if they are not already in it
            if(!get_vertex(g,a)) { add_vertex(g,a); }
        if(!get_vertex(g,b)) { add_vertex(g,b); }

        //if the edge doens't exist then add the edge, otherwise modify it
            if(_find_edge(g,a,b) == 0){
                if(w == 0) { w = ""; }
                _add_edge(g,a,b,w);
            }
```

```
            else {
                    if(w == 0) { w = ""; }
                    _modify_edge(g,a,b,w);
                }
            }
        else {
                add_vertex(g,a);
            }
        }
        else{
            if(b){
            delete_edge(g,a,b);
            }
        else{
                delete_vertex(g,a);
            }
        }
    }

    void add_vertex(struct graph *g, struct map *data){

        if(g == 0){
            printf("graph not found! add_new_vertex()\n");
            return ;
        }

        if (data == 0){
            printf("vertex not found! add_new_vertex()\n");
            return ;
        }

        //create a new vertex with the data
        struct vertex *v = _new_vertex(data);

        ++(g -> vertex_count);

        //traverse list until end, then add new node to the end
        struct vertex *current = g -> vertex_head;

        //if we have no verices yet, then add the first vertex to the list
        if (g -> vertex_head == 0){
            g -> vertex_head = v;
            return;
        }

        //otherwise traverse until we can't and then add
        while(current -> next_vertex){
            current = current -> next_vertex;
        }

        current -> next_vertex = v;
        return ;
```

```
}

void delete_vertex(struct graph *g, struct map *data)
{

    if(g == 0){
        printf("graph not found: delete_vertex()!\n");
        return;
    }
    if(g -> vertex_count == 0){
        printf("no vertex to delete in: delete_vertex()\n");
        return;
    }

    //get the vertex to be deleted
    struct vertex *to_delete = get_vertex(g, data);
    struct vertex * prev; //used to fix vertices list in graphs later

    if(!to_delete){
        printf("vertex not found\n");
        return;
    }

    //traverse and try to find the node needed to delete
    struct vertex *current = g -> vertex_head;

    //accounts for if the deleted node is the head of the vertex
    if((g -> vertex_head) == to_delete){
        struct vertex *tmp = (g -> vertex_head) -> next_vertex;
        g -> vertex_head = tmp;
    }

    //otherwise traverse the list until we see the delete node as our next node
    else {
        while (current -> next_vertex){
            /*if we found the node that we are looking for, make current skip
its next node and point to the node after*/
            //how do we compare these two?
            if (current -> next_vertex == to_delete){
                struct vertex *tmp = current -> next_vertex;
                struct vertex *next_node = tmp -> next_vertex;
                current -> next_vertex = next_node;
                break;
            }

            current = current -> next_vertex;
        }
    }

    //go through the list and delete the nodes from each edge list
    //remove like how we did above where we just redirect the next pointer
    struct vertex *current2 = g -> vertex_head;
```

```
    while(current2){

        struct edge *current_edge = current2 -> connected_edges;
        if(current_edge != 0){
            if (current_edge -> to == to_delete){
                struct edge *tmp = current_edge -> next;
                current_edge = tmp;
            }
            else{
                while (current_edge -> next){
                    struct edge *tmp = current_edge -> next;
                    if(tmp -> from  == to_delete){
                        current_edge -> next = tmp -> next;
                        _free_edge(tmp);
                    }

                    current_edge = current_edge -> next;

                }
            }
        }


        current2 = current2 -> next_vertex;
    }

    _free_vertex(to_delete);
    --(g -> vertex_count);

    return;

}


/*
 * finds a vertex given the data and the graph teh vertex  should be in
 * returns null if no vertex found
 */

struct vertex * get_vertex(struct graph *g, struct map *data)
{

    if (g == 0){
        printf("graph not found. get_vertex() failed.");
    }

    if (data == 0){
        printf("data doesn't exist. get_vertex() failed.");

    }
```

```
    //traverse the graph's list of vertices until we find the node, return it
    struct vertex *current = g -> vertex_head;
    while(current){

        if (current -> data == data){
            return current;
        }

        current = current -> next_vertex;

