GRAILanguage

graph rendering articulate innovation

COMS-4115
Programming Languages and Translators
Project Final Report

Aashima Arora(aa3917)
Jiaxin Su(js4722)
Riva Tropp(rtt2114)
Rose Sloan (rns2144)

1 Introduction

GRAIL (Graph Rendering Articulate Innovation Language) is a language focuses on allowing users to build and manipulate graph while using relatively simple syntax. The most notable feature of this language is that it implements complete type inference for all expressions and functions, allowing users to simply create graphs with custom fields contained in their nodes and edges.

Graphs can be used to model a number of mathematical and real world problems, including social network graphs, transportation networks, utility graphs, document link graphs, packet flow, neural networks, dependency modeling, and much more. However, in most existing languages, graph construction, particularly for graphs that require a significant amount of data stored on nodes or edges, is quite difficult or at the very least syntactically complex. We seek to reduce this complexity through the use of type inference and simple record structures.

2 Language Tutorial

2.1 Using GRAIL

2.1.1 Hello World

In the grail folder,type *make* and then run make-ext.sh. Make creates the **grail.native** file that can accept a .gl file as input and generate the llvm output. The linking with the external display is done in the file make-ext.sh. To run your .gl file, run it as

```
./make_ext.sh hello_world.gl
```

A GRAIL Hello World example :

```

main(){
  print("Hello, World!");
  return 0;
}
```

Listing 1: GRAIL Hello World

Output: Hello, World!

The above code illustrates that:

- main() is a required function.
- print() is an built-in function that can be used to display a string.

2.2 Data Manipulation

2.2.1 Primitive Types

Due to the elegant type inference, the primitives can be declared as illustrated:

```
main() {
    var_str = "String"
    var_bool = true;
    var_int = 1;
```

```
    print(var_str);
    printbool(var_bool);
    printint(var_int);
    return 0;
}
```

Listing 2: GRAIL Primitives

Output:

String1
1

2.2.2 Derived Types

1. **Lists:** Lists can be declared in the following manner using an array like notation.

```
main()
{
  a = [1, 2, 3, 4, 5];
  printint(a[4]);
  return 0;
}
```

Listing 3: GRAIL Primitives

Output:

5

2. **Records:** Records form the nodes in the graph. They can have various attributes.

```
main()
{
  myrec = {a: "yeah", b: 2};
  print(myrec.a);
  printint(myrec.b);
  return 0;
}
/* init & access in record that has more than one types */
```

Listing 4: GRAIL Records(Nodes in Graph)

Output: yeah2

3. **Edges:** Edges form the connection between two records in the graph. The edges can either be directed, or undirected. It is enforced that only two nodes that have the same record structure can be connected to form an edge. The edge also has an attached record, which may be of a different structure than its nodes.

```
main() {
  x = {a: 1, b: 2};
  y = {a: 3, b: 4};
  e = x->y with {weight: 1};
  z = e.from;
  printint(z.a);
  return 0;
}
```

Listing 5: GRAIL Edges(Edges in Graph)

Output: 1

4. **Graphs:** Graphs are a collction of nodes and edges.

```
main() {
  a = {key: 1, cap: 10};
  b = {key: 2, cap: 10};
  c = {key: 3, cap: 15};
  d = {key: 4, cap: 20};
  x = a->a with {weight: 10};
  y = b->b with {weight: 20};
  graph = (a, b, x, y, c->d) with {weight: 2};
  return 0;
}
```

Listing 6: GRAIL Graph

2.2.3 Control Flow

These are the various control flow statements built into GRAIL.

1. **If Statements**

```
main() {
  a = 5;
  if (a < 3) {
    print("Bigger");
  }
  else if (a == 5) {
    print("Equal");
  }
  return 0;
}
```

Listing 7: GRAIL If Statements

Output:

Equal

2. **For Loops**

```
main()
{
  for ( a = 5; a >= 0; a = a - 1;) {
    printbool(true);
  }
  print("Complete");
  return 0;
}
```

Listing 8: GRAIL For Loops

Output:

1
1
1
1
1
1
Complete

3. **For In Loops**

```
main()
{
  a = ["a", "b", "c"];
  for ( x in a ) {
    print(x);
  }
  return 0;
}
```

Listing 9: GRAIL For In Loops

Output:

abc

4. **While Loops**

```
main()
{
  i = 5;
  while (i > 0) {
    printint(i);
    i = i - 1;
  }
  print("42");
  return 0;
}
```

Listing 10: GRAIL While Loops

Output:

5
4
3
2
1
42

5. **Function Calls**

```
f(a) {
  a = a + 1;
  return a;
}

main() {
  x = f(3);
  printint(x);
  return 0;
}
```

Listing 11: GRAIL Functions

Output:

4

2.3 Example: Petersen Graph in GRAIL

The following example code constructs and displays the Petersen graph in GRAIL. As we can see, this can be done in under 50 lines of code, all of which are simple and readable.

```
main()
{
  //construct the Petersen graph

  petenodes = [{key: 1}, {key: 2}, {key: 3}, {key: 4}, {key: 5},
          {key: 6}, {key: 7}, {key: 8}, {key: 9}, {key: 10}];
  pete = ({key: 0}) with {weight:1};

  for(n in petenodes){
    pete &= n;
  }

  for(i = 0; i < 5; i += 1;){
    pi = petenodes[i];
    po = petenodes[i+5];
    pete .&= pi--po;
    if(i == 0){
      p2 = petenodes[2];
      p3 = petenodes[3];
      pete .&= pi -- p2;
      pete .&= pi -- p3;
    }

    if(i == 1){
      p3 = petenodes[3];
      p4 = petenodes[4];
      pete .&= pi -- p3;
      pete .&= pi -- p4;
    }

    if(i == 2){
      p4 = petenodes[4];
      pete .&= pi -- p4;
    }
  }

  for(i = 5; i < 9; i += 1;){
    pi = petenodes[i];
    pplus = petenodes[i+1];
    pete .&= pi--pplus;
    if(i == 5){
         p9 = petenodes[9];
         pete .&= pi--p9;
    }
  }

    display(pete);
}
```

Listing 12: GRAIL Petersen Graph

Output:

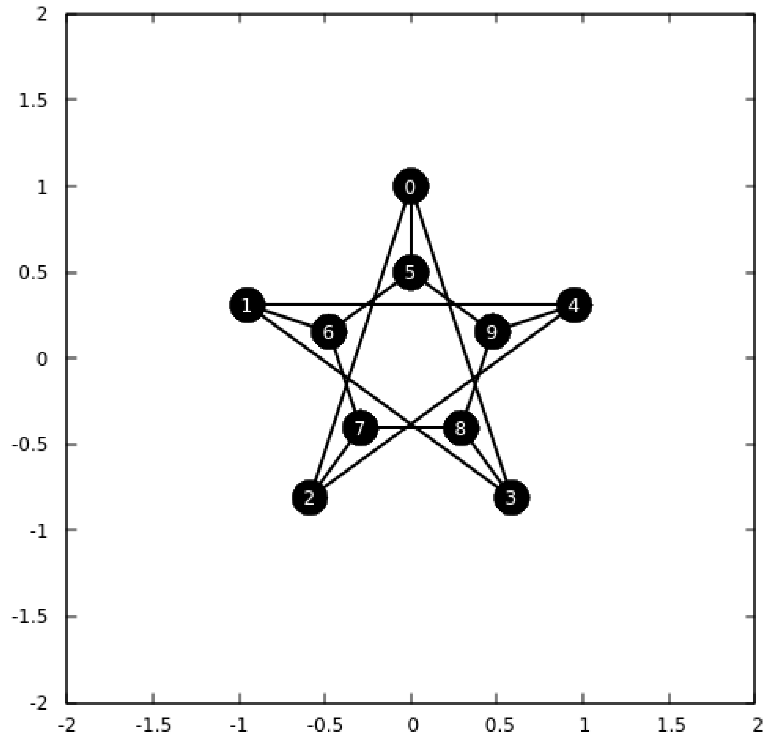


Figure 1: Petersen Graph

3 Language Reference Manual

3.1 Tokens

GRAIL tokens are separated by one or more whitespace characters. Comments delimited by /* and */ or single-line comments beginning with // are also ignored. Comments may not be nested.

3.1.1 Identifiers

An identifier is a sequence of characters, all of which must be either alphanumeric or the underscore (_) character. The first character must be a letter. Uppercase and lowercase letters are considered distinct but the choice of case in identifiers holds no significance to the compiler. Identifiers are used for variables, features of records, and function names.

3.1.2 Reserved Words

The following identifiers are reserved and may not be used elsewhere:

```
dir, edges, else, false, for, free, from,
if, in, nodes, rel, return, to, true, while, with
```

3.1.3 Constants

Integers are a 32-bit sequence of digits with no floating point:

```
var = 3;
```

Floats are 32-bit floating point numbers:

```
var = 3.0;
```

Characters are single characters enclosed by single quotation marks.

```
var = 'a';
```

Boolean constants are represented by the keywords true and false. Booleans may take on only these two values.

```
var = false;
```

String literals are a series of characters delimited by double quotation marks. Strings cannot be nested, though a double quotation mark can appear inside a string by using the escape sequence. Backslashes must similarly be escaped with another backslash.

3.2 Types

3.2.1 Primitive Types

GRAIL has 5 primitive types: *boolean, character, integer, float, and void.* A boolean is a true/false Boolean value. A char is a single member of the ASCII character set. An integer is any mathematical integer. A float is a rational floating point number. The void type is a null type, used in functions that return no variables.

3.2.2 Derived types

1. **Record:**

   Records are user-definable data-structure consisting of comma-separated pairs of keys (which must be unique within the record) and data. The data may be any primitive or derived type . Records inside a graph are called nodes, and all nodes in a graph must contain the same record type.

2. **Edges:** An edge connects two nodes and can be directed or undirected. It consists of two parts, a descriptor, which describes the connection between two nodes (directed or undirected, and in which direction). It also contains a record containing information about the edge (such as, for example, a weight). If the edge is declared in a graph constructor or as part of a statement in which it is added to a graph, it need not be declared with an attached record, as there is a default record for all edges in the graph.

   ```
   e = u->v with {weight: 4}; //where u and v are nodes
   ```

   An edge's structure (including types of fields) may not be altered. An edge always has to, from, dir, and rel fields that yield the node pointed to by the edge, the node that extends from the edge, a boolean set to true if the edge is directed, and the edge's attached record.

3. **Lists:** Lists are arrays of primitives or objects of the same type. The type of a list is the type of the first element inserted into a list (which must be the same type as the other elements in the list).
4. **Graphs:** Graphs are collections of nodes and the edges connecting them with a default edge record defined.

3.3 Expressions

Expressions, consisting of type-compatible operators or groups of operators separated by operands, are outlined below.

3.3.1 Primitive Literals and Identifiers

Literals of primitive types and identifiers referencing previously defined variables can be expressed in the format shown in the tokens section. Identifiers must be assigned (using an assignment statement) before they can be accessed.

3.3.2 Lists

Lists can be declared in the format shown below.

```
[item1, item2, item3];
```

We can access or update the nth item of a list using the syntax:

```
lst[n]
```

In this case, lst must be an identifier.

3.3.3 Function Calls

Syntax for function calls is as follows:

```
functionname(parameter1, parameter2);
```

Parameters can be of any expression format, as long as they are of a type that can be inputted into the given function. (Because of the type inference features, some functions may take variables of multiple types. The types of these function arguments are resolved at compile time.)

3.3.4 Records

Records can be initialized as follows.

```
{field1: value1, field2: value2, field3: value}
```

We can access or update the values of fields using the dot operator shown below:

```
recordname.fieldname
```

3.3.5 Edges

We can declare edge literals using one of the following three formats.

```
node1 -> node2 with rec
node1 <- node2 with rec
node1 -- node2 with rec
```

The first two constructors produce directed edges (the first going from node1 to node2, the second going from node2 to node 1). The third produces an undirected edge. The expressions node1 and node2 must be previously initialized variables of the same record type. If (and only if) the edge is declared inside a graph constructor or a graph addition operation, the "with rec" syntax may be omitted. If it is, the edge's record will be initialized as the default record of the graph.

We can use the keywords from, to, dir, and rel to access the edge's origin endpoint, the edge's destination endpoint, a boolean equal to true if the edge is directed, and attached record. We do this using dot operator syntax as follows:

```
e = node1 -> node2 with rec;
n = e.from;
```

3.3.6 Graphs

Graphs are initialized using a list of expressions and a default edge record as follows:

```
(x, y, z, x->y) with {field1:value1,field2:value}
```

The expressions provided may be edges, nodes, or both. All nodes must be of the same type, and all edges must have the same node type as the nodes initialized in the graph and the same record type as the default record of the graph. If edges are declared in the graph constructor as edge literals, their endpoints will be added to the graph. Otherwise, nodes must be declared separately (or explicitly added to the graph using a graph addition operation) to be included.

We can use the keywords nodes and edges to get lists objects containing each of the nodes or each of the edges in the graph. Again, we do this using dot operator syntax.

3.3.7 Unary Operations

!expr is logical not and may be applied to expressions of the boolean type. -expr is numeric negation and returns the value of the expr multiplied by negative one. It may be applied to expressions of type int and float.

3.3.8 Equality and Comparison

All equality and comparison operations can be invoked using the syntax "expr1 operator expr2".

The `==` operator may be used to compare any two objects of the same type and returns whether they are structurally equal. Similarly, `!=` can compare any two expressions of the same type and returns true when they are not structurally equal.

The `<`, `>`, `<=`, and `>=` operators correspond to less than, greater than, less than or equal to, and greater than or equal to respectively. These operations may be applied only to ints and floats.

3.3.9 Arithmetic Operations

All binary mathematical operations can be invoked using the syntax `expr1 operator expr2`.

We use the operators `+, -, *`, and `/` to perform addition, subtraction, multiplication, and division respectively on integers. We use the operators `.+, .-, .*`, and `./` to perform addition, subtraction, multiplication, and division respectively on floats.

Additionally, we can use the syntax

```
x += i;
y .+= f;
```

as shorthand for

```
x = x + i;
y = y .+ f;
```

3.3.10 Logical operation

We can use the syntax

```
expr1 && expr2
expr1 || expr2
```

to return, respectively, the logical and and logical or of the two expressions. Both expressions must be of the boolean type.

3.3.11 List Addition

The expression `l ^ i` returns a list containing the elements of l with an additional element i as the last element, as long as i is of the same type as the items of l. We can use the shorthand `l ^= i;` to represent `g = g ^ i;`.

3.3.12 Graph Addition

The expression `g & n` returns a graph with the same nodes and edges as g, as well as a new node n. Similarly, the expression `e & n` returns a graph with the same nodes and edges as g, as well as a new node e.

As with list addition, we can use the syntax `g &= n;` and `g .&= e;` as shorthand for `g = g & n;` and `g = g .& e;`.

3.4 Statements

Statements executes in sequence. They do not have values and are executed for their effects. The statements in our language are classified in the following groups:

- Expression statement
- Assignment statement
- Conditional Statement
- Loop Statement

3.4.1 Expression Statement

In certain cases, we may want to evaluate an expression purely for its side effects. (For example, we may wish to call a print function.) The syntax to do so is as follows:

```
expr;
```

3.4.2 Assignment Statement

Assignment statements are used to assign an identifier to the value of the expression. We have two types of assignment statements, using following formats:

```
lvalue = expr;
lvalue .= expr;
```

The first statement simply stores the given expression in the location indicated by the lvalue. If the expression is a derived type, it will serve as a pointer to the given expression, so if the expression is updated, the value stored in the lvalue will change. (For example, if `y` is a list and we perform `x = y;`, when we change the items of y, it will change the items in x.) In contrast, the second statement returns a deep copy of expr and stores it in the provided lvalue.

Acceptable lvalues are identifiers, list items (e.g. listvariable[5]), and fields of records (e.g. recordvariable.fieldname).

3.4.3 Conditional Statement

Conditional statements use the expression (which must be of a boolean type) as conditional test to decide which block of statements will get executed. They have the following formats:

```
if (expression) { statement(s) }
if (expression) { statement(s) } else { statement(s) }
if (expression) { statement(s) } else if (expression) { statement(s) } else { statement(s) }
```

3.4.4 Loop Statement

We support while, for, and for-in loops

```
while (expression) { statement(s) }
for (init expression; cond expression; execution expression; ) { statement(s) }
for (variable in listname){ statement(s)}
```

The while loop takes one expression as the conditional expression to check if the available variables or expressions qualify, which determine if the body statement(s) will be executed or not. The standard for loop takes three expressions : initialization expression, conditional expression, and execution expression. The initialization expression will be executed when the for loop is initiated. The conditional expression is the test expression to check if the condition(s) is satisfied, which corresponds to if the body statement(s) will be executed. The execution expression will be executed after every time the body statement(s) is executed. The for-in loop iterates over a list, assigning the variable to each member of the list in order and performing the provided statements.

3.5 Scope

GRAIL is a statically scoped language. The scope of a formal parameter of a function is the entire body of the function, and local variables remain in scope only within the function in which they are initialized.

Function names are visible in the bodies of functions defined later in the document. Every program must contain a main function (defined function main()).

3.6 Built-In Functions

3.6.1 Print

Prints to standard output.

```
print (string);
printint(int);
printbool(bool);
```

3.6.2 Display

Displays a graph using the gnuplot external library.

```
display(graph);
```

3.6.3 Size

Returns the size of a list.

```
size(list)
```

4 Project Plan

4.1 Process

As a group, we met at least one a week (and frequently more often) to discuss our progress, merge code, address any language design concerns, and delegate tasks for the next week. Changes to our language design or the structure of the code were discussed as a group, either at these meetings or over text or email for minor changes. Additionally, many weeks, we met with our TA Danny during his office hours to discuss the development of our language and ask for advice on implementing various features.

Group members tested their own features as they implemented them before pushing to master, and we maintained a regression test suite, adding tests as we implemented more features, throughout the project (often adding test cases that group members had written while testing individual features). In addition to testing our code by compiling to LLVM and running the LLVM interpreter, we also created a script to compile our code to a "typer" mode, which took in a GRAIL program and returned the same code but with the types of each expression printed. This typer mode was valuable both for testing typer and for determining whether certain bugs were arising in the typer or in the codegen.

4.2 Programming Style Guide

The following style guidelines were used by the group:

- Give variables sensible names
- Comment any code where it is non-obvious what the code is doing
- Indent using spaces, not tabs
- Break overly long lines of code into multiple lines

4.3 Project Timeline

Date	Milestone
February 1	Pick language concept, begin design
February 8	Proposal complete
February 20	LRM first draft complete
March 1	Edited LRM complete, git repository created
March 20	Scanner and parser complete
March 27	Hello world finished
April 20	Control flow and primitive types working in codegen
April 28	Type inference complete for all types
May 10	Everything done

4.4 Roles of Team Members

The roles of each team member are outlined below. It is worth noting that roles shifted substantially after the "hello world" demonstration, as much of the coding work up to that point centered on implementing the front end and getting the basic type inference structure working. After that point, we shifted to working more heavily on codegen, which was very bare bones at that point.

Rose Sloan (manager)

- Scanner and parser
- Much of codegen (control flow, lists, graphs, binary operations, and deep copy)

Riva Tropp (language guru)

- Language design
- Type inference
- Static semantic checking

Aashima Arora (systems architect)

- Early type inference (pre-hello world)
- Portions of codegen (variable assignment, records, edges)
- External library linkage

Jiaxin Su (tester)

- Early codegen (pre-hello world)
- Create, update, and maintain the regression test suite

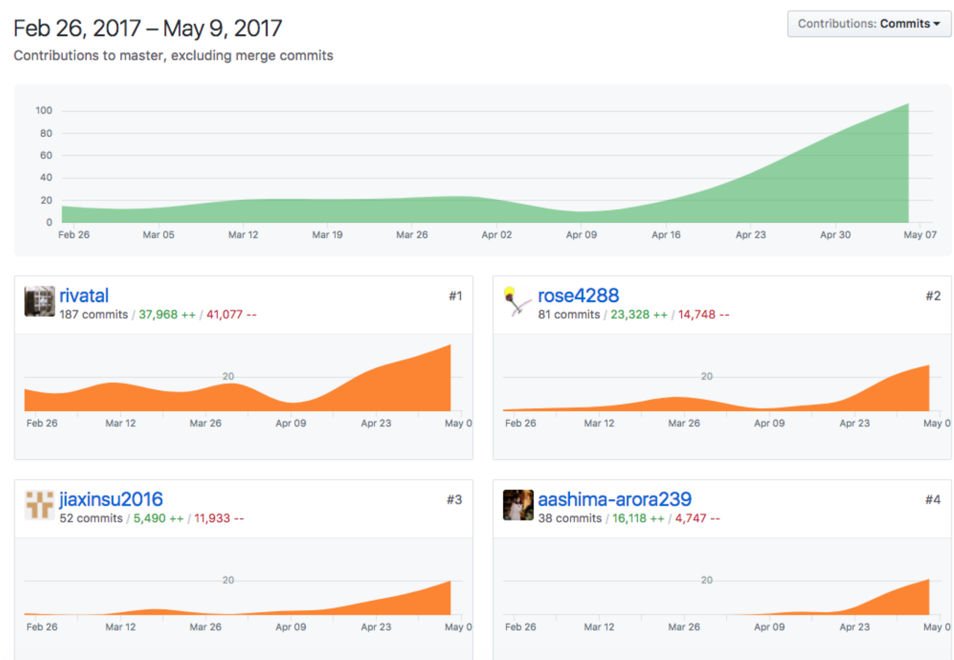
4.5 Software Used

The bulk of our code is written in OCaml and compiles to LLVM. We also have a C script to integrate the external library gnuplot, which is used by our display function. The regression test suite is run through a shell script that calls our code and the LLVM interpreter. (We also used the LLVM interpreter frequently when testing individual cases.) To compile to typer mode, we use the run.sh shell script, which in turn calls an awk script.

