n2n:
A Relational Graphing Language

Nicholas Falba (nrf2118), Jialun Liu (jl4347),
Elisheva Aeder (ea2621), Johan Mena (jmm2371)

December 17, 2014

Contents

Chapter 1

Introduction

In todays world there is an escalating interest in relationships. Be it the connection of people on a social networking platform, of family members in a family tree, of variables in a mathematical equation, or the connection of trains in a city subway system, we are constantly trying to find networks of people and things and analyze how they interrelate. Most often, relationships are implemented in a programming setting via the graph data structure containing a set of nodes and edges defining connections between the nodes. In standard programming languages, however, graphs can be tedious to create and manipulate, requiring the creation of separate immutable classes for nodes and edges, and they tend not to focus on the possibility of having various types of nodes and relationships within one graphs structure. n2n is a language designed with the connected graph as the primary data structure providing a higher level of abstraction to perform common operations on connected graph data structures with a specific focus on how nodes of a graph interrelate, and the types of relationships they have.

1.1 Language Goals

Graphing in many languages can be difficult and tedious with regards to creating nodes, relating them, adding or removing data from them, and finding existing relationships between nodes. n2n was designed to be an intuitive language focused on programming directed connected graph data structures, not only making it easier to perform basic operations on the graph, but also providing the ability to specify types of nodes and relationships that can model information stored in graphs in real settings.

1.2 Overview

The primary components of an n2n program are Graph objects and the Nodes and Relationships they contain. Node and Relationship (Rel) constructors are named and can hold a set of primitive data fields which must be specified when an literal of that Node or Rel type is instantiated. Associations are made between nodes in a graph by adding Node1-Relationship-Node2 tuples to a graph, which simultaneously adds both the nodes and the relationship to the graph. Inserting and removing data from a Node or Rel, like inserting and removing Node-Rel-Node tuples from a graph, is done with ease, as is performing searches and other graphing algorithms with the instantiated graphs.

Chapter 2

Language Description

An n2n graph is a collection of Node-Relationship-Node tuples. Node and relationship constructors are defined when their name and the data types they contain are specified. Once their types are created, Node-Rel-Node tuples can be added to a graph representing a relationship from the first Node to the second. The Nodes and Rels of the tuple are either named instances of the newly created Node and Rel constructors (ids), or unnamed literals of some Node or Rel constructor with the required data fields specified as defined by their constructors. The two node literals in the tuple do not have to be of the same type. It is possible to have more than one Relationship from Node A to Node B, and to have a different Relationship connecting Node B to Node A than exists from A to B.

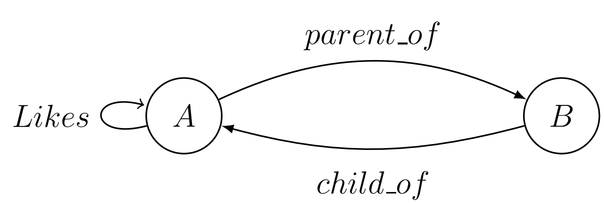


Figure 2.1: Simple Graph Example where one Relationship connects Node A to Node B

2.1 A More Descriptive Example

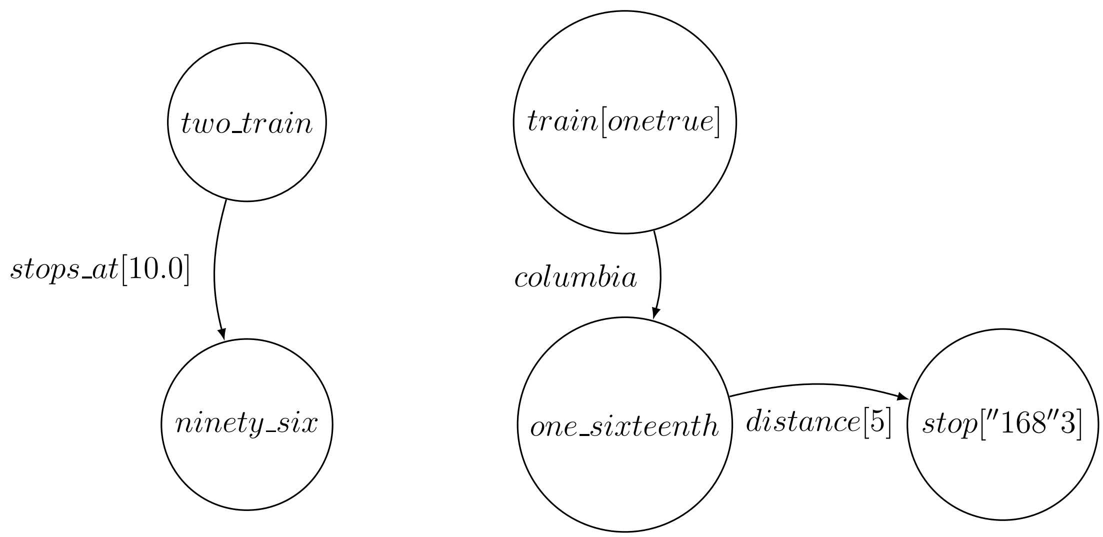
To make things clearer, consider the graph of the NYC subway system, specifically the red line (with 1, 2, and 3 trains). In the example below, we create a graph called *red_line*, Node constructors *subway_stop*, and *train*, and Rel constructors *stops_at*, and *distance*. The *subway_stop* Node type holds two data fields: a string that defines the name of a *subway_stop*, and an integer representing the number of different trains that stop there. When a new Node of type *subway_stop* is created such as the id named *one_sixteenth*, values for its string and integer data fields must be defined. When the *red_line* graph is created, the arguments of its constructor are Node-Rel-Node tuples which are either Node ids that have been instantiated already (as in the case with *one_sixteenth*), or a Node or Rel literal, defined within the Graph constructor by setting its primitive data fields within brackets right there! (This is done in the example when the *subway_stop*, *ninety_six* is inserted into the graph.)

```
subway_stop: Node = {name: String, num_trains_stop_here
    : Int}
train: Node = {line: String, local: Bool}
stops_at: Rel = {stop_rate: Double}
distance: Rel = {dist_by_num_stops: Int}

one_sixteenth: Node = subway_stop[''one_sixteenth'', 1]
two_train: Node = train[''two'', false]
columbia: Rel = stops_at[7.0]

red_line: Graph = <
        one_sixteenth distance[ 5 ] subway_stop[``
            one_sixty_eight'', 3]
        two_train stops_at[ 10.0 ] subway_stop[``
            ninety_six'', 4]
        train[``one'',true] columbia one_sixteenth
>
```

Functions can then be defined to manipulate and perform graph algorithms on the defined Graphs, similar to the way functions are defined in languages such as C or Java. As n2n is specific to operating with Graphs, it supports basic binary and unary operations, but does not support features such as loops which are unnecessary in n2n. Graph algorithms that



would require looping in other languages, can be done more simply using n2ns specialized functions.

2.2 Built in Functions

One attribute of n2n that eases the manipulation of its Graph types is having built in functions that are native to the language. There are four built in functions:

1. A print function that takes in any valid expression or expression list as arguments and prints the value of the expression. For example,

```
print(node_or_rel_instance)
```

   prints the type of the node instance and the values of the data fields it holds.

```
print(Graph)
```

   prints the Node and Rel literals contained in the Graph
   The print function will also print primitive data types as it does in Java.

2. A find_many function searches the graph for a specified pattern defined in the function arguments and returns a list of Nodes or Relationships. Arguments that the find_many function accepts are:

   (a) (node_literal) returns a List of Nodes in the graph that match this node_literal with its data fields set to specified values.

   (b) (Node, Rel) searches the graph for Nodes matching the passed Node id or literal that have a Rel (id or literal) relationship with one or more other Nodes in the graph. It returns a List of the Nodes on the other side of these relationships.

   (c) (Rel, Node) does the inverse in that it searches the graph for Nodes being pointed at by the passed Rel and returns the Nodes that point to them.

   (d) (node, node) returns a List of Rels that connect the first Node id or literal with the second.

3. A *map* function is called by a List or Graph of Node ids or constructors and accepts a set of statements as arguments. The function performs the statements on each Node in the List or Graph, and returns the resulting Graph or List (the Graph or List which is the result of performing the statements on the elements on the instances in the list).

4. A *neighbors* function is called by a Graph instance and accepts one Node instance as an argument. It returns a List of Nodes that argument has a Relationship towards within that caller graph (as in he has a relationship directed from himself to the these other nodes).

2.3 Compiling and Running an n2n Program

In the src file with the Makefile and the source code, run *make all* which compiles all n2n source files. Use the ./ n2n operator to run your program as follows:

```
$ ./ n2n -[option] source\_files.n2n
```

This will compile the n2n code to Java and run the program.
Options for compiling n2n code:

1. -a for printing the AST

2. -s for running semantic analysis on the source code
3. -c is for compiling the whole program
4. -j is for compiling and running the java code of the program

2.4 Sample Code

This is a small program that tests the find_many function. It first creates two global Node types or constructors. Then within the main method it creates Node and Rel literals and inserts Node-Rel-Node tuples into the graph. It then calls the find_many function which finds all *actor* Nodes whose name data field is set to "Keanu" and whose age could be anything.

```
actor: Node = { name: String, age: Int };
actedIn: Rel = { role: String };
movie: Node = { title: String, year: Int };

fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        Leo: Node = actor["Leo", 20];

        neo: Rel = actedIn["Neo"];
        jordan: Rel = actedIn["Jordan"];
        nelson: Rel = actedIn["Nelson"];

        matrix: Node = movie["Matrix", 1999];
        wolf: Node = movie["Wolf", 1994];
        sweet_nov: Node = movie["Sweet November",
           2000];

        Cast: Graph = <
                Keanu neo matrix,
                Leo jordan wolf,
                Keanu nelson sweet_nov
        >;

```

```
node_lit: List<Node>;
node_lit = Cast.find_many(actor["Keanu", _]);

print(node_lit);
print("\n");
```

This program outputs:

```
[actor{ name = Keanu, age = 35 }]
```

which is a list of the Nodes of the graph whose name is Keanu.

Chapter 3

Language Reference Manual

N2n enables the easy creation and manipulation of directed, connected graph types. Graphs, Nodes, and Relationships (Rel) are native data types in the language. Node and Rel types may enclose a set of primitive types. The language includes graph operators to add and remove Node-Rel-Node tuples from the graph, and element operators that allow the addition or subtraction of data fields from Node and Rel instances. Functions native to the language allow for easy searching and manipulating of the graph data structures. A programmer can create and manipulate the graphs while n2n handles their storage under the hood.

3.1 Lexical Conventions

An n2n program, written with the Unicode character set, consists of global variables and global functions that can be used in a main function from which the program is run. Its functions contain various types of tokens such as expression operators, keywords, built-in function names, identifiers, and other separators. All identifiers, keywords, and functions names are case sensitive. In general blanks, tabs, and comments are ignored except as they serve to separate tokens. Semi-colons indicate the end of a statement.

3.1.1 Comments

Comments are introduced and terminated by two semi-colons For example:

```
;; This is a comment ;;
```

3.1.2 Identifiers

An identifier is a sequence of letters and digits with no spaces between them. The first character must be alphabetic. Underscore characters may be included in the identifier.

3.1.3 Keywords

The following are reserved for use as keywords and may not be used otherwise:

Int	Graph	Void	true	find_many
Double	Node	if	false	main
String	Rel	else	map	return
Bool	List	fn	neighbors	

3.1.4 Constants

An integer constant is a sequence of digits that are assumed to be in base 10. A double constant represents a fractional number. It consists of an integer part, a decimal point, and a fractional part. Either the integer part or the fraction part but not both may be missing. Both are comprised of a series of digits. Boolean constants may hold only the values true or false. String constants are one or more characters surrounded by double quotes. Special characters may be escaped within a string the same as they can be escaped in Java.

3.2 Identifiers, Overloading, and Scoping. Oh my!

The type of the identifier determines the meaning of the values in the identifiers storage. The location and lifetime of an identifier are determined by the scope in which it is declared. A function sets a scope for a variable so that any variable declared in a function only exists within that function and are discarded on return. Any variable declared outside functions are global and visible by the entire program, independent of any function. If a local variable shares its name with a global variable, the local variable (within the function) has precedence.

3.3 Data Types

3.3.1 Primitive types

Primitive types include Int, Double, Bool, and Strings. The type Void can be used as a return type for functions.

3.3.2 Complex types

Complex types include Graph, Node, Rel, and List

1. Node literals are the elements stored in graphs. A Node constructor define a Node type and is created when its name and the primitive data fields it encapsulates are defined. When a Node literal is created (which may or may not have an id), it must provide values for the data fields defined in its Node constructor.

2. Rel is a directed, named relationship between Nodes. There can be a different Rel between Node1 and Node2 than exists between Node2 and Node1. A Rel type is declared and defined like Nodes with a name and primitive data fields associated with the Rel.

3. Graph is a collection of Node-Rel-Node tuples. It can be thought of as a list of relationships, and the nodes that the names relationships connect.

4. List maintains an ordered collection of elements

3.4 Declarations

3.4.1 Variable Declaration and Assignment

The type of the identifier determines the meaning of the values in the identifiers storage. Thus the type must be explicitly specified when declaring an identifier as follows:

Declaration

An identifier name, followed by a colon, its type and possibly followed by an assignment via the assignment operator (=).

```
var_name : type
```

Or

```
var_name : type = expr
```

(where expr here refers to some expression being assigned to the identifier).

3.4.2 Assignment

Once an identifier has been declared, to assign or reassign its value simply use its name followed by the assignment operator and an expression. For example:

```
var_name = expr
```

More specifically:

1. Nodes and Rel: There are two types of Node/Rel declarations:

   (a) The declaration of a Node constructor
   To define a new Node or Rel node type, declare an identifier as above by specifying its name followed by a colon and its type, the assign character, = and the declaration of data fields within curly braces. For example:

   ```
   node_constr1: Node = {s: String, b: Bool}
   ```

   (b) The declaration of Node literal.
   A Node literal can be declared explicitly by assigning it to an id, or instantiated within Graph objects. When declared explicitly, the expression following Node objects assignments is surrounded by square brackets in which the fields of data objects associated with that node are set. For example:

   ```
   node_id_of_type1: Node = node_type1[''my_name''
       , true]
   ```

2. Graph: To create a graph, declare its name and type (Graph) as above. Following the assignment operator there is a list of zero or more Node-Rel-Node tuples surrounded by <and >. The Nodes and Rels in the

tuples can either be ids representing pre-existing literals or definitions of new Node/Rel literals whose constructors are called and data fields defined within the creation of this graph. For example:

```
graph1: Graph = <
        keanu    acted_in[``John'']    movie[``
           Constantine'', 2003],
        actor[``Keanu''] acted_in[``Neo'']     movie
           [``Matrix'', 1999]
>
```

This creates a graph, *graph1*, with two Node-Rel-Node relationships. The first has has a Node id *keanu* which was defined before the declaration of *graph1*, as well as a Node literal of type movie whose String and Int data fields are given values within this instantiation of *graph1*.

3.4.3 Function Declaration

A function is declared by specifying the name, parameters, and return type of the function as follows:

```
fn func_name ( parameter-list ) -> return_type {
   statements }
```

The parameters are formatted as follows:

```
name:type
```

and are separated by commas. The body of the function is a list of statements that may include new variable declarations and function calls. Functions may be recursive. Note that functions create scope so that a variable declared within a function with the same name as a global variable defined outside of a function will take precedence over the global within that function. (More in section Scope) There must be a function called main in an n2n program from which the program is run

3.5 Expressions

1. Identifier: An identifier is an expression provided it has been suitably declared.

2. Primitive Literals of types Int, Double, Bool, and String are also expressions.

3. Complex Literals of types Graph, Node, Rel, or List

4. Binary Operators include Additive and Multiplicative operators as enumerated below

5. Unary Operators

6. Graph Operators

7. Node and Rel Operators

8. Access: To access a primitive data field of a Node or Rel instance use the dot operator as in:

   ```
   node1.named_attribute
   ```

   node1 must be an existing Node id that has a primitive data field with the name *named_attribute*. The access expression may be followed by the assignment operator (=) in order to change the value of the primitive data type held in the Node or Rel. For example:

   ```
   keanu.visited = true
   ```

   updates the "visited" field in the *keanu* node to be true

9. Call
   To call an existing n2n function, type the name of the function followed by the function arguments (if any) within parentheses. For example:

   ```
   dijkstra(graph1);
   ```

10. Built-in function calls including the *print*, *find_many*, *neighbors*, *map* functions

3.6 Operators

3.6.1 Unary Operators

-*expression* results in the negative of the expression
! expression results in true if the value of the expression is false and true if the expression value is false

3.6.2 Multiplicative Operators

```
Expression * expression
```

The * operator indicates multiplication. If both operands are of type Int the result is Int. Same with Double. If one is Int and the other Double, the result is Double.
/indicates division. The same type considerations as for multiplication apply.
The % operator indicates the remainder from the division of the first expression by the second. Both operands must be of type Int.