    }

    return 0;

}

/*
 * modifies the vertex given the new data, deletes old node from graph g and
adds new one to g
 */
void modify_vertex(struct graph *g, struct map *old, struct map *new)
{

    if(g == 0){
        printf("graph not found. modify_vertex() failed.");
        return ;
    }

    if(old == 0){
        printf("no old data. modify_vertex() failed.");
        return ;
    }

    if (new == 0){
        printf("no new data. modify_vertex() failed.");
        return ;
    }

    delete_vertex(g, old);
    add_vertex(g, new);

    return;

}

/*
 * add edge function, creates a new edge between two edges, only does it one
way
 * it will make the edge from g to v
 *
```

```
 */
void _add_edge(struct graph *g, struct map *v, struct map *f, char *data)
{

    if (g == 0){
        printf("graph not found. failed at add_edge().");
        return;
    }
    if (v == 0 || f == 0){
        printf("map not found, invalid data. failed at add_edge().");
        return;

    }


    if(!get_vertex(g,v)) { add_vertex(g,v); }
    if(!get_vertex(g,f)) { add_vertex(g,f); }

    //this will find the edges in the graph and return them
    struct vertex *v_vertex = get_vertex(g, v);
    struct vertex *f_vertex = get_vertex(g, f);

    int i = _find_edge(g, v, f);
    if (i == 0){
        //go through each of the edge lists
        struct edge *new_edge_v = _new_edge(v_vertex, f_vertex, data);
        //all the edges for that vertex
        struct edge *v_edges = v_vertex -> connected_edges;

        if(v_vertex -> connected_edges == 0){
            v_vertex -> connected_edges  = new_edge_v;
            ++ (g -> edge_count);
            return;
        }

        while(v_edges -> next != NULL){
            v_edges = v_edges -> next;
        }

        v_edges -> next = new_edge_v;
        }

    else{
        printf("There is already an edge between the two vertices!\n");
    }

}

void add_wedge(struct graph *g, struct map *v, char *data, struct map *f) {
    _add_edge(g, v, f, data);
}
```

```
void add_edge(struct graph *g, struct map *v, struct map *f) {
    _add_edge(g, v, f, "");
}
/*
 * deletes an edge between the two given vertices in graph g
 */
void delete_edge(struct graph *g, struct map *v, struct map *f)
{

    if (g == 0){
        printf("Graph not found. delete_edge() failed.");
        return;
    }

    if(v ==  0 || f == 0){
        printf("Maps don't exist. delete_edge() failed.");
        return;
    }

    struct vertex *v_vertex = get_vertex(g, v);
    struct vertex *f_vertex = get_vertex(g, f);

    //if either of the edges are in the graph then fail
    if (v_vertex == 0 || f_vertex == 0){
        printf("vertex not found. failed at add_edge()");
        return;
    }

    struct edge *v_edges = v_vertex -> connected_edges;
    while (v_edges -> next != 0){
        struct edge *tmp = v_edges -> next;
        if (tmp -> to == f_vertex){
            v_edges -> next = tmp -> next;
            _free_edge(tmp);
            return;
        }
        v_edges = v_edges -> next;
    }

}

int _find_edge(struct graph *g, struct map *v, struct map *f)
{

    if (g == 0){
        printf("Graph not found. find_edge() failed.");
        return 0;
    }

    if(v ==  0 || f == 0){
        printf("Maps don't exist. find_edge() failed.");
```

```
            return 0;
        }

        struct vertex *v_vertex = get_vertex(g, v);
        struct vertex *f_vertex = get_vertex(g, f);

        //if either of the vertex are in the graph then fail
        if (v_vertex == 0 || f_vertex == 0){
            printf("vertex not found. failed at find_edge()\n");
            return 0;
        }

        struct edge *v_edges = v_vertex -> connected_edges;
        while(v_edges != 0){

            if (v_edges -> to == f_vertex){
                return 1;
            }

            v_edges = v_edges -> next;

        }

        return 0;