We combined and tracked our code using a github repository. Each team member pulled code to their own computer or virtual machine and used their own choice of programming environment.

4.6 Project Log

The github graphs, showing who made commits when are shown below. As a note, Aashima's commits were not tied to her github account until April that are not reflected on her graph. The full github log (not included here for brevity's sake) shows that she made 16 commits before then.



An overview of what happened when is as follows.

Date	Milestone
February 28	Git repository created
March 5	Scanner created
March 15	Parser created
March 19	Type inference started
March 21	Scanner and parser complete
March 22	Very basic type inference complete
March 25	Codegen created
March 25	Regression test suite created
March 27	Hello world finished
April 4	Type inference working for function calls
April 4	Control flow added to codegen
April 16	Binary operations on primitives in codegen
April 18	Variable assignment in codegen
April 21	Lists, records, and dot operations added to typer
April 23	Script to compile code to typer mode created
April 26	All types added to typer
May 4	List implementation working in codegen
May 6	For in, list addition, and records in codegen
May 7	Edges working in codegen
May 8	Graphs added to codegen, all supported features added
May 10	Everything debugged and prepared for submission

5 Architectural Design

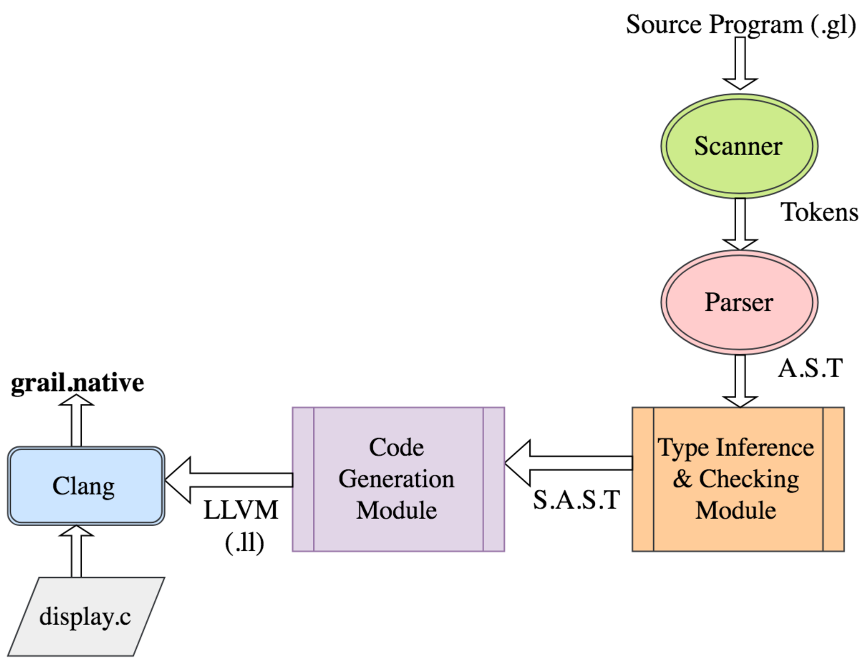


Figure 2: Architecture of GRAIL compiler

The architecture of the GRAIL compiler consists of the following major components: scanner, parser, type inference and type checker, and code generator, shown in Figure 1 above. The scanner, parser, type inference and type checker modules constitute the front end, and the code generation module, which generate the LLVM output, along with the display component form the back end of the GRAIL compiler. Except for the display module, all components have been implemented in OCaml.

The OCaml compiler emits **grail.native** as the final binary. The entry point of the compiler is *grail.ml*. A source program with a *.gl* extension is passed as the input to *scanner.ml* and *parser.ml*, which convert it into AST format.The AST is passed to *infer.ml* which is the module responsible for inferring the types of each function and expression. Along with type inference, this module is also responsible for all semantic checking. After type checking, the semantically-checked SAST is passed to codegen.ml. The output of codegen.ml is LLVM which is linked with a display.c file and compiled to binary code using clang compiler.

5.1 Components

1. **Scanner -** The scanner takes as input a grail(.gl) source program and generates tokens for identifiers, keywords, operators, and values, as specified by the lexical conventions. {Rose Sloan}
2. **Parser -** The parser takes in the generated tokens and generates an abstract syntax tree (AST) based on GRAIL Syntax. {Rose Sloan}
3. **Typer -** The typer is responsible for inferring the types in the AST as well as semantic checking. The type inference module uses the classic Hindley-Milner type system to infer function return types and

expression types, which may be either primitive or derived types. The typer checks that the AST is semantically correct and, if it is, returns the SAST annotating each of the above mentioned expressions with their types. {Riva Tropp, Aashima Arora}

4. **Code Generator -** The code generator traverses the SAST tree to generate code in post-order fashion. The code generator converts each expression into LLVM build commands. All items of primitive types are converted to LLVM primitive types, while the derived types are generated as various structures. The LLVM output can be piped to a *.ll* file, which links into clang for graph visualization. {Rose Sloan, Aashima Arora}

5. **Gnuplot Display : display.c -** The display function that is built-in into GRAIL uses the output of the LLVM to obtain the graph. The graph is read from the memory using structures compatible with the LLVM output. The subcomponents of the graph like nodes and edges are parsed from the graph structure and written into files in a format that can be understood by gnuplot. The files are then read by a gnuplot script which plots the graph. {Aashima Arora}

6 Test Plan

6.1 Testing Phases

6.1.1 Unit Testing

Our unit testing focuses on testing specific functionalities of the compiler. Our "tests/new-tests" folder contains the entire set of integration test suites,including pass-tests that should pass and fail-tests that should fail. These can be run in GRAIL folder by entering "make" and "./testall.sh" in the terminal.

We tested syntax and lexical conventions of our language respectively. The goal of our test suites is to check the GRAIL compiler work with our language end-to-end, from the scanner to the code generation phase. The generated codes are then compiled with lli and llvm-link, and run to verify the expected output with the actual output.

- Arithmetic Operators were tested by calculations and comparisons. GRAIL has binary operators, unary operators, value binding (= and .=), and equality (==). We tested them by declaring variables, assigning them, make them perform calculations or comparisons, and printing the result to the screen to ensure their correctness. Control flow structures might be used in these kinds of test cases.
- Control flow was unit-tested. Available control flow structures are if/else, if/else if, for, for in, while, and return. Should-fail cases were also tested, such as the while loop condition does not return a boolean value.
- GRAIL supports type inference. Due to this feature, then our language should follow the inference syntax convention. We also provide test cases for this.
- Our primitive types are int, float, boolean, char, and string. Testing was done on those types by assigning various types to the same variable to see if the compiler throws type dis-match error. Also, we test the declaration and usage of different types to verify the correctness. We use built-in print functions for to check if the expected output matches the actual printed output.
- The built-in functions were tested (print() for string, printbool() for boolean, printint() for int, display() for displaying graphs, size() for getting list length). We have designed specific test cases to verify if these functions work well in the context.

- We also tested if literals, like integer/double/boolean/characters/strings, can work with other basic features like function calls, function return, and various data structures. They were tested individually and verified by using if statements or printing them to the stdout.
- We have specific test cases for comments, which has format like ”//” and ”/* ... */”. The former is single line comment, while the later is multi-line comments. Moreover, we put comments randomly in the test suite codes to ensure the compiler scanned them successfully.

6.1.2 Other Testing Methods

For debugging purposes, we have extra testing methods that are for specific parts of the compiler, inclusing parser, typer and code generation. We tested these three components individually throughout the semester.

For parser, we have a test script file named ”parserize.ml” for obtaining output from the parser directly. The user can feed grail source codes into the ”parserize” executable from the stdin, which will be produced after entering ”make” in the parser folder. In this way, we will be able to see the code output from the parser.

For our typer, we created a type-checker to check types. From the stdin, the user can feed grail source codes into the grail executable, which will be produced after entering ”./run.sh” command in GRAIL folder. This program will spit out the corresponding output from the typer.

We also made heavy use of the LLVM interpreter to check if the obtained LLVM codes were correct for debugging purpose. The output of the GRAIL compiler can be piped to the LLVM interpreter (called using lli in the command line) to test LLVM outputs.

6.2 Automation

Our Shell script test file in GRAIL folder, takes in all the files in the ”$tests/new-tests$”folder and compiles all the files (must have extension ”gl”) in that directory to LLVM code that can be executed with LLVM interpreter. Furthermore, the llvm-link will compile the produced llvm codes, and create corresponding executables. The Shell script will run the executables and store the produced output in .out files or error messages in .err files. These .out and .err files will be stored in the ”$test-output$” folder in GRAIL folder. If the user enters ”make clean” command in GRAIL folder, the ”$test-output$” folder will be removed.

6.3 Test Suites

We tested the following features of our language:

- Primitive data assignment and operations Our test suites covered basic declaration and assignment of primitive types. We also covered cases while these types work with arithmetic or binary operations. These tests indirectly stress how GRAIL follows type inference syntax convention, and check if two sides of the assignment sign (”=”), the equality sign (”==”), more binary or unary operators have matched types.
- Control flow Our control flow structures are while(...) ..., for (...; ...; ...;), for(... in ...) ..., if ... else ... , and if ... else if .... For while, we checked if the condition expression return boolean value or not. For structures related to if loop, we tested if each block of the structure is accessible. For structures related to for loop, we tested if each of the three condition expressions work with the code block below. We also tried to iterate through loops and verified the number iterations of loops, trying to stress the language’s ability to handle nested blocks.

- List assignment and declaration

  Our test cases tried to declare and assign values to list data structures by specifying the elements in the list, and check for the deep copy feature, which allows items in the list or list to swap / manipulate their values.

- Record assignment and declaration

  Our test cases tried to declare and assign values to record data structures by specifying the elements in the record, and check for the deep copy feature, which allows items in the list or list to swap / manipulate their values. In the context of graph, records are treated as nodes of the graph.

- Edge assignment and declaration

  Our test cases tried to declare and assign values to Edge data structures by specifying the elements in the Edge, like the start node and the end node. our test cases also checked for the deep copy feature, which allows items in the edge or edge to swap / manipulate their values.

- Graph assignment and declaration

  Our test cases tried to declare and assign values to graph data structures by specifying the elements in the graph, and check for the deep copy feature.

- Functions for data structures For list, we checked if we have access each of the elements in the list by iterating through the list, and printing corresponding values. We also checked if the elements in the list share the same type, and prepared negative cases for type mismatch cases. In addition, we checked if the size() function works on specific list by printing out the expected list length.

  For record, we checked if we could access specific fields in the data structure by using the dot function and printing out its values. For edges and graphs, we checked if we could access the nodes in corresponding data structures by .nodes(), .from, and .to functions. We also checked if we could display graphs by display().

- Function calls We prepared test cases like hello-world, gcd, and more to test if the function call work in various function context.

6.4 Grail to LLVM

Below are our sample codes.

Sample 1 in Grail:

```

main(){
  c = {station: "49th St Station", line: "1", lat:39.9436, lon:75.2167, capacity:1500,
    service: [0,1,1,1,1,1,1]};
  d = {station: "116th St Station", line: "1", lat:39.56, lon:75.456, capacity:750, service:
     [0,1,1,1,1,1,0]};
    g = (c -- d) with {distance: 1};
  c.station = "168th";
  size(c.service);
}
```

Sample 1 in LLVM:

```
; ModuleID = 'Grail'

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt1 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@fmt2 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
```

```
@fmt3 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@fmt4 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt5 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@str = private unnamed_addr constant [2 x i8] c"1\00"
@str6 = private unnamed_addr constant [16 x i8] c"49th St Station\00"
@str7 = private unnamed_addr constant [2 x i8] c"1\00"
@str8 = private unnamed_addr constant [17 x i8] c"116th St Station\00"
@str9 = private unnamed_addr constant [6 x i8] c"168th\00"

declare i32 @printf(i8*, ...)

declare i32 @sample_display(i32)

define i32 @"size!1"({ i32*, i32 } %x) {
entry:
  %x1 = alloca { i32*, i32 }
  store { i32*, i32 } %x, { i32*, i32 }* %x1
  ret i32 1
}

define i32 @"size!2"({ i32*, i32 } %x) {
entry:
  %x1 = alloca { i32*, i32 }
  store { i32*, i32 } %x, { i32*, i32 }* %x1
  ret i32 1
}

define void @main() {
entry:
  %strct = alloca { i32*, i32 }
  %lst = alloca i32, i32 7
  %ptr = getelementptr inbounds i32* %lst, i32 0
  store i32 0, i32* %ptr
  %ptr1 = getelementptr inbounds i32* %lst, i32 1
  store i32 1, i32* %ptr1
  %ptr2 = getelementptr inbounds i32* %lst, i32 2
  store i32 1, i32* %ptr2
  %ptr3 = getelementptr inbounds i32* %lst, i32 3
  store i32 1, i32* %ptr3
  %ptr4 = getelementptr inbounds i32* %lst, i32 4
  store i32 1, i32* %ptr4
  %ptr5 = getelementptr inbounds i32* %lst, i32 5
  store i32 1, i32* %ptr5
  %ptr6 = getelementptr inbounds i32* %lst, i32 6
  store i32 1, i32* %ptr6
  %p0 = getelementptr inbounds { i32*, i32 }* %strct, i32 0, i32 0
  %p1 = getelementptr inbounds { i32*, i32 }* %strct, i32 0, i32 1
  store i32* %lst, i32** %p0
  store i32 7, i32* %p1
  %lst7 = load { i32*, i32 }* %strct
  %0 = alloca { i32, float, i8*, float, { i32*, i32 }, i8* }
  %ptr8 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %0, i32 0,
    i32 0
  store i32 1500, i32* %ptr8
  %ptr9 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %0, i32 0,
    i32 1
  store float 0x4043F8C7E0000000, float* %ptr9
  %ptr10 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %0, i32 0,
    i32 2
  store i8* getelementptr inbounds ([2 x i8]* @str, i32 0, i32 0), i8** %ptr10
  %ptr11 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %0, i32 0,
    i32 3
  store float 0x4052CDDE60000000, float* %ptr11
  %ptr12 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %0, i32 0,
    i32 4
```

```
store { i32*, i32 } %lst7, { i32*, i32 }* %ptr12
%ptr13 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %0, i32 0, i32 5
store i8* getelementptr inbounds ([16 x i8]* @str6, i32 0, i32 0), i8** %ptr13
%1 = load { i32, float, i8*, float, { i32*, i32 }, i8* }* %0
%c = alloca { i32, float, i8*, float, { i32*, i32 }, i8* }
store { i32, float, i8*, float, { i32*, i32 }, i8* } %1, { i32, float, i8*, float, { i32*, i32 }, i8* }* %c
%strct14 = alloca { i32*, i32 }
%lst15 = alloca i32, i32 7
%ptr16 = getelementptr inbounds i32* %lst15, i32 0
store i32 0, i32* %ptr16
%ptr17 = getelementptr inbounds i32* %lst15, i32 1
store i32 1, i32* %ptr17
%ptr18 = getelementptr inbounds i32* %lst15, i32 2
store i32 1, i32* %ptr18
%ptr19 = getelementptr inbounds i32* %lst15, i32 3
store i32 1, i32* %ptr19
%ptr20 = getelementptr inbounds i32* %lst15, i32 4
store i32 1, i32* %ptr20
%ptr21 = getelementptr inbounds i32* %lst15, i32 5
store i32 1, i32* %ptr21
%ptr22 = getelementptr inbounds i32* %lst15, i32 6
store i32 0, i32* %ptr22
%p023 = getelementptr inbounds { i32*, i32 }* %strct14, i32 0, i32 0
%p124 = getelementptr inbounds { i32*, i32 }* %strct14, i32 0, i32 1
store i32* %lst15, i32** %p023
store i32 7, i32* %p124
%lst25 = load { i32*, i32 }* %strct14
%2 = alloca { i32, float, i8*, float, { i32*, i32 }, i8* }
%ptr26 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %2, i32 0, i32 0
store i32 750, i32* %ptr26
%ptr27 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %2, i32 0, i32 1
store float 0x4043C7AE20000000, float* %ptr27
%ptr28 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %2, i32 0, i32 2
store i8* getelementptr inbounds ([2 x i8]* @str7, i32 0, i32 0), i8** %ptr28
%ptr29 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %2, i32 0, i32 3
store float 0x4052DD2F20000000, float* %ptr29
%ptr30 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %2, i32 0, i32 4
store { i32*, i32 } %lst25, { i32*, i32 }* %ptr30
%ptr31 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %2, i32 0, i32 5
store i8* getelementptr inbounds ([17 x i8]* @str8, i32 0, i32 0), i8** %ptr31
%3 = load { i32, float, i8*, float, { i32*, i32 }, i8* }* %2
%d = alloca { i32, float, i8*, float, { i32*, i32 }, i8* }
store { i32, float, i8*, float, { i32*, i32 }, i8* } %3, { i32, float, i8*, float, { i32*, i32 }, i8* }* %d
%4 = alloca { i32 }
%ptr32 = getelementptr inbounds { i32 }* %4, i32 0, i32 0
store i32 1, i32* %ptr32
%5 = load { i32 }* %4
%g = alloca { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }, { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }, { i32 } }
%ptr33 = getelementptr inbounds { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }, { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }, { i32 } }* %g, i32 0, i32 2
store { i32 } %5, { i32 }* %ptr33
%c34 = load { i32, float, i8*, float, { i32*, i32 }, i8* }* %c
%c35 = load { i32, float, i8*, float, { i32*, i32 }, i8* }* %c
```

```
%strct36 = alloca { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }
%lst37 = alloca { i32, float, i8*, float, { i32*, i32 }, i8* }, i32 2
%ptr38 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %lst37, i32 0
store { i32, float, i8*, float, { i32*, i32 }, i8* } %c34, { i32, float, i8*, float, { i32*, i32 }, i8* }* %ptr38
%ptr39 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %lst37, i32 1
store { i32, float, i8*, float, { i32*, i32 }, i8* } %c35, { i32, float, i8*, float, { i32*, i32 }, i8* }* %ptr39
%p040 = getelementptr inbounds { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }* %strct36, i32 0, i32 0
%p141 = getelementptr inbounds { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }* %strct36, i32 0, i32 1
store { i32, float, i8*, float, { i32*, i32 }, i8* }* %lst37, { i32, float, i8*, float, { i32*, i32 }, i8* }** %p040
store i32 2, i32* %p141
%lst42 = load { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }* %strct36
%6 = alloca { i32 }
%ptr43 = getelementptr inbounds { i32 }* %6, i32 0, i32 0
store i32 1, i32* %ptr43
%7 = load { i32 }* %6
%8 = alloca { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }
%ptr44 = getelementptr inbounds { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }* %8, i32 0, i32 0
store { i32, float, i8*, float, { i32*, i32 }, i8* }* %c, { i32, float, i8*, float, { i32*, i32 }, i8* }** %ptr44
%ptr45 = getelementptr inbounds { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }* %8, i32 0, i32 1
store { i32, float, i8*, float, { i32*, i32 }, i8* }* %d, { i32, float, i8*, float, { i32*, i32 }, i8* }** %ptr45
%ptr46 = getelementptr inbounds { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }* %8, i32 0, i32 2
store i1 false, i1* %ptr46
%ptr47 = getelementptr inbounds { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }* %8, i32 0, i32 3
store { i32 } %7, { i32 }* %ptr47
%9 = load { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }* %8
%strct48 = alloca { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }
%lst49 = alloca { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }
%ptr50 = getelementptr inbounds { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }* %lst49, i32 0
store { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } } %9, { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }* %ptr50
%p051 = getelementptr inbounds { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }* %strct48, i32 0, i32 0
%p152 = getelementptr inbounds { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }* %strct48, i32 0, i32 1
store { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }* %lst49, { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }** %p051
store i32 1, i32* %p152
%lst53 = load { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }* %strct48
%ptr54 = getelementptr inbounds { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }, { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }, { i32 } }* %g, i32 0, i32 0
store { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 } %lst42, { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }* %ptr54
%ptr55 = getelementptr inbounds { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32
```

```
}, { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }, { i32 } }* %g, i32 0, i32 1
  store { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 } %lst53, { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }* %ptr55
  %g56 = load { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }, { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }, { i32 } }* %g
  %g57 = alloca { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }, { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }, { i32 } }
  store { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }, { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }, { i32 } } %g56, { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, i32 }, { { { i32, float, i8*, float, { i32*, i32 }, i8* }*, { i32, float, i8*, float, { i32*, i32 }, i8* }*, i1, { i32 } }*, i32 }, { i32 } }* %g57
  %ptr58 = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %c, i32 0, i32 5
  store i8* getelementptr inbounds ([6 x i8]* @str9, i32 0, i32 0), i8** %ptr58
  %ext_val = getelementptr inbounds { i32, float, i8*, float, { i32*, i32 }, i8* }* %c, i32 0, i32 4
  %10 = load { i32*, i32 }* %ext_val
  %strct59 = alloca { i32*, i32 }
  store { i32*, i32 } %10, { i32*, i32 }* %strct59
  %tmp = getelementptr inbounds { i32*, i32 }* %strct59, i32 0, i32 1
  %len = load i32* %tmp
  ret void
}
```