3.6.3 Additive Operators

expression + *expression*: The result is the sum of the expressions. The same type considerations as for multiplication apply.
expression - *expression*: The result it the difference of the operands.

3.6.4 Relational Operators

e <e and e >e and e <= e and e >=e all yield false if the specified relation is false and true if it is true. These operators can only be applied for types Int and Double.
Same for the equality operators: == (equal to) and != (not equal to). *expression* && *expression* returns true if both expressions are true and false otherwise
Expression ’—”—’ *expression* returns true if either expression is true and false otherwise

3.6.5 Graph and Node Operators

Node-Rel-Node tuples may be added or removed from a Graph using the ˆ+ and ˆ- operators. For example:

```
Graph_name ^+ node1 rel1 node2
```

The format of the Node-Rel-Node tuple is the same as in Graph instantiation (above). To add or remove a data field from a Node or Rel instance, use the [+] and [-] operators respectively. The data field should have the form name: type

```
node_example [+] visited: Bool
```

3.7 Built in Functions

One attribute of n2n that eases the manipulation of its Graph types is having built in functions that are native to the language. There are four built in functions:

1. A print function that takes in any valid expression or expression list as arguments and prints the value of the expression. For example,

   ```
   print(node_or_rel_instance)
   ```

   prints the type of the node instance and the values of the data fields it holds.

   ```
   print(Graph)
   ```

   prints the Node and Rel literals contained in the Graph
   The print function will also print primitive data types as it does in Java.

2. A find_many function searches the graph for a specified pattern defined in the function arguments and returns a list of Nodes or Relationships. Arguments that the find_many function accepts are:

   (a) (node_literal) returns a List of Nodes in the graph that match this node_literal with its data fields set to some value. The underscore character is used to match again any value.

For example, a graph with an actor Node type or constructor may contain multiple actor literals whose data fields are set differently. In this example consider an actor that contains types name:String and age:Int. To find all actor Nodes whose first argument, *name*, is set to "John" and whose second argument *age* is set to 25:

```
john_actors: List<Node> = find_many(actor[''
    John'',25]);
```

To pattern match against fewer fields of the Node literal, simply replace a data field with an underscore. For example, to find all *actors* named John regardless of their age:

```
john_actors: List<Node> = find_many(actor[''
    John'',_]);
```

Pattern matching against all underscores will return all Nodes of type *actor*.

(b) (Node, Rel) searches the graph for Nodes matching the passed Node id or literal that have a Rel (id or literal) relationship with one or more other Nodes in the graph. It returns a List of the Nodes on the other side of these relationships. For example, in a graph of *movie* and *actor* constructors, to find movie Nodes for which *keanu* has an *acted_in* relationship with:

```
keanu_movies: List<Node> = find_many(keanu
    acted_in)
```

This will search the graph for matches on a node *keanu* that has a relationship of type *acted_in* and return a List of the *movie* Nodes on the other side of the relationship.

(c) (Rel, Node) does the inverse in that it searches the graph for Nodes being pointed at by the passed Rel and returns the Nodes that point to them. For example:

```
neo_actors: List<Node> = find_many(acted_in[''
    Neo''] matrix)
```

Finds Nodes that have a relationship of *acted_in* with its data field set to "Neo" in Node matrix. Here too, data fields can be replaced with underscores to match against zero or more data fields of the Node or Rel literals.

(d) (node, node) returns a List of Rels that connect the first Node id or literal with the second. For example: Both

```
connection: List<Node> = find_many(actor['Neo'']
    ]_matrix)
```

returns all Rels between *actors* whose name field is set to "Neo" and the Node id named *matrix* and

```
relationship: List<Node> = find_many(movie[\_]
    actor[\_])
```

returns all Rels between any *movie* Node and any *actor* Node.

3. A *map* function is called by a List or Graph of Node ids or constructors and accepts a set of statements as arguments. The function performs the statements on each Node in the List or Graph, and returns the resulting Graph or List (the Graph or List which is the result of performing the statements on the elements on the instances in the list). For example:

```
mutated_graph: Graph = Cast.map(node in {node [+]
    visited:Bool;});
```

returns a Graph called *mutated_graph* which adds a boolean field called *visited* to each Node in graph *Cast*.

4. A *neighbors* function is called by a Graph instance and accepts one Node instance as an argument. It returns a List of Nodes that argument has a Relationship towards within that caller graph (as in he has a relationship directed from himself to the these other nodes).

3.8 Statements

A statement is anything that can make up a line of n2n code.

3.8.1 Statement Blocks

A block of code is any list of statements surrounded by curly braces as shown:

```
{
        statement;
        statement;
        ...
}
```

3.8.2 Expressions

Any of the named expressions given above are expression statements if followed by a semicolon.

3.8.3 Conditional statements

Conditional statements are composed of two parts: a required if; and an optional else part.

1. if (expression) {

    statements

   }

2. if (expresion) {

    statements

   } else {

    statements

   }

In the first case, the compiler will execute *statements* if the result of evaluating *expression* is the boolean value of true.

The second case is like the first but if the result of evaluating *expression* is the boolean value of false, it will evaluate the *statements* inside the else part of the conditional.

3.8.4 Return Statements

To return a function to its caller use the keyword *return* as follows:

> return expression

This will return the result of evaluating expression.

3.8.5 Variable Declarations

The declaration of a variable is a statement that can exist in an n2n program either globally or within functions definitions.

Chapter 4

From Plan to Finished Product

4.1 Planning

The planning of our project consisted of coming up with a language idea, determining team responsibilities, and setting a timeline for the execution of our language compiler. We had weekly, then biweekly in person meetings to plan, discuss, and execute our project. Over the course of the semester many aspects of our initial plan changed. Although we initially agreed upon a language and its specifications, our initial model was worked and reworked and expanded and reconfigured many times throughout the semester as the project progressed.

4.2 Development

At the beginning of each project stage, each team member took on a portion of the project and worked on it both individually and with other team members. For many parts of the project our designated roles overlapped; we worked together, we corrected each others mistakes, and redesignated tasks, though often our task assignments overlapped. It would be difficult to outline each team members responsibilities as most pieces of our compiler were touched by almost all of our hands. We met once, and, sometimes, twice every week to discuss issues that we had with the code, to formulate new decisions about the project, and to keep one another up to date with new developments. We also used git with Github for our version control so that each team member had access to our entire code base at all times.

4.3 Team Responsibilities

The initial responsibilities designated to each team member were generally not binding, as each team member took on various roles as our project progressed throughout the semester. We were each involved with multiple parts of the project, which are delineated below:

Team Member	Roles
Nicholas Falba	Parser, Semantic Check, SAST
Johan Mena	Compiler Executable, Java Backend, Testing, Git Ninja
Jialun Liu	Scanner, Parser, AST, Testing
Elisheva Aeder	AST, SAST, Code Gen, Final Report

4.4 Project Timeline

The following was the initial timeline the we set out for our project: The

Date	Accomplishment
September 10	First Meeting
September 24	Project Proposal
October 20	Scanner, Parser, AST Completed
October 27	Language Reference Manual
November 24	Semantic Check Completed
December 1	Code Generation Completed
December 10	Testing Completed
December 17	Final Report and Presentation

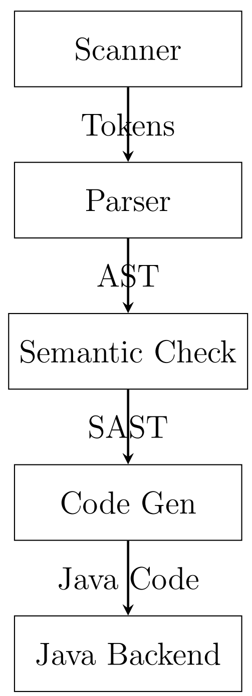
actual flow of events was not as smooth as the above. While coding each new part of the compiler, previously completed sections had to be revised and updated based on new overall language updates and newly discovered errors. Most aspects of the compiler were not fully complete until the final stages of testing.

4.5 Development Environment

N2n was developed in the Mac OS X environment. The compiler was written in OCaml, using ocamlc to compile .ml files, ocamllex for the scanner, and ocamlyacc for the parser. The generated Java code is compiled using the java compiler. The compiling process was automated with Makefiles. Testing and verification was run with shell scripts. Github was used to host our git repository.

Chapter 5

Architectural Design



The n2n compiler takes a .n2n file as input into the scanner which outputs tokens. It then gets passed to the parser which outputs an AST. It passes through the semantic check and produces an SAST and then the code generation outputs Java source code which is compiled and run. The code generation includes a Java library which allows for the functions required for Graph, Node, and Rel creation and manipulation.

5.1 Scanner

The scanner takes in a N2N source file and outputs tokens according to the lexical rules of our language.

5.2 Parser

The parser takes in the tokens from the scanner and uses our grammatical rules to decide if the input are in the language that is specified by our context free grammar. During parsing, it produces an abstract syntax tree that defines the structure of the given program.

5.3 Abstract Syntax Tree

The AST defines the structure of the N2N language, including various types of structures, for example, primitive and complex types, builtin function calls, user-defined functions and the program itself.

5.4 Semantic Check

The semantic check receives an abstract syntax tree as an argument and prints a semantically checked tree in a similar format. For the AST functions, their semantically checked counterparts (SAST functions), hold more information. In the case of expressions, this is useful for type checking. And in the case of Graph Elements (Nodes and Relationships), we keep track of element type, identifiers, and all the field information for each Node or Rel across the semantic check to assist in our back-end methods. To keep track of which identifiers exist and the types for them, we store this information in a list in the environment. The types are segregated in the environment based on what needs to be stored there. Nodes and relationships each have their own section, since the types are so similar in structure but used differently. This is to avoid confusion when checking, since theyre stored in separate lists. Graphs are stored in their own area since you store Node, Rel, Node triples in them, in addition to an Id. Lists are stored separately since they are the only type you cant mutate directly. And the rest of the types (Int, Bool, String, and Double) have their own table since they all need to keep similar information. And there are two sets of tables, one for the global and one for the local scope, as different variables will be visible depending on the scope. Moreover, functions are defined and added to the environment as well as their return type, so you can check to make sure a function is returning what it is supposed to. Variables and functions are added to the variable tables when they are instantiated and defined, respectively. The semantic check makes sure that only certain expressions appear in places that are allowed by the syntax and semantics of the language, and makes sure the output of individual expressions and statements is what is expected by the language. The output is a set of semantically checked functions in tree format used by our code generating function to output java code.

5.5 Code Generation (code_gen.ml)

The code_gen takes the semantically checked abstract syntax tree generated from the semantic check and then converts if to the java code. Here we match types of expressions with primitive types(Int, Double, String, Bool) and complex types(Node, Rel, Graph, List); if we find a primitive type we just insert the java equivalent syntax, if a complex type we then need to insert the code according to methods we defined in our java backend.

5.6 Java Backend

(Graph.java Node.java Relationship.java) The backend has three infrastructure classes:

5.6.1 Relationship

Relationships are the glue that holds all the nodes that are connected to it.

5.6.2 Node

Node is the class that holds the data for each one of the members of the Graph. Nodes and relationships also keep track of the data they contain.

5.6.3 Graph

Graph is the main class, and is used to store the set of relationships that connect all of the nodes in the graph. It also provides overloaded finder methods and functionality for adding, removing or printing out Relationships and Nodes. Most methods were implemented using Java 8's stream api, for readability and performance. There are also unit tests for most of the finder methods, to ensure reliability of the functionality.

5.7 Command Line Interface (n2n.ml)

The N2N command line is the file that puts all of these modules together in succession. It opens the source program that has been passed to it and runs it through the scanner, parser, semantic check and code generation to produce a java program and finally compiles the program using the java. Our compiler supports several different functionalities, for instance, printing the AST of the program, running semantic check over the source program, generating the java program that then be compiled.

Chapter 6

Test Plan

6.1 Development of the Test Plan

Our test plan consists of a test suite of comprehensive tests meant to check all of the functionalities of a n2n program, one script to test each stage of the compiler, e.g. printing the AST, running semantic check over the source program and generating the java code from the source program. We also have an automated test script that runs over all of the test cases and checks them against the expected output files. The test suite and scripts were primarily authored by Nicholas Ray Falba, Johan M. Mena and Jialun Liu.

6.2 Test Suites

Our test suite is listed in the table below. These test cases were chosen because they represent the smallest building blocks of a n2n program, making it possible for us to test each functionality individually and easily identify the location of bugs. There are also some more complicated tests which allow for testing multiple functionalities of the program integrated together.

6.3 Component Testing

```
./test.sh [option] source_files.n2n
```

File	Functionality Tested
arithmetic1.n2n	Checks basic arithmetic, adding two numbers and printing the result
arithmetic2.n2n	Checks basic arithmetic with a number of operations
comparison1.n2n	Checks comparison between two numbers and prints correct Bool output
comparison2.n2n	Checks equality operations on numbers and prints the correct Bool output
declaration.n2n	Checks local and global variable declaration and scope visibility
escapestr1.n2n	Checks to make sure that new line escape strings work
escapestr2.n2n	Checks to make sure that tab escape strings work
escapestr3.n2n	Checks non-main function call to print escape string
escapestr5.n2n	Same as above except different escape string
float.n2n	Checks output of integer division
float2.n2n	Checks output of same expression above except for Doubles
helloworld.n2n	Checks print of "Hello World"
init1.n2n	Chekcs to see if we can declare and initialize at the same time
functions1.n2n	Checks function call
functions2.n2n	Testing assignment to the output of a function call
functions3.n2n	Testing nested function calls
find_many1.n2n	Testing find_many for (Node, Node) case
find_many2.n2n	Testing find_many for (Node, Rel) case
find_many3.n2n	Testing find_many for (Rel, Node) case
find_many4.n2n	Testing find_many for (Node_literal) case
graph.n2n	Testing graph instantiation for Node and Rel variables
graph2.n2n	Testing graph instantiation for Node and Rel literals
graphInsDel.n2n	Testing graph insert and remove after instantiation
if.n2n	Testing if statements and dangling else problem
map.n2n	Test built-in function map
neighbor.n2n	Testing basic neighbor function call
node.n2n	Testing node declaration and instantiation
nodeAccess.n2n	Testing access on field of a Node
nodeInsDel.n2n	Testing Data_InsertRemove operators
rel.n2n	Testing rel instantiation and initialization
relAccess.n2n	Testing access on field of a Rel
simple.n2n	Testing multiple global variable declaration
simple2.n2n	Testing basic local variable assignment

test.sh - semantic check – outputs semantic check trace - all methods of semantic check that have been visited for a program and if there's error it prints error message and if no error it prints "passed." The option is either semantic_check in which case the above program is run and the output trace sent to a file at a given location if there is an error. Or you could run it with *ast* as an argument in which case it just runs the print function of the ast.

```
#!/bin/bash

rm -f ../test/sem_output/*.out
echo

arg_array=( $@ )
test_instruction=${arg_array[0]}
test_message=""
test_output=""
for (( i = 1; i < $#; i++))
do
if [[ $test_instruction == "ast" ]]
then
        test_message="Printing ast for ${arg_array[$i
            ]}: "
        test_output=$(./n2n -a ${arg_array[i]} 2>&1)
elif [[ $test_instruction == "semantic_check" ]]
then
        test_message="Checking semantic_check for ${
            arg_array[$i]}: "
        test_output=$(./n2n -s ${arg_array[$i]} 2>&1)
elif [[ $test_instruction == "java" ]]
then
        test_message="Generating java code for ${
            arg_array[$i]}: "
        if [[ ${arg_array[$i+1]} == *".java" ]]
        then
                test_output=$(./n2n -j ${arg_array[$i]}
                    ${arg_array[$++i]} 2>&1)
        else
                test_output=$(./n2n -j ${arg_array[$i]}
                    2>&1)
        fi
```

```
fi
if [[ $test_output == *"Parse_error"* ]]
then
        echo $test_message"Failed Parsing"
        echo
elif [[ $test_instruction != "ast" && $test_output !=
   *"Passed Semantic Analysis"* ]]
then
        filename=${arg_array[$i]}
        back_filename=${filename##*/}
        echo $test_message"Failed Semantic Check. 
           Output in file ../test/sem_output/${
           back_filename%.*}.out"
        echo
        printf "$test_output" > ../test/sem_output/${
           back_filename%.*}.out
else
        if [[ $test_instruction == "ast" ]]
        then
        echo $test_message"Passed."
        echo
        printf "$test_output\n"
        echo
        else
        echo $test_message"Passed."
        echo
        fi
fi
done
```