    }

    void _modify_edge(struct graph *g, struct map *v, struct map *f, char *data)
    {

        if (g == 0){
            printf("Graph not found. find_edge() failed.");
            return ;
        }

        if(v ==  0 || f == 0){
            printf("Maps don't exist. find_edge() failed.");
            return ;
        }

        struct vertex *v_vertex = get_vertex(g, v);
        struct vertex *f_vertex = get_vertex(g, f);

        //if either of the edges are in the graph then fail
        if (v_vertex == 0 || f_vertex == 0){
            printf("vertex not found. failed at add_edge()");
            return ;
        }

        struct edge *v_edges = v_vertex -> connected_edges;
        while(v_edges != 0){
```

```
        if (v_edges -> to == f_vertex){
            v_edges -> data = data;
            return;
        }

        v_edges = v_edges -> next;

    }

}

/*
 * This will create a new graph that has the nodes that are in both
 * g and h. The nodes are not connected.
 */
struct graph * intersection_graph(struct graph *g, struct graph *h)
{

    if (g == 0 || h == 0){
        printf("Graph doesn't exist. intersection_graphs() failed.");
        return 0;
    }

    struct graph *i = new_graph();

    // if either graphs are empty, return the new empty graph
    if (g -> vertex_count == 0 ||  h -> vertex_count == 0){
        return i;
    }

    struct vertex *current = g -> vertex_head;

    while (current){

        struct map *d = current -> data;

        if (get_vertex(h, d)){
            add_vertex(i, d);
        }

        current = current -> next_vertex;
    }

    return i;
}

/*
 * Union of two graphs, returns a new graph with all the nodes in both graphs,
 * unconnected.
 */
```

```
struct graph * union_graph(struct graph *g, struct graph *h)
{

    if (g == 0 || h == 0){
        printf("Graph doesn't exist. union_graphs() failed.");
        return 0;
    }

    struct graph *i = new_graph();

    //if both graphs are empty, return the new empty graph
    if (g -> vertex_count == 0 && h -> vertex_count == 0){
        return i;
    }

    struct vertex *current_g = g -> vertex_head;
    struct vertex *current_h = h -> vertex_head;

    while (current_g){
        add_vertex(i, current_g -> data);
        current_g = current_g -> next_vertex;
    }

    while (current_h){
        if (!get_vertex(i, current_h -> data)){
            add_vertex(i, current_h -> data);
        }
        current_h = current_h -> next_vertex;

    }

    return i;
}

/*
 * Adds the two given graphs together and returns resulting graph.
 */
struct graph * add (struct graph *g, struct graph *h){

    if (g == 0 || h == 0){
        printf("Graph doesn't exist. union_graphs() failed.");
        return 0;
    }

    struct graph *i = new_graph();

    //if both graphs are empty, return the new empty graph
    if (g -> vertex_count == 0 && h -> vertex_count == 0){
        return i;
    }

```

```
    struct vertex *current_g = g -> vertex_head;
    struct vertex *current_h = h -> vertex_head;

    while (current_g){
        add_vertex(i, current_g -> data);
        current_g = current_g -> next_vertex;
    }

     while (current_h){
        add_vertex(i, current_h -> data);
        current_h = current_h -> next_vertex;
    }

    return i;

}


/*
 * Given a graph and a node, return all the edges given that node.
 */
struct list *_get_edges(struct graph *g, struct map *data){

    if (g == 0){
        printf("Graph doesn't exist. union_graphs() failed.");
        return 0;
    }

    if (data == 0){
        printf("Data doesn't exist. get_edges() failed.");
    }

    struct vertex *current_vertex = get_vertex(g, data);

    struct list *edges_queue = make_list();
    struct edge *current_edge = current_vertex -> connected_edges;
    while(current_edge){
        add_tail(edges_queue, current_edge -> data);
        current_edge = current_edge -> next;
    }
    return edges_queue;
}