Sample 2 in Grail:

```

main(){
a = {weight:4};
b .= a;
b.weight = 5;
c = {weight: 2};
d = {weight: 2};

y = 5;
if(c == d){
      y = 3;
}

e = a -- c with {weight: 1};

}
```

Sample 2 in LLVM:

```

; ModuleID = 'Grail'

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt1 = private unnamed_addr constant [4 x i8] c"%f\0A\00"

declare i32 @printf(i8*, ...)

declare i32 @sample_display(i32)

define void @main() {
entry:
  %0 = alloca { i32 }
  %ptr = getelementptr inbounds { i32 }* %0, i32 0, i32 0
```

```
  store i32 4, i32* %ptr
  %1 = load { i32 }* %0
  %a = alloca { i32 }
  store { i32 } %1, { i32 }* %a
  %a1 = load { i32 }* %a
  %strct = alloca { i32 }
  %strct2 = alloca { i32 }
  store { i32 } %a1, { i32 }* %strct2
  %tmp = getelementptr inbounds { i32 }* %strct2, i32 0, i32 0
  %val = load i32* %tmp
  %tmp3 = getelementptr inbounds { i32 }* %strct, i32 0, i32 0
  store i32 %val, i32* %tmp3
  %rec = load { i32 }* %strct
  %b = alloca { i32 }
  store { i32 } %rec, { i32 }* %b
  %ptr4 = getelementptr inbounds { i32 }* %b, i32 0, i32 0
  store i32 5, i32* %ptr4
  %2 = alloca { i32 }
  %ptr5 = getelementptr inbounds { i32 }* %2, i32 0, i32 0
  store i32 2, i32* %ptr5
  %3 = load { i32 }* %2
  %c = alloca { i32 }
  store { i32 } %3, { i32 }* %c
  %4 = alloca { i32 }
  %ptr6 = getelementptr inbounds { i32 }* %4, i32 0, i32 0
  store i32 2, i32* %ptr6
  %5 = load { i32 }* %4
  %d = alloca { i32 }
  store { i32 } %5, { i32 }* %d
  %y = alloca i32
  store i32 5, i32* %y
  %c7 = load { i32 }* %c
  %d8 = load { i32 }* %d
  %strct9 = alloca { i32 }
  store { i32 } %c7, { i32 }* %strct9
  %strct10 = alloca { i32 }
  store { i32 } %d8, { i32 }* %strct10
  %tmp11 = getelementptr inbounds { i32 }* %strct10, i32 0, i32 0
  %val12 = load i32* %tmp11
  %tmp13 = getelementptr inbounds { i32 }* %strct9, i32 0, i32 0
  %val14 = load i32* %tmp13
  %tmp15 = icmp eq i32 %val14, %val12
  %tmp16 = mul i1 %tmp15, true
  br i1 %tmp16, label %then, label %else

merge:                                            ; preds = %else, %then
  %6 = alloca { i32 }
  %ptr17 = getelementptr inbounds { i32 }* %6, i32 0, i32 0
  store i32 1, i32* %ptr17
  %7 = load { i32 }* %6
  %8 = alloca { { i32 }*, { i32 }*, i1, { i32 } }
  %ptr18 = getelementptr inbounds { { i32 }*, { i32 }*, i1, { i32 } }* %8, i32 0, i32 0
  store { i32 }* %a, { i32 }** %ptr18
  %ptr19 = getelementptr inbounds { { i32 }*, { i32 }*, i1, { i32 } }* %8, i32 0, i32 1
  store { i32 }* %c, { i32 }** %ptr19
  %ptr20 = getelementptr inbounds { { i32 }*, { i32 }*, i1, { i32 } }* %8, i32 0, i32 2
  store i1 false, i1* %ptr20
  %ptr21 = getelementptr inbounds { { i32 }*, { i32 }*, i1, { i32 } }* %8, i32 0, i32 3
  store { i32 } %7, { i32 }* %ptr21
  %9 = load { { i32 }*, { i32 }*, i1, { i32 } }* %8
  %e = alloca { { i32 }*, { i32 }*, i1, { i32 } }
  store { { i32 }*, { i32 }*, i1, { i32 } } %9, { { i32 }*, { i32 }*, i1, { i32 } }* %e
  ret void

then:                                             ; preds = %entry
```

```
  store i32 3, i32* %y
  br label %merge

else:                                          ; preds = %entry
  br label %merge
}
```

6.5 Testing Roles

Jiaxin created the testing infrastructure, including automation of regression tests by Shell scripts, and the tester for parser, which spit out the outputs from the parser after we feeding source GRAIL codes into the compiler. Jiaxin also designed test cases, and reported bugs to the member responsible for the code (Rose, Riva and Aashima), who would in turn find and solve the reported error.

7 Lessons Learned

7.1 Rose Sloan

Lessons learned in this project can be split into two categories: lessons learned about programming a compiler and lessons learned about developing software in a group. I will briefly discuss both.

I worked on two portions of the code: the scanner/parser and the codegen module. The former mostly drew upon knowledge I already had, as I am quite familiar with CFGs and parsing from my background in natural language processing, so that portion of the project mostly taught me about the specifics of the ocamlyacc format. Codegen, however, provided much more of a challenge. I learned a lot about how LLVM works, particularly how it uses pointers and structs. (In fact, I am now comfortable reading LLVM code, which is something I certainly couldn't say before this project.) My number one piece of advice to anyone in the future working on this project (or at least to anyone working on derived types in codegen) is to know and love the LLVM getelementptr instruction. It's a little confusing (so much so that there's an FAQ about it on the LLVM website, which is both lengthy and quite helpful), but once you get comfortable with it, it makes most operations on derived types infinitely easier.

As far as working with a group goes, the number one thing I would advise is talking to your group about any large structural changes as soon as possible. Throughout the project, there will be a number of times when you either have to change the structure of the AST or SAST or change the arguments or return type of a function. When you make these changes, you will most likely break someone else's code. In general, I recommend having one group member who's broadly familiar with most of the code and can update everything after someone makes one of these changes and get everything back to a point where all the code compiles. (I often served this role for our group.) It's a little tedious, but if compatibility updates can be made quickly and correctly, it really helps everyone make progress on the project as a whole.

7.2 Jiaxin Su

As the tester for this compiler project, I learn a lot in terms of Shell scripting, organizing tests, and working with the rest of the project team to ensure the reliability of the compiler. Since our test infrastructure is in Shell, I was required to learn to read and write bash in a short period of time. My Shell scripting skills is drastically improved by the end of the class. It is also interesting to explore various languages (OCaml, LLVM, AWK, Shell, and our own Grail) in one single course. This kind of exploration definitely improved my programming sense, which will be helpful in the long run.

Furthermore, I discovered that organizing test suites and writing good test cases were not easy at all. First of all, I had to know what need to be tested and how to test them: should we test them in the function context or just in small, separate main function? Since I have to write grail codes and the expected output, I had to have a good understanding about the language syntax. The last main thing I learn is that a tester are required to be a good OCaml code reader and have a good understanding what the entire project (not only as a whole but also in every specific part) so that he or she knows how to work with the rest of the team and understand the team's needs.

The main advice I have for the perspective students who will take PLT in the future is that don't be afraid of asking questions. In order to work well with your team, sometimes you just have to be "stupid" and ask whatever you do not know, even if it is a very trivial thing. Good communications definitely will help improve your teamwork experience.

7.3 Aashima Arora

I think the final result of GRAIL was extremely rewarding. Along with that, I do believe that there are quite a lot of key takeaways from this project.

As far as software development aspect is concerned, I learned how to effectively collaborate with peers on a large scale programming project and evenly distribute duties. Though I have done that in the past since I have worked in the industry for about 3 years, I definitely have a few best practices to take away from working with my three teammates Riva, Rose, and Jiaxin. I enjoyed working with each of them. I loved working on type inference and I think it is coolest thing to have ever happened to syntax. I also enjoyed contributing to Codegen and becoming familiar with something as low level as LLVM. I am a systems person so it wasn't that hard except a few annoying things that would come up sometimes. LLVM was quite understandable and not that hard to debug as well. I also integrated the GNU plot for graph displays and although it was quite a task, it worked out very well in the end. It definitely made me understand the intricacies of getting a C Program to poke into the LLVM output and manipulate it accordingly.

Overall, I learned some very interesting things and I totally concur with Prof. Edward's choice of OCaml for building the compiler because it did make things easier from implementation perspective. It's a pretty language and I learned it very well this semester. That is one more language in my pocket. Some pointers for a great final project from my end would be - start early, get involved in each component, have tests ready and do stress testing as much as you can.

7.4 Riva Tropp

Undergoing the process of writing a language gave me a new appreciation for the languages we use; how much thought goes into everything from scoping to equality, and how different choices can make languages great for some things and terrible for others.

Working on a team was a great experience, and I learned loads about github, synchronizing programming environments, and communicating effectively. I also learned tons of OCaml, my first functional programming language, which was a whole new paradigm for thinking about code. I also gained a respect for good type inference, including OCaml's (which would cheerfully spit out which of its types were throwing an error in my typer, no matter how convoluted the code). In terms of suggestions, communication is paramount, even if it can get annoying. Meeting times need to be set and confirmed, goals restated, important updates passed along to others if it will affect their code. I would also recommend, if there is an issue that could be solved in two different parts of the code, taking some time to discuss which makes the most sense, and what would be the issues involved. This happened several times between typer and codegen and it was not always intuitive where the change would be better.

A All Code

1. **scanner.mll** Authors - Rose Sloan

```
(* Ocamllex scanner for GRAIL *)

{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| "//"     { oneline lexbuf }
| '"'      { str (Buffer.create 16) lexbuf }
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| ';'      { SEMI }
| ','      { COMMA }
| '.'      { DOT }
| '+'      { PLUS }
| ".+"     { FPLUS }
| '-'      { MINUS }
| ".-"     { FMINUS }
| '*'      { TIMES }
| ".*"     { FTIMES }
| '/'      { DIVIDE }
| "./"     { FDIVIDE }
| "&"      { ADD }
| ".&"     { EADD }
| "+="     { PLUSEQ }
| ".+="    { FPLUSEQ }
| "&="     { ADDEQ }
| ".&="    { EADDEQ }
| '^'      { CARAT }
| "^="     { CARATEQ }
| '='      { ASSIGN }
| ".="     { COPY }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "--"     { DASH }
| "->"    { RARROW }
| "<-"     { LARROW }
| ':'      { COLON }
| "else"   { ELSE }
| "false"  { FALSE }
| "for"    { FOR }
| "free"   { FREE }
| "from"   { FROM }
| "to"     { TO }
| "rel"    { REL }
| "dir" { DIRECTED }
| "edges" { EDGES }
| "nodes" { NODES }
| "if"     { IF }
```

```
| "in"     { IN }
| "return" { RETURN }
| "true"   { TRUE }
| "type"   { TYPE }
| "while"  { WHILE }
| "with"   { WITH }
| ['0'-'9']+ as lxm { INTLIT(int_of_string lxm) }
| ['0'-'9']*'.'['0'-'9']* as lxm { DOUBLELIT(float_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '''(_ as mychar)''' { CHARLIT(mychar) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

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

and str strbuf = parse
  '"' { STRINGLIT( Buffer.contents strbuf ) }
| '\\' '"' { Buffer.add_char strbuf '"'; str strbuf lexbuf}
| '\\'  { Buffer.add_char strbuf '\\'; str strbuf lexbuf}
| [^ '\\' '"']+ { Buffer.add_string strbuf (Lexing.lexeme lexbuf); str strbuf lexbuf }
| eof { raise (Failure ("Unterminated String")) }
| _ { raise ( Failure("Problem with string")) }
```

Listing 13: scanner.mll

2. **parser.mly**
   Authors - Rose Sloan

```
%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS DIVIDE ASSIGN NOT DOT COLON
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE INT BOOLEAN VOID
%token TIMES LBRACKET RBRACKET DASH RARROW LARROW
%token ACCIO CHAR DOUBLE EDGE EMPTY
%token TO FROM IN RECORD TYPE WITH FREE DIRECTED EDGES NODES REL
%token FPLUS FMINUS FTIMES FDIVIDE ADD EADD CARAT
%token PLUSEQ FPLUSEQ ADDEQ EADDEQ COPY CARATEQ
%token <int> INTLIT
%token <char> CHARLIT
%token <float> DOUBLELIT
%token <string> STRINGLIT
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN COPY PLUSEQ FPLUSEQ ADDEQ EADDEQ CARATEQ
%nonassoc COLON
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ IN
%left ADD EADD CARAT
%left DOT
%nonassoc NOWITH
```

```
%nonassoc GRAPH
%nonassoc WITH
%nonassoc RBRACKET
%nonassoc LARROW RARROW DASH
%left PLUS MINUS FPLUS FMINUS
%left TIMES DIVIDE FTIMES FDIVIDE
%right NOT NEG

%start program
%type <Ast.program> program

%%


program:
  decls EOF { $1 }

decls:
   { [] }
 | decls_list { List.rev $1 }

 decls_list:
   func { [$1] }
 | decls_list func { $2::$1 }

func:
   func_dec LBRACE  stmt_list RBRACE { Fbody($1, List.rev $3) }

func_dec:
  ID LPAREN formals_opt RPAREN { Fdecl($1, $3) }

formals_opt:
    { [] }
  | formal_list   { List.rev $1 }

formal_list:
    ID                   { [$1] }
  | formal_list COMMA ID { $3 :: $1 }

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

stmt:
   expr SEMI  { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | IF LPAREN expr RPAREN LBRACE stmt_list RBRACE { If($3, List.rev $6, []) }
  | IF LPAREN expr RPAREN LBRACE stmt_list RBRACE ELSE LBRACE stmt_list RBRACE   { If(
    $3, List.rev $6, List.rev $10) }
  | IF LPAREN expr RPAREN LBRACE stmt_list RBRACE ELSE IF LPAREN expr RPAREN LBRACE
    stmt_list RBRACE  { If($3, List.rev $6, [If($11, List.rev $14, [])]) }
  | FOR LPAREN stmt expr SEMI stmt RPAREN LBRACE stmt_list RBRACE { For($3, $4, $6,
    List.rev $9) }
  | FOR LPAREN expr IN expr RPAREN LBRACE stmt_list RBRACE { Forin($3, $5, List.rev $8)
     }
  | expr ASSIGN expr SEMI { Asn($1, $3, true) }
  | expr COPY expr SEMI { Asn($1, $3, false) }
  | expr PLUSEQ expr SEMI { Asn($1, Binop($1, Add, $3), true) }
  | expr FPLUSEQ expr SEMI { Asn($1, Binop($1, Fadd, $3), true) }
  | expr ADDEQ expr SEMI { Asn($1, Binop($1, Gadd, $3), true) }
  | expr EADDEQ expr SEMI { Asn($1, Binop($1, Eadd, $3), true) }
  | expr CARATEQ expr SEMI { Asn($1, Binop($1, Ladd, $3), true)}
  | WHILE LPAREN expr RPAREN LBRACE stmt_list RBRACE { While($3, List.rev $6) }

  expr:
```

```
    INTLIT           { IntLit($1) }
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(false) }
  | STRINGLIT        { StrLit($1) }
  | CHARLIT          { CharLit($1) }
  | DOUBLELIT        { FloatLit($1) }
  | ID               { Id($1) }
  | LBRACKET actuals_opt RBRACKET { List($2)}
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | ID LBRACKET expr RBRACKET { Item($1, $3) }
  | expr DOT ID { Dot($1, $3) }
  | expr DOT FROM { Dot($1, "from") }
  | expr DOT TO { Dot($1, "to") }
  | expr DOT REL { Dot($1, "rel") }
  | expr DOT DIRECTED { Dot($1, "dir") }
  | expr DOT EDGES { Dot($1, "edges") }
  | expr DOT NODES { Dot($1, "nodes") }
  | 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 FPLUS   expr { Binop($1, Fadd,   $3) }
  | expr FMINUS  expr { Binop($1, Fsub,   $3) }
  | expr FTIMES  expr { Binop($1, Fmult,  $3) }
  | expr FDIVIDE expr { Binop($1, Fdiv,   $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 IN     expr { Binop($1, In,    $3) }
  | expr ADD    expr { Binop($1, Gadd, $3) }
  | expr EADD   expr { Binop($1, Eadd, $3) }
  | expr CARAT  expr { Binop($1, Ladd, $3)}
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr         { Unop(Not, $2) }
  | expr RARROW expr with_opt { Edge($1, To, $3, $4) }
  | expr LARROW expr with_opt { Edge($1, From, $3, $4) }
  | expr DASH expr with_opt  { Edge($1, Dash, $3, $4) }
  | LPAREN RPAREN WITH expr { Graph([], $4) }
  | LPAREN expr RPAREN WITH expr { Graph([$2], $5) }
  | LPAREN graph_list RPAREN WITH expr { Graph($2, $5) }
  | LBRACE rec_opt RBRACE { Record($2) }
  | LPAREN expr RPAREN %prec NOWITH { $2 }


with_opt:
   %prec NOWITH { Noexpr }
  | WITH expr { $2 }

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

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


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

```


rec_opt:
     { [] }
  | rec_list  { List.rev $1 }

rec_list:
    ID COLON expr                  { [($1, $3)] }
  | rec_list COMMA ID COLON expr { ($3, $5) :: $1 }
```

Listing 14: parser.mly

3. **ast.ml**

Authors - Rose Sloan, Riva Tropp

```
type id = string

(* type eop =   make all op *)


type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
          And | Or | In | Fadd | Fsub | Fmult | Fdiv | Gadd | Eadd | To | From | Dash |
     Ladd


type uop = Neg | Not

type primitiveType =
  | TInt
  | TBool
  | TString
  | TFloat
  | TChar
  | T of string
  | TVoid
  | TList of primitiveType
  | TRec of primitiveType * ((id * primitiveType) list) (*the entire type is explicit
    in TRec*)
  | TEdge of primitiveType * primitiveType * primitiveType (*name of type, node type,
    edge type*)
  | TGraph of primitiveType * primitiveType * primitiveType

type expr =
    IntLit of int
  | BoolLit of bool
  | StrLit of string
  | CharLit of char
  | FloatLit of float
  | Id of string
  | List of expr list
  | Call of string * expr list
  | Item of string * expr
  | Dot of expr * string
  | Unop of uop * expr
  | Binop of expr * op * expr
  | Edge of expr * op * expr * expr
  | Graph of expr list * expr
  | Record of (string * expr) list
  | Noexpr

(* annotated expr -> expr with types *)
type aexpr =
  | AIntLit of int * primitiveType
  | ACharLit of char * primitiveType
  | ABoolLit of bool * primitiveType
```

```
  | AStrLit of string * primitiveType
  | AFloatLit of float * primitiveType
  | AId of string * primitiveType
  | ABinop of aexpr * op * aexpr * primitiveType
  | AUnop of uop * aexpr * primitiveType
  | ACall of string * aexpr list * astmt list * string * primitiveType
  | AList of aexpr list * primitiveType          (*Make sure to check that the primitive
     type is only a TList*)
  | AItem of string * aexpr * primitiveType
  | ARecord of (string * aexpr) list * primitiveType
  | ADot of aexpr * string * primitiveType
  | AEdge of aexpr * op * aexpr * aexpr * primitiveType
  | AGraph of aexpr list * aexpr * primitiveType
  | ANoexpr of primitiveType

and astmt =
  | AAsn of aexpr * aexpr * bool * primitiveType
  | AIf of aexpr * astmt list * astmt list
  | AFor of astmt * aexpr * astmt * astmt list
  | AWhile of aexpr * astmt list
  | AReturn of aexpr * primitiveType
  | AExpr of aexpr
  | AForin of aexpr * aexpr * astmt list


and stmt =
  | Asn of expr * expr * bool
  | If of expr * stmt list * stmt list
  | While of expr * stmt list
  | For of stmt * expr * stmt * stmt list
  | Forin of expr * expr * stmt list
  | Return of expr
  | Expr of expr

type stmt_list = stmt list

type func_dec = Fdecl of id * id list

(*name, formals, return type*)
type afunc_dec = AFdecl of id * (id * primitiveType) list * primitiveType

type func = Fbody of func_dec * stmt list
type afunc = AFbody of afunc_dec * astmt list


type sast_afunc = {
  typ : primitiveType;
  fname : string;
  formals : (string * primitiveType) list;
  body: astmt list
}

type program = func list
```

Listing 15: ast.ml

4. **astutils.ml**
   Authors - Riva Tropp