6.4 Automation

We use an automated test script that takes .n2n files, compiles them into Java code, runs the programs, and compares their outputs with the expected outputs. It prints ok if the tests are correct, and returns the difference if not. The automated test script is run via the command:

```
./test_all.sh [source files]
```

And this is its code:

```
#!/bin/bash

COMPILER="../n2n -c"
TESTFILES=$@

rm errors.out

for f in $TESTFILES
do
        LEN=$((${#f}-4))
        OUTFILENAME="${f:0:$LEN}.output"
        TESTFILENAME="${f:0:$LEN}.out"
  echo -n "Test $f: "
        $COMPILER $f > $OUTFILENAME 2>> errors.out

  DIFF=$(diff -w $OUTFILENAME $TESTFILENAME)
  if [ "$DIFF" == "" ]
        then
                echo " OK"
        else
                echo " FAIL"
    echo "=========="
    echo "Expected:"
    echo "$(cat ${TESTFILENAME})"
    echo "=========="
    echo "But was:"
    echo "$(cat ${OUTFILENAME})"
        fi

  rm -f $OUTFILENAME
done
```

6.5 n2n to Java

6.5.1 DFS Search Algorithm

The following n2n program performs a DFS search on a graph and prints all Nodes in the graph:

```
member: Node = {Name: String, Age: Int, visited: Bool};
relation: Rel = {Relation: String};

fn DFS(N: Node, M: Graph)->Int {
        if(N.visited == false){
                print(N);
                print("\n");
                Neighbor: List<Node> = M.neighbors(N);
                N.visited = true;
                Neighbor.map(node in { DFS(node, M); })
                   ;
        }
        return 0;
}

fn main()->Void {
        John: Node = member["John Hamilton", 49, false
           ];
        Mary: Node = member["Mary Lance", 47, false];
        Johan: Node = member["Johan Hamilton", 21,
           false];
        Sara: Node = member["Sara Hamilton", 20, false
           ];

        Mother: Rel = relation["Mother of"];
        Father: Rel = relation["Father of"];
        Child: Rel = relation["Child of"];
        Brother: Rel = relation["Brother of"];
        Sister: Rel = relation["Sister of"];

        Family: Graph = <
                John Father Johan,
```

```
                John Father Sara,
                Mary Mother Johan,
                Mary Mother Sara,
                Johan Brother Sara,
                Johan Child John,
                Johan Child Mary,
                Sara Sister Johan,
                Sara Child John,
                Sara Child Mary
        >;

        DFS(Johan, Family);
}
```

The output of this n2n program looks like:

```
member{ visited = false, Age = 21, Name = Johan
   Hamilton }
member{ visited = false, Age = 47, Name = Mary Lance }
member{ visited = false, Age = 20, Name = Sara Hamilton
    }
member{ visited = false, Age = 49, Name = John Hamilton
    }
```

The Java code that this program compiles to is:

```
package com.n2n;

import java.util.*;

class Main {

        String relation = "relation";
String member = "member";
;
public static void main(String[] args) {

                Node John = new Node("member", new
                   HashMap<String, Object>() {{
        put("Name", "John Hamilton");
```

```
put("Age", 49);
put("visited", false);

}}
);;
        Node Mary = new Node("member", new HashMap<
           String, Object>() {{
        put("Name", "Mary Lance");
put("Age", 47);
put("visited", false);

}}
);;
        Node Johan = new Node("member", new HashMap<
           String, Object>() {{
        put("Name", "Johan Hamilton");
put("Age", 21);
put("visited", false);

}}
);;
        Node Sara = new Node("member", new HashMap<
           String, Object>() {{
        put("Name", "Sara Hamilton");
put("Age", 20);
put("visited", false);

}}
);;
        Relationship Mother = new Relationship("
           relation", new HashMap<String, Object>() {{
        put("Relation", "Mother of");

}}
);;

      Relationship Father = new Relationship("relation"
         , new HashMap<String, Object>() {{
```

```
        put("Relation", "Father␣of");

}}
);;
        Relationship Child = new Relationship("relation
           ", new HashMap<String, Object>() {{
        put("Relation", "Child␣of");

}}
);;
        Relationship Brother = new Relationship("
           relation", new HashMap<String, Object>() {{
        put("Relation", "Brother␣of");

}}
);;
        Relationship Sister = new Relationship("
           relation", new HashMap<String, Object>() {{
        put("Relation", "Sister␣of");

}}
);;
        Graph Family = new Graph(Arrays.asList(new
           Graph.Member<>(John, Father, Johan), new
           Graph.Member<>(John, Father, Sara), new
           Graph.Member<>(Mary, Mother, Johan), new
           Graph.Member<>(Mary, Mother, Sara), new
           Graph.Member<>(Johan, Brother, Sara), new
           Graph.Member<>(Johan, Child, John), new
           Graph.Member<>(Johan, Child, Mary), new
           Graph.Member<>(Sara, Sister, Johan), new
           Graph.Member<>(Sara, Child, John), new Graph
           .Member<>(Sara, Child, Mary)));;
        DFS(Johan, Family);
        }
public static int DFS(Node N, Graph M) {
if(N.getValueFor("visited").equals(false)) {
                System.out.print(N);
```

```
            System.out.print("\n");
            Set<Node> Neighbor = M.neighbors(N);
            N.getData().put("visited", true);
            for( Node node : Neighbor ){
DFS(node, M);
            }

            }
            else {
            }

return 0;
            }

}
```

This does not include the Java backend files which are included below in the appendix.

Chapter 7

Lessons Learned

7.1 Nicholas Falba

Its a clich to say, start earlier, but its true. Specifically, get your front-end working as early as possible. And once you have a solid front-end (Parser, Scanner, AST), writing the semantic check will be way simpler and you should start on that too. This way you can get a feel for whats going to work and what isnt early on. This will give you a head start and crucial time management information in implementing the full check.
Also, practice Ocaml. Even a little bit. Learning by doing is how I wrote the semantic check. You dont want to do that. Learn beforehand.
Also resolve group differences early or actually make your group run like a dictatorship. Very early on, if you can agree on a syntax, itll go a long way. Dont wait until a month in to voice your concerns.
Also learn all about git. We only had one person in our group who knew git and it caused a bunch of minor roadblocks, but these could and should have been easily avoided.

7.2 Jialun Liu

This course is very interesting and I have definitely learned a lot. The most important thing is really to think early and start early, meaning that we should carefully consider what the syntax and rules we are going to use for our language. Finding out some errors in parser and insufficiency of the rule defined are going to a disaster in the later stage of the project, it makes us change everything in the flow of the compiler and it makes us really suffered. Setting up the flow of the entire compiler is another important issue, we should have set it up at the very early stage so that it would thing clear about what we are really doing and figure out bugs a lot easier. Testing a very important issue of every project, as the tester, I didnt do well in this part, I should have conducted the tests as we moving along different parts of the compiler. Also, shell scripts for automated tests is a very useful thing it saved us a lot of effort of going through and debugging the compiler. Also, make some intuitive comment in the semantic check would be very helpful when the source program fails to go through the semantic check. Also, Ocaml is a very powerful language, all the pattern matching and function nesting make the language extremely useful. It would be a good start to get yourself familiar with Ocaml since that will save quite a lot a effort when writing semantic check in the later stage of the project. And it would definitely be a good idea to spend more time to play around with that after the project. In terms of project management, we had a great time working together, but we should have made a clear deadline for each part of the project so that it wouldnt become the case that we have to do a whole bunch of things in the last two weeks of the semester.

7.3 Johan Mena

Don't pay attention to the lessons of othersit worked for them but it might not work for you. Pick your teammates carefully. Or maybe just at random.

7.4 Elisheva Aeder

Learn Ocaml! And fast! One of my biggest struggles throughout the project was my lack of fluency in Ocaml. Spending more time on it in earlier stages would have been helpful. Also, presentations and reports take a lot longer than you'd think. Start earlier. Assign a leader and let them lead. Or have better defined roles.

Chapter 8

Appendix

8.1 Scanner

```
{
  open Parser
  exception LexError of string

  let verify_escape s =
  if String.length s = 1 then (String.get s 0)
  else
  match s with
     "\\n" -> '\n'
   | "\\t" -> '\t'
   (*| "\\" -> '\'*)
   | "\"" ->  '"'
   (*| "\'"␣->␣␣'''*)
   | c -> raise (Failure("unsupported character " ^ c))
}

(* Regular Definitions *)

let digit = ['0'-'9']
let decimal = ((digit+ '.' digit*) | ('.' digit+))

(* Regular Rules *)
```

```
rule token = parse
    | [' ' '\n' '\t']          { token lexbuf }
    | ';'                      { TERMINATION } (*
       terminate the code*)
    | ";;"                     { block_comment lexbuf }
       (* block comment*)

    | '('                      { LPAREN }
    | ')'                      { RPAREN }
    | '{'                      { LBRACE }
    | '}'                      { RBRACE }
    | ']'                      { RBRACKET }
    | '['                      { LBRACKET }

    | '+'                      { PLUS }
    | '-'                      { MINUS }
    | '*'                      { TIMES }
    | '/'                      { DIVIDE }
    | '%'                      { MOD }

(*  | ';'                        { SEMI }*)
    | ':'                      { COLON }
    | ","                      { COMMA }
    | '='                      { ASSIGN }
    | "->"                     { ARROW }
    | "^"                      { CONCAT }
    | "."                      { ACCESS }

    | "=="                     { EQ }
    | "!="                     { NEQ }
    | '<'                      { LT }
    | "<="                     { LEQ }
    | ">"                      { GT }
    | ">="                     { GEQ }
    | "!"                      { NOT }
    | "&&"                     { AND }
    | "||"                     { OR }
```

```
    | "if"                          { IF }
    | "else"                        { ELSE }

(*    | "elif"                        { ELIF }*)
    | "map"                         { MAP }
    | "find_many"                   { FINDMANY }

    | "fn"                          { FUNCTION}
    | "return"                      { RETURN }

    | "in"                          { IN }

    | "^+"                          { GRAPH_INSERT }
    | "^-"                          { GRAPH_REMOVE }
    | "[+]"                         { DATA_INSERT }
    | "[-]"                         { DATA_REMOVE }
    | "neighbors"                   { NEIGHBORS }

    | "Graph"                       { GRAPH }
    | "Rel"                         { REL }
    | "Node"                        { NODE }
    | "List"                        { LIST }
    | "Int"                         { INT }
    | "Double"                      { DOUBLE }
    | "String"                      { STRING }
    | "Bool"                        { BOOL }
    | "Void"                        { VOID }
    | '_'                           { ANY }

 | digit+ as lit
                                                                   {
     INT_LITERAL(int_of_string lit) }
 | decimal as lit
                                                                   {
     DOUBLE_LITERAL(float_of_string lit) }
 | '"' ([^ '"']* as lit) '"'
                                              { STRING_LITERAL(
```

```
    lit) }
  | ("true" | "false") as lit
                                                    { BOOL_LITERAL(
    bool_of_string lit) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as
    lit { ID(lit) }  (*every ID should start with a
    letter*)
  | eof { EOF }
  | _ as char { raise (Failure("illegal character " ^
    Char.escaped char)) }

and block_comment = parse
   ";;" { token lexbuf }
| eof  { raise (LexError("unterminated block_comment!")
   ) }
| _    { block_comment lexbuf }

(*
  | "new_node"

                  { NEW_NODE }
  | "new_rel"

                  { NEW_REL }
  | "add_field"

       { ADD_FIELD}
*)
```

8.2 Parser

```
%{
   open Ast
%}

```

```
%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET
   ANY
%token TERMINATION COMMA ASSIGN COLON ARROW CONCAT
   ACCESS
%token PLUS MINUS TIMES DIVIDE MOD
%token EQ NEQ LT LEQ GT GEQ AND OR NOT
%token IF ELSE
%token MAP IN FINDMANY
%token FUNCTION RETURN
%token GRAPH_INSERT GRAPH_REMOVE DATA_INSERT
   DATA_REMOVE NEIGHBORS
%token GRAPH REL NODE INT DOUBLE STRING BOOL VOID LIST
%token <int> INT_LITERAL
%token <string> STRING_LITERAL ID
%token <float> DOUBLE_LITERAL
%token <bool> BOOL_LITERAL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right GRAPH_INSERT GRAPH_REMOVE DATA_INSERT
   DATA_REMOVE
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left CONCAT
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left NEG NOT

%start program
%type <Ast.program> program

%%

program:
 | /* nothing */                    { ([], []) }
```

```
 | program global_var_declaration { ($2 :: fst $1), snd
     $1 }
 | program function_declaration   { fst $1, ($2 :: snd
    $1) }

var_declaration:
  | ID COLON n2n_type
                                                    { Var($3,
     $1) } /* foo: String */
  | ID COLON n2n_type ASSIGN LBRACE formal_list RBRACE
        { Constructor($3, $1, List.rev $6)} /* movie:
     Node = { title: String, year: Int } */
  | expr ASSIGN expr
                                                        {
     Access_Assign($1, $3) } /* foo: String = "lolomg",
      matrix: Node = movie[  Matrix  , 1999] */
  | ID COLON n2n_type ASSIGN expr
                              { Var_Decl_Assign($1, $3,
      $5)} /* Split the declaration and initializatio
     for complex */

global_var_declaration:
  var_declaration TERMINATION { $1 }

n2n_type:
  | INT                  { Int }
  | STRING               { String }
  | DOUBLE               { Double }
  | BOOL                 { Bool }
  | GRAPH                { Graph }
  | NODE                 { Node }
  | REL                  { Rel }
  | LIST LT n2n_type GT { List($3)}
  | VOID                 { Void }

function_declaration:
 | FUNCTION ID LPAREN formal_parameters RPAREN ARROW
    n2n_type LBRACE statements RBRACE /* fn foo (bar:
```

```
    Int) -> Bool { ... } */
     {
       { fname = $2;
         formals = $4;
         body = List.rev $9;
         return_type = $7;
       }
     }

formal_parameters:
  | /* nothing */ { [] }
  | formal_list    { List.rev $1 }

formal_list:
  | parameter                   { [$1] }      /* foo:
     Int */
  | formal_list COMMA parameter { $3 :: $1 } /* foo:
     Int, bar: String */

parameter:
  | ID COLON n2n_type     { Formal($3, $1) } /* foo: Int
      */

statements:
  | /* nothing */         { [] }
  | statements statement { $2 :: $1 }

statement:
  | expr TERMINATION
                                                      { Expr($1
     ) } /* 1 + 2 */
  | RETURN expr TERMINATION
                                               { Return($2) }
     /* return 1 + 2 */
  | LBRACE statements RBRACE
                                              { Block(List.rev
     $2) } /* { 1 + 2 \n 3 + 4 } */
```

```
  | IF LPAREN expr RPAREN statement %prec NOELSE
                  { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN statement ELSE statement
                { If($3, $5, $7) }
  | var_declaration TERMINATION
                                  { Var_Decl($1) } /*
     actor: Node, number: Int, graph_example: Graph */

expr:
  | literal                      { Literal($1) } /* 42,
      "Jerry", 4.3, true */
  | complex_literal              { Complex($1) } /*
     constructor(literal,literal), { node rel node,
     node_literal rel_literal node_literal } */
  | binary_operation             { $1 } /* 4 + 3, "
     Johan" ^ "Mena" */
  | graph_operation              { $1 }
  | graph_element_operation      { $1 }
  | unary_operation              { $1 } /* -1 */
  | ID                           { Id($1) } /* actor,
     number, graph_example */
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) } /*
     fucntion_ID_String_param("Keanu") */
  | built_in_function_call       { Func($1) }
  | LPAREN expr RPAREN           { $2 } /* (4 + 6) */
  | ID ACCESS ID                 { Access($1, $3)}

literal:
  | INT_LITERAL                  { Int_Literal($1) } /*
      4, 3, 27 */
  | STRING_LITERAL               { String_Literal($1) }
      /* "Me", "You", "Bill Clinton" */
  | DOUBLE_LITERAL               { Double_Literal($1) }
      /* 4.2, 3.7, 7.4 */
  | BOOL_LITERAL                 { Bool_Literal($1) }
     /* true, false */
  | ANY                          { Any }
```

```
complex_literal:
  | ID LBRACKET literal_opt RBRACKET    { Graph_Element
     ($1, $3) } /* actor("Keanu"), "true" */
  | LT graph_components GT              { Graph_Literal
     ($2) } /* < graph_literals > */

graph_component:
  graph_type graph_type graph_type       {
     Node_Rel_Node_Tup($1, $2, $3) } /* :: or commas?
     */

graph_components:
  |                           { [] }
  | graph_component_list      { List.rev $1 }

graph_component_list:
  | graph_component                              { [$1] }
  | graph_component_list COMMA graph_component { $3 ::
     $1 }

graph_type:
  | ID                              { Graph_Id($1) }
  | complex_literal                 { Graph_Type($1) }

literal_opt:
  |                 { [] }
  | literal_list  { List.rev $1 }

literal_list:
  | literal                         { [$1] }
  | literal_list COMMA literal    { $3 :: $1 }

actuals_opt:
  | /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
  | expr                        { [$1] }
```

```
  | actuals_list COMMA expr { $3 :: $1 }

binary_operation:
  | expr PLUS   expr

    { Binop($1, Add,    $3) }
  | expr MINUS  expr

    { Binop($1, Sub,    $3) }
  | expr TIMES  expr

    { Binop($1, Mult,   $3) }
  | expr DIVIDE expr

    { Binop($1, Div,    $3) }
  | expr MOD    expr

    { Binop($1, Mod,    $3) }
  | expr EQ     expr

    { Binop($1, Equal,  $3) }
  | expr NEQ    expr

    { Binop($1, Neq,    $3) }
  | expr LT     expr

    { Binop($1, Less,   $3) }
  | expr LEQ    expr

    { Binop($1, Leq,    $3) }
  | expr GT     expr

    { Binop($1, Greater,  $3) }
  | expr GEQ    expr

    { Binop($1, Geq,    $3) }
  | expr AND    expr
```

```
    { Binop($1, And, $3) }
  | expr OR      expr

    { Binop($1, Or, $3) }
  | expr CONCAT expr

    { Binop($1, Concat, $3) }

graph_operation:
  | expr GRAPH_INSERT LPAREN graph_component RPAREN
                              { Grop($1, Graph_Insert, $4
    )}/* ^+ */
  | expr GRAPH_REMOVE LPAREN graph_component RPAREN
                              { Grop($1, Graph_Remove, $4
    )}/* ^- */

graph_element_operation:
  | expr DATA_INSERT parameter
                                              { Geop($1,
    Data_Insert, $3) }/* [+] */
  | expr DATA_REMOVE parameter
                                              { Geop($1,
    Data_Remove, $3) }/* [-] */

unary_operation:
  | NOT expr                        { Unop(Not, $2) }
  | MINUS expr %prec NEG            { Unop(Neg, $2) }

built_in_function_call:
  | ID ACCESS find_many                          { Find_Many(
    $1, $3) } /* graph_example.find_many(...)*/
  | ID ACCESS map_function                       { Map($1, $3
    ) } /* graph_or_list_example.map(...) */
  | ID ACCESS neighbors_function                 {
    Neighbors_Func($1, $3) } /* graph_example.
    neighbors(node_ID) */

map_function:
```

```
  | MAP LPAREN ID IN LBRACE statements RBRACE RPAREN {
     Map_Func($3, $6) }  /* map(node in {...}) */

neighbors_function:
  | NEIGHBORS LPAREN ID RPAREN { $3 }

find_many:
  | FINDMANY LPAREN complex_literal RPAREN
                      { Find_Many_Node($3) } /* Find
     all nodes that match a literal, i.e. find_many(
     actor("Neo")) returns all nodes of type actor that
      have the name field equal to "Neo" */
  | FINDMANY LPAREN graph_type COMMA graph_type RPAREN
          { Find_Many_Gen($3, $5) } /* Return what's␣
     missing,␣i.e.␣nodes␣pointed␣to,␣nodes␣pointed␣from
     ,␣or␣rel␣between␣nodes␣*/
```