/*
 * Given a graph and a node, return the list of vertices that the vertex given
connected to.
 */
struct list *get_edge_neighbors(struct graph *g, struct map *data){

    if (g == 0){
        printf("Graph doesn't exist. union_graphs() failed.");
```

```
        return 0;
    }
    if (data == 0){
        printf("Data doesn't exist. get_edges() failed.");
    }

    struct vertex *current_vertex = get_vertex(g, data);

    struct list *edges_queue = make_list();
    struct edge *current_edge = current_vertex -> connected_edges;
    while(current_edge){
        add_tail(edges_queue, current_edge->to->data);
        current_edge = current_edge -> next;
    }
    return edges_queue;

}

struct list *get_all_vertices(struct graph *g){

    struct list *all_vertices = make_list();

    struct vertex *v = g -> vertex_head;

    while(v != 0){
        add_tail(all_vertices, v->data);
        v = v -> next_vertex;
    }

    return all_vertices;

}

/*
 * prints out all the edges and nodes of a graph
 */
void printg(struct graph *g){


    struct vertex *v = g -> vertex_head;

    while (v){
        struct map *tmp = v->data;

        printf("%s", "vertex data:\n");
        print_vertex(tmp);


        _print_edge(v -> connected_edges);

```

```
            v = v -> next_vertex;

            printf("\n");

        }

    }

    void print_vertex(struct map *m){

        struct map_node *current = m->node_head;
        while (current != NULL) {

            printf("\"%s\" : \"%s\"", current->key, current->value);
            if (current->next != NULL)
                printf(" , ");
            else
                printf("\n");
            current = current->next;
        }

    }

    void _print_edge(struct edge *e){

        if (e){
            while(e != 0){
                printf("%s: %s\n", "Edge data", e->data);
                printf("Connected to: ");
                print_vertex(e->to->data);

                e = e -> next;

            }
        }
    }


    /*
     * Graph clean up functions.
     */

    void _clean_graph(struct graph *G){
        if(G == NULL)
        {
            printf("Are you seriously trying to free a null graph?\n");
            return;
        }
        _free_all_vertex(G);
        free(G);

```

```
}

void _free_vertex(struct vertex *v){
    if(v){
        v -> next_vertex = 0;
        _free_edge(v -> connected_edges);
        free_map(v -> data);
        free(v);
    }

}


void _free_edge(struct edge* e){

    if(e){

        e -> to = 0;
        e -> from = 0;
        free(e);

    }
}

void _free_all_vertex(struct graph *g)
{

    struct vertex * current = g -> vertex_head;
    while(current)
    {
        struct vertex *tmp = current;
        _free_adjacency_row(current);
        current = current -> next_vertex;
        _free_vertex(tmp);
    }

}


void _free_adjacency_row(struct vertex *V)
{

    struct edge * current = V -> connected_edges;
    while(current)
    {
        struct edge *tmp = current;
        current = current -> next;
        _free_edge(tmp);

    }
}
```

```
 
 
/*
 * LIST METHODS
 */

/*
 * Initializes an empty list.
 */
struct list * make_list() {

    struct list *l;
    l = malloc(sizeof(struct list));
    if (l == NULL)
        return NULL;

    l->size = 0;
        l->head = 0;
    return l;
}

/*
 * Returns the size of a list.
 */
int size(struct list *l) {

    return l->size;
}

/*
 * Returns the element at index i of the list.
 */
void * list_get(struct list *l, int i) {

    if (l->head == NULL || i >= l->size || i < 0)
        return NULL;

    struct list_node *current = l->head;
    int j = 0;
    while (j != i) {
        current = current->next;
                ++j;
    }

    return current->data;
}

/*
 * Sets the element at index i to data.
 * Will return 1 if successful (valid i)
 * and 0 otherwise.
```

```
 */
int set(struct list *l, int i, void *data) {

    if (l->head == NULL || i >= l->size || i < 0)
        return 0;

    struct list_node *current = l->head;
    int j = 0;
    while (j != i) {
        current = current->next;
        ++j;
    }

    current->data = data;
    return 1;
}

/*
 * Adds to the front of the list.
 * Returns a 1 if successful and 0 otherwise.
 */
int add_head(struct list *l, void *data) {

    struct list_node *node = (struct list_node *)malloc(sizeof(struct
list_node));
    if (node == NULL)
        return 0;

    node->data = data;
    node->next = l->head;
    l->head = node;
    ++l->size;
    return 1;
}