```
open Ast

(*Let the strings begin *)
let string_of_op (op: op) =
  match op with
  | Add -> "+" | Mult -> "*" | Less -> "<" | Greater -> ">"
```

```
  | Or -> "||" | And -> "&&" | Sub -> "-" | Div -> "/" | Fadd -> ".+"
  | Equal -> "==" | Neq -> "-" | Leq -> "<=" | Geq -> ">=" | Fsub -> ".-"
  | Fmult -> ".*" | Fdiv -> "./" | To -> "<-" | From -> "->" | Dash -> "--"
  | In -> "in" | Gadd -> "&" | Eadd -> ".&" | Ladd -> "^"

let string_of_uop (uop: uop) =
  match uop with
  |Neg -> "-"
  |Not -> "not "

let rec string_of_type (t: primitiveType) =
  match t with
  | TRec(s, l) -> (Printf.sprintf "record %s" (string_of_type s))
  | TInt -> "int"
  | TBool -> "bool"
  | TFloat -> "float"
  | TString -> "str"
  | TChar -> "char"
  | TVoid -> "void"
  | TEdge(name, a, b) -> Printf.sprintf "edge %s (%s) with %s" (string_of_type name) (string_of_type a) (string_of_type b)
  | TGraph(name, a, b) -> Printf.sprintf "graph %s (%s) with %s" (string_of_type name) (string_of_type a) (string_of_type b)
  | TList(x) -> "list of " ^ (string_of_type x)
  | T(x) -> Printf.sprintf "any %s" x

let string_of_tuple (t: id * primitiveType) =
  match t with
    (a, b) -> a ^ " " ^ string_of_type b

let rec string_of_aexpr (ae: aexpr): string =
  match ae with
  | AIntLit(x, t)  -> Printf.sprintf "(%s: %s)" (string_of_int x) (string_of_type t)
  | ABoolLit(b, t) -> Printf.sprintf "(%s: %s)" (string_of_bool b) (string_of_type t)
  | AFloatLit(f, t) -> Printf.sprintf "(%s: %s)" (string_of_float f) (string_of_type t)
  | AStrLit(b, t) -> Printf.sprintf "(%s: %s)" (b) (string_of_type t)
  | ACharLit(c, t) -> Printf.sprintf "(%s: %s)" (String.make 1 c) (string_of_type t)
  | AId(x, t) -> Printf.sprintf "(%s: %s)" x (string_of_type t)
  | ADot(s,entry,t) -> Printf.sprintf "(%s.%s : %s)" (string_of_aexpr s) entry (string_of_type t)
  | AItem(s, e1, t) -> Printf.sprintf "(%s[%s] : %s)" s (string_of_aexpr e1) (string_of_type t)
  (*   | ASubset(_,_,t) -> Printf.sprintf "(%s)" (string_of_type t)
  *)  | ABinop(e1, op, e2, t) ->
    let s1 = string_of_aexpr e1 in let s2 = string_of_aexpr e2 in
    let sop = string_of_op op in let st = string_of_type t in
    Printf.sprintf "(%s %s %s: %s)" s1 sop s2 st
  | AUnop(op, e1, t) ->
    let s1 = string_of_aexpr e1 in let sop = string_of_uop op in let st = string_of_type t in
    Printf.sprintf "(%s%s: %s)" sop s1 st
  | ACall(id, aelist, _,id2, t) ->
    let s1 = List.map(fun a -> (string_of_aexpr (a))) aelist in
    let l = String.concat "," s1 in Printf.sprintf "(call %s(%s)) : %s" id2 l (string_of_type t)
  | ARecord(aexprs, t) ->
    let rec helper l str : string =
      (match l with
         [] -> str
       |(id, aexpr) :: t -> helper t (id ^ " " ^ string_of_aexpr aexpr ^ str))
    in
    (*     ignore(print_string ("list is length " ^ string_of_int (List.length aexprs))); *)
    ((string_of_type t) ^ "{" ^ (helper aexprs "") ^ "}")
  | AEdge(e1, op, e2, e3, t) -> Printf.sprintf "%s %s %s %s : %s" (string_of_aexpr e1)
```

```
    (string_of_op op) (string_of_aexpr e2) (string_of_aexpr e3) (string_of_type t)
  | AList(elist, t) -> Printf.sprintf "(%s : %s)" (string_of_aexpr_list elist) (string_of_type t)
  | AGraph(elist, e1, t) -> Printf.sprintf "(%s %s : %s)" (string_of_aexpr_list elist) (string_of_aexpr e1) (string_of_type t)
  | ANoexpr(_) -> ""

and string_of_aexpr_list l =
  match l with
    [] -> ""
  |h :: t -> string_of_aexpr h ^ string_of_aexpr_list t

and string_of_astmt (l: astmt) =
  let str =
    match l with
    | AReturn(aexpr,typ) -> "return " ^ string_of_aexpr aexpr ^ "; " ^ string_of_type typ ^ "\n";
    | AAsn(ae1,ae2,_,_) -> string_of_aexpr ae1 ^ " = " ^ string_of_aexpr ae2 ^ "; ";
    | AExpr(aexpr) -> " " ^ string_of_aexpr aexpr ^ "; "
    | AIf(e, s1, s2) ->
      let a = "if (" ^ string_of_aexpr e ^ ") {" ^ string_of_astmt_list s1 ^ "; " in
      let b =  (match s2 with
            [] -> ""
          |rest -> string_of_astmt_list rest) in (a ^ b)
    | AFor(as1, ae1, as2, astmts) ->
      "for (" ^ string_of_astmt as1  ^ string_of_aexpr ae1 ^ " ; " ^ string_of_astmt as2
      ^ string_of_astmt_list astmts
    | AWhile(ae1, astmts) -> "while (" ^ string_of_aexpr ae1 ^ ") {" ^ string_of_astmt_list astmts ^ "}"
    | AForin(id, aexpr, astmts) -> "for (" ^ string_of_aexpr id ^ " in " ^ string_of_aexpr aexpr ^ "){" ^ string_of_astmt_list astmts
  in str ^ "\n"

and string_of_astmt_list (stmts : astmt list) : string =
  let s1 = List.map(fun a -> (string_of_astmt (a))) stmts in let l = String.concat "" s1 in l

and string_of_stmt (l: stmt)=
  match l with
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | Asn(e1,e2,_) -> string_of_expr e1 ^ " = " ^ string_of_expr e2 ^ ";\n"
  | Expr(expr) -> " " ^ string_of_expr expr ^ ";\n"
  | If(e, s1,  s2) -> let a = "if (" ^ string_of_expr e ^ ") {" ^ string_of_stmt_list s1 ^ "; }" in
    let b =
      match s2 with
        [] -> ""
      |rest -> string_of_stmt_list rest in
    (a ^ b)
  | For(s1, e1, s2, astmts) -> "for (" ^ string_of_stmt s1 ^ string_of_expr e1 ^ string_of_stmt s2 ^" ) {\n" ^
                                  string_of_stmt_list astmts ^ "}"
  | While(e1, stmts) -> "while (" ^ string_of_expr e1 ^ "){\n" ^ string_of_stmt_list stmts ^ "}"
  | Forin(s, e, stmts) -> "for (" ^ string_of_expr s ^ " in " ^ string_of_expr e ^ "){" ^ string_of_stmt_list stmts

and string_of_stmt_list (stmts : stmt list) : string =
  let s1 = List.map(fun a -> (string_of_stmt (a))) stmts in let l = String.concat "" s1 in l

and string_of_expr (e: expr): string =
  match e with
  | IntLit(x) -> string_of_int x
```

```
  | BoolLit(b) -> string_of_bool b
  | StrLit(b) -> b
  | FloatLit(f) -> string_of_float f
  | CharLit(c) -> String.make 1 c
  | Id(s) -> s
  | Dot(a, b) -> ((string_of_expr a) ^ "." ^ b)
  (*   | Subset(s,e) -> Printf.sprintf "%s[%s]" s (string_of_expr e)*)
  | Binop(e1, op, e2) ->
    let s1 = string_of_expr e1 and s2 = string_of_expr e2 in
    let sop = string_of_op op in
    Printf.sprintf "(%s %s %s)" s1 sop s2
  | Unop(uop, e1) ->
    let s1 = string_of_expr e1 in
    let sop = string_of_uop uop in
    (Printf.sprintf "(%s%s)" sop s1)
  | Call(id, e) ->
    let s1 = List.map(fun a -> (string_of_expr (a))) e in let l = String.concat "," s1 in Printf.sprintf "(call %s(%s))" id l
  | Record(exprs) ->
    let rec helper l str : string =
      (match l with
          [] -> str
        |(s, e) :: t -> helper t (str ^ s ^ ": " ^ (string_of_expr e)))
    in ("{" ^ (helper exprs "") ^ "}")
  | Edge(e1, op, e2, e3) -> Printf.sprintf "%s %s %s %s" (string_of_expr e1) (string_of_op op) (string_of_expr e2) (string_of_expr e3)
  | List(elist) -> Printf.sprintf "(%s)" (string_of_expr_list elist)
  | Item(l, e) -> Printf.sprintf "%s[%s]" l (string_of_expr e)
  | Graph(elist, e) -> Printf.sprintf "(%s) with %s" (string_of_expr_list elist) (string_of_expr e)
  | Noexpr -> ""

and string_of_expr_list l =
  match l with
    [] -> ""
  |h :: t -> string_of_expr h ^ string_of_expr_list t


let string_of_func (func: sast_afunc) =
  let header = func.fname in
  let formals = "(" ^ String.concat ", " (List.map (fun (a,b) -> a ^ ": " ^ string_of_type b) func.formals) ^ "){ : " ^ string_of_type func.typ ^ "\n"
  in let body = String.concat "" (List.map string_of_astmt func.body) ^ "}\n"
  in header ^ formals ^ body
(*   let t = "Type :" ^ string_of_type func.typ
     in let name =
       " Name : " ^ func.fname
     in let formals = "(" ^ String.concat ", " (List.map fst func.formals) ^ ")\n{\n"
     in let body =
       String.concat "" (List.map string_of_astmt func.body) ^ "}\n"
     in t ^ name ^ formals ^body
*)
(*Maps a variable to its name in the environment*)
let map_id_with (fname: string )(id: string) : string =
  (*    ignore(print_string("map_id_with " ^ fname ^ "#" ^ id ^ "\n"));  *)
  (fname ^ "#" ^ id)

let map_func_id (fname: string) (calln: string): string =
  (*    ignore(print_string("map_id_with " ^ fname ^ "#" ^ id ^ "\n"));  *)
  (fname ^ "!" ^ calln)

(*Store variables with record names*)
let map_id_rec (rname: string) (id: string) : string =
(*   ignore(print_string ("getting name: " ^ rname ^ ";" ^ id ^ "\n"));  *)
```

```
  rname ^ ";" ^ id
```

Listing 16: astutils.ml

5. **codegen.ml**
   Authors - Rose Sloan, Aashima Arora

```
(* report errors found during code generation *)
exception Error of string

module L = Llvm
module A = Ast
module C = Char

module StringMap = Map.Make(String)
module TypeMap = Map.Make(String)

let translate (functions) =
  (* define *)
  let context = L.global_context () in
  let the_module = L.create_module context "Grail"
  and i32_t = L.i32_type context
  and i8_t  = L.i8_type  context
  and i1_t  = L.i1_type  context
  and str_t = L.pointer_type (L.i8_type context)
  and float_t = L.float_type context
  and void_t= L.void_type context
  and pointer_t = L.pointer_type
  in
  let tymap = (ref TypeMap.empty)
  in let rec ltype_of_typ = function
      A.TInt -> i32_t
    | A.TChar -> i8_t
    | A.TBool -> i1_t
    | A.TVoid -> void_t
    | A.TString -> str_t
    | A.TFloat -> float_t
    | A.TList t -> L.struct_type context [|L.pointer_type (ltype_of_typ t); i32_t|]
    | A.TRec(tany,tlist) ->
            let tname = (match tany with A.T s -> s | _ -> raise(Failure "the typer
    somehow gave us wrong input")) in
            let struct_name = ("struct."^tname) in
            if TypeMap.mem struct_name !tymap
            then
               TypeMap.find struct_name !tymap
            else

          let ret_types = Array.of_list(List.map (fun (_,t) -> ltype_of_typ t) tlist)
    in
                let record_t = L.struct_type context ret_types in
                tymap := TypeMap.add ("struct."^tname) record_t !tymap;
                record_t
    |A.TEdge(tany,trec1,trec2) ->
          let tname = (match tany with A.T s -> s | _ -> raise(Failure "the typer
    somehow gave us wrong input")) in
          let struct_name = ("struct."^tname) in
          if TypeMap.mem struct_name !tymap
          then
              TypeMap.find struct_name !tymap
          else
          let ret_types =
                              [  pointer_t (ltype_of_typ trec1);
                                 pointer_t (ltype_of_typ trec1);
                                 ltype_of_typ A.TBool;
                                 ltype_of_typ trec2;
```

```
                          ]
          in
          let all_ret_types = Array.of_list(ret_types) in
          let edge_t = L.struct_type context all_ret_types in
          tymap := TypeMap.add ("struct."^tname) edge_t !tymap;
          edge_t
   | A.TGraph(tany, nt, et) ->
          let tname = (match tany with A.T s -> s | _ -> raise(Failure "the typer somehow gave us wrong input")) in
          let struct_name = ("struct."^tname) in
          if TypeMap.mem struct_name !tymap
          then
              TypeMap.find struct_name !tymap
          else
          let ereltyp = (match et with A.TEdge(_, _, rel) -> rel | _ -> raise(Failure "wrong edge type")) in
          let ret_types = [| (ltype_of_typ (A.TList nt)); (ltype_of_typ (A.TList et)); (ltype_of_typ ereltyp)|]
          in let graph_t = L.struct_type context ret_types in
          tymap := TypeMap.add ("struct."^tname) graph_t !tymap;
          graph_t
   | _ -> raise(Failure "provided a bad type")
   in
(* Declare printf(), which the print built-in function will call *)
 let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
 let printf_func = L.declare_function "printf" printf_t the_module in

(* Declare sample_display(),for displaying a sample graph *)
 let display_t = L.function_type i32_t [| i32_t |] in
 let display_func = L.declare_function "sample_display" display_t the_module in

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

 (* Fill in the body of the given function *)
 let build_function_body afunc =
   let (the_function, _) = StringMap.find afunc.A.fname 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 in
   let float_format_str = L.build_global_stringptr "%f\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 (n, t) p = L.set_value_name n p;
       let local = L.build_alloca (ltype_of_typ t) n builder in
       ignore (L.build_store p local builder);
       StringMap.add n local m in

     List.fold_left2 add_formal StringMap.empty afunc.A.formals (Array.to_list (L.params the_function))
   in
   let lookup n map = try StringMap.find n map
     with Not_found -> raise (Failure ("undeclared variable " ^ n))
   in

```

```
        (* Invoke "f builder" if the current block does not already
       have a terminal (e.g., a branch). *)
    let add_terminal builder f =
          match L.block_terminator (L.insertion_block builder) with
            Some _ -> ()
          | None -> ignore (f builder)
    in

    let get_list_type t = (*quick utility function to map TList to the list's type*)
    (match t with
      A.TList x -> x
    |  _ -> raise(Failure "problem typing lists"))
  in

  let get_graph_types t = (*maps TGraph to the node and edge types*)
  (match t with
    A.TGraph(_, nt, et) -> (nt, et)
  | _ -> raise(Failure "not a graph")
  )
in

  let rec compare e1 e2 t builder = (*implements structural equality*)
    (match t with
    A.TInt | A.TChar | A.TBool -> (L.build_icmp L.Icmp.Eq e1 e2 "tmp" builder, builder)
    | A.TFloat -> (L.build_fcmp L.Fcmp.Oeq e1 e2 "tmp" builder, builder)
    | A.TRec(_, fields) -> let rec1 = L.build_alloca (ltype_of_typ t) "strct" builder in ignore(L.build_store e1 rec1 builder);
        let rec2 = L.build_alloca (ltype_of_typ t) "strct" builder in ignore(L.build_store e2 rec2 builder);
        compare_fields 0 fields rec1 rec2 builder
    | A.TEdge(_, trec1, trec2) -> let ed1 = L.build_alloca (ltype_of_typ t) "edge" builder in ignore(L.build_store e1 ed1 builder);
        let ed2 = L.build_alloca (ltype_of_typ t) "edge" builder in ignore(L.build_store e2 ed2 builder);
        let (fromcomp, builder) = compare (L.build_load (L.build_load (L.build_struct_gep ed1 0 "tmp" builder) "val" builder) "val" builder)
                                          (L.build_load (L.build_load (L.build_struct_gep ed2 0 "tmp" builder) "val" builder) "val" builder) trec1 builder in
        let (tocomp, builder) = compare (L.build_load (L.build_load (L.build_struct_gep ed1 1 "tmp" builder) "val" builder) "val" builder)
                                          (L.build_load (L.build_load (L.build_struct_gep ed2 1 "tmp" builder) "val" builder) "val" builder) trec1 builder in
        let (dircomp, builder) = compare (L.build_load (L.build_struct_gep ed1 2 "tmp" builder) "val" builder)
                                          (L.build_load (L.build_struct_gep ed2 2 "tmp" builder) "val" builder) A.TBool builder in
        let (relcomp, builder) = compare (L.build_load (L.build_struct_gep ed1 3 "tmp" builder) "val" builder)
                                          (L.build_load (L.build_struct_gep ed2 3 "tmp" builder) "val" builder) trec2 builder in
        (L.build_mul fromcomp (L.build_mul tocomp (L.build_mul dircomp relcomp "tmp" builder) "tmp" builder) "tmp" builder, builder)

    | A.TGraph(_, ntyp, etyp) -> let ereltyp = (match etyp with A.TEdge(_, _, rel) -> rel | _ -> raise(Failure "wrong edge type")) in
        let g1 = L.build_alloca (ltype_of_typ t) "graph" builder in ignore(L.build_store e1 g1 builder);
        let g2 = L.build_alloca (ltype_of_typ t) "graph" builder in ignore(L.build_store e2 g2 builder);
        let (nodescomp, builder) = compare (L.build_load (L.build_struct_gep g1 0 "tmp" builder) "val" builder)
                                          (L.build_load (L.build_struct_gep g2 0 "tmp" builder) "val" builder) (A.TList ntyp) builder in
```

```
      let (edgescomp, builder) = compare (L.build_load (L.build_struct_gep g1 1 "tmp" builder) "val" builder)
                                         (L.build_load (L.build_struct_gep g2 1 "tmp" builder) "val" builder)  (A.TList etyp) builder in
      let (relcomp, builder) = compare (L.build_load (L.build_struct_gep g1 2 "tmp" builder) "val" builder)
                                        (L.build_load (L.build_struct_gep g2 2 "tmp" builder) "val" builder) ereltyp builder in
      (L.build_mul nodescomp (L.build_mul edgescomp relcomp "tmp" builder) "tmp" builder, builder)
  | A.TList(_) -> compare_list e1 e2 t builder
  | A.TString -> raise(Failure "comparison of strings not supported")
  | _ -> raise(Failure "bad type provided to comparison operation")
  )

and compare_fields n fields rec1 rec2 builder = (*comparison for records*)
(match fields with
  [] -> (L.const_int i1_t 1, builder) (*empty recs are equal*)
| (_,t)::tl -> let (cmp, builder) = compare (L.build_load (L.build_struct_gep rec1 n "tmp" builder) "val" builder)
                                            (L.build_load (L.build_struct_gep rec2 n "tmp" builder) "val" builder) t builder in
               let (restcmp, builder) = compare_fields (n+1) tl rec1 rec2 builder in
               (L.build_mul cmp restcmp "tmp" builder, builder)
)


  and compare_list lst1 lst2 t builder =  (*comparison for lists*)
    let list_typ = get_list_type t in
    let struct1 = L.build_alloca (ltype_of_typ t) "strct" builder in ignore(L.build_store lst1 struct1 builder);
    let struct2 = L.build_alloca (ltype_of_typ t) "strct" builder in ignore(L.build_store lst2 struct2 builder);

    let len1 = L.build_load (L.build_struct_gep struct1 1 "tmp" builder) "len" builder
    and len2 = L.build_load (L.build_struct_gep struct2 1 "tmp" builder) "len" builder in

    let comp_val = L.build_icmp L.Icmp.Eq len1 len2 "tmp" builder in
    let comp_loc = L.build_alloca i1_t "loc" builder in ignore(L.build_store comp_val comp_loc builder);
    let merge_bb = L.append_block context "merge" the_function in

    let then_bb = L.append_block context "compare" the_function in

    ignore (L.build_cond_br comp_val then_bb merge_bb builder);