8.3 AST

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

type grop =
```

```
  | Graph_Insert
  | Graph_Remove

type geop =
  | Data_Insert
  | Data_Remove

type uop =
  | Not
  | Neg

type n2n_type =
  | Int
  | String
  | Bool
  | Double
  | Graph
  | Node
  | Rel
  | List of n2n_type
  | Void

type formal =
  | Formal of n2n_type * string

type var_decl =
  | Var of n2n_type * string
  | Constructor of n2n_type * string * formal list
  | Var_Decl_Assign of string * n2n_type * expr
  | Access_Assign of expr * expr

and expr =
  | Literal of literal
  | Complex of complex_literal
  | Id of string
  | Binop of expr * op * expr
  | Grop of expr * grop * graph_component
  | Geop of expr * geop * formal
```

```
  | Unop of uop * expr
  | Access of string * string
  | Call of string * expr list
  | Func of built_in_function_call

and literal =
  | Int_Literal of int
  | Double_Literal of float
  | String_Literal of string
  | Bool_Literal of bool
  | Any

and built_in_function_call =
  | Find_Many of string * find_many
  | Map of string * map_function
  | Neighbors_Func of string * string

and complex_literal =
  | Graph_Literal of graph_component list
  | Graph_Element of string * literal list

and map_function =
  | Map_Func of string * statement list

and find_many =
  | Find_Many_Node of complex_literal
  | Find_Many_Gen of graph_type * graph_type

and graph_component =
  | Node_Rel_Node_Tup of graph_type * graph_type *
    graph_type

and graph_type =
  | Graph_Id of string
  | Graph_Type of complex_literal

and statement =
  | Block of statement list
```

```
  | Expr of expr
  | Return of expr
  | If of expr * statement * statement
  | Var_Decl of var_decl

type func_decl = {
  fname : string;
  formals : formal list;
  body : statement list;
  return_type : n2n_type;
}

type program = var_decl list * func_decl list

let string_of_binop = function
  | Add          -> "+"
  | Sub          -> "-"
  | Mult         -> "*"
  | Div          -> "/"
  | Mod          -> "mod"
  | Equal        -> "=="
  | Neq          -> "!="
  | Less         -> "<"
  | Leq          -> "<="
  | Greater      -> ">"
  | Geq          -> ">="
  | And          -> "&&"
  | Or           -> "||"
  | Concat       -> "^"

let string_of_grop = function
  | Graph_Insert -> "[+]"
  | Graph_Remove -> "[-]"

let string_of_geop = function
  | Data_Insert  -> "^+"
  | Data_Remove  -> "^-"
```

```
let string_of_unop = function
  | Neg -> "-"
  | Not -> "!"

let rec string_of_n2n_type = function
  | Int          -> "Int"
  | String       -> "String"
  | Bool         -> "Bool"
  | Double       -> "Double"
  | Graph        -> "Graph"
  | Node         -> "Node"
  | Rel          -> "Rel"
  | List(t)      -> "List<" ^ string_of_n2n_type t ^ ">"
  | Void         -> "Void"

let string_of_formal = function
  | Formal(the_type, id) -> id ^ " : " ^
     string_of_n2n_type the_type

let string_of_literal = function
  | Int_Literal(l)    -> string_of_int l
  | Double_Literal(l) -> string_of_float l
  | String_Literal(l) -> "\"" ^ l ^ "\""
  | Bool_Literal(l)   -> string_of_bool l
  | Any               -> "_"

let rec string_of_expr = function
  | Literal(l) -> string_of_literal l
  | Complex(c) -> string_of_complex_literal c
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      string_of_binop o ^ " " ^
      string_of_expr e2
  | Grop(e1, grop, e2) ->
      string_of_expr e1 ^ " " ^
      string_of_grop grop ^ " " ^
      string_of_graph_component e2
  | Geop(e1, geop, e2) ->
```

```
      string_of_expr e1 ^ " " ^
      string_of_geop geop ^ " " ^
      string_of_formal e2
  | Unop(o, e) ->
      string_of_unop o ^ " " ^
      string_of_expr e
  | Id(s)      -> s
  | Access(id1, id2) -> id1 ^ "." ^ id2
  | Call(func_id, actuals) ->
      func_id ^ "(" ^ String.concat ", " (List.map
         string_of_expr actuals) ^ ")"
  | Func(k) -> string_of_built_in_fdecl k

and string_of_var_decl = function
 | Var(the_type, id) -> id ^ " : " ^ string_of_n2n_type
     the_type
 | Constructor(the_type, id, formal_list) ->
      id ^ " : " ^ string_of_n2n_type the_type ^ " = {
         " ^ String.concat ", " (List.map
         string_of_formal formal_list) ^ "}"
 | Var_Decl_Assign(id, the_type, expr) ->
      id ^ " : " ^ string_of_n2n_type the_type ^ " = "
         ^ string_of_expr expr
 | Access_Assign(e1, e2) ->
      string_of_expr e1 ^ " = " ^ string_of_expr e2

and string_of_complex_literal = function
  | Graph_Literal(graph_type_l) -> "( " ^ String.concat
      "," (List.map string_of_graph_component
     graph_type_l) ^ " )"
       (*Modification needed*)
  | Graph_Element(id, el) ->
      id ^ "[" ^ String.concat ", " (List.map
         string_of_literal el) ^ "]"

and string_of_graph_component = function
  | Node_Rel_Node_Tup(n1, r1, n2) ->
```

```
      string_of_graph_type n1 ^ " " ^
         string_of_graph_type r1 ^ " " ^
         string_of_graph_type n2

and string_of_built_in_fdecl = function
  | Find_Many(graph_id, s) -> graph_id ^ "." ^
     string_of_find_many s
  | Map(l, s) -> l ^ "." ^ string_of_map s
  | Neighbors_Func(l, node) ->
      l ^ ".neighbors (" ^ node ^ ")"

and string_of_find_many = function
  | Find_Many_Node(graph_type) ->
      "find_many (" ^ string_of_complex_literal
         graph_type ^ ")"
  | Find_Many_Gen(graph_type1, graph_type2) ->
      "find_many (" ^ string_of_graph_type graph_type1
         ^ "," ^ string_of_graph_type graph_type2 ^ ")"

and string_of_map = function
  | Map_Func(element_id, s) ->
      "map (" ^ element_id ^ " in " ^ " { " ^ String.
         concat "\n" (List.map string_of_statement s) ^
          " } )"

and string_of_graph_type = function
  | Graph_Id(id)                -> id
  | Graph_Type(graph_element) ->
     string_of_complex_literal graph_element

and string_of_statement = function
    Block(l) -> "{" ^ String.concat "\n" (List.map
       string_of_statement l) ^ "}"
  | Expr(l) -> string_of_expr l ^ "\n"
  | Return(l) -> "return" ^ string_of_expr l ^ "\n"
  | If(e, l , p) ->
    (match p with
```

```
        Block([]) -> "if ( " ^ string_of_expr e ^ " )\n
            " ^ string_of_statement l
      | _ -> "if ( " ^ string_of_expr e ^ " )\n" ^
          string_of_statement l ^ "else\n " ^
          string_of_statement p )
  | Var_Decl(v) -> string_of_var_decl v

(*****************************************************************************



let string_of_func_decl fdecl=
  "fn" ^ (fdecl.fname) ^ "(" ^ String.concat ", " (List
     .map string_of_formal fdecl.formals)
  ^ ") -> " ^ (string_of_n2n_type fdecl.return_type) ^
     "{" ^ String.concat "\n" (List.map
     string_of_statement fdecl.body) ^ "}"

let string_of_program (vars, funcs) =
  String.concat "\n" (List.map string_of_var_decl (List
     .rev vars)) ^ (if (List.length vars) > 0 then "\n"
      else "") ^
  String.concat "\n" (List.map string_of_func_decl (
     List.rev funcs))
```

8.4 Semantic Check

```
open Ast
open Sast

exception Error of string;;

type environment = {
functions: func_decl list;
scope: string;
node_types: (string * formal list) list;
rel_types: (string * formal list) list;
```

```
locals: var_scope;
globals: var_scope;
has_return: bool;
return_val: expr;
return_type: n2n_type;
}

and var_scope = {
prims: (string * n2n_type * expr) list;
nodes: (string * string * (string * n2n_type * expr)
   list) list; (* Form (id, node_type, field storage
   list) *)
rels: (string * string * (string * n2n_type * expr)
   list) list; (* Form (id, node_type, field storage
   list) *)
graphs: (string * graph_component list) list;
lists: (string * n2n_type * expr list) list (*Form (
   list_id, type, list contents) *)
}

let beginning_scope = { prims = []; nodes = []; rels =
   []; graphs = []; lists = [] }

let beginning_environment = { functions = []; globals =
    beginning_scope; locals = beginning_scope; scope =
   "global"; node_types = []; rel_types = [];
   has_return = false; return_val = Id("None");
   return_type = Void}

(*For debugging *)

let rec print_type ty =
match ty with
Node -> "Node"
| Rel -> "Rel"
| Graph -> "Graph"
| String -> "String"
| Int -> "Int"
```

```
| Double -> "Double"
| Bool -> "Bool"
| Void -> "Void"
| List(t) -> "List<" ^ print_type t ^ ">"

(* End debugging *)

let get_id_from_expr ex =
match ex with
Id(v) -> v
| _ -> raise(Error("Trying to get id from a non-id
   expression\n"))

let add_tuple_to_list env t =
match t with
(id,ty,fl) ->
let (list_to_update, location) = if List.exists (fun (
   fid,_,_) -> fid=id) env.locals.nodes then (env.
   locals.nodes, "nl")
else if List.exists (fun (fid,_,_) -> fid=id) env.
   locals.rels then (env.locals.rels, "rl")
else if List.exists (fun (fid,_,_) -> fid=id) env.
   globals.nodes then (env.globals.nodes, "ng")
else (env.globals.rels, "rg") in
let (new_list, location) =
let updated_list = List.fold_left (fun l (vid,vty,vfl)
   -> if vid=id then (id,ty,fl)::l else (vid,vty,vfl)::
   l ) [] list_to_update in
(List.rev updated_list, location) in
match location with
"nl" -> let new_vt = {env.locals with nodes = new_list}
    in
        {env with locals = new_vt}
| "rl" -> let new_vt = {env.locals with rels = new_list
   } in
        {env with locals = new_vt}
| "ng" -> let new_vt = {env.globals with nodes =
   new_list} in
```

```
        {env with globals = new_vt}
| "rg" -> let new_vt = {env.globals with rels =
   new_list} in
        {env with globals = new_vt}
| _ -> raise(Error("??"))

let check_arithmetic_binary_op t1 t2 =
match (t1, t2) with
| (Int, Int) -> Int
| (Int, Double) -> Double
| (Double, Int) -> Double
| (Double, Double) -> Double
| (_,_) -> raise(Error("Binary operation fails, wrong
   element type"))

let check_equality t1 t2 =
prerr_string(print_type t1 ^ " " ^print_type t2 ^"\n");
if t1 = t2 then Bool else
match (t1, t2) with
| (Int, Double) -> Bool
| (Double, Int) -> Bool
| (_, _) -> raise(Error("Equality operation fails,
   arguments not same type"))

let check_logic t1 t2 =
match(t1, t2) with
| (Int, Int) -> Bool
| (Int, Double) -> Bool
| (Double, Int) -> Bool
| (Double, Double) -> Bool
| (String, String) -> Bool
| (_,_) -> raise(Error("Logical operation fails,
   arguments not of correct types"))

let get_literal_type l = match l with
Int_Literal(i) -> Int
| Double_Literal(d) -> Double
| String_Literal(s) -> String
```