/*
 * Adds to the end of the list.
 * Returns a 1 if successful and 0 otherwise.
 */
int add_tail(struct list *l, void *data) {

    struct list_node *node = (struct list_node *)malloc(sizeof(struct
list_node));
    if (node == NULL) {
        return 0;
    }
    node->data = data;
    node->next = NULL;

    /* if the list is empty, this node is the head */
  if (l->head == NULL) {
      ++l->size;
```

```
            l->head = node;
            return 1;
    }
      struct list_node *current = l->head;
      while (current->next != NULL) {
          current = current->next;
      }

      /* current is now the last node in the list */
      current->next = node;
      ++l->size;
      return 1;
  }

  /*
   * Returns data from the head of the list and removes it.
   */
  void * remove_head(struct list *l) {

      if (l->head == NULL)
          return NULL;

      struct list_node *oldHead = l->head;
      l->head = oldHead->next;
      void *data = oldHead->data;
      free(oldHead);
      l->size -= 1;
      return data;
  }

  /*
   * Returns data from the tail of a list and removes it.
   */
  void * remove_tail(struct list *l) {

      if (l->head == NULL)
          return NULL;

      if (l->head->next == NULL) {
          return remove_head(l);
      }

      struct list_node *slow = l->head;
      struct list_node *fast = l->head->next;

      while (fast->next != NULL) {
          slow = fast;
          fast = fast->next;
      }

      /* slow is now the second to last node in the list */
      void *data = fast->data;
```

```
    slow->next = NULL;
    l->size -= 1;
    free(fast);
    return data;
}

/*
 * Prints out list of elements in a list.
 * Used for testing.
 */
void printl(struct list *l) {

    printf("[");
    struct list_node *current = l->head;
    while (current != NULL)
        {
            if(current -> next == NULL)
            {
                printf("%d", *(int *) current -> data);
            }
            else
            {
                printf("%d,", *(int *)current->data);
            }
            current = current->next;
    }

    printf("]\n");
}

/*
 * Build a new list that is concatenating two lists.
 */
struct list * concat(struct list * a, struct list * b) {
    struct list *  new_list = make_list();
    struct list_node * current = a -> head;
    while (current) {
        add_tail(new_list, current -> data);
        current = current -> next;
    }
    current = b -> head;
    while(current) {
        add_tail(new_list, current -> data);
        current = current -> next;
    }
    return new_list;

}

/* For use of primitive int casted to void * for generic linked list*/
int add_head_int (struct list * l, int data)
{
    int * d = malloc(sizeof(int));
```

```
    *d = data;
    return add_head(l, d);
}

int add_tail_int (struct list * l, int data)
{
    int * d = malloc(sizeof(int));
    *d = data;
    return add_tail(l, d);
}

int list_get_int(struct list * l, int index)
{
    void * answer = list_get(l, index);
    return *(int *) answer;
}

int list_set_int(struct list * l, int index, int E)
{
    int answer = list_get_int(l, index);
    int * d = malloc(sizeof(int));
    *d = E;
    set(l, index, (void *) d);
    return answer;
}

int remove_head_int(struct list * l)
{
    void * answer = remove_head(l);
    return *(int *) answer;
}

int remove_tail_int(struct list * l)
{
    void * answer = remove_tail(l);
    return *(int *) answer;
}

/* For use of linked list using doubles*/
int add_head_dec (struct list * l, double data)
{
    double * d = malloc(sizeof(double));
    *d = data;
    return add_head(l, d);
}

int add_tail_dec (struct list * l, double data)
{
    double * d = malloc(sizeof(double));
    *d = data;
    return add_tail(l, d);
}
```

```
double list_get_dec(struct list * l, int index)
{
    void * answer = list_get(l, index);
    return *(double *) answer;
}

double list_set_dec(struct list * l, int index, double E)
{
    double answer = list_get_dec(l, index);
    double * d = malloc(sizeof(double));
    *d = E;
    set(l, index, (void *) d);
    return answer;
}

double remove_head_dec(struct list * l)
{
    void * answer = remove_head(l);
    return *(double *) answer;
}

double remove_tail_dec(struct list * l)
{
    void * answer = remove_tail(l);
    return *(double *) answer;
}