    (*compare list elements by checking size equality and then effectively using for-in loop*)
    let then_builder = L.builder_at_end context then_bb in
    let lstvals1 = L.build_load (L.build_struct_gep struct1 0 "tmp" then_builder) "lst" then_builder and
    lstvals2 = L.build_load (L.build_struct_gep struct2 0 "tmp" then_builder) "lst" then_builder  in
    let elind = L.build_alloca i32_t "ind" then_builder in ignore(L.build_store (L.const_int i32_t 0) elind then_builder);

    let pred_bb = L.append_block context "checklimits" the_function in
    ignore (L.build_br pred_bb then_builder);

    let body_bb = L.append_block context "comparison" the_function in
    let body_builder = L.builder_at_end context body_bb in
    let ind = L.build_load elind "i" body_builder in
    let p1 = L.build_in_bounds_gep lstvals1 [|ind|] "ptr" body_builder and p2 = L.
```

```
    build_in_bounds_gep lstvals2 [|ind|] "ptr" body_builder in
      let el1 = (L.build_load p1 "tmp" body_builder) and el2 = (L.build_load p2 "tmp" body_builder) in
      let (elcomp, body_builder) = compare el1 el2 list_typ body_builder in
      let comp_val = L.build_mul (L.build_load comp_loc "tmp" body_builder) elcomp "tmp" body_builder in
      ignore(L.build_store comp_val comp_loc body_builder);

      ignore(L.build_store (L.build_add (L.build_load elind "tmp" body_builder) (L.const_int i32_t 1) "inc" body_builder) elind body_builder);
      add_terminal body_builder (L.build_br pred_bb);

      let pred_builder = L.builder_at_end context pred_bb in
      let bool_val = L.build_icmp L.Icmp.Slt (L.build_load elind "tmp" pred_builder) len1 "comp" pred_builder in

      ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);

      let end_builder = L.builder_at_end context merge_bb in
      (L.build_load comp_loc "tmp" end_builder, end_builder)
  in
  let rec assign_array ar els n builder = (*stores elements, starting with element n in ar, returns ar*)
    match els with
      [] -> ar
    | e::tl -> let p = L.build_in_bounds_gep ar [|(L.const_int i32_t n)|] "ptr" builder in
                       ignore(L.build_store e p builder); assign_array ar tl (n+1) builder
    in

    let add_to_list lst el t builder = (*adds element el to the end of lst*)
      let list_typ = get_list_type t and newstruct = L.build_alloca (ltype_of_typ t) "strct" builder in
      let oldstruct = L.build_alloca (ltype_of_typ t) "strct" builder in ignore(L.build_store lst oldstruct builder);

      let oldlen = L.build_load (L.build_struct_gep oldstruct 1 "tmp" builder) "len" builder
      and oldlst = L.build_load (L.build_struct_gep oldstruct 0 "tmp" builder) "lst" builder in
      let newlen = L.build_add (L.const_int i32_t 1) oldlen "len" builder in
      ignore(L.build_store newlen (L.build_struct_gep newstruct 1 "tmp" builder) builder);

      let newlst = L.build_array_alloca(ltype_of_typ list_typ) newlen "lst" builder  in
      ignore(L.build_store el (L.build_in_bounds_gep newlst [|oldlen|] "ptr" builder) builder);

      let elind = L.build_alloca i32_t "ind" builder in ignore(L.build_store (L.const_int i32_t 0) elind builder);


      (*copy over old list elements by effectively using a for-in loop*)
      let pred_bb = L.append_block context "checklimits" the_function in
      ignore (L.build_br pred_bb builder);

      let body_bb = L.append_block context "assignment" the_function in
      let body_builder = L.builder_at_end context body_bb in
      let ind = (L.build_load elind) "i" body_builder in
      let oldp = L.build_in_bounds_gep oldlst [|ind|] "ptr" body_builder and newp = L.build_in_bounds_gep newlst [|ind|] "ptr" body_builder
      in ignore(L.build_store (L.build_load oldp "tmp" body_builder) newp body_builder);

```

```
    ignore(L.build_store (L.build_add (L.build_load elind "tmp" body_builder) (L.const_int i32_t 1) "inc" body_builder) elind body_builder);
    add_terminal body_builder (L.build_br pred_bb);

    let pred_builder = L.builder_at_end context pred_bb in
    let bool_val = L.build_icmp L.Icmp.Slt (L.build_load elind "tmp" pred_builder) oldlen "comp" pred_builder in

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

    let end_builder = L.builder_at_end context merge_bb in
    ignore(L.build_store newlst (L.build_struct_gep newstruct 0 "tmp" end_builder) end_builder);
    (L.build_load newstruct "strct" end_builder, end_builder)

  in

  let int_ops op =
       (match op with
          A.Add       -> L.build_add
        | A.Sub      -> L.build_sub
        | A.Mult     -> L.build_mul
        | A.Div      -> L.build_sdiv
        | 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
        | _ -> raise (Failure "wrong operation applied to ints")
       )
  in

  let float_ops op =
    (match op with
       A.Fadd       -> L.build_fadd
     | A.Fsub      -> L.build_fsub
     | A.Fmult     -> L.build_fmul
     | A.Fdiv      -> L.build_fdiv
     | A.Equal    -> L.build_fcmp L.Fcmp.Oeq
     | A.Neq      -> L.build_fcmp L.Fcmp.One
     | A.Less     -> L.build_fcmp L.Fcmp.Ult
     | A.Leq      -> L.build_fcmp L.Fcmp.Ole
     | A.Greater -> L.build_fcmp L.Fcmp.Ogt
     | A.Geq      -> L.build_fcmp L.Fcmp.Oge
     | _ -> raise (Failure "wrong operation applied to floats")
    )
  in

  let bool_ops op =
  (match op with
    A.And      -> L.build_and
  | A.Or       -> L.build_or
  | _ -> raise (Failure "wrong operation applied to bools") )

in

  let list_ops e1 e2 t op builder =
  (match op with
    A.Ladd -> add_to_list e1 e2 t builder
  | _ -> raise(Failure "wrong operation applied to lists")
  )

in
```

```

    let graph_ops e1 e2 t op builder =
      let (ntyp, etyp) = get_graph_types t in
      let gstruct = L.build_alloca (ltype_of_typ t) "strct" builder in ignore(L.build_store e1 gstruct builder);
      (match op with
        A.Gadd -> let oldns = L.build_load (L.build_struct_gep gstruct 0 "ptr" builder) "nodes" builder in
                  let (newns, builder) =  add_to_list oldns e2 (A.TList ntyp) builder in
                  ignore(L.build_store newns (L.build_struct_gep gstruct 0 "tmp" builder) builder);
                  ((L.build_load gstruct "g" builder), builder)
      | A.Eadd -> let oldes = L.build_load (L.build_struct_gep gstruct 1 "ptr" builder) "nodes" builder in
                  let (newes, builder) =  add_to_list oldes e2 (A.TList etyp) builder in
                  ignore(L.build_store newes (L.build_struct_gep gstruct 1 "tmp" builder) builder);
                  ((L.build_load gstruct "g" builder), builder)
      | _ -> raise(Failure "wrong operation applied to graphs")
      )

in

  let rec copy_list lst t builder = (*deep copy for lists*)
    let list_typ = get_list_type t and newstruct = L.build_alloca (ltype_of_typ t) "strct" builder in
    let oldstruct = L.build_alloca (ltype_of_typ t) "strct" builder in ignore(L.build_store lst oldstruct builder);

    let len = L.build_load (L.build_struct_gep oldstruct 1 "tmp" builder) "len" builder
    and oldlst = L.build_load (L.build_struct_gep oldstruct 0 "tmp" builder) "lst" builder in
    ignore(L.build_store len (L.build_struct_gep newstruct 1 "tmp" builder) builder);

    let newlst = L.build_array_alloca(ltype_of_typ list_typ) len "lst" builder  in
    let elind = L.build_alloca i32_t "ind" builder in ignore(L.build_store (L.const_int i32_t 0) elind builder);


    (*copy over old list elements by effectively using a for-in loop*)
    let pred_bb = L.append_block context "checklimits" the_function in
    ignore (L.build_br pred_bb builder);

    let body_bb = L.append_block context "assignment" the_function in
    let body_builder = L.builder_at_end context body_bb in
    let ind = L.build_load elind "i" body_builder in
    let oldp = L.build_in_bounds_gep oldlst [|ind|] "ptr" body_builder and newp = L.build_in_bounds_gep newlst [|ind|] "ptr" body_builder in
    let oldel = (L.build_load oldp "tmp" body_builder) in let (newel, body_builder) = copy oldel list_typ body_builder in
    ignore(L.build_store newel newp body_builder);

    ignore(L.build_store (L.build_add (L.build_load elind "tmp" body_builder) (L.const_int i32_t 1) "inc" body_builder) elind body_builder);
    add_terminal body_builder (L.build_br pred_bb);

    let pred_builder = L.builder_at_end context pred_bb in
    let bool_val = L.build_icmp L.Icmp.Slt (L.build_load elind "tmp" pred_builder) len "comp" pred_builder in

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

```

```
    let end_builder = L.builder_at_end context merge_bb in
    ignore(L.build_store newlst (L.build_struct_gep newstruct 0 "tmp" end_builder) end_builder);
    (L.build_load newstruct "strct" end_builder, end_builder)

and copy e t builder = (*returns a deep copy of e and the builder at the end of copy*)
  (match t with
      A.TInt -> (e, builder) (*no need to deep copy for primitive types*)
    | A.TChar -> (e, builder)
    | A.TBool -> (e, builder)
    | A.TVoid -> (e, builder)
    | A.TString -> (e, builder)
    | A.TFloat -> (e, builder)
    | A.TList _ -> copy_list e t builder
    | A.TRec(_, fields) -> let newrec = L.build_alloca (ltype_of_typ t) "strct" builder in
        let oldrec = L.build_alloca (ltype_of_typ t) "strct" builder in ignore(L.build_store e oldrec builder);
        copy_fields 0 fields newrec oldrec builder
    | A.TEdge(_, trec1, trec2) -> let newe = L.build_alloca (ltype_of_typ t) "edge" builder in
        let olde = L.build_alloca (ltype_of_typ t) "edge" builder in ignore(L.build_store e olde builder);
        let (newfrom, builder) = copy (L.build_load (L.build_load (L.build_struct_gep olde 0 "tmp" builder) "val" builder) "val" builder) trec1 builder in
        let (newto, builder) = copy (L.build_load (L.build_load (L.build_struct_gep olde 1 "tmp" builder) "val" builder) "val" builder) trec1 builder in
        let (newdir, builder) = copy (L.build_load (L.build_struct_gep olde 2 "tmp" builder) "val" builder) A.TBool builder in
        let (newrel, builder) = copy (L.build_load (L.build_struct_gep olde 3 "tmp" builder) "val" builder) trec2 builder in
        let frompoint = L.build_alloca (ltype_of_typ trec1) "fromp" builder in ignore(L.build_store newfrom frompoint builder);
        let topoint = L.build_alloca (ltype_of_typ trec1) "top" builder in ignore(L.build_store newto topoint builder);
        ignore(L.build_store frompoint (L.build_struct_gep newe 0 "tmp" builder) builder);
        ignore(L.build_store topoint (L.build_struct_gep newe 1 "tmp" builder) builder);
        ignore(L.build_store newdir (L.build_struct_gep newe 2 "tmp" builder) builder);
        ignore(L.build_store newrel (L.build_struct_gep newe 3 "tmp" builder) builder);
        (L.build_load newe "edge" builder, builder)
    | A.TGraph(_, ntyp, etyp) -> let ereltyp = (match etyp with A.TEdge(_, _, rel) -> rel | _ -> raise(Failure "wrong edge type")) in
        let newg = L.build_alloca (ltype_of_typ t) "graph" builder in
        let oldg = L.build_alloca (ltype_of_typ t) "graph" builder in ignore(L.build_store e oldg builder);
        let (newnodes, builder) = copy (L.build_load (L.build_struct_gep oldg 0 "tmp" builder) "val" builder) (A.TList ntyp) builder in
        let (newedges, builder) = copy (L.build_load (L.build_struct_gep oldg 1 "tmp" builder) "val" builder) (A.TList etyp) builder in
        let (newrel, builder) = copy (L.build_load (L.build_struct_gep oldg 2 "tmp" builder) "val" builder) ereltyp builder in
        ignore(L.build_store newnodes (L.build_struct_gep newg 0 "tmp" builder) builder);
        ignore(L.build_store newedges (L.build_struct_gep newg 1 "tmp" builder) builder);
        ignore(L.build_store newrel (L.build_struct_gep newg 2 "tmp" builder) builder);
        (L.build_load newg "graph" builder, builder)
    | _ -> raise(Failure "not a valid type")

  )

and copy_fields n fields newrec oldrec builder = (*deep copy for records*)
  (match fields with
```

```
    [] -> (L.build_load newrec "rec" builder, builder)
  | (_,t)::tl -> let (newval, builder) = copy (L.build_load (L.build_struct_gep oldrec n "tmp" builder) "val" builder) t builder in
                  ignore(L.build_store newval (L.build_struct_gep newrec n "tmp" builder) builder); copy_fields (n+1) tl newrec oldrec builder
  )

in

let get_expr_type e = (*a quick utility function to map an aexpr to its type*)
  (match e with
    A.AIntLit(_, t) -> t
  | A.ACharLit(_, t) -> t
  | A.ABoolLit(_, t) -> t
  | A.AStrLit(_, t) -> t
  | A.AFloatLit(_, t) -> t
  | A.AId(_, t) -> t
  | A.ABinop(_, _, _, t) -> t
  | A.AUnop(_, _, t) -> t
  | A.ACall(_, _, _, _, t) -> t
  | A.AList(_, t) -> t
  | A.AItem(_, _, t) -> t
  | A.ARecord(_, t) -> t
  | A.ADot(_, _, t) -> t
  | A.AEdge(_, _, _, _, t) -> t
  | A.AGraph(_, _, t) -> t
  | A.ANoexpr(t) -> t)

in

  let rec build_expressions l builder local_var_map = (*builds all aexprs in l, updating builder appropriately: basically combine map and fold*)
    (match l with
      [] -> ([], builder)
    | e::tl -> let (exp, newbuilder) = aexpr builder local_var_map e in
           let (other_exps, endbuilder) =
           build_expressions tl newbuilder local_var_map in (exp::other_exps, endbuilder))

  and build_list_from_els l t builder local_var_map = (*builds a list LLVM object given a list of its elements*)
    let list_typ = get_list_type t and (els, newbuilder) = build_expressions l builder local_var_map in
    let struct_var = L.build_alloca (ltype_of_typ t) "strct" newbuilder in
    let ar_var = L.build_array_alloca (ltype_of_typ list_typ) (L.const_int i32_t (List.length l)) "lst" newbuilder in
    let init_list = assign_array ar_var els 0 newbuilder in
    let p0 = L.build_struct_gep struct_var 0 "p0" newbuilder and p1 = L.build_struct_gep struct_var 1 "p1" newbuilder in
    ignore(L.build_store init_list p0 newbuilder); ignore(L.build_store (L.const_int i32_t (List.length l)) p1 newbuilder);
    (L.build_load struct_var "lst" newbuilder, newbuilder)

  and build_edge_with_record e1 op e2 erec typ builder local_var_map = (*builds an edge using the LLVM record erec*)
          let (directed,from,into) =
           match op with
             | A.Dash -> (false,e1,e2)
             | A.To -> (true,e1,e2)
             | A.From -> (true,e2,e1)
             | _ -> raise(Failure("undefined edge type"))

             in let get_ptr e =
                  (match e with
                    A.AId(n,_) ->
```

```
                        (try StringMap.find n local_var_map
                        with Not_found ->
                        raise (Failure ("undeclared variable " ^ n)))

                    | _-> raise (Failure ("Not supported.Node must be declared"))
                  )

            in let argslist =
            [   get_ptr from;
                get_ptr into;
                fst (aexpr builder local_var_map (A.ABoolLit(directed,A.TBool)));
                erec
            ]

         in let loc = L.build_alloca (ltype_of_typ typ) "" builder
     in let rec populate_structure fields i =
      match fields with
      | [] -> L.build_load loc "" builder
      | hd :: tl ->
            ( let eptr = L.build_struct_gep loc i "ptr" builder
                in ignore(L.build_store hd eptr builder);
          populate_structure tl (i+1)
              )
    in (populate_structure argslist 0, builder)


  and aexpr builder local_var_map = function
        A.AIntLit(i, _) -> (L.const_int i32_t i, builder)
      | A.ABoolLit(b, _) -> (L.const_int i1_t (if b then 1 else 0), builder)
      | A.AStrLit(s, _) -> (L.build_global_stringptr s "str" builder, builder)
      | A.ACharLit(c, _) -> (L.const_int i8_t (C.code c), builder)
      | A.AFloatLit(f, _) -> (L.const_float float_t f, builder)
      | A.AId(s,_) -> (L.build_load (lookup s local_var_map) s builder, builder)
      | A.AList(l, t) ->  build_list_from_els l t builder local_var_map
      | A.AItem(s, e, _) -> let strct = lookup s local_var_map in let arp = L.build_struct_gep strct 0 "tmp" builder in
                          let ar = L.build_load arp "tmpar" builder and (ad, builder) = aexpr builder local_var_map e in
                          let p = L.build_in_bounds_gep ar [|ad|] "ptr" builder in (L.build_load p "item" builder, builder)
      | A.ACall ("print", [e], _, _, _) -> let (e', builder') =  (aexpr builder local_var_map e) in
                                    (L.build_call printf_func [| e' |] "printf" builder', builder')
      | A.ACall ("sample_display", [e], _, _, _) -> let (e', builder') =  (aexpr builder local_var_map e) in
                                    (L.build_call display_func [| e' |] "sample_display" builder', builder')
      |A.ACall("display",[e],_,_,_) ->
        let t = get_expr_type e in
        let graph_display_t = L.function_type i32_t [|ltype_of_typ t|] in
        let graph_display_func = L.declare_function "display" graph_display_t the_module in let (e', builder') =  (aexpr builder local_var_map e) in
                                    (L.build_call graph_display_func [| e' |] "graph_display" builder', builder')

      | A.ACall("printint", [e], _, _, _) | A.ACall ("printbool", [e], _, _, _) -> let (e', builder') =  (aexpr builder local_var_map e) in
                                      (L.build_call printf_func [| int_format_str ; e' |] "printf" builder', builder')
      | A.ACall("printfloat", [e], _, _, _) -> let (e', builder') =  (aexpr builder local_var_map e) in
                              (L.build_call printf_func [| float_format_str ; e' |] "printf" builder', builder')
      |A.ACall("size", [e], _, _, _) -> let (e', builder') = (aexpr builder
```

```
local_var_map e) in
    let strct = L.build_alloca (L.type_of e') "strct" builder' in ignore(L.build_store e' strct builder');
    (L.build_load (L.build_struct_gep strct 1 "tmp" builder') "len" builder', builder')
  | A.ACall (_, act, _, callname, _) ->
    let (fdef, fdecl) = try StringMap.find callname function_decls with Not_found -> raise (Failure ("undeclared function " ^ callname)) in
    let (actuals', builder') = build_expressions (List.rev act) builder local_var_map in
    let actuals = List.rev actuals' in
    let result = (match fdecl.A.typ with A.TVoid -> ""
                                    | _ -> callname ^ "_result") in
    (L.build_call fdef (Array.of_list actuals) result builder', builder')
  | A.AUnop(op, e, _) ->
        let (e', builder') = aexpr builder local_var_map e in
       ((match op with
        A.Neg     -> L.build_neg
       | A.Not     -> L.build_not) e' "tmp" builder', builder')
  | A.ABinop (e1, op, e2, t) ->     let (e1', builder1) = aexpr builder local_var_map e1
    in let (e2', builder') = (
      match e2 with
      A.AEdge(n1, op, n2, rel, typ) ->
        (match rel with
          A.ANoexpr _ -> let g = L.build_alloca (ltype_of_typ t) "g" builder1 in ignore(L.build_store e1' g builder1);
                            build_edge_with_record n1 op n2 (L.build_load (L.build_struct_gep g 2 "ptr" builder1) "tmp" builder1) typ builder1 local_var_map
        | _ -> aexpr builder1 local_var_map e2
        )
    |  _  -> aexpr builder1 local_var_map e2 )
  in
  let et = get_expr_type e1 in
  (match op with
  A.Equal -> compare e1' e2' et builder
| A.Neq -> let (compval, builder) = compare e1' e2' et builder in (L.build_sub (L.const_int i1_t 1) compval "tmp" builder, builder)
| _ ->  (match et with
     | A.TFloat -> ((float_ops op) e1' e2' "tmp" builder', builder')
     | A.TBool -> ((bool_ops op) e1' e2' "tmp" builder', builder')
     | A.TList _ -> list_ops e1' e2' t op builder'
     | A.TGraph(_,_,_) -> graph_ops e1' e2' t op builder'
     | _ -> ((int_ops op) e1' e2' "tmp" builder', builder')
    ))
    | A.ANoexpr _ -> (L.const_int i32_t 0, builder)
  | A.ARecord(alist,trec) ->
        let (argslist, builder) = build_expressions (List.map (fun f -> (snd f)) alist) builder local_var_map
     in let loc = L.build_alloca (ltype_of_typ trec) "" builder
 in let rec populate_structure fields i =
  match fields with
  | [] -> L.build_load loc "" builder
  | hd :: tl ->
        ( let eptr = L.build_struct_gep loc i "ptr" builder
            in ignore(L.build_store hd eptr builder);
      populate_structure tl (i+1)
          )
in (populate_structure argslist 0, builder)
   | A.AEdge(e1,op,e2,item,typ) -> let (rel, builder) = aexpr builder local_var_map item in
      build_edge_with_record e1 op e2 rel typ builder local_var_map
  | A.ADot(e1,entry,_) ->
       let rec match_name lst n =
         match lst with
```