```
| Bool_Literal(b) -> Bool
| _ -> raise(Error("Should not be using Any here"))

let get_type_from_id var_table id =
if List.exists (fun (lid,_,_) -> lid=id) var_table.
   lists then
(let (_,ty,_) = List.find(fun (lid,_,_) -> lid=id)
   var_table.lists in List(ty))
else if List.exists (fun (nid,_,_) -> nid=id) var_table
   .nodes then Node
else if List.exists (fun (rid,_,_) -> rid=id) var_table
   .rels then Rel
else if List.exists (fun (gid, _) -> gid=id) var_table.
   graphs then Graph
else let (_,ty,_) = try List.find (fun (vid, _, _) ->
   vid=id) var_table.prims with
Not_found -> raise Not_found in ty

let check_for_var_existence var_table id =
(List.exists (fun (gid, _) -> gid=id) var_table.graphs
|| List.exists(fun (rid,_,_) -> rid=id) var_table.rels
|| List.exists(fun (nid,_,_) -> nid=id) var_table.nodes
|| List.exists(fun (vid,_,_) -> vid=id) var_table.prims
|| List.exists(fun (lid,_,_) -> lid=id) var_table.lists
   )

let check_if_id_is_node env id =
if List.exists (fun (nid, _, _ ) -> prerr_string("
   Looking for: "^id^ " but finding " ^nid^ ".\n"); nid
    = id) env.locals.nodes then true
else if List.exists (fun (nid, _, _) -> prerr_string("
   Looking for: "^id^ " but finding " ^nid^ ".\n"); nid
    = id) env.globals.nodes then true
else false

let set_default_val ty =
match ty with
Int -> Literal(Int_Literal(0))
```

```
| Double -> Literal(Double_Literal(0.0))
| Bool -> Literal(Bool_Literal(false))
| String -> Literal(String_Literal(""))
| Graph -> Complex(Graph_Literal([]))
| Node | Rel -> Complex(Graph_Element("", []))
| _ -> raise(Error("Not a primitive type, YOU FOOL!"))


let get_new_env env func =
let new_locals = List.fold_left (fun l f -> let (t, id)
    = (match f with
Formal(ty, vid) -> (ty, vid)) in
if check_for_var_existence env.locals id then raise(
   Error("Variable: " ^ id ^ " already exists in local
   scope"))
else (match t with
Node -> let new_nodes = (id, id, []) :: l.nodes in {l
   with nodes = new_nodes}
| Rel -> let new_rels = (id, id, []) :: l.rels in {l
   with rels = new_rels}
| Graph -> let new_graphs = (id, []) :: l.graphs in {l
   with graphs = new_graphs}
| List(ty) -> let new_lists = (id, ty, []) :: l.lists
   in {l with lists = new_lists}
| Void -> raise(Error("Can't have a variable of type
   Void"))
|   -> let new_prims = (id, t, set_default_val t) :: l.
   prims in {l with prims = new_prims}) ) env.locals
   func.formals and
new_functions = func :: env.functions in
{env with functions = new_functions; locals =
   new_locals; scope = func.fname; return_type = func.
   return_type}

let update_prim_table var_table id ty v =
let does_exist = check_for_var_existence var_table id
   in
match does_exist with
```

```
true␣->␣raise(Error("Variable␣to␣declare:␣"␣^id^␣"␣
    already␣exists"))
|␣false␣->
let␣new_prims␣=␣(id,␣ty,␣v)::var_table.prims␣in
{var_table␣with␣prims␣=␣new_prims}

let␣rec␣gen_tuple_list␣fl␣l␣tl␣=
match␣fl,␣l␣with
[],␣[]␣->␣List.rev␣tl
|␣h1::t1,␣h2::t2␣->
(match␣h1,␣h2␣with
Formal(ty,␣id),␣lit␣->␣let␣new_tl␣=␣(id,␣ty,␣Literal(
    lit))␣::␣tl␣in
␣␣␣␣␣␣␣␣gen_tuple_list␣t1␣t2␣new_tl
)
|␣␣␣->␣raise(Error("You're such a failure"))

let update_node_or_rel_table env var_table id idt ty ex
    =
let does_exist = check_for_var_existence var_table id
    in
match does_exist with
true -> raise(Error("Variable to declare already exists
    "))
| false ->
let l = (match ex with
Complex(Graph_Element(s, ll)) -> ll
| _ -> raise(Error("Just cry. Stop what you're␣doing␣
    and␣cry")))␣and
forml␣=(match␣ty␣with
Node␣->␣let␣(_,␣fl)␣=␣List.find␣(fun␣(fid,␣_)␣->␣
    prerr_string("Looking␣for␣"␣^␣idt␣^␣"␣constructor,␣
    finding:␣"␣^␣fid␣^␣"\n");
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣fid=idt)␣env.node_types
    ␣in␣fl
|␣Rel␣->␣let␣(_,␣fl)␣=␣List.find␣(fun␣(fid,␣_)␣->␣
    prerr_string("Looking␣for␣"␣^␣idt␣^␣"␣constructor,␣
    finding:␣"␣^␣fid␣^␣"\n");
```

```
                                fid=idt) env.rel_types in fl
|   -> raise (Error("Wow. Do you even go here?")))) in
let tl = gen_tuple_list forml l [] in
(match ty with
Node -> let new_nodes = (id, idt, tl)::var_table.nodes in
        {var_table with nodes = new_nodes}
| Rel -> let new_rels = (id, idt, tl)::var_table.rels in
        {var_table with rels = new_rels}
|   -> raise (Error("You called the wrong var assign method you fool!")))

let update_graph_table var_table id v =
let does_exist = check_for_var_existence var_table id in
match does_exist with
true -> raise (Error("Variable to declare already exists"))
| false ->
let va = (match v with
|Complex(Graph_Literal(l)) -> l
|Func(Map(s, e1)) -> let (_, l) = List.find (fun (gid, _) -> gid=s) var_table.graphs in l
| Call(id, _) -> [];
|   -> raise (Error("Calling the wrong function, you fool!"))) in
let new_graphs = (id, va)::var_table.graphs in
                {var_table with graphs = new_graphs}

let update_list_table var_table id ty v =
let does_exist = check_for_var_existence var_table id in
match does_exist with
true -> raise (Error("Variable to declare already exists"))
| false ->
```

```
let new_lists = (id, ty, v)::var_table.lists in
        {var_table with lists = new_lists}

let rec check_expr env expr =
match expr with
| Literal(l) -> get_literal_type l
| Complex(c) -> (match c with
Graph_Literal(nrn_list) -> check_nrn_list env nrn_list
| Graph_Element(id, lit_list) ->
    check_node_or_rel_literal env id lit_list)
| Id(v) -> prerr_string ("check_expr: " ^ v ^ " id
    called\n");
(try get_type_from_id env.locals v with
Not_found -> try get_type_from_id env.globals v with
Not_found -> raise (Error ("Id does not appear in program
    ")))
| Unop(u, e) -> (match u with
Not -> if check_expr env e = Bool then Bool else raise
    (Error ("Using NOT on a non-boolean expr"))
| Neg -> if check_expr env e = Double then Double
else if check_expr env e = Int then Int
else raise (Error ("Using a neg on a non int or float
    expr")))
| Binop(e1, op, e2) ->
let t1 = (match e1 with
Access(_, _) -> check_expr env e2
| _ -> check_expr env e1) and t2 = check_expr env e2 in
let binop_t = (match op with
  Add -> check_arithmetic_binary_op t1 t2
| Sub -> check_arithmetic_binary_op t1 t2
| Mult -> check_arithmetic_binary_op t1 t2
| Div -> check_arithmetic_binary_op t1 t2
| Mod -> if (t1, t2) = (Int, Int) then Int else raise (
    Error ("Using MOD on a non-integer expr"))
| Equal -> check_equality t1 t2
| Neq -> check_equality t1 t2
| Less -> check_logic t1 t2
| Leq -> check_logic t1 t2
```

```
| Greater -> check_logic t1 t2
| Geq -> check_logic t1 t2
| And -> if(t1, t2) = (Bool, Bool) then Bool else raise
   (Error("Using AND on a non-boolean expr"))
| Or -> if(t1, t2) = (Bool, Bool) then Bool else raise
   (Error("Using OR on a non-boolean expr"))
| Concat -> if (t1, t2) = (String, String) then String
   else raise (Error("Using Concat on non-string expr")
   )) in binop_t
| Grop(e1, grop, gc) -> let t1 = check_expr env e1 in
(match t1 with
Graph -> (match grop with
| Graph_Insert ->
        (match gc with
        Node_Rel_Node_Tup(n1, r, n2) -> if
   check_nrn_expr env n1 r n2 then Graph else
                raise(Error("Invalid input for
   node_rel_node tuple"))
        )
| Graph_Remove ->
        (match gc with
        Node_Rel_Node_Tup(n1, r, n2) -> if
   check_nrn_expr env n1 r n2 then Graph else
                raise(Error("Invalid input for
   node_rel_node tuple"))
        ))
| _ -> raise(Error("Trying to perform a graph insert on
   a non-graph")))
| Geop(e1, geop, f) -> let t = check_expr env e1 and tf
   = (match f with
Formal(ty, s) -> ty;
) in
(match t with
Node | Rel -> (match tf with
Int | String | Bool | Double -> t
| _ -> raise(Error("Can only insert or remove field of
   primitive type")))
```

```
|␣␣␣->␣raise(Error("Can␣only␣insert␣field␣into␣a␣Node␣
    or␣Rel")))
|␣Access(idl,␣idr)␣->
prerr_string(idl␣^␣"."␣^␣idr␣^␣"␣called\n");
let␣t␣=␣(try␣get_type_from_id␣env.locals␣idl␣with
Not_found␣->␣try␣get_type_from_id␣env.globals␣idl␣with
Not_found␣->␣raise(Error("Can't find left identifier"))
    )in
(match t with
| Node -> t
| Rel ->  t
| _ -> raise(Error("Trying to access something that is
    not a node or rel")))
| Call("print", el) -> prerr_string("Print function is
    being called\n"); List.iter(fun e -> ignore(
    check_expr env e) ) el; Void
| Call(id, el) -> let func = (try List.find (fun f ->
    prerr_string("Looking for function: " ^ id ^ " but
    finding function: " ^f.fname^".\n");

Not_found -> raise(Error("Function definition not found
    "))) in
(try List.iter2 (fun e f -> let ty = (match f with
```

```
Formal
(
t
,
_
)
->
t
)
in
let t =
check_expr
env
e
in
if
t
<>
ty
then
raise
(
Error
("
Argument
does
```

```
not

match

expected

argument

type
")
)
)

el

func
.
formals

with
Invalid_argument s -> raise(Error("Entered the wrong
   number of arguments into function"))); func.
   return_type
| Func(fname) -> (match fname with
Find_Many(id,e1) -> prerr_string("Checking find_many on
    "^id^ ".\n");
if List.exists (fun (gid, _) -> gid = id) env.locals.
   graphs then
        let lt = check_find_many_arguments env e1 in
           List(lt)
else if List.exists (fun (gid,_) -> gid = id) env.
   globals.graphs then
        let lt = check_find_many_arguments env e1 in
           List(lt)
else raise(Error("Could not find List or Graph ID: " ^
   id ^ " to run find_many on"))
| Neighbors_Func(id, nid) ->
```

```
if List.exists (fun (gid, _) -> gid = id) env.locals.
   graphs then
         if check_if_id_is_node env nid then List(Node)
            else raise(Error("Locals: Argument to
            neighbors must be a node id"))
else if List.exists (fun (gid,_) -> gid = id) env.
   globals.graphs then
         if check_if_id_is_node env nid then List(Node)
            else raise(Error("Globals: Argument to
            neighbors must be a node id"))
else raise(Error("Could not find List or Graph ID: " ^
   id ^ " to run neighbors on"))
| Map(id,e2) ->
prerr_string("Checking collection id: " ^ id ^ " for
   map\n");
let ty = try get_type_from_id env.locals id with
         Not_found -> try get_type_from_id env.globals
            id with
         Not_found -> raise(Error("Could not find List
            ID: " ^ id ^ " to run map on")) in
         prerr_string(id ^ " is a " ^ print_type ty ^
            ".\n");
         match ty with
         List(t) -> List(t)
         | Graph -> Graph
         | _ -> raise(Error("Trying to run map on non-
            list or graph. Id: "^id^".\n")))

and check_map_func env ty map_func =
match map_func with
Map_Func(id, stmt_list) ->
let new_locals = (match ty with
Graph -> let new_nodes = (id, id, []) :: env.locals.
   nodes in
{env.locals with nodes = new_nodes}
| List(t) -> (match t with
Node -> let new_nodes = (id, id, []) :: env.locals.
   nodes in
```

```
{env.locals with nodes = new_nodes}
| Rel -> let new_rels = (id, id, []) :: env.locals.rels
    in
{env.locals with rels = new_rels}
| Graph -> let new_graphs = (id, []) :: env.locals.
   graphs in
{env.locals with graphs = new_graphs}
| List(t2) -> raise(Error("Cannot have List of Lists"))
| Void -> raise(Error("Cannot have List of Voids"))
| _ -> let new_prims = (id, t, set_default_val t) ::
   env.locals.prims in
{env.locals with prims = new_prims})
| _ -> raise(Error("Cannot have a map function operate
   on a non-collection"))) in
let new_env = {env with locals = new_locals} in
let (checked_stmts, up_env) = get_checked_statements
   new_env stmt_list [] in
checked_stmts

and check_find_many_arguments env e =
match e with
Find_Many_Node(complex) -> prerr_string("Find_many is
   the general type\n"); (match complex with
Graph_Element(s, ll) -> let t =
   check_node_or_rel_literal_matching env s ll in
if t = Node then t else raise(Error("Find_many_node
   does not have node as argument"))
| Graph_Literal(gcl) -> raise(Error("Find_many_node
   does not have node as argument!")))
| Find_Many_Gen(gt1, gt2) -> let t1 = check_graph_type
   env gt1 and t2 = check_graph_type env gt2 in
(match (t1, t2) with
(Node, Rel) -> prerr_string("Find_many is returning a
   node list\n"); Node
| (Rel, Node) -> prerr_string("Find_many is returning a
    node list\n"); Node
| (Node, Node) -> prerr_string("Find_many is returning
   a rel list\n"); Rel
```

```
| (Rel, Rel) -> raise(Error("Cannot have two rel
   arguments in Find_Many"))
| (_,_) -> raise(Error("Must have (Node, Node), (Rel,
   Node), or (Node, Rel) as arguments to find_many")))

and is_node env id =
let isNode = List.exists (fun (fid, _) -> fid = id) env
   .node_types in
isNode

and is_rel env id =
let isRel = List.exists (fun (fid, _) -> fid = id) env.
   rel_types in
isRel

and check_node_literal env id lit_list =
let (_, l) = List.find (fun (fid, _) -> fid = id) env.
   node_types in
(try List.iter2 (fun lit f -> let t2 = (match f with
Formal(ty,_) -> ty
) in
if(lit = Any) then raise(Error("Cannot have a complex
   type with any value. Can only use in find_many
   matching.")) else
let t1 = get_literal_type lit in
if t1 <> t2 then raise(Error("Type mismatch between
   arguments and expected type for given node object.")
   )) lit_list l with
Invalid_argument s -> raise(Error("Lists have unequal
   sizes. Check number of literals in your assignment.\
   n" ^
"Constructor list size: " ^ string_of_int (List.length
   l) ^ "\n" ^ "Literal List size: " ^ string_of_int(
   List.length lit_list) ^ "\n"))); Node

and check_rel_literal env id lit_list =
let (_, l) = List.find (fun (fid, _) -> fid = id) env.
   rel_types in
```

```
(try List.iter2 (fun lit f -> let t2 = (match f with
Formal(ty,_) -> ty
) in
if(lit = Any) then raise(Error("Cannot have a complex
   type with any value. Can only use in find_many
   matching.")) else
let t1 = get_literal_type lit in
if t1 <> t2 then raise(Error("Type mismatch between
   arguments and expected type for given rel object."))
   ) lit_list l with
Invalid_argument s -> raise(Error("Lists have unequal
   sizes. Check number of literals in your assignment.\
   n" ^
"Constructor list size: " ^ string_of_int(List.length l
   ) ^ "\n" ^ "Literal List size: " ^ string_of_int(
   List.length lit_list) ^ "\n"))); Rel

and check_node_or_rel_literal env id lit_list =
if is_node env id then check_node_literal env id
   lit_list
else if is_rel env id then check_rel_literal env id
   lit_list
else raise(Error("Could not find constructor for your
   node or rel"))

and check_node_literal_matching env id lit_list =
let (_, l) = List.find (fun (fid, _) -> fid = id) env.
   node_types in
(try List.iter2 (fun lit f -> let t2 = (match f with
Formal(ty,_) -> ty
) in
let t1 = if(lit = Any) then t2 else get_literal_type
   lit in
if t1 <> t2 then raise(Error("Type mismatch between
   arguments and expected type for given node object.")
   )) lit_list l with
Invalid_argument s -> raise(Error("Lists have unequal
   sizes. Check number of literals in your assignment.\
```