/* For use of linked list using strings*/
int add_head_str (struct list * l, char * data)
{
    /*char * d = malloc(strlen(data));
    d = data;*/
    return add_head(l, (void *) data);
}

int add_tail_str (struct list * l, char * data)
{
    /*char * d = malloc(strlen(data));
    d = data;*/
    return add_tail(l, (void *) data);
}

char * list_get_str(struct list * l, int index)
{
    void * answer = list_get(l, index);
    return (char *) answer;
}

char * list_set_str(struct list * l, int index, char * E)
{
    char * answer = list_get_str(l, index);
```

```
    /*char * d = malloc(strlen(E));
    d = E;*/
    set(l, index, (void *) E);
    return answer;
}

char * remove_head_str(struct list * l)
{
    void * answer = remove_head(l);
    return (char *) answer;
}

char * remove_tail_str(struct list * l)
{
    void * answer = remove_tail(l);
    return (char *) answer;
}

/* For use of linked list using maps*/
int add_head_map (struct list * l, struct map * data)
{
    return add_head(l, (void *) data);
}

int add_tail_map (struct list * l, struct map * data)
{
    return add_tail(l, (void *) data);
}

struct map * list_get_map(struct list * l, int index)
{
    void * answer = list_get(l, index);
    return (struct map *) answer;
}

struct map *list_set_map(struct list * l, int index, struct map * E)
{
    struct map * answer = list_get_map(l, index);
    set(l, index, (void *) &E);
    return answer;
}

struct map * remove_head_map(struct list * l)
{
    void * answer = remove_head(l);
    return (struct map *) answer;
}

struct map *remove_tail_map(struct list * l)
{
    void * answer = remove_tail(l);
    return (struct map *) answer;
```

```
}

/* Misc functions: strops, random numbers, atoi, etc.*/
int * random_int(int minimum_number, int max_number)
{
    int * answer = malloc(sizeof(int));
    *answer = rand() % (max_number + 1 - minimum_number) + minimum_number;
    return answer;
}

char * myItoa(int num)
{
    char * result = malloc(sizeof(char) * 50);
    sprintf(result, "%d", num);
    return result;
}

int * myAtoi(char * str)
{
    int * result = malloc(sizeof(int));
    int res = 0;  // Initialize result
    int sign = 1;  // Initialize sign as positive
    int i = 0;  // Initialize index of first digit

    // If number is negative, then update sign
    if (str[0] == '-')
    {
        sign = -1;
        i++;  // Also update index of first digit
    }

    // Iterate through all digits and update the result
    for (; str[i] != '\0'; ++i)
    {
        if(str[i] == '0' || str[i] == '1' || str[i] == '2' || str[i] == '3'
                || str[i] == '4' || str[i] == '5' || str[i] == '6' ||
                str[i] == '7' || str[i] == '8' || str[i] == '9')
        {
            res = res*10 + str[i] - '0';
        }
        else
        {
            result = NULL;
        }
    }
    // Return result with sign
    *result = sign*res;
    return result;
}

char * concat_string(char * a, char * b)
{
    char * c = malloc(strlen(a) + strlen(b) + 1);
```

```
    sprintf(c, "%s%s", a, b);
    c[strlen(a) + strlen(b)] = '\0';
    return c;
}

int length(char * s)
{
    return strlen(s);
}

int str_comp(char * a, char * b)
{
    int ans = strcmp(a, b);
    //fprintf(stderr, "Didn't seg yet: %d\n", ans);
    return (ans == 0);
}

// Ocaml views chars as ints. Ocaml should have a convert back! C.decode()?
// See Ocaml Char module!
char * get_char(char * s, int idx)
{
    int size = strlen(s);
    if(idx < 0 || idx > size)
    {
        return NULL;
    }
    else
    {
    char * i = malloc(sizeof(char) * 2);
    i[0] = s[idx];
    i[1] = '\0';
        return i;
    }
}
```