```
                | [] -> raise (Failure ("Not found"))
                | h :: t -> if h = n then 0 else
                            1 + match_name t n
        in
        let t = get_expr_type e1 in
        (match t with
          A.TRec(_, alist) -> let mems = List.map fst alist in
          (match e1 with
            | A.AId(name,_) ->
                let index = match_name mems entry
                in let load_loc = lookup name local_var_map
                in let ext_val = L.build_struct_gep load_loc index "ext_val" builder
                in (L.build_load ext_val "" builder, builder)
            |_ ->
                let (e',builder) = aexpr builder local_var_map e1
                in
                let loc = L.build_alloca (L.type_of e') "e" builder in
                let _ = L.build_store e' loc builder
                in
                let mems =
               List.map (fun (id,_)  -> id) alist

                in let index = match_name mems entry
                in let ext_val = L.build_struct_gep loc index "ext_val" builder
                in (L.build_load ext_val "" builder, builder))
          | A.TEdge(_, _, _) ->  let (e', builder) = aexpr builder local_var_map e1 in
            let loc = L.build_alloca (L.type_of e') "e" builder in ignore(L.build_store e' loc builder);
            (match entry with
              "from" -> (L.build_load (L.build_load (L.build_struct_gep loc 0 "ptr" builder) "from" builder) "from" builder, builder)
            | "to" -> (L.build_load (L.build_load (L.build_struct_gep loc 1 "ptr" builder) "to" builder) "to" builder, builder)
            | "dir" -> (L.build_load (L.build_struct_gep loc 2 "ptr" builder) "dir" builder, builder)
            | "rel" -> (L.build_load (L.build_struct_gep loc 3 "ptr" builder) "rel" builder, builder)
            | _ -> raise( Failure "dot not supported with this keyword")
            )
          | A.TGraph(_, _, _) ->  let (e', builder) = aexpr builder local_var_map e1 in
            let loc = L.build_alloca (L.type_of e') "e" builder in ignore(L.build_store e' loc builder);
            (match entry with
              "nodes" -> (L.build_load (L.build_struct_gep loc 0 "ptr" builder) "nodes" builder, builder)
            | "edges" -> (L.build_load (L.build_struct_gep loc 1 "ptr" builder) "edges" builder, builder)
            | _ -> raise( Failure "dot not supported with this keyword")
            )
          | _ -> raise(Failure "dot not supported on this type")
      )

      | A.AGraph(lst, rel, t) ->
        let rec split_lists l =
          match l with
            [] -> ([], [], [])
            | h::tl -> let (nodes, edges, ids) = split_lists tl in
            let typ = get_expr_type h in
            (match typ with
              A.TRec(_, _) -> (match h with
                              A.AId(name, _) -> if List.mem name ids then (nodes, edges, ids)
                                                 else (h::nodes, edges, name::ids)
```

```
                              |            _ -> (h::nodes, edges, ids))
             | A.TEdge(_, _, _) -> (match h with
                                A.AEdge(node1, o, node2, r, ty) ->
                                  let (newnodes, newids) =
                                    (match node1 with
                                      A.AId(name, _) -> if List.mem name ids then (nodes, ids) else (node1::nodes, name::ids)
                                    | _ -> (nodes, ids)
                                    )
                                  in let (newnodes, newids) =
                                    (match node2 with
                                      A.AId(name, _) -> if List.mem name newids then (newnodes, newids) else (node1::newnodes, name::newids)
                                    | _ -> (newnodes, newids)
                                    )
                                  in let newe =
                                    (match r with
                                      A.ANoexpr(_) -> A.AEdge(node1, o, node2, rel, ty)
                                    | _ -> h)
                                  in (newnodes, newe::edges, newids)

                                | _ -> (nodes, h::edges, ids))
             | _ -> raise(Failure "wrong type given to graph constructor")
                         )
       in

       let (nodes, edges, _) = split_lists lst in
       let (grel, builder) = aexpr builder local_var_map rel in
       let graph = L.build_alloca (ltype_of_typ t) "g" builder in
       ignore(L.build_store grel (L.build_struct_gep graph 2 "ptr" builder) builder);

       let (ntyp, etyp) = get_graph_types t in
       let (nlist, builder) = build_list_from_els nodes (A.TList ntyp) builder local_var_map in
       let (elist, builder) = build_list_from_els edges (A.TList etyp) builder local_var_map in
       ignore(L.build_store nlist (L.build_struct_gep graph 0 "ptr" builder) builder);
       ignore(L.build_store elist (L.build_struct_gep graph 1 "ptr" builder) builder);
       (L.build_load graph "g" builder, builder)




(* Build the code for the given statement; return the builder for
   the statement's successor *)

in let rec astmt (builder, local_var_map) = function
      A.AExpr(e) -> ((snd (aexpr builder local_var_map e)), local_var_map)
    | A.AReturn(e, t) -> (match t with
          A.TVoid -> ignore(L.build_ret_void builder); (builder, local_var_map)
        | _ -> let (e', builder') = (aexpr builder local_var_map e) in ignore(L.build_ret e' builder'); (builder', local_var_map))
    | A.AAsn(s, e, b, t) ->
       let (e', builder') = aexpr builder local_var_map e in
       let (e', builder') = if b then (e', builder') else copy e' t builder' in
       let 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

       (match s with
         A.AId(name, _) ->
```

```
          let local_var_map = if StringMap.mem name local_var_map
          then local_var_map
          else add_local local_var_map (t,name) in
          ignore (L.build_store e' (lookup name local_var_map) builder'); (builder', local_var_map)
          | A.AItem(name, adr, _) ->
          let arp = L.build_struct_gep (lookup name local_var_map) 0 "tmp" builder and (ad, builder') = aexpr builder' local_var_map adr in
          let ar = L.build_load arp "tmpar" builder' in
          let p = L.build_in_bounds_gep ar [|ad|] "ptr" builder' in ignore(L.build_store e' p builder'); (builder', local_var_map)
        | A.ADot(r, entry, _) ->
          let rec match_name lst n =
            (match lst with
              [] -> raise (Failure ("Not found"))
              | h :: t -> if h = n then 0 else
                          1 + match_name t n)
          in
          let name =
            (match r with
              A.AId(s, _) -> s
            | _ -> raise(Failure("invalid lvalue"))
            )
          in
          let rtype = get_expr_type r in
          let alist = (match rtype with
             A.TRec(_, l) -> l
            | _ -> raise( Failure "wrong type provided to record") )
              in
          let mems = List.map fst alist in
          let index = match_name mems entry in
          let recval = (lookup name local_var_map) in
          let ptr = L.build_struct_gep recval index "ptr" builder' in
          ignore(L.build_store e' ptr builder'); (builder', local_var_map)
          | _ -> raise(Failure "invalid lvalue"))



      | A.AIf (predicate, then_stmt, else_stmt) ->
        let (bool_val, builder) = aexpr builder local_var_map predicate in
        let merge_bb = L.append_block context "merge" the_function in

        let then_bb = L.append_block context "then" the_function in
        add_terminal (fst (List.fold_left astmt ((L.builder_at_end context then_bb), local_var_map) then_stmt))
          (L.build_br merge_bb);

        let else_bb = L.append_block context "else" the_function in
        add_terminal (fst (List.fold_left astmt ((L.builder_at_end context else_bb), local_var_map) else_stmt))
          (L.build_br merge_bb);

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

      | A.AWhile (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 (fst (List.fold_left astmt ((L.builder_at_end context body_bb), local_var_map) body))
          (L.build_br pred_bb);

        let pred_builder = L.builder_at_end context pred_bb in
```

```
        let (bool_val, pred_builder) = aexpr pred_builder local_var_map 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, local_var_map)


      | A.AFor (s1, e2, s3, body) -> List.fold_left astmt (builder, local_var_map)
                                       [s1 ; A.AWhile(e2, List.rev (s3::(List.rev body)))]


      | A.AForin (e1, e2, body) ->
      let ind = (match e1 with A.AId(s, t) -> (s, t) | _ -> raise(Failure "invalid for loop")) in let lst = L.build_alloca (ltype_of_typ (A.TList (snd ind))) "lst" builder in
      let (e2', builder) = (aexpr builder local_var_map e2)
      in ignore(L.build_store e2' lst builder);
      let elvar = L.build_alloca (ltype_of_typ (snd ind)) (fst ind) builder in
      let local_var_map = StringMap.add (fst ind) elvar local_var_map in
      let ar = L.build_load (L.build_struct_gep lst 0 "tmp" builder) "ar" builder and endlst = L.build_load (L.build_struct_gep lst 1 "tmp" builder) "end" builder in
      let elind = L.build_alloca i32_t "ind" builder in ignore(L.build_store (L.const_int i32_t 0) elind builder);

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

      let body_bb = L.append_block context "while_body" the_function in
      let body_builder = L.builder_at_end context body_bb in
      let p = L.build_in_bounds_gep ar [|(L.build_load elind) "i" body_builder|] "ptr" body_builder in ignore(L.build_store (L.build_load p "tmp" body_builder) elvar body_builder);
      let (endbody_builder, new_local_var_map) = (List.fold_left astmt ((L.builder_at_end context body_bb), local_var_map) body) in
      ignore(L.build_store (L.build_add (L.build_load elind "tmp" endbody_builder) (L.const_int i32_t 1) "inc" endbody_builder) elind endbody_builder);
      add_terminal endbody_builder (L.build_br pred_bb);

      let pred_builder = L.builder_at_end context pred_bb in
      let bool_val = L.build_icmp L.Icmp.Slt (L.build_load elind "tmp" pred_builder) endlst "comp" pred_builder in

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



  (* Build the code for each statement in the function *)
  in let (builder,_) = List.fold_left
         astmt (builder,local_vars) afunc.A.body
  in
  (* Add a return if the last block falls off the end *)
  add_terminal builder (match afunc.A.typ with
        A.TVoid -> L.build_ret_void
        | t -> L.build_ret (L.const_int (ltype_of_typ t) 0)
      )
  in


List.iter build_function_body functions;
the_module
```

Listing 17: codegen.ml

6. **grail.ml**
Authors - Riva Tropp, Aashima Arora

```
open Ast
open Astutils

module NameMap = Map.Make(String)
type environment = primitiveType NameMap.t
type genvironment = (primitiveType * (string * primitiveType) list * stmt list) NameMap.t

let  builtins = ref [
                   ("print", (TVoid, [("x", TString)], []));
                   ("sample_display",(TInt, [("x",TInt)],[Return(IntLit(0))]));
                   ("printint", (TVoid, [("x", TInt)], []));
                   ("printfloat", (TVoid, [("x", TFloat)], []));
                   ("printbool", (TVoid, [("x", TBool)], []));
                   ("printchar", (TVoid, [("x", TChar)], []));
                   ("size", (TInt, [("x", TList(Infer.gen_new_type()))], [Return(IntLit(1))]));
                   ("display", (TVoid, [("x", TGraph(Infer.gen_new_type(), Infer.gen_new_type(), Infer.gen_new_type()))], []))]

let parse (s) : Ast.program =
  Parser.program Scanner.token (s)

(*https://www.rosettacode.org/wiki/Sort_using_a_custom_comparator#OCaml*)
let mycmp l1 l2 =
  (if String.length l1 <> String.length l2 then
    compare (String.length l2) (String.length l1)
  else
    String.compare (String.lowercase l1) (String.lowercase l2))
(*   |_ -> raise(failwith("formal not a string")) *)

(*Extra checking functions*)
let check_overload (e: Ast.func) (genv : genvironment) : unit =
  match e with
    Fbody(Fdecl(fname, _), _) ->
    if(NameMap.mem fname genv)
    then (raise (failwith ("function " ^ fname ^ " already defined.")))
    else ()

(*checks for shared formals*)
let check_formals (s: string list) : unit =
  let rec helper l =
    match l with
    | x :: y :: xs ->
    if (x = y) then (raise (failwith ("Error: Shared formal.")))
    else (helper (y :: xs))
    |_ -> ()
  in helper (List.sort mycmp s)

(* Culls uncalled functions (whose variables are still typed as any) from the sast. *)
let rec enforce_no_any(funcs: Ast.afunc list) : Ast.afunc list =
  match funcs with
  |[] -> []
  |AFbody(AFdecl(name, aformals, ret), stmts) :: tail ->
  let toss =
    List.fold_left (fun hasany f ->
    (match f with | (_,T(_)) -> true | _ -> hasany)) false aformals
  in
  let toss = match ret with T(_) -> true | _ -> toss in
  let toss = if(List.length stmts = 0) then(true) else(toss) in
  if(toss) then(enforce_no_any tail) else(List.hd funcs :: enforce_no_any tail)

```

```
let rec get_formals(formals: string list)(func: string) : (id * primitiveType) list =
  check_formals formals;
  List.map (fun i -> (map_id_with func i, Infer.gen_new_type())) formals

let infer_func (e: Ast.func) (genv : genvironment) : (Ast.afunc list * genvironment) =
  check_overload e genv;
  match e with
  |Fbody(Fdecl(fname, formals), stmts) ->
    let ids = get_formals formals fname in
    let env = List.fold_left (fun m (i,t) -> NameMap.add i t m) NameMap.empty ids in
    let genv = NameMap.add fname (Infer.gen_new_type(),ids,stmts) genv in
    Infer.infer_func (env, genv, [], []) e

let grail (ast: Ast.afunc list) (input) : Ast.afunc list =
  let rec get_sast(p: Ast.program) (genv : genvironment) (l : Ast.afunc list) : Ast.
    afunc list  =
    match p with
    [] -> let li = enforce_no_any (List.rev l) in li
    |hd :: tl -> let (afuncs, genv) =
                   infer_func hd genv
    in get_sast tl genv (afuncs @ l) in
   let rec addbuiltins l genv =
    match l with
    |[] -> genv
    |(a, b) :: t -> let genv = NameMap.add a b genv in addbuiltins t genv
  in let genv = addbuiltins !builtins NameMap.empty
  in
  get_sast (parse (input)) genv []


let format_sast_codegen (ast : Ast.afunc) : Ast.sast_afunc =
  match ast with
    AFbody(AFdecl(name, aformals, t), astmts) ->
    { typ = t;
      fname = name;
      formals = aformals;
      body = astmts
    }

(*Interpreter for debugging purposes*)
(* let rec interpreter (ast: Ast.sast_afunc list) : Ast.sast_afunc list =
  print_string "> ";
  let input = Lexing.from_channel stdin in
    try
      (*do for func*)
       let ast = List.map format_sast_codegen (grail [] (input)) in ast (* in
    interpreter (pre_ast @ ast) *)
    with
    | Failure(msg) ->
      if msg = "lexing: empty token" then [] @ interpreter (ast)
      else (print_endline msg; [] @ interpreter(ast))
    | _ -> print_endline "Syntax Error"; [] @  interpreter (ast)

    let say() =
      let str = "Welcome to Grail, the awesomest language!\n"  in
      print_string str

      let rec display (input: Ast.sast_afunc list) : unit =
      match input with
      [] -> ()
      | h :: t ->
      print_string (string_of_func h);
      display t;;

      say();
```

```
      let l = interpreter([]) in display l *)

   let compile() =
   let file =
   try
   Lexing.from_channel stdin
    with
    | _ -> raise(failwith("Syntax Error"))
    in
   let sast =
   List.map format_sast_codegen (grail [] file)
   in
   let m = Codegen.translate sast in
   Llvm_analysis.assert_valid_module m;
   print_string (Llvm.string_of_llmodule m);;
   compile()

(*
    let sast = List.map format_sast_codegen (grail [] file) in    let m = Codegen.
    translate sast in
   Llvm_analysis.assert_valid_module m; print_string

   (Llvm.string_of_llmodule m);;
   compile();
 *)
```

Listing 18: grail.ml

7. **infer.ml**
   Authors - Riva Tropp, Aashima Arora

```
open Ast
open Astutils

module NameMap = Map.Make(String)
type environment = primitiveType NameMap.t
type genvironment = (primitiveType * (string * primitiveType) list * stmt list) NameMap
    .t
type recs = primitiveType list
type funcs = afunc list
type allenv = environment * genvironment * recs * funcs
(* local, global, records, functions*)
(* Unknown type,  resolved type. eg.[(T, TInt); (U, TBool)] *)
type substitutions = (id * primitiveType) list

let callstack = Stack.create()

let map_id (id: string) : string =
  let fname = Stack.top callstack in
  (map_id_with fname id)

(*One for function templates generated, one for type variables*)
let func_variable = ref 1
let type_variable = ref 1

(* generates a new unknown type placeholder.
 * returns T(string) of the generated int *)
let gen_new_type () =
  let c1 = !type_variable in
  incr type_variable;
  T(string_of_int c1)

let get_func_name (id: id) : string =
  let calln = !func_variable in
  incr func_variable;
```

```
  map_func_id id (string_of_int   calln)

let get_id (e: expr) : string =
  match e with
  |Id(str) -> str
  |_ -> raise(failwith(string_of_expr e ^ " is not an id."))

let get_subtype (t: primitiveType) : primitiveType =
  match t with
  |TList(st) -> st
  | T(_) -> t
  |x -> raise(failwith("error: " ^ string_of_type x ^ " not iterable."))

let type_of (ae: aexpr): primitiveType =
  match ae with
  | AIntLit(_, t) | ABoolLit(_, t) | AStrLit(_,t) | AFloatLit(_, t) | ACharLit(_,t) ->
    t
  | AId(_, t) -> t
  | ABinop(_, _, _, t) -> t
  | AItem(_,_,t) -> t
  | ACall(_, _, _,_, t) -> t
  | AList(_, t) -> t
  | ARecord(_,t) -> t
  | AEdge(_,_,_,_,t) -> t
  | ADot(_,_,t) -> t
  | AUnop(_,_,t) -> t
  | ANoexpr(t) -> t
  | AGraph(_,_,t) -> t

(*Generate unique record type based on fields*)
let gen_new_rec (fieldslist : (id * aexpr) list) : primitiveType =
  let fields = List.map (fun (a, b) -> a, type_of b) fieldslist
  in TRec(gen_new_type(), fields)

let get_rec (recs: recs) (fieldslist: (id * aexpr) list) : primitiveType =
  let rec helper (l : recs) (curr : ((id * primitiveType) list)) (rl : recs) =
  match l with
  |[] -> let newtype = gen_new_rec(fieldslist) in newtype
  |TRec(name, fl) :: t ->
  if(fl = curr)
  then(TRec(name, fl))
  else(helper t curr rl)
  |_ -> raise(failwith("error"))
in helper recs (List.map (fun (a, b) -> a, type_of b) fieldslist) recs

(*Ensures an expression is a conditional (e.g. for predicate statements)*)
let check_bool (e: aexpr) : unit =
(*     print_string "Checking bool"; *)
  if(type_of e != TBool)
  then(raise(failwith ((string_of_aexpr e) ^ " not a boolean.")))
  else ()

(*A checking function for something like the first field of a for*)
let check_asn (a: stmt) : unit =
  (*   print_string "Checking assign\n";*)
  match a with
    Asn(_,_,_) -> ()
  |_ -> raise(failwith ((string_of_stmt a) ^ " not an assignment statement."))

let format_formal (formal: (string * primitiveType) * aexpr) : string * primitiveType =
  match formal with
  ((x, _), e) -> (x, type_of e)

(* Updates the name map for the formals with the types of the actuals. *)
let update_types_formals (stufflist: ((id * primitiveType) * aexpr) list) (env:
```

```
    environment) (id: string): environment =
  List.fold_left (fun e f -> let id, typ = format_formal f in
    NameMap.add (map_id id) typ e) env stufflist