```
n" ^
"Constructor list size: " ^ string_of_int (List.length l) ^ "\n" ^ "Literal List size: " ^ string_of_int(List.length lit_list) ^ "\n"))); Node

and check_rel_literal_matching env id lit_list =
let (_, l) = List.find (fun (fid, _) -> fid = id) env.rel_types in
(try List.iter2 (fun lit f -> let t2 = (match f with
Formal(ty,_) -> ty
) in
let t1 = if(lit = Any) then t2 else get_literal_type lit in
if t1 <> t2 then raise(Error("Type mismatch between arguments and expected type for given rel object."))) lit_list l with
Invalid_argument s -> raise(Error("Lists have unequal sizes. Check number of literals in your assignment.\n" ^
"Constructor list size: " ^ string_of_int(List.length l) ^ "\n" ^ "Literal List size: " ^ string_of_int(List.length lit_list) ^ "\n"))); Rel

and check_node_or_rel_literal_matching env id lit_list =
if is_node env id then check_node_literal_matching env id lit_list
else if is_rel env id then check_rel_literal_matching env id lit_list
else raise(Error("Could not find constructor for your node or rel"))

and check_graph_ID env id =
if List.exists (fun (nid,_,_) -> nid=id) env.locals.nodes then Node
else if List.exists (fun (rid,_, _) -> rid=id) env.locals.rels then Rel
```

```
else if List.exists (fun (rid,_,_) -> rid = id) env.
   globals.rels then Rel
else if List.exists (fun (nid,_,_) -> nid=id) env.
   globals.nodes then Node
else raise(Error("Id (" ^id^") doesn't exist"))

and check_graph_type env gt =
let t = (match gt with
Graph_Id(gid) -> check_graph_ID env gid
|Graph_Type(complex) -> (match complex with
| Graph_Element(id, lit_list) ->
   check_node_or_rel_literal env id lit_list
| _ -> raise(Error("There is no spoon")))) in t

and check_nrn_expr env n1 r n2 =
let t1 = check_graph_type env n1 and t2 =
   check_graph_type env n2 and tr = check_graph_type
   env r in
match (t1, tr, t2) with
(Node, Rel, Node) -> true
| _ -> false

and check_nrn_list env nrn_list =
List.iter (fun nrn_expr ->
match nrn_expr with
Node_Rel_Node_Tup(n1, r, n2) -> if check_nrn_expr env
   n1 r n2 != true then
                                                
           raise(Error("Combination is not a Node-Rel-
   Node combination"))) nrn_list; Graph

and get_type_from_constructor env id =
if List.exists (fun (fid, _) -> fid=id) env.node_types
   then Node
else Rel

and get_field_list env id ll =
```

```
let (_,l) = try List.find (fun (fid, _) -> fid=id) env.
    node_types with
Not_found -> List.find (fun (fid, _) -> fid=id) env.
    rel_types in
get_field_lists l ll []

and get_field_lists l ll fl =
match l, ll with
| [], [] -> List.rev fl
| headl::taill, headll::tailll ->
(match headl with
Formal(ty, head) -> let sast_literal = (match headll 
    with
Int_Literal(i) -> SInt_Literal(i)
| Double_Literal(d) -> SDouble_Literal(d)
| Bool_Literal(b) -> SBool_Literal(b)
| String_Literal(s) -> SString_Literal(s)
| Any -> SAny) in
let new_fl = (head, ty, sast_literal) :: fl in
get_field_lists taill tailll new_fl)
| _ -> raise(Error("Lit list does not match constructor
     list"))

and get_scomplex env complex =
(match complex with
Graph_Literal(gcl) -> SGraph_Literal(get_sgc_list env 
    gcl []))
| Graph_Element(str, ll) -> SGraph_Element((
    get_type_from_constructor env str, str), 
    get_field_list env str ll))

and get_sgc_list env gcl sgcl =
match gcl with
[] -> List.rev sgcl
| head :: tail -> let new_gcl = get_sgc env head :: 
    sgcl in
get_sgc_list env tail new_gcl
```

```
and get_sgc env gc =
(match gc with
Node_Rel_Node_Tup(n1, r, n2) -> SNode_Rel_Node_tup(get_sgraph_type env n1, get_sgraph_type env r, get_sgraph_type env n2))

and get_sgraph_type env gt =
(match gt with
Graph_Id(s) -> SGraph_Id(s)
| Graph_Type(complex) -> SGraph_type(get_scomplex env complex))

and get_sformal form =
(match form with
Formal(ty, s) -> SFormal(ty, s))

and get_sformal_list fl sfl =
match fl with
[] -> List.rev sfl
| head :: tail -> let new_sfl = (get_sformal head) :: sfl in
get_sformal_list tail new_sfl

and get_sfm env fm =
match fm with
Find_Many_Node(complex) -> SFind_Many_Node(get_scomplex env complex)
| Find_Many_Gen(gt1, gt2) -> SFind_Many_Gen(get_sgraph_type env gt1, get_sgraph_type env gt2)

and get_smap env ty mf =
match mf with
Map_Func(s, sl) -> SMap_Func(s, check_map_func env ty mf)

and get_sbuilt_in_function_call env f =
match f with
Find_Many(s, fm) -> SFindMany(s, get_sfm env fm)
```

```
| Map (s, mf) -> SMap(s, (check_expr env (Id(s))), 
    get_smap env (check_expr env (Id(s))) mf)
| Neighbors_Func(s1,s2) -> SNeighbors_Func(s1,s2)

and get_sexpr env ex = match ex with
Literal(l) -> (match l with
Int_Literal(i) -> SLiteral(SInt_Literal(i), Int)
| Double_Literal(d)-> SLiteral(SDouble_Literal(d), 
    Double)
| String_Literal(s) -> SLiteral(SString_Literal(s), 
    String)
| Bool_Literal(b) -> SLiteral(SBool_Literal(b), Bool)
| Any -> SLiteral(SAny, String))
| Id(v) -> SId(v, check_expr env ex)
| Unop(u, e) -> SUnop(u, get_sexpr env e, check_expr 
    env ex)
| Binop(e1, op, e2) -> SBinop(get_sexpr env e1, op, 
    get_sexpr env e2, check_expr env ex)
| Grop(e, grop, gc) -> SGrop(get_sexpr env e, grop, 
    get_sgc env gc, check_expr env ex)
| Geop (e, geop, form) -> SGeop(get_sexpr env e, geop, 
    get_sformal form, check_expr env ex)
| Access(str, str2) -> SAccess(str, str2, check_expr 
    env ex)
| Call(str, el) -> SCall(str, List.map (fun e -> 
    get_sexpr env e) el, check_expr env ex)
| Func(f) -> SFunc(get_sbuilt_in_function_call env f, 
    check_expr env ex)
| Complex(comp) -> SComplex(get_scomplex env comp, 
    check_expr env ex)

and resolve_envs old_env new_env =
let new_prims = List.map (fun (id, ty, e) -> let v = (
    try List.find (fun (vid, _, _) -> vid=id) new_env.
    locals.prims with
Not_found -> (id, ty, e)) in v) old_env.locals.prims
and new_nodes = List.map (fun (id, idt, l) -> let v = (
    try List.find (fun (vid, _, _) -> vid=id) new_env.
```

```
    locals.nodes with
Not_found -> (id, id, l)) in v) old_env.locals.nodes
and new_rels = List.map (fun (id, idt, l) -> let v = (
    try List.find (fun (vid,_,_) -> vid=id) new_env.
    locals.rels with
Not_found -> (id, idt, l)) in v) old_env.locals.rels
and new_graphs = List.map (fun (id, gcl) -> let v = (
    try List.find (fun (gid,__) -> gid=id) new_env.
    locals.graphs with
Not_found -> (id, gcl)) in v) old_env.locals.graphs
and new_lists = List.map (fun (id, ty, el) -> let v = (
    try List.find (fun (lid,__,__) -> lid=id) new_env.
    locals.lists with
Not_found -> (id, ty, el)) in v) old_env.locals.lists 
    in
{new_env with locals = {prims = new_prims; nodes = 
    new_nodes; rels = new_rels; graphs = new_graphs; 
    lists = new_lists}}

and check_stmt env stmt =
prerr_string("Calling check_stmt\n"); match stmt with
| Block(stmt_list) ->
prerr_string("Calling Block from check_stmt\n");
let new_env = env in
let (checked_stmts, up_env) = List.fold_left (fun (l, e
    ) s -> let (checked_statment, up_e) = check_stmt e s
     in
                                                
                                                (
    checked_statment :: l, up_e)) ([], env) stmt_list in
let resolved_env = resolve_envs new_env up_env in
(SBlock(List.rev checked_stmts), resolved_env)

| Expr(e) ->
prerr_string("Calling expression from check_stmt");
let new_env = (match e with
Geop(e1, geop, formal) ->
let norid = get_id_from_expr e1 in
```

```
let (id,t,fl) = try List.find(fun (lid,_,_) -> lid=
    norid) env.locals.nodes with
Not_found -> try List.find(fun (lid,_,_) -> lid=norid)
    env.locals.rels with
Not_found -> try List.find(fun (lid,_,_) -> lid=norid)
    env.globals.nodes with
Not_found -> try List.find (fun (lid,_,_) -> lid=norid)
     env.globals.rels with
Not_found -> raise(Error("Id: " ^ norid ^ " not found as
    a node or rel.\n")) in
let (id, t, new_field_list) = (match formal with
Formal(ty,vid) ->
        (match geop with
                Data_Insert -> if List.exists (fun (
    vcid,_,_) -> vcid=vid) fl then raise(Error("Your
    field: " ^ vid ^ " already exists in node: " ^ id ^ ".\n
    "))
                else (id, t, (vid, ty, set_default_val
    ty) :: fl)
                | Data_Remove -> if List.exists (fun (
    vcid,t,_) -> (vcid=vid && t=ty)) fl then
                        let new_fl = List.fold_left (
    fun l (vcid,vcty,vcfl) -> if (vcid=vid) then
                                                
                                                l
    else (vcid,vcty,vcfl) :: l) [] fl in
                        (id, t, new_fl) else raise(
    Error("Field to remove does not exist"))
        )
) in add_tuple_to_list env (id, id, new_field_list)
| _ -> env) in (SExpr(get_sexpr env e), new_env)

| Return(e) ->
prerr_string("Return from check_stmt");
let t1 = check_expr env e in
(if not((t1=env.return_type)) then
raise (Error("Incompatible Return Type")));
```

```
let new_env = {env with has_return = true; return_type
    = t1; return_val = e } in
(SReturn(get_sexpr env e), new_env)

| If(e,s1,s2) ->
prerr_string("Calling If from check_stmt\n");
let t1 = check_expr env e in
(if not(t1=Bool) then
raise (Error("If statement must be a boolean")));
let (st1,new_env)= check_stmt env s1 in
let (st2,new_env2) = check_stmt new_env s2 in
(SIf((get_sexpr env e), st1, st2),new_env2)

| Var_Decl(decl) ->
prerr_string("Calling Var_decl from check_stmt\n");
let (checked_stmt, up_env) =
(match decl with
Var(ty, id) -> prerr_string("Local_Var: Checking " ^ id
    ^ "\n");
let new_table = (match ty with
Node | Rel-> update_node_or_rel_table env env.locals id
    id ty (set_default_val ty)
| Graph -> update_graph_table env.locals id (
    set_default_val ty)
| List(t) -> prerr_string("Var: Id: " ^ id ^ " is a
    List\n");
        (match t with
        List(ty) -> raise(Error("Can't have List of
    Lists! THAT'S INSANE"))
        | _ -> update_list_table env.locals id t [])
| Void -> raise(Error("Can't declare void"))
| _ -> update_prim_table env.locals id ty (
    set_default_val ty)) in
let new_env = {env with locals = new_table} in
(SVar(ty, id), new_env)
| Var_Decl_Assign(id, ty, e) -> prerr_string("
    Var_Decl_Assign: Checking " ^ id ^ "\n");
let t_ex = check_expr env e in
```

```
if (t_ex = ty) then
        let new_table = (match ty with
        Node | Rel-> let idt = (match e with
                Complex(Graph_Element(s, _)) -> s
                | _ -> raise(Error("Trying to assign
                   non Node or Rel to non Node or Rel")
                   )) in  update_node_or_rel_table env
                   env.locals id idt ty e
        | Graph -> update_graph_table env.locals id e
        | List(t) -> update_list_table env.locals id t
          []
        | Void -> raise(Error("Can't␣declare␣void"))
␣␣␣␣␣␣␣␣|␣␣␣->␣update_prim_table␣env.locals␣id␣ty␣e)␣in
␣␣␣␣␣␣␣␣let␣new_env␣=␣{env␣with␣locals␣=␣new_table}␣in
␣␣␣␣␣␣␣␣(SVar_Decl_Assign(id,␣ty,␣get_sexpr␣env␣e),␣
   new_env)
else
␣␣␣␣␣␣␣␣raise(Error("Type␣mismatch␣in␣local␣variable␣
   assignment"))
|␣Access_Assign(e1,␣e2)␣->
prerr_string("Access_Assign␣being␣called␣from␣check_stm
   \n");
let␣tl␣=␣check_expr␣env␣e1␣and␣tr␣=␣check_expr␣env␣e1␣
   in
prerr_string("tl␣=␣"␣^␣print_type␣tl␣^␣"␣tr␣=␣"␣^␣
   print_type␣tr␣^␣".\n"␣);
if␣(tl␣=␣tr)␣then
␣␣␣␣␣␣␣␣(SAccess_Assign(get_sexpr␣env␣e1,␣get_sexpr␣env
   ␣e2),␣env)
else
␣␣␣␣␣␣␣␣raise(Error("Type␣mismatch␣in␣assignment!"))
|␣Constructor(_,_,_)␣->␣raise(Error("Can't declare
   constructor locally"))) in (SVar_Decl(checked_stmt),
    up_env)

and get_checked_statements env stmts checked_statments
   =
match stmts with
```

```
| stmt :: tail ->
let (checked_statement, new_env) = check_stmt env stmt
   in
get_checked_statements new_env tail (checked_statement
   ::checked_statments)
| [] -> (List.rev checked_statments, env)

let check_function env func =
prerr_string("Starting to check function: " ^ func.
   fname ^ ".\n");
let (sfstatements, up_env) = get_checked_statements (
   get_new_env env func) func.body [] in
({sfname = func.fname; sformals = get_sformal_list func
   .formals []; sbody = sfstatements; sreturn_type =
   func.return_type}, {up_env with locals = env.locals
   })

let rec check_functions env funcs checked_funcs =
let checked_functions =
(match funcs with
func :: tail ->
let (checked_func, up_env) = check_function env func in
check_functions up_env tail (checked_func ::
   checked_funcs)
| [] -> checked_funcs) in
checked_functions

let check_global env var =
let (checked_global, up_env) =
(match var with
Var(ty, id) -> prerr_string("Global_Var: Checking " ^
   id ^ "\n");
let new_table = (match ty with
Node | Rel-> update_node_or_rel_table env env.globals
   id id ty (set_default_val ty)
| Graph -> update_graph_table env.globals id (
   set_default_val ty)
| List(t) -> update_list_table env.globals id t []
```

```
| Void -> raise(Error("Can't␣declare␣void"))
|␣␣␣->␣update_prim_table␣env.globals␣id␣ty␣(
   set_default_val␣ty))␣in
let␣new_env␣=␣{env␣with␣globals␣=␣new_table}␣in
(SVar(ty,␣id),␣new_env)
|␣Var_Decl_Assign(id,␣ty,␣e)␣->␣prerr_string("
   Var_Decl_Assign:␣Checking␣"␣^␣id␣^␣"\n");
let␣t_ex␣=␣check_expr␣env␣e␣in
if␣(t_ex␣=␣ty)␣then
␣␣␣␣␣␣␣␣let␣new_table␣=␣(match␣ty␣with
␣␣␣␣␣␣␣␣Node␣|␣Rel->␣let␣idt␣=␣(match␣e␣with
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Complex(Graph_Element(s,␣_))␣->␣s
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣␣␣->␣raise(Error("Trying␣to␣assign␣
   non␣Node␣or␣Rel␣to␣non␣Node␣or␣Rel")))␣in␣␣
   update_node_or_rel_table␣env␣env.globals␣id␣idt␣ty␣e
␣␣␣␣␣␣␣␣|␣Graph␣->␣update_graph_table␣env.globals␣id␣e
␣␣␣␣␣␣␣␣|␣List(t)␣->␣update_list_table␣env.globals␣id␣t
   ␣[]
␣␣␣␣␣␣␣␣|␣Void␣->␣raise(Error("Can't declare void"))
        | _ -> update_prim_table env.globals id ty e)
           in
        let new_env = {env with globals = new_table} in
        (SVar_Decl_Assign(id, ty, get_sexpr env e),
           new_env)
else
        raise(Error("Type mismatch in global variable
           assignment"))
| Access_Assign(e1, e2) -> let tl = check_expr env e1
   and tr = check_expr env e1 in
if (tl = tr) then
        (SAccess_Assign(get_sexpr env e1, get_sexpr env
            e2), env)
else
        raise(Error("Type mismatch in assignment!"))
| Constructor(ty, id, l) ->
prerr_string("Constructor " ^ id ^ " being created\n" )
   ;
let list_to_check = (match ty with
```

```
         Node -> env.node_types
         | Rel -> env.rel_types
         | _ -> raise(Error("Can't declare constructors 
            of non Node or Rel types"))) in
let does_type_exist = List.exists (fun (cid, _) -> cid=
   id) list_to_check in
if does_type_exist then
        raise(Error("Already have a constructor of this
   name for this type"))
else
        let new_constructors = (id, l) :: list_to_check
    in
        let new_env = (match ty with
        Node -> if List.exists (fun (vid, _) -> vid=id) 
   env.rel_types then
                raise(Error("Can't have a constructor
   for both Node and Rel types")) else
                {env with node_types = new_constructors
                   }
         | Rel -> if List.exists (fun (vid,_) -> vid=id)
             env.node_types then
                raise(Error("Can't have a constructor 
                   for both Rel and Node types"))
                        else {env with rel_types = 
   new_constructors}
        |  _ -> raise(Error("Can only have constructors 
   for Node and Rel types"))) in
        (SConstructor(ty, id, get_sformal_list l []), 
   new_env)) in
(checked_global, up_env)

let rec check_globals_and_update_env env vars 
   checked_vars =
let (checked_globals, new_env) =
(match vars with
| var :: tail ->
let (checked_global, up_env) = check_global env var in
```

```
check_globals_and_update_env up_env tail (
   checked_global::checked_vars)
| [] -> (checked_vars, env))
in (checked_globals, new_env)

let run_program program =
let (vars, funcs) = program in
let env = beginning_environment in
let (checked_globals, new_env) =
   check_globals_and_update_env env (List.rev vars) []
   in
let checked_functions = check_functions new_env (List.
   rev funcs) [] in
SProg(checked_globals, checked_functions)
```

8.5 SAST

```
open Ast

type sformal =
  SFormal of Ast.n2n_type * string

type svar_decl =
    SVar of Ast.n2n_type * string
  | SConstructor of Ast.n2n_type * string * sformal
     list
  | SVar_Decl_Assign of string * Ast.n2n_type * sexpr
  | SAccess_Assign of sexpr * sexpr

and sexpr =
    SLiteral of sliteral * Ast.n2n_type
  | SId of string * Ast.n2n_type
  | SBinop of sexpr * Ast.op * sexpr * Ast.n2n_type
  | SGrop of sexpr * Ast.grop * sgraph_component * Ast.
     n2n_type
  | SGeop of sexpr * Ast.geop * sformal * Ast.n2n_type
```

```
  | SUnop of Ast.uop * sexpr * Ast.n2n_type
  | SAccess of string * string * Ast.n2n_type
  | SCall of string * sexpr list * Ast.n2n_type
  | SFunc of sbuilt_in_function_call * Ast.n2n_type
  | SComplex of scomplex_literal * Ast.n2n_type

and sliteral =
  SInt_Literal of int
  | SDouble_Literal of float
  | SString_Literal of string
  | SBool_Literal of bool
  | SAny

and sbuilt_in_function_call =
  SFindMany of string * sfind_many
  | SMap of string * Ast.n2n_type * smap_function
  | SNeighbors_Func of string * string

and sfind_many =
  | SFind_Many_Node of scomplex_literal
  | SFind_Many_Gen of sgraph_type * sgraph_type

and smap_function =
  | SMap_Func of string * sstatement list

and scomplex_literal =
  | SGraph_Literal of sgraph_component list
  | SGraph_Element of (Ast.n2n_type * string) * (string
      * Ast.n2n_type * sliteral) list

and sgraph_component =
  SNode_Rel_Node_tup of sgraph_type * sgraph_type *
    sgraph_type

and sgraph_type =
  | SGraph_Id of string
  | SGraph_type of scomplex_literal
```

```
and sstatement =
  SBlock of sstatement list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstatement * sstatement
  | SVar_Decl of svar_decl

type sfunc_decl = {
  sfname : string;
  sformals : sformal list;
  sbody : sstatement list;
  sreturn_type : Ast.n2n_type;
}

type sprogram =
  SProg of svar_decl list * sfunc_decl list
```

8.6 Code Generation

```
open Ast
open Sast
open Printf

let imports =
  "package com.n2n;\n\n" ^
  "import java.util.*;\n\n"

let set_default_val ty = match ty with
    Int -> "0"
    | String -> "\"\""
    | Double -> "0.0"
    | Bool -> "false"
    | _ -> "\"\""


 let rec gen_var_type = function
```

```
    Int       -> "int"
  | Double    -> "double"
  | Bool      -> "boolean"
  | String    -> "String"
  | Void      -> "void"
  | Rel       -> "Relationship"
  | Node      -> "Node"
  | Graph     -> "Graph"
  | List(ty)  -> "Set<" ^ gen_var_type ty ^ ">"

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

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

let gen_literal lit = match lit with
  | SInt_Literal(i)       -> string_of_int i
  | SDouble_Literal(d)    -> string_of_float d
  | SBool_Literal(b)      -> string_of_bool b
  | SString_Literal(str)  -> "\"" ^ str ^ "\""
  | SAny                  -> "Any"

let rec gen_literal_list ll = match ll with
```

```
  | [] -> ""
  | head::[] -> gen_literal head
  | head::tail -> gen_literal head ^ ", " ^
     gen_literal_list tail

let rec gen_expr expr = match expr with
  | SLiteral(l,t)           -> gen_literal l
  | SId(v,t)                -> v
  | SComplex(c,t)           -> gen_scomplex c
  | SUnop(u, e, t)          -> gen_unop u ^ "(" ^
     gen_expr e ^ ")"
  | SBinop(e1, op, e2, t)   -> (match e1, op with
            | SAccess(_,_,_), Equal | SAccess(_,_,_),
               Neq -> gen_expr e1 ^ ".equals(" ^
               gen_expr e2 ^ ")"
            | _ ->gen_expr e1 ^ gen_binop op ^ gen_expr
                e2)
  | SAccess(e1,e2, t)       -> e1 ^ ".getValueFor(\"" ^
      e2 ^"\")"
  | SCall(id, e1, t)        -> if(id="print") then
     gen_print e1 else id ^ "(" ^ gen_expr_list e1 ^ ")
     "
  | SFunc(fname, t)         -> gen_sfunc fname
  | SGrop(e1, grop, nrn, t) -> gen_expr e1 ^
     gen_graph_op grop ^ gen_nrn_tup nrn ^ ")"
  | SGeop(e1, geop, f1, t)  -> gen_expr e1 ^
     gen_graph_elem_op geop ^
      (match f1 with
        SFormal(_, id) -> id) ^
      (match geop with
        Data_Insert ->  let t =
          (match f1 with
            SFormal(ty,_) -> ty) in "\"," ^ (
                set_default_val t) ^ ")"
        | Data_Remove -> "\")")

and gen_expr_list expr_list = match expr_list with
  | [] -> ""
```

```
  | head::[] -> gen_expr head
  | head::tail -> gen_expr head ^ ", " ^ gen_expr_list
     tail

and gen_formal h = match h with
  SFormal(type_spec, id) -> gen_var_type type_spec ^ "
     " ^ id

and gen_formal_list fl = match fl with
  | [] -> ""
  | head::[] -> gen_formal head
  | head::tail -> gen_formal head ^ ", " ^
     gen_formal_list tail

and gen_print p = match p with
  | []          -> ""
  | head::[]    -> "System.out.print(" ^ gen_expr head ^
      ")"
  | head::tail -> "System.out.print(" ^ gen_expr head ^
      ")" ^ gen_print tail

and gen_sgraph_type gt = match gt with
  | SGraph_Id(id)    -> id
  | SGraph_type(s1) -> (match s1 with
    SGraph_Element(element_type, field_info) ->
       gen_gt_instantiation element_type field_info
    | _ -> "")

and gen_gt_instantiation element_type field_info =
   match element_type with
  | (graph_element_type, id) -> "new " ^ gen_var_type
     graph_element_type ^ "(\"" ^ id ^ "\"" ^
  ", new HashMap<String, Object>() {{\n\t" ^ (
     gen_graph_elem element_type field_info "") ^ "\n
     }})\n"

and gen_scomplex c = match c with
```

```
  | SGraph_Literal(nrn_list) ->
     gen_node_rel_node_tup_list nrn_list (* Should just
      be a list of graph elements *)
  | SGraph_Element(element_type, field_info) ->
     gen_element_instantiation element_type field_info

and gen_element_instantiation element_type field_info =
    match element_type with
  | (graph_element_type, id) -> "\"" ^ id ^ "\"" ^
  ", new HashMap<String, Object>() {{\n\t" ^ (
     gen_graph_elem element_type field_info "") ^ "\n
     }}\n"

and gen_graph_elem element_type field_info out_str =
   match element_type with
  | (n2n_type, id) -> (match field_info with
    | (field_name, field_type, field_value)::tail ->
       let put_string = (if (field_type = String) then
      sprintf "put(%s, %s);\n" ("\"" ^ field_name ^
        "\"") (if field_value = SAny then "\"Any\""
        else (gen_literal field_value))
    else
      sprintf "put(%s, %s);\n" ("\"" ^ field_name ^
        "\"") (if field_value = SAny then "\"Any\""
        else (gen_literal field_value))) in
      let new_str = out_str ^ put_string in (
        gen_graph_elem element_type tail new_str)
    | [] -> out_str)

and gen_node_rel_node_tup_list nrn_tup = match nrn_tup
   with
  | [] -> ""
  | head::[] -> gen_nrn_tup head
  | head::tail -> gen_nrn_tup head ^ ", " ^
     gen_node_rel_node_tup_list tail

and gen_nrn_tup nrn_tup = match nrn_tup with
```

```
  SNode_Rel_Node_tup(sg1, sg2, sg3) -> "new Graph.
     Member<>(" ^ gen_sgraph_type sg1 ^ ", " ^
     gen_sgraph_type sg2 ^ ", " ^ gen_sgraph_type sg3 ^
      ")"

and gen_sfunc fname = match fname with
     (* TOASK How call Map and Neighbors function? And
        Graph/Data inserts *)
     | SFindMany(id, sfm) -> id ^ ".findMany(" ^
        gen_find_many sfm ^ ")"
     | SMap(id, ty, smf) ->
         (match ty with
           Graph -> "for( " ^ "Node " ^ gen_map id ty
              smf
           | List(t) -> "for( " ^ gen_var_type t ^ " " ^
               gen_map id ty smf
           | _ -> raise Not_found)
     | SNeighbors_Func(id1, id2) -> id1 ^ ".neighbors("
        ^ id2 ^ ")"

and gen_find_many sfind = match sfind with
   | SFind_Many_Node(scomp) -> gen_scomplex scomp
   | SFind_Many_Gen(gt1, gt2) -> gen_sgraph_type gt1 ^
     ", " ^ gen_sgraph_type gt2

(* TODO: Figure out what to pass into the Map function
   when created in Javac Backedn *)
(* Cuz this is not right! *)
and gen_map id ty smap = match smap with
   | SMap_Func(nid, sl) -> (match ty with
       Graph -> nid ^ " : " ^ id ^ ".getMapSet() ){\n" ^
           gen_sstmt_list sl ^ "}\n"
       | List(t) -> nid ^ " : " ^ id ^ " ){\n" ^
          gen_sstmt_list sl ^ "}\n"
       | _ -> raise Not_found)


and gen_sstmt stmt = match stmt with
```

```
  | SBlock(stmt_list) -> gen_sstmt_list stmt_list
  | SExpr(expr) -> (match expr with
        SFunc(SMap(_,_,_), _) -> gen_expr expr ^ "\n\
          t"
        | _ -> gen_expr expr ^ ";\n\t")
  | SReturn(expr) -> "return " ^ gen_expr expr ^ ";\n\t
    "
  | SIf(expr,s1,s2) -> "if(" ^ gen_expr expr ^ ") {\n\t
    \t" ^ gen_sstmt s1 ^ "}\n\telse {\n\t" ^ gen_sstmt
     s2 ^ "}\n\n"
  | SVar_Decl(vdec) -> gen_var_dec vdec ^";\n\t"

and gen_sstmt_list stmt_list = match stmt_list with
  | [] -> ""
  | head::[] -> gen_sstmt head
  | head::tail -> gen_sstmt head ^ gen_sstmt_list tail

and gen_var_dec dec = match dec with
  | SVar(ty,id) -> gen_var_type ty ^ " " ^ id
  | SConstructor(ty,id,formals) -> "String " ^id^ " = "
     ^ "\"" ^id^ "\";\n"
  | SAccess_Assign(e1, e2) -> (match e1 with
     SAccess(el, er, t) -> el ^ ".getData().put(" ^
        "\"" ^ er ^ "\", " ^ gen_expr e2 ^ ")"
     |_-> gen_expr e1 ^ " = " ^ gen_expr e2)
  | SVar_Decl_Assign(id,ty,e) -> (match ty with
    | Int | Double | Bool | String -> gen_var_type ty ^
       " " ^ id ^ " = " ^ gen_expr e ^ ";"
    | Rel | Node -> gen_var_type ty ^ " " ^ id ^ " =
       new " ^ gen_var_type ty ^ "(" ^ gen_expr e ^ ")
       ;"
    | List(_) -> (match e with
        SFunc(fname, t) ->
          (match fname with
          _ -> gen_var_type ty ^ " " ^ id ^ " = " ^
             gen_sfunc fname)
        | _ -> gen_var_type ty ^ " " ^ id ^ " = new " ^
            gen_var_type ty ^ "(" ^ gen_expr e ^ ");")
```

```
    | Graph -> (match e with
        SFunc(fname, t) ->
          (match fname with
          SMap(_,_,_) -> gen_sfunc fname
        | _ -> raise Not_found)
        | SCall(s, e1, t) -> gen_var_type ty ^ " " ^ id
            ^ " = " ^ s ^ "(" ^ gen_expr_list e1 ^ ");"
        |_ ->gen_var_type ty ^ " " ^ id ^ " = new Graph
            (Arrays.asList(" ^ gen_expr e ^ "));")
    | Void -> "void")(* impossible case *)

and gen_var_dec_list var_dec_list = match var_dec_list
   with
  | [] -> ""
  | head::[] -> gen_var_dec head
  | head::tail -> gen_var_dec head ^ gen_var_dec_list
     tail

and gen_global_var_dec_list var_dec_list = match
   var_dec_list with
  | [] -> ""
  | head::[] -> "static " ^ gen_var_dec head ^ ";"
  | head::tail -> gen_var_dec head ^ gen_var_dec_list
     tail

(* TODO: These dont currently exist in java backend *)
and gen_graph_op grop = match grop with
  | Graph_Insert -> ".insert("
  | Graph_Remove -> ".remove("

and gen_graph_elem_op geop = match geop with
  | Data_Insert -> ".getData().put(\""
  | Data_Remove -> ".getData().remove(\""

and gen_func_dec func =
  if(func.sfname = "main") then "public static void
     main(String[] args) {\n\n\t\t" ^ gen_sstmt_list
     func.sbody ^ "}\n"
```

```
  else "public static " ^ gen_var_type func.
     sreturn_type ^ " " ^ func.sfname ^
  "(" ^ gen_formal_list func.sformals ^ ") {\n" ^
     gen_sstmt_list func.sbody ^ "}\n"

and gen_func_dec_list fl = match fl with
  | [] -> ""
  | head::[] -> gen_func_dec head
  | head::tail -> gen_func_dec head ^ gen_func_dec_list
      tail

let prog_gen = function
  SProg(checked_globals, checked_functions) ->
    imports ^
    "class Main {\n\n\t" ^
    gen_global_var_dec_list checked_globals ^ ";\n" ^
    gen_func_dec_list checked_functions ^
    "\n}\n"
```

8.7 Java Backend

8.7.1 Graph.java