(* In graph, the type of the edges can be inferred from the type of the graph *)
let enforce_type (ae: aexpr) (nt: primitiveType): aexpr =
  match ae with
  | AId(a, t) -> AId(a, nt)
  | ABinop(a,b,c,t) ->  ABinop(a,b,c,nt)
  | AItem(a,b,t) -> AItem(a,b,nt)
  | ACall(a,b,c,d, t) -> ACall(a,b,c,d,nt)
  | AList(a, t) -> AList(a, nt)
  | ARecord(a,t) -> ARecord(a,nt)
  | AEdge(a,b,c,d,t) -> AEdge(a,b,c,d,nt)
  | ADot(a,b,t) -> ADot(a,b,nt)
  | AUnop(a,b,t) -> AUnop(a,b,nt)
  | ANoexpr(t) -> ANoexpr(nt)
  | AGraph(a,b,t) ->  AGraph(a,b,nt)
  | ABoolLit(a,t) -> ABoolLit(a,nt)
  | ACharLit(a,t) -> ACharLit(a,nt)
  | AIntLit(a,t) -> AIntLit(a,nt)
  | AStrLit(a,t) -> AStrLit(a,nt)
  | AFloatLit(a,t) -> AFloatLit(a,nt)

(*Comparator used in annotating records.*)
let comp (x: id * expr) (y: id * expr) : int =
  match x, y with
  |(a,_), (b,_) -> if(a = b) then(0) else(if(a < b) then(-1) else(1))

(*Helper function for annotating records (check for duplicate fields)*)
let rec has_dups l =
  match l with
  |(a,_) :: (b,c) :: tail -> if(a = b) then(true) else(has_dups((b,c)::tail))
  |[]| _ -> false

(*finds the variable in the map*)
let find_in_map(id: string) (env: environment): primitiveType =
   let mapped = map_id id in        (*in astutils*)
   if (NameMap.mem mapped env)
   then (NameMap.find mapped env)
   else (raise(failwith(mapped ^ " not found@79")))

(* Runs over all the nodes and edges and assigns them the type *)
let enforce_node_consistency (plist: aexpr list) (typ: primitiveType) =
  let rec helper pl typ =
  match pl with
  |[] -> []
  |h :: tl ->
  let enforced = match type_of h with
  |TRec(_,_)| T(_) | TVoid  -> h
  |TEdge(_,_,_) -> enforce_type h typ
  |x -> raise(failwith(string_of_aexpr h ^ " should not be in constructor."))
  in enforced :: helper tl typ
in helper plist typ

(* Split the graph constructor into two lists based on their types *)
let rec split_types (aelist: aexpr list) : (primitiveType list * primitiveType list) =
  let rec helper l edgelist nodelist : (primitiveType list * primitiveType list) =
  (match l with
  |[] -> edgelist, nodelist
  |h :: t ->
  let et1 = type_of h in
  (match et1 with
   |TRec(_,_) -> helper t edgelist (et1 :: nodelist)
   |TEdge(_,n,_) -> helper t (et1 :: edgelist) (n :: nodelist)
```

```
    |T(_) | TVoid -> helper t edgelist nodelist
    |_ -> raise(failwith(string_of_type et1 ^ " not a graph type."));
  ))
  in (helper aelist [] [])

(*Searches a list of record fields for a particular id and gets its type*)
let rec get_field_type (elist: (id * primitiveType) list) (id: id) :primitiveType =
  if(List.mem id ["from"; "to"; "rel"])
  then(gen_new_rec([]))
  else(
  match elist with
  [] -> raise (failwith (id ^ "  not defined @ 133"))
  |(field, typ) :: tail -> if(field = id) then(typ) else(get_field_type tail id))

let rec check_field (fields: ((id * primitiveType) * (id * primitiveType))) : unit =
  match fields with
  |(id1, t1), (id2, t2) -> if(id1 = id2) then(check_compatible_types (t1,t2)) else(
    raise(failwith("mismatched fields " ^ id1 ^ " & " ^ id2)))

and check_compatible_types (t: primitiveType * primitiveType) : unit =
  match t with
    |T(_), a | TVoid, a | a, TVoid | a, T(_) -> ()
    |TList(_), TList(T(_)) | TList(T(_)), TList(_) -> ()
    |TRec(a, b), TRec(c, d) -> ignore(let fieldslists = List.combine b d in List.map (
    fun a -> check_field a) fieldslists); ()
    |TEdge(_, b, c), TEdge(_,e,f) -> ignore(check_compatible_types (b,e));
    check_compatible_types (c,f)
    |TGraph(a, b, c), TGraph(_, e, f) -> ignore(check_compatible_types (b,e));
    check_compatible_types (c,f)
    |a, b -> if(a = b) then () else raise(failwith("type mismatch: " ^ (string_of_type a
    ) ^ ","  ^(string_of_type b) ^ "@118"))

(*Ensures all members of a list share the same type.*)
let rec check_type_consistency (tl: primitiveType list) : unit =
  match tl with
  |x :: y :: t ->
  ignore(check_compatible_types (x,y));
  check_type_consistency (y :: t)
  |[] | _ -> ()

(* A function is a list of statements. Each statement's expressions are inferred here.
The result is annotated and passed into the sast. *)
let rec infer_stmt_list (allenv: allenv) (e: stmt list) : (allenv * astmt list) =
    let rec helper allenv astmts stmts  : (allenv * astmt list) =
     match stmts with
       [] -> (allenv, List.rev astmts)
       |fst :: snd :: tail ->
       let allenv, ae = type_stmt allenv fst in
       let allenv, ae2 = type_stmt allenv snd in
       (match ae with
       |AReturn(ae, _) -> raise(failwith("error: unreachable statment " ^
     string_of_astmt ae2));
       |_ -> (helper allenv (ae2 :: ae :: astmts) tail))
       |x :: tail -> let allenv, ae = type_stmt allenv x in helper allenv (ae :: astmts)
      tail
  in helper allenv [] e

and type_stmt (allenv: allenv) (e: stmt) : allenv * astmt  =
  let allenv, astmt = infer_stmt allenv e in
  let _, genv,recs,funcs = allenv in
  let env,_,recs,_ = update_map allenv astmt in
  ((env, genv, recs,funcs), astmt)

and infer_stmt (allenv: allenv) (e: stmt): (allenv * astmt) =
(*   ignore(print_string ("\ninferring " ^ (string_of_stmt e)));   *)
```

```
let env, genv, recs, funcs = allenv in
let allenv, inferred_astmt =
match e with
| Asn(e1, e2, switch) ->
  let ae2 = infer_expr allenv e2 in
  let typ = type_of ae2 in
  let ae1, env =
        match e1 with
        |Id(a) ->
        let id = map_id a in
        let env =  (* if a variable is first encountered here, add it to env*)
        if NameMap.mem (id) env
        then (let otype = type_of (infer_expr allenv e1) in
              ignore(check_compatible_types (otype, typ)); env)
        else (NameMap.add id (gen_new_type()) env) in
        AId(a, typ), env
        |Item(a,_)|Dot(Id(a),_) ->
        let id = map_id a in
        if(NameMap.mem id env)
        then(infer_expr (env, genv, recs, funcs) e1, env)
        else(raise(failwith(id ^ " not defined.")))
        |x -> raise(failwith(string_of_expr x ^ " is not a valid lval"))
  in
  (allenv, AAsn(ae1, ae2, switch, typ))
| Return(expr) ->
  let aexpr = infer_expr allenv expr in
  let allenv = env, genv, recs, funcs in
  (allenv, AReturn(aexpr, type_of aexpr))
| Expr(expr) ->
  let aexpr = infer_expr allenv expr in
  let allenv = env, genv, recs, funcs in
  (allenv, AExpr(aexpr))
| If(expr, s1, s2) ->
  (* Statement blocks only modify the environment in the block *)
  let conditional = infer_expr allenv expr in
  (check_bool conditional);
  let ((_,genv,_,funcs), as1) = infer_stmt_list allenv s1 in
  let ((_,genv,_,funcs), as2) = infer_stmt_list (env,genv,recs,funcs) s2 in
  let allenv = env, genv, recs, funcs in
  (allenv, AIf(conditional, as1, as2))
| While(e1, s1s) ->
  let ae1 = infer_expr allenv e1 in ignore(check_bool ae1);
  let ((_,genv,_,_funcs), as1s) = infer_stmt_list allenv s1s in
  let allenv = env, genv, recs, funcs in
  (allenv, AWhile(ae1, as1s))
| For(s1, e1, s2, stmts) ->
  (check_asn s1);
  (check_asn s2);
  let outerenv = allenv in
  let (allenv, as1) = type_stmt allenv s1 in
  let ae1 = infer_expr allenv e1
  in (check_bool ae1);
  let (allenv, as2) = (type_stmt allenv s2) in
  let _, astmts = infer_stmt_list allenv stmts in
   (outerenv, AFor(as1, ae1, as2, astmts))
| Forin(e1, e2, stmts) ->
  let outerenv = allenv in
  let env, genv, recs, funcs = allenv in
  let id = (get_id e1) in
  let ae2 = infer_expr allenv e2 in
  let it = get_subtype (type_of ae2) in
  let env = NameMap.add (map_id id) it env in
  let allenv = env, genv, recs, funcs in
  let aid = infer_expr allenv e1 in
  let _, astmts = infer_stmt_list allenv stmts in  (*change type_stmt to update the
```

```
    map*)
     (outerenv, AForin(aid, ae2, astmts))
    in let env, genv, recs, funcs = allenv in
    let funcs = update_funcs inferred_astmt funcs genv
  in ((env,genv,recs,funcs), inferred_astmt)

(*Called from annotate_stmt, infers expressions inside statements.*)
and infer_expr (allenv: allenv) (e: expr): (aexpr)  =
  let annotated_expr = annotate_expr allenv e in
  let constraints = collect_expr annotated_expr in
  let subs = unify constraints in
  let ret = apply_expr subs annotated_expr in ret

(*Step 1 of HM: annotate expressions with what we know of their types.*)
and annotate_expr (allenv: allenv) (e: expr) : aexpr =
let env, genv, recs, funcs = allenv in
  match e with
  | IntLit(n) -> AIntLit(n, TInt)
  | BoolLit(b) -> ABoolLit(b, TBool)
  | StrLit(s) -> AStrLit(s,TString)
  | FloatLit(f) -> AFloatLit(f, TFloat)
  | CharLit(c) -> ACharLit(c, TChar)
  | Noexpr -> ANoexpr(gen_new_type())
  | Id(x) ->
    let typ = find_in_map x env in
    (match typ with
     |t ->  AId(x, t))
  | Item(s, e) ->
    let et1 = annotate_expr allenv e in
    let typ = find_in_map s env in
    (match typ with
     |TVoid -> raise (failwith (s ^ " not defined @ 115."))
     |TList(t) -> AItem(s, et1, t)
     |T(a) -> AItem(s, et1, gen_new_type())
     |t -> raise (failwith (string_of_type (t) ^ " not a list.")))
  | Binop(e1, op, e2) ->
    let et1 = annotate_expr allenv e1
    and et2 = annotate_expr allenv e2
    and new_type = gen_new_type () in
    ABinop(et1, op, et2, new_type)
  | Unop(uop, e1) ->
    let et1 = annotate_expr allenv e1 and t = gen_new_type() in
    AUnop(uop, et1, t)
  | Dot(e1, entry) ->
    let ae1 = annotate_expr allenv e1 in
    let et1 = type_of ae1 in
    let sae1 = string_of_aexpr ae1 in
    let typ =
         (match et1 with
          |TRec(str, elist) ->
          get_field_type elist entry
          |TGraph(_,n,e) ->
          (match entry with
          |"edges" -> TList(e)
          |"nodes" -> TList(n)
          |_ -> raise(failwith(entry ^ " not a field."))
          )
          |TEdge(a,n,e) ->
          (match entry with
          |"from" |"to" -> n
          |"dir" -> TBool
          |"rel" -> e
          | _ -> raise(failwith(entry ^ " not a field."))
          )
          |T(x) -> T(x)
```

```
          |x -> raise(failwith (sae1 ^ " not a record.")))
    in ADot(ae1, entry, typ)
  | List(e) ->
    let ael = List.map (fun a -> annotate_expr allenv a) e in
    let len = List.length ael in
    if (len = 0)
    then (AList(ael, TList(gen_new_type())))
    else (ignore(check_type_consistency (List.map (fun a -> type_of a) ael));
        let tl = List.nth ael (len-1) in
        let t = (type_of (tl)) in
      AList(ael, TList(t)))
  | Call(id, elist) ->
    let aelist = List.map (fun a -> infer_expr allenv a) elist in
    let callingfunc = Stack.top callstack in
    Stack.push id callstack;
    let (oldtype, aformals, stmts) =
      if (NameMap.mem id genv)
      then (NameMap.find id genv)
      else (raise (failwith "function not defined @ 147")) in
    if(id=callingfunc)
    then(ACall(id, aelist, [], id, oldtype)) (*no infinite loops. Give the correct
    statements here?? *)
    else(
    ignore(let len = List.length aformals in
    if (List.length aelist != len)
    then(raise(failwith("error: " ^ id ^ " takes " ^ (string_of_int len) ^ " formal/s")
    ))
    else());
    (* Here we reinfer the function for the call by mapping the formals to the actuals.
     *)
    let env = update_types_formals (List.combine aformals aelist) env id in
    let allenv = env, genv, recs, funcs in
    ignore(check_formals aformals allenv);
    let (_, astmts) = (infer_stmt_list allenv stmts) in
    let t = get_return_type astmts in
    ignore(Stack.pop callstack);
    let in_id = get_func_name id in (* the id for this call of the function. *)
    ACall(id, aelist, astmts, in_id, t))
| Record(pairlist) ->
    let rec helper(l: (string * expr) list) =
    match l with
    [] -> []
    |(id, expr) :: tl ->
    (id, (annotate_expr allenv expr)) :: helper tl
    in let apairlist = helper (List.sort comp pairlist) in
    ignore(if(has_dups pairlist) then(raise(failwith("error: duplicate record entry")))
     else());
    let typ = get_rec recs apairlist in
    ARecord(apairlist, typ)
 | Edge(e1, op, e2, e3) ->
   let ae1 = annotate_expr allenv e1 and
       ae2 = annotate_expr allenv e2 and
       ae3 = annotate_expr allenv e3 in
      AEdge(ae1, op, ae2, ae3, TEdge(gen_new_type(), type_of ae1, type_of ae3))
 | Graph(elist, tedge) ->
   let aelist = List.map (fun a -> infer_expr allenv a) elist in
   let edgelist, nodelist = split_types aelist in
   ignore(check_type_consistency (edgelist));
   ignore(check_type_consistency (nodelist));

   let atemplate = infer_expr allenv tedge in
   let etype = type_of atemplate in
   let ntype =
   if(List.length nodelist = 0) then(gen_new_rec([])) else(List.hd nodelist) in
   let edgetype = if(List.length edgelist = 0)
```

```
                   then(TEdge(gen_new_type(), gen_new_type(), gen_new_type()))
                   else(List.hd edgelist) in
    let gtype = match edgetype with
    |TEdge(name, nt, et) -> TEdge(name, ntype, etype)
    |_ -> raise(failwith("error"));
    in
    let aelist = enforce_node_consistency aelist (gtype) in
    AGraph(aelist, atemplate, TGraph(gen_new_type(), ntype, gtype))
  (*a. check the list for consistency between nodes and edges. (which could be noexprs or lists themselves, or type of e.)
    b. Edge template type imposes constraints on nodes.
    c. what if there are no nodes? Graph should be a trec of any, and should be overwritable when the first node comes in.
   Remember, edges have nodes in them. *)

(*Ensures actuals and their corresponding formals have compatible types. *)
and check_formals (aformals: (id * primitiveType) list) (allenv: allenv)  : unit =
(*   ignore(print_string("checking formals\n"));  *)
  let env, _,_,_ = allenv in
  List.iter(fun (id, t) -> let nt = find_in_map id env in ignore(check_compatible_types (nt,t))) aformals

(*Step 2 of HM: Collect constraints*)
and collect_expr (ae: aexpr) : (primitiveType * primitiveType) list =
  match ae with
  | AIntLit(_,_) | ABoolLit(_,_) | AStrLit(_,_) | AFloatLit(_,_)
  | ACharLit(_,_) | ARecord(_,_) | AGraph(_,_,_) | AId(_,_) -> []
  | AUnop(uop, ae1, t) ->
    let et1 = type_of ae1 in
    let opc = match uop with
    | Not -> [(et1, TBool); (t, TBool)]
    | Neg -> [(et1, TInt); (t, TInt)]
  in (collect_expr ae1) @ opc
  | ABinop(ae1, op, ae2, t) ->
    let et1 = type_of ae1 and et2 = type_of ae2 in
    let opc = match op with
      | Add | Mult | Sub | Div -> [(et1, TInt); (et2, TInt); (t, TInt)]
      (* we return et1, et2 since these are generic operators *)
      | Greater | Less | Equal | Geq | Leq | Neq -> check_compatible_types(et1, et2); [(t, TBool)]
      | And | Or -> [(et1, TBool); (et2, TBool); (t, TBool)]
      | Fadd | Fsub | Fmult | Fdiv -> [(et1, TFloat); (et2, TFloat); (t, TFloat)]
      | Ladd -> [(et1, TList(et2)); (t, TList(et2))]
      | In ->
      (match et2 with |TList(x) ->
              [(et1, x);
              (et2, TList(gen_new_type()));
              (t, TBool)]
                      | _ -> raise(failwith("Error @330")))
      | Gadd ->  [(t, et1)]
      | Eadd ->
       (match et1, et2 with |TGraph(name, n, e), TEdge(_,_,_) -> [(t, et1); (et2, e)]
                            | _ -> [(t, et1)])
      | _ -> raise(failwith("error"))
     in
    (collect_expr ae1) @ (collect_expr ae2) @ opc
  (*opc appended at the rightmost since we apply substitutions right to left *)
  | AEdge(ae1, op, ae2, ae3, t) ->
    let et1 = type_of ae1 and et2 = type_of ae2 and et3 = type_of ae3 in
    let opc = match op with
        | To | From | Dash ->
        (match et1, et2 with
        |TRec(_,_), TRec(_,_) -> ignore(check_compatible_types (et1,et2)); []
        | _ -> raise(failwith("error: " ^ string_of_aexpr ae1 ^ " and " ^ string_of_aexpr ae2 ^ " must be nodes."))
```

```
          )
          | _ -> raise(failwith((string_of_op op) ^ " not an edge operator."))
    in
    ignore(match et3 with
       | TRec(_,_) | T(_) -> ()
       | _ -> raise(failwith("error: " ^ string_of_aexpr ae3 ^ " not a record.")));
    (collect_expr ae1) @ (collect_expr ae2) @ opc @ (collect_expr ae3)
  | ADot(ae1, _, _) -> []
  | AItem(s, ae1, t) -> collect_expr ae1 @ [((type_of ae1), TInt)]
  | ACall(_, _, _, _, _)
  | ANoexpr(_) -> []
  | AList(ael, t) -> []

(*Step 3 of HM: unify constraints*)
and unify (constraints: (primitiveType * primitiveType) list) : substitutions =
  match constraints with
  | [] -> []
  | (x, y) :: xs ->
    (* generate substitutions of the rest of the list *)
    let t2 = unify xs in
    (* resolve the LHS and RHS of the constraints from the previous substitutions *)
    let t1 = unify_one (apply t2 x) (apply t2 y) in
    (*     ignore(print_string ("after unify one\n")); *)
    t1 @ t2
and unify_one (t1: primitiveType) (t2: primitiveType) : substitutions =
  match t1, t2 with
  | TInt, TInt | TBool, TBool | TString, TString | TFloat, TFloat | TVoid, TVoid -> []
  | T(x), z | z, T(x) -> [(x, z)]
  | TList(x), TList(y) -> unify_one x y
  | TGraph(name1, a, b), TGraph(name2, c, d) -> unify_one a c @ unify_one b d
  | TEdge(name1, n1, e1), TEdge(name2, n2, e2) ->
    unify_one name1 (TEdge(name2, n2, e2))
  | TRec(a, b), TRec(c, d) ->
    ignore(let fieldslists = List.combine b d in List.map (fun x -> check_field x) fieldslists);
    unify_one a c
  | _ -> raise (failwith "mismatched types@502")
and substitute (u: primitiveType) (x: id) (t: primitiveType) : primitiveType =
  match t with
  | TInt | TBool | TString | TFloat | TList(_) | TChar| TVoid-> t
  | T(c) | TRec(T(c),_)  | TEdge(T(c),_,_) | TGraph(T(c),_,_) -> if c = x then u else t
  | _ -> raise(failwith("error"))
and apply (subs: substitutions) (t: primitiveType) : primitiveType =
  List.fold_right (fun (x, u) t -> substitute u x t) subs t

(*Step 4: Final application of substitutions*)
and apply_expr (subs: substitutions) (ae: aexpr): aexpr =
  match ae with
  | ABoolLit(b, t) -> ABoolLit(b, apply subs t)
  | AIntLit(n, t) -> AIntLit(n, apply subs t)
  | AStrLit(s,t) -> AStrLit(s, apply subs t)
  | ACharLit(c,t) -> ACharLit(c, apply subs t)
  | AFloatLit(f, t) -> AFloatLit(f, apply subs t)
  | AId(s, t) -> AId(s, apply subs t)
  | AGraph(aelist, e, t) -> AGraph(apply_expr_list subs aelist, e, apply subs t) (*no apply on the edge template, right?*)
  | AList(e, t) -> AList(apply_expr_list subs e, apply subs t)
  | ABinop(e1, op, e2, t) -> ABinop(apply_expr subs e1, op, apply_expr subs e2, apply subs t)
  | AUnop(op, e1, t) -> AUnop(op, apply_expr subs e1, apply subs t)
  | ARecord(e1, t) -> ARecord(e1, apply subs t)
  | AItem(s, e1, t) -> let ae1 = apply_expr subs e1 in (* ignore(check_int ae1); *) AItem(s, ae1, apply subs t)
  | ACall(name, e, astmts, id, t) -> ACall(name, e, astmts, id, apply subs t)
  | ADot(id, entry, t) -> ADot(apply_expr subs id, entry, apply subs t) (*Am I handling
```

```
    this right?*)
  | AEdge(e1, op, e2, e3, t) -> AEdge(apply_expr subs e1, op, apply_expr subs e2, apply_expr subs e3, apply subs t)
  | ANoexpr(t) -> ANoexpr(t) (*is this okay?*)
and apply_expr_list (subs: substitutions) (ae: aexpr list)  : aexpr list =
  let rec helper (ae: aexpr list) (res: aexpr list) =
    match ae with
      [] -> List.rev res
    |h :: t ->  helper t (apply_expr subs h :: res)
  in helper ae []

(*Helper function for update map*)
and assign (ae: aexpr) (ae2: aexpr) (env: environment) : environment =
  let t = type_of ae2 in
  let env =
  match ae with
  |AId(str, _) -> NameMap.add (map_id str) t env
  |ADot(AId(_, TRec(T(recname), _)), str, _) -> NameMap.add (map_id (map_id_rec recname str)) t env
  |AItem(str, _, _) -> NameMap.add (map_id str) (TList(t)) env
  |_ -> raise(failwith("error: " ^ string_of_aexpr ae ^ " not a valid lvalue@534."))
in env

(*Updates environment*)
and update_map (allenv: allenv) (a: astmt) : allenv =
  let env, genv, recs, funcs = allenv in
  match a with
  |AAsn(ae1, ae2, _,_) ->
    let env, recs = (update_map_recs (type_of ae2) (env, recs)) in
    let env = assign ae1 ae2 env in
   env, genv, recs, funcs
  |_ -> allenv

(* get the template we generate from call. *)
and update_funcs (a: astmt) (funcs: funcs) (genv: genvironment) : funcs =
  match a with
  |AReturn(ae, _)
  |AExpr(ae)
  |AWhile(ae, _)
  |AAsn(_,ae, _,_) -> apply_update ae funcs genv
  |AIf(ae, s1, s2) -> let funcs = apply_update ae funcs genv in
                      let funcs = List.fold_left (fun a b -> update_funcs b a genv) funcs s1 in
                      List.fold_left (fun a b -> update_funcs b a genv) funcs s2
  |AFor(s1, ae, s2, s3s) ->
                      let funcs = update_funcs s1 funcs genv in
                      let funcs = apply_update ae funcs genv in
                      let funcs = update_funcs s2 funcs genv in
                      List.fold_left (fun a b -> update_funcs b a genv) funcs s3s
  |AForin(_,_,s1s) -> List.fold_left (fun a b -> update_funcs b a genv) funcs s1s
and apply_update (call: aexpr) (funcs: funcs) (genv: genvironment ) : funcs =
  match call with
  | AIntLit(_,_)| ABoolLit(_,_) | AFloatLit(_,_) | AStrLit(_,_) | ACharLit(_,_) | AId(_,_) -> funcs
  | AItem(_, e, _) | AUnop(_, e, _) | ADot(e,_,_) -> apply_update e funcs genv

  | ABinop(e1, _, e2, _) -> let funcs = apply_update e1 funcs genv in apply_update e2 funcs genv
  | AEdge(e1, _, e2, e3, _) -> let funcs = apply_update e1 funcs genv in
                               let funcs = apply_update e2 funcs genv in
                               apply_update e3 funcs genv
  | AList(elist, _)  -> List.fold_left (fun a b -> apply_update b a genv) funcs elist
  | AGraph(elist, e1, _) -> let funcs = List.fold_left (fun a b -> apply_update b a genv) funcs elist in
                            apply_update e1 funcs genv
```

```
  | ANoexpr(_) -> funcs
  |ACall(name, aelist, astmts, id, t) ->
   let funcs = List.fold_left (fun a b -> apply_update b a genv) funcs aelist in
   let (_, aformals, _) =
      if (NameMap.mem (name) genv)
      then (NameMap.find (name) genv)
      else (raise (failwith "function not defined @ 601")) in
  let flist = List.combine aformals aelist in
  let aformals = List.map format_formal flist in
  ((AFbody(AFdecl(id, aformals, t), astmts)) :: funcs)
  |_ -> funcs

 (*Used when an expression itself changes the environment, i.e, in records or calls
 that are secretly records. *)
 and update_map_recs (t: primitiveType) (env, recs: environment * recs) : environment *
     recs =
    (match t with
    |TRec(T(tname), elist) ->
      let rec helper l env =
      (match l with
      |[] -> env
      |(field, fieldtype) :: tail ->
      let env = NameMap.add (map_id (map_id_rec tname field)) fieldtype env in helper
    tail env
      )
      in
      let recs = if(NameMap.mem tname env)
      then(recs)
      else(t :: recs) in
      let env = helper elist env in
    (env, recs)
    |_ -> env, recs)

(*Returns the type for functions.*)
and grab_returns (r: astmt list) : primitiveType list =
  match r with
  | [] -> []
  | h :: tail ->
    (match h with
     |AReturn(_, t) ->
       t :: grab_returns tail
     |AIf(_, x, y) ->
       let ifs =  grab_returns x  @ grab_returns y in
       if(ifs != [])
       then(raise(failwith("error-- predicate return")))
       else(grab_returns tail)
     |AFor(_, _, _, y) ->
      let fors = grab_returns y in
      if(fors != [])
      then(raise(failwith("error-- predicate return")))
      else (grab_returns tail)
      |AForin(_, _, y) ->
      let fors = grab_returns y in
      if(fors != [])
      then(raise(failwith("error-- predicate return")))
      else (grab_returns tail)
     |AWhile(_, y) ->
      let whiles = grab_returns y in
      if(whiles != [])
      then(raise(failwith("error-- predicate return")))
      else (grab_returns tail)
     | _ -> grab_returns tail)
and get_return_type(r: astmt list) : primitiveType =
  let returns = grab_returns r in
  let rec find_type l : primitiveType =
```

```
    match l with
      [] -> TVoid     | [t] -> t
    | x :: y :: tail ->
      raise (failwith "Error: multiple returns.");
  in (find_type returns)

(* Infer formals from function statements. *)
and infer_formals (formals: string list) (env: environment): (string * primitiveType)
      list =
  let rec helper f env aformals =
  match f with
  |[] -> List.rev aformals
  | h :: tail ->
    let t = find_in_map h env in
    helper tail env ((h, t) :: aformals) in
  helper formals env []

(*The calling method for this file. Infers all types for a func (statements, formals),
    and
outputs an annotated func. *)
and infer_func (allenv: allenv) (f: func) : (afunc list * genvironment) =
(*   ignore(print_string("inferring new func\n")); *)
  let env, genv, recs, funcs = allenv in
  match f with
  |Fbody(decl, stmts) ->
    ignore(match decl with Fdecl(fname, _)-> Stack.push fname callstack); (*set scope*)
    let ((_,genv,_,funcs), istmts) = infer_stmt_list allenv stmts (*infer the function
    statments*)
    in let ret_type = get_return_type istmts
    in match decl with
    |Fdecl(fname, formals) ->          (*add function to NameMap*)
      if NameMap.mem fname genv
      then(
        let aformals = infer_formals formals env in
        let genv = NameMap.add fname (ret_type, aformals, stmts) genv in
        let allenv = env, genv, recs, funcs in
        let ((env, genv, recs, funcs), astmts) = infer_stmt_list allenv stmts in
        (ignore(Stack.pop callstack));
        let toss =  List.fold_left (fun hasany h -> match h with |(_,T(_)) -> true |_
    -> hasany) false aformals in
        let funcs =
        match ret_type with
        T(_) -> funcs
        |_ -> if(toss) then(funcs) else(AFbody(AFdecl(fname, aformals, ret_type),
    astmts) :: funcs)
      in funcs, genv)
      else raise (failwith "function not defined @ 412")

(*Credit to: https://github.com/prakhar1989/type-inference/blob/master/infer.ml*)
```

Listing 19: infer.ml

8. **display.c**
   Authors - Aashima Arora

```
/* Currently customized coordinate generation for petersen graph, displays petersen
    graph when fed with pete.gl.*/

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

#define PI 3.14159
```

```
#define _GNU_SOURCE
#define MAX 50
#define MAX_NODE_STORE 1000


typedef struct {
    int key;
}node;

typedef struct {
    int w;
}attr;

typedef struct {
    node* n1;
    node* n2;
    int directed;
    attr weight;
}edge;

typedef struct {
    node* node_list;
    int size;
}lst_node;

typedef struct {
    edge* edge_list;
    int size;
}lst_edge;

typedef struct {
    lst_node nodes;
    lst_edge edges;
    attr def_weight;
}graph;

typedef struct {
    int  key[MAX_NODE_STORE];
    int count;
}node_tbl;

node_tbl lookup;
typedef struct {
    int nodes[MAX];
    int to[MAX];
    int from[MAX];
    int weights[MAX];
    int num_nodes;
    int num_edges;
    int directed;
} Node_info;

int display_graph(Node_info* info, int directed)
{
    FILE* fp = fopen("pnts.dat","w");
    FILE* fe = fopen("edges.dat","w");
    float x, y ;



    for(int i = 0; i <5; i++)
    {
       x = (cos(PI/2 + ((2*PI)/5)*i));
       y = (sin(PI/2 + ((2*PI)/5)*i))    ;
```

```
        fprintf(fp,"%d\t%f\t%f\n",i,x,y);

    }
    for(int i = 5; i < info->num_nodes - 1 ; i++)
    {
       x =(0.5*cos(PI/2 + ((2*PI)/5)*i));
       y =(0.5*sin(PI/2 + ((2*PI)/5)*i));
       fprintf(fp,"%d\t%f\t%f\n",i,x,y);

    }

    for(int i = 0; i < info->num_edges; i++)
    {
        fprintf(fe,"%d\t%d\t%d\t%d\t%d\n", info->from[i] - 1,
        info->to[i] - 1,info->weights[i],-1,1);
    }

    fclose(fp);
    fclose(fe);
    if(directed)
        system("gnuplot gnuplot_dir.sh -persist");
   else
        system("gnuplot gnuplot.sh -persist");

   return 0 ;
}

int set_mapping_node_addr(node* n1, int size)
{
     int found = 0;
     for(int i = 0; i < size; i++)
     {
        found = 0;
        for(int j = 0; j < lookup.count; j++)
        {
            if(n1[i].key == lookup.key[j])
                found = 1;
        }

        if(!found)
            lookup.key[lookup.count++] = n1[i].key;
      }
      return 0;
}

int get_mapping_node_addr(node* n1) {
     int i = 0;
      for(i = 0; i < lookup.count; i++)
      {
          if(n1->key == lookup.key[i])
             return i;
      }
      return -1;
}

int fill_edge_info(int* to, int* from, int* weight, edge* edges, int size,int
    default_weight) {

   int directed = 0;
   for(int i = 0; i < size; i++) {

        to[i] = get_mapping_node_addr(edges[i].n1);
        from[i] = get_mapping_node_addr(edges[i].n2);
        weight[i] = edges[i].weight.w;
        if(weight[i] == 0)
```

```
            weight[i] = default_weight;
        if(edges[i].directed == 1)
            directed = 1;
    }

    return directed;
}

int display(graph g) {

    int directed = 0;
    node d_nodes[MAX];
    edge d_edges[MAX];
    memcpy(d_nodes,g.nodes.node_list,g.nodes.size*sizeof(node));
    memcpy(d_edges,g.edges.edge_list,g.edges.size*sizeof(edge));
    printf("EDGES - %d\n",g.edges.size);
    printf("NODES - %d\n",g.nodes.size);
    printf("DEFAULT WEIGHT %d\n",g.def_weight.w);
    Node_info n1;
    n1.num_nodes = g.nodes.size;
    n1.num_edges = g.edges.size;
    set_mapping_node_addr(d_nodes,g.nodes.size);
    directed = fill_edge_info(n1.to,n1.from,n1.weights,d_edges,g.edges.size,g.
    def_weight.w);

    /*
    for(int k = 0; k < g.nodes.size; k++) {
        printf("\n - node - %p, key - %d\n",&d_nodes[k],d_nodes[k].key);
     }
     */

    for(int k = 0; k < g.edges.size; k++) {
        printf("\nfrom key1 - %d -> to key2 - %d\n",
        d_edges[k].n1->key,d_edges[k].n2->key);
     }

    return display_graph(&n1,directed);
}
```

Listing 20: display.c

9. **gnuplot.sh**
   Authors - Aashima Arora

```
set xr [-2:2]
set yr [-2:2]

set size square
flePnts = 'pnts.dat'
fleEdges = 'edges.dat'

loadEdges = sprintf('< gawk '' \
    FNR==NR{x[$1]=$2;y[$1]=$3;next;} \
    {printf "%%f\t%%f\n%%f\t%%f\n\n", x[$1], y[$1], x[$2], y[$2];} \
'' %s %s', flePnts, fleEdges);


loadWeights = sprintf('< gawk '' \
    FNR==NR{x[$1]=$2;y[$1]=$3;next;} \
    {printf "%%f\t%%f\t%%s\n", (x[$1]+x[$2])/2 + $4, (y[$1]+y[$2])/2 + $5, $3} \
'' %s %s', flePnts, fleEdges);
plot \
    loadEdges using 1:2 with lines lc rgb "black" lw 2 notitle, \
    flePnts using 2:3:(0.1) with circles fill solid lc rgb "black" notitle, \
```

```
    flePnts using 2:3:1 with labels tc rgb "white" font "Arial Bold" notitle ,
```

Listing 21: gnuplot.sh

10. **gnuplot-dir.sh**
    Authors - Aashima Arora

```
set xr [0:50]
    set yr [0:50]

    set size square

    set style arrow 1 head filled size screen 0.025,10,40 lc rgb "black" lw 2

    flePnts = 'pnts.dat'
    fleEdges = 'edges.dat'

    loadEdges = sprintf('< gawk '' \
        FNR==NR{x[$1]=$2;y[$1]=$3;next;} \
        {printf "%%f\t%%f\t%%f\t%%f\n\n", x[$1], y[$1], (x[$2]-x[$1]), (y[$2]-y[$1]);}
    \
    '' %s %s', flePnts, fleEdges);

    loadWeights = sprintf('< gawk '' \
        FNR==NR{x[$1]=$2;y[$1]=$3;next;} \
        {printf "%%f\t%%f\t%%s\n", (x[$1]+x[$2])/2 + $4, (y[$1]+y[$2])/2 + $5, $3} \
    '' %s %s', flePnts, fleEdges);

    plot \
        loadEdges using 1:2:3:4 with vectors arrowstyle 1  notitle, \
        flePnts using 2:3:(0.6) with circles fill solid lc rgb "black" notitle, \
        flePnts using 2:3:1 with labels tc rgb "white" font "Arial Bold" notitle, \
        loadWeights using 1:2:3 with labels tc rgb "red" center font "Arial Bold"
    notitle
```

Listing 22: gnuplot-dir.sh

11. **make-ext.sh**
    Authors - Aashima Arora

```
if [ -n "$1" ]; then
   FILE="$1"
else
   echo -n "File name is a required argument. Enter a .gl file. "
   exit
fi
./grail.native <"$FILE" > out.ll &&
clang -o final out.ll ./external/disp.c -lm

mv final bin/
cd bin
./final
```

Listing 23: make-ext.sh

12. **run.sh**
    Authors - Riva Tropp

```
clear
ocamllex scanner.mll
ocamlyacc parser.mly
ocamlc -c  ast.ml
ocamlc -c astutils.ml
ocamlc -c parser.mli
```

```
ocamlc -c scanner.ml
ocamlc -c parser.ml
awk -f imode.awk > igrail.ml
awk -f idebug.awk $1 > infer2
mv infer.ml backupinfer.ml
mv infer2 infer.ml
ocamlc -c infer.ml
ocamlc -c igrail.ml
ocamlc -o grail parser.cmo scanner.cmo astutils.cmo infer.cmo igrail.cmo
rm igrail.ml
mv backupinfer.ml infer.ml
```

Listing 24: run.sh

13. **imode.awk**
    Authors - Riva Tropp

```
BEGIN{
    file = "grail.ml"

    while((getline < file) > 0){
  if(match($0, /(\(\*)(\s+let rec interpreter)/)){
      print gensub(/(\(\*)(\s+let rec interpreter)/, "\\2", "g")
  }
  else if(match($0, /.*display l \*\)/))
      print gensub(/(.*display l)( \*\))/, "\\1", "g")

  else if(match($0, /let compile/))
      print "(* " $0
  else if(match($0, /compile\(\);/))
      print $0 " *)"
  else
      print $0
    }
    if(close(file))
  print file " failed to close"
}
```

Listing 25: imode.awk

14. **idebug.awk**
    Authors - Riva Tropp

```
BEGIN{
    file = "infer.ml"

    while((getline < file) > 0){
        if(ARGV[1] == "d" && match($0, /(\(\*)(\s+ignore\(print_string.*)(\*\))/))
            print gensub(/(\(\*)(\s+ignore\(print_string.*)(\*\))/, "\\2", "g")

        else
            print $0
    }
    if(close(file))
        print file " failed to close"
}
```

Listing 26: idebug.awk

15. **testall.sh**
    Authors - Jiaxin Su

```
#!/bin/sh
#Run testcases under dir /tests
```

```
#make clean
#make

# Path to the LLVM interpreter
# Riva's path
 LLI="/usr/bin/lli"
 LLL="/usr/bin/llvm-link"
# Jiaxin's path
# LLI="/usr/local/opt/llvm/bin/lli"
# LLL="/usr/local/opt/llvm/bin/llvm-link"

# coloring notes
# success = green
# warning or err = red
# help or neutral things = yellow
NC='\033[0m'
YELLOW='\033[1;33m'
GREEN='\033[0;32m'
RED='\033[0;31m'

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

# Set time limit for all operations
ulimit -t 30

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

keep=0

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

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

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

# Run <args>
# Report the command, run it, and report any errors
Run() {
```

```
    echo $* 1>&2
    eval $* || {
        SignalError "$1 failed on $*"
        return 1
    }
}

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

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

    echo -n "$basename..."

    echo 1>&2
    echo "${YELLOW} ###### Testing $basename ${NC}" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
  #  Run "clang -emit-llvm -o list.bc -c src/list.c" &&
    Run "$GRAIL" "<" $1 ">" "${basename}.ll" &&
    Run "$LLL" "${basename}.ll" "-o" "a.out" &&
    chmod +x a.out &&
    Run "$LLI" "a.out" ">" "${basename}.out"&&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
        if [ $keep -eq 0 ] ; then
            mv ${basename}.out ./test_output/
      mv ${basename}.ll ./test_output/
            mv ${basename}.diff ./test_output/
      rm -f $generatedfiles
        fi
        echo "${GREEN}OK ${NC}"
        echo "${GREEN} ###### SUCCESS ${NC}" 1>&2
    else
        echo "${RED} ###### FAILED ${NC}" 1>&2
        mv ${basename}.out ./test_output/
        mv ${basename}.ll ./test_output/
        mv ${basename}.diff ./test_output/
        globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.gl//'`
    reffile=`echo $1 | sed 's/.gl$//'`
```

```
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "${YELLOW} ###### Testing $basename ${NC}" 1>&2

    generatedfiles=""

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

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
        if [ $keep -eq 0 ] ; then
            rm -f $generatedfiles
        fi
        echo "${GREEN}OK ${NC}"
        echo "${GREEN} ###### SUCCESS ${NC}" 1>&2
    else
        echo "${RED} ###### FAILED ${NC}" 1>&2
        mv ${basename}.err ./test_output/
        mv ${basename}.diff ./test_output/
        globalerror=$error
    fi
}

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

shift `expr $OPTIND - 1`

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

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

mkdir test_output

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

for file in $files
do
    case $file in
        *test-*)
            Check $file 2>> $globallog
            ;;
```

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

# cat testall.log

exit $globalerror
```

Listing 27: testall.sh