```
package com.n2n;

import java.util.*;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Graph {

    private Set<Relationship> relationships = new
       HashSet<>();

```

```java
/**
 * A class that encapsulates {Node, Relationship,
   Node} triplets. Used
 * when constructing graphs to enforce this type
   union.
 *
 * @param <N> A node.
 * @param <R> A relationship.
 */
public static class Member<N, R> {
    private N from;
    private R rel;
    private N to;

    public Member(N from, R rel, N to) {
        this.from = from;
        this.rel  = rel;
        this.to   = to;
    }

    public N getFrom() { return from; }
    public R getRel()  { return rel; }
    public N getTo()   { return to; }
}

public Set<Relationship> getRelationships(){
    return this.relationships;
}

public Graph(List<Member<Node, Relationship>>
   relatedMemberList) {
    addToGraph(relatedMemberList);
}

private void addToGraph(List<Member<Node,
   Relationship>> relatedMemberList) {
    relatedMemberList.stream().forEach((members) ->
        {
```

```
            members.getRel().addNodes(members.getFrom()
                , members.getTo());
            relationships.add(members.getRel());
        });
    }

    /**
     * An operation for finding nodes based on loose
        relationship equality.
     *
     * Loose equality is defined as a match on the '
        type' of the relationship.
     *
     * Example:
     * + Find movies in which Keanu acted_in
     *       keanu_movies: List<Node> = find_many(keanu
        acted_in)
     *
     *   Will search the graph for matches on a 'keanu'
         node that has a relationship of type 'acted_in
        '
     *
     * @param node The source node from which
        relationships start.
     * @param relationshipType The relationship type (
        its name) that joins source node and potential
        target nodes.
     * @return The set of target nodes, or an empty set
          if no nodes are found.
     */
    public Set<Node> findMany(Node node, String
       relationshipType) {
        return findManyHelper(r -> r.looselyEquals(
            relationshipType), r -> r.getNodesFrom(node)
            .stream());
    }
```

```
public Set<Node> findMany(String type, Map<String,
    Object> data) {
     return findMany(new Node(type, data));
}

/**
 * An operation for finding nodes based on strict
     relationship equality.
 *
 * Strict equality is defined as a match on the '
     type' of the relationship and all the fields
     provided in the
 * relationship.
 *
 * Example:
 * + Find movies in which Keanu acted_in as Neo
 *        keanu_movies: List<Node> = find_many(keanu
     acted_in[   Neo   ])
 *
 *    Will search the graph for matches on a 'keanu'
      node that has a relationship of type 'acted_in
     ' with
 *    first field of the relationship as "Neo"
 *
 * @param node The source node.
 * @param relationship A relationship that joins
     source node and target nodes.
 * @return The set of target nodes, or an empty set
      if no nodes are found.
 */
public Set<Node> findMany(Node node, Relationship
    relationship) {
     return findManyHelper(r -> r.strictlyEquals(
         relationship), r -> r.getNodesFrom(node).
         stream());
}

/**
```

```
 * An operation for finding nodes based on an
   inverse, loose relationship equality.
 *
 * Loose equality is defined as a match on the '
   type' of the relationship.
 *
 * Example:
 * + Find movies in which Keanu acted_in
 *       matrix_actors: List<Node> = find_many(
   acted_in matrix)
 *
 *   Will search the graph for actor nodes that
   point to node 'matrix' through an 'acted_in'
   relationship
 *
 * @param node The source node from which
   relationships start.
 * @param relationshipType The relationship type (
   its name) that joins source node and potential
   target nodes.
 * @return The set of target nodes, or an empty set
     if no nodes are found.
 */
public Set<Node> findMany(String relationshipType,
  Node node) {
    return findManyHelper(r -> r.looselyEquals(
       relationshipType), r -> r.getNodesTo(node).
       stream());
}

/**
 * An operation for finding nodes based on an
   inverse, strict relationship equality.
 *
 * Strict equality is defined as a match on the '
   type' of the relationship and all the fields
   provided in the
 * relationship.
```

```
 *
 * Example:
 * + Find actors that acted_in as 'Neo' in 'matrix'
 *
 *       neo_actors: List<Node> = find_many(acted_in
    (   N e o   ) matrix)
 *
 *     Will search the graph for actor nodes that
    point to node 'matrix' through an 'acted_in'
    relationship with
 *     first field as 'Neo'
 *
 * @param node The destination node at which the
    relationship ends.
 * @param relationship A relationship that joins
    source node and target nodes.
 * @return The set of target nodes, or an empty set
      if no nodes are found.
 */
public Set<Node> findMany(Relationship relationship
   , Node node) {
    return findManyHelper(r -> r.strictlyEquals(
       relationship), r -> r.getNodesTo(node).
       stream());
}

/**
 * Finds all relationships that join leftNode and
    RightNode.
 * TODO: Ugly and inefficient. Fix me.
 *
 * @param leftNode Left side of the relationship.
 * @param rightNode Right side of the relationship.
 * @return A set that contains relationships
    between the two nodes.
 */
public Set<Relationship> findMany(Node leftNode,
   Node rightNode) {
```

```
    Set<Relationship> result = new HashSet<>();
    result.addAll(relationshipFinder(leftNode,
       rightNode));
    result.addAll(relationshipFinder(rightNode,
       leftNode));
    return result;
}

public Set<Node> findMany(Node target) {
    Set<Node> nodes = new HashSet<>();
    for (Relationship relationship : relationships)
        {
        nodes.addAll(relationship
                .getAll()
                .stream()
                .filter(node -> node.looselyEquals(
                   target)).collect(Collectors.
                   toList()));
    }
    return nodes;
}

public Set<Node> neighbors(Node target) {
    return getNodesFromRelationships(r -> r.
       getNodesFrom(target).stream());
}

public Set<Node> getMapSet() {
    return getNodesFromRelationships(r -> r.getAll
       ().stream());
}

private Set<Node> getNodesFromRelationships(
   Function<Relationship, Stream<? extends Node>>
   mapper) {
    return relationships.stream().flatMap(mapper).
       collect(Collectors.toSet());
}
```

```
private Set<Relationship> relationshipFinder(Node
    left, Node right) {
    Set<Relationship> result = new HashSet<>();
    for (Relationship relationship : relationships)
        {
        Set<Node> nodesFromLeft = relationship.
            getNodesFrom(left);
        Set<Node> nodesToRight = relationship.
            getNodesTo(right);
        if (!nodesToRight.isEmpty()) {
            boolean modified = nodesFromLeft.
                retainAll(nodesToRight);
            if (modified) {
                result.add(relationship);
            }
        }
    }
    return result;
}

private Set<Node> findManyHelper(Predicate<
    Relationship> predicate, Function<Relationship,
    Stream<? extends Node>> mapper) {
    return relationships.stream()
            .filter(predicate)
            .flatMap(mapper)
            .collect(Collectors.toSet());
}

public void insert(List<Member<Node, Relationship>>
    relatedMemberList) {
    addToGraph(relatedMemberList);
}

@Override
public String toString() {
    return "Graph{" +
```

```
                "relationships=" + relationships +
                '}';
    }
}
```

8.7.2 Node.java

```
package com.n2n;

import java.util.Map;

public class Node {

    private String type;
    private Map<String, Object> data;

    public Node(String type, Map<String, Object> data)
       {
        this.type = type;
        this.data = data;
    }

    public Object getValueFor(String field) {
        return this.data.get(field);
    }

    public Map<String, Object> getData() {
        return this.data;
    }

    public boolean looselyEquals(Object other) {
        if (this == other) return true;
        if (other == null || getClass() != other.
           getClass()) return false;

        Node otherNode = (Node) other;
```

```
    return type.equals(otherNode.type) &&
        dataLooselyEquals(otherNode.getData());
}

private boolean dataLooselyEquals(Map<String,
    Object> other) {
    /*if (!this.data.keySet().containsAll(other.
        keySet())) {
          return false;
    }*/

    for (Map.Entry<String, Object> entry : other.
        entrySet()) {
          if (!entry.getValue().equals("Any") && !
              this.data.get(entry.getKey()).equals(
              entry.getValue())) {
                return false;
          }
    }
    return true;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass())
        return false;

    Node node = (Node) o;

    return data.equals(node.data) && type.equals(
        node.type);
}

@Override
public int hashCode() {
    int result = type.hashCode();
    result = 31 * result + data.hashCode();
```

```
        return result;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        int count = this.data.size();
        int i=0;
        sb.append(type).append("{ ");
        for (Map.Entry<String, Object> data : this.data
            .entrySet()) {
            sb.append(data.getKey()).append(" = ").
               append(data.getValue());
            if (i < count-1)
                sb.append(", ");
            i++;
        }
        sb.append(" }");
        return sb.toString();
    }
}
```

8.7.3 Relationship.java

```
package com.n2n;

import java.util.*;

public class Relationship {

    private String type;
    private Map<String, Object> data;
    private Map<Node, Set<Node>> fromTo = new HashMap
       <>();
    private Map<Node, Set<Node>> toFrom = new HashMap
       <>();
```

```
public Relationship(String type, Map<String, Object
    > data) {
    this.type = type;
    this.data = data;
}

public Relationship(String type) {
    this(type, Collections.emptyMap());
}

public Object getValueFor(String field) {
    return this.data.get(field);
}

public Set<Node> getAll() {
    Set<Node> nodes = new HashSet<>();
    nodes.addAll(fromTo.keySet());
    nodes.addAll(toFrom.keySet());
    return nodes;
}

public void addNodes(Node from, Node to) {
    if (fromTo.containsKey(from)) {
        fromTo.get(from).add(to);
    } else {
        fromTo.put(from, new HashSet<>(Arrays.
            asList(to)));
    }
    if (toFrom.containsKey(to)) {
        toFrom.get(to).add(from);
    } else {
        toFrom.put(to, new HashSet<>(Arrays.asList(
            from)));
    }
}

public Set<Node> getNodesFrom(Node from) {
```

```
        return fromTo.containsKey(from) ? new HashSet
            <>(fromTo.get(from)) : Collections.emptySet
            ();
    }

    public Set<Node> getNodesTo(Node to) {
        return toFrom.containsKey(to) ? new HashSet<>(
            toFrom.get(to)) : Collections.emptySet();
    }

    public boolean looselyEquals(String type) {
        return this.type.equals(type);
    }

    public boolean strictlyEquals(Relationship
        relationship) {
        return this.equals(relationship);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Relationship that = (Relationship) o;

        return type.equals(that.type) && data.equals(
            that.data);
    }

    @Override
    public int hashCode() {
        int result = type.hashCode();
        result = 31 * result + data.hashCode();
        return result;
    }
```

```
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        int count = this.data.size();
        int i = 0;
        sb.append(type).append("{ ");
        for (Map.Entry<String, Object> data : this.data
            .entrySet()) {
            sb.append(data.getKey()).append(" = ").
                append(data.getValue());
            if (i < count-1)
                sb.append(", ");
            i++;
        }
        sb.append(" }");
        return sb.toString();
    }
}
```

8.8 Tests

8.8.1 arithmetic1.n2n

```
fn main () -> Void {
        print(2+4);
}
```

8.8.2 arithmetic2.n2n

```
fn main () -> Void {
        print(3-6);
        print("\n");
        print(6*3);
        print("\n");
        print(6/3);
```

```
        print("\n");
        print(15%4);
        print("\n");
}
```

8.8.3 comparison1.n2n

```
fn main () -> Void {
        print(2>3);
        print("\n");
        print(2<3);
        print("\n");
}
```

8.8.4 comparison2.n2n

```
fn main () -> Void {
        print(2==4);
        print("\n");
        print(2!=4);
        print("\n");
        print(2>=4);
        print("\n");
        print(2<=4);
        print("\n");
}
```

8.8.5 declaration.n2n

```
;;Test the multiple declaration of a variable local
   scoping;;

a: Int = 100;

fn main () -> Void {
```

```
        print(a);
        print("\n");
        a = 300;
        print(a);
        print("\n");
        a: Int = 1;   ;;Not sure what happens when
           declared again;;
        print(a);
        print("\n");

        b: Int = 200;
        print(b);
        print("\n");
}
```

8.8.6 escapestr1.n2n

```
fn  main() -> Void {
        a: String = "a\n";
        print(a);
}
```

8.8.7 escapestr2.n2n

```
fn  main() -> Void {
        a: String = "a\t";
        print(a);
        print("\n");
}
```

8.8.8 escapestr3.n2n

```
fn escapestr3() -> Void {
        a: String = "a\\";
        print(a);
```

```
}

fn main() -> Void {
  escapestr3();
}
```

8.8.9 escapestr5.n2n

```
fn  escapestr3() -> Void {
        a: String = "a\'";
        print(a);
}

fn main() -> Void {
  escapestr3();
}
```

8.8.10 float.n2n

```
fn main () -> Void {
        print(8/6);
}
```

8.8.11 float2.n2n

```
fn main () -> Void {
        print(8.0/6.0);
}
```

8.8.12 helloworld.n2n

```
fn main () -> Void {
        print("hello world");
}
```

8.8.13 init1.n2n

```
fn main() -> Void {
        a: Int = 66;
        b: Int = 123456;
        print(a);
        print("\n");
        print(b);
        print("\n");

}
```

8.8.14 functions1.n2n

```
fn foo () -> Void {
        print(5);

}

fn main () -> Void {
        foo();
        print("\n");

}
```

8.8.15 functions2.n2n

```
fn foo () -> Int {
        ;;Do something;;
        return 5;

}

fn main () -> Void {
        a: Int = foo();
        print(a);
```

```
        print("\n");
}
```

8.8.16 functions3.n2n

```
fn foo (c: Int) -> Int {
        ;;Do something;;
        return c+5;
}

fn bar (a: Int) -> Int {
        b: Int = a;
        return foo(b);
}

fn main () -> Void {
        a: Int = bar(61);
        print(a);
}
```

8.8.17 find_many1.n2n

```
actor: Node = { name: String, age: Int };
actedIn: Rel = { role: String };
movie: Node = { title: String, year: Int };

fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        Leo: Node = actor["Leo", 20];

        neo: Rel = actedIn["Neo"];
        jordan: Rel = actedIn["Jordan"];

        matrix: Node = movie["Matrix", 1999];
        wolf: Node = movie["Wolf", 1994];
```

```
        Cast: Graph = <
                Keanu neo matrix,
                Leo jordan wolf
        >;

        missing_rel: List<Rel>;
        missing_rel = Cast.find_many(Keanu, matrix);

        print(missing_rel);

}
```

8.8.18 find_many2.n2n

```
actor: Node = { name: String, age: Int };
actedIn: Rel = { role: String };
movie: Node = { title: String, year: Int };

fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        Leo: Node = actor["Leo", 20];

        neo: Rel = actedIn["Neo"];
        jordan: Rel = actedIn["Jordan"];
        nelson: Rel = actedIn["Nelson"];

        matrix: Node = movie["Matrix", 1999];
        wolf: Node = movie["Wolf", 1994];
        sweet_nov: Node = movie["Sweet November",
           2000];

        Cast: Graph = <
                Keanu neo matrix,
                Leo jordan wolf,
                Keanu nelson sweet_nov
```

```
        >;

        point_to: List<Node>;
        point_to = Cast.find_many(Keanu, neo);

        print(point_to);

}
```

8.8.19 find_many3.n2n

```
actor: Node = { name: String, age: Int };
actedIn: Rel = { role: String };
movie: Node = { title: String, year: Int };

fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        Leo: Node = actor["Leo", 20];

        neo: Rel = actedIn["Neo"];
        jordan: Rel = actedIn["Jordan"];
        nelson: Rel = actedIn["Nelson"];

        matrix: Node = movie["Matrix", 1999];
        wolf: Node = movie["Wolf", 1994];
        sweet_nov: Node = movie["Sweet November",
           2000];

        Cast: Graph = <
                Keanu neo matrix,
                Leo jordan wolf,
                Keanu nelson sweet_nov
        >;

        point_from: List<Node>;
        point_from = Cast.find_many(neo, matrix);
```

```
        print(point_from);
        print("\n");

}
```

8.8.20 find_many4.n2n

```
actor: Node = { name: String, age: Int };
actedIn: Rel = { role: String };
movie: Node = { title: String, year: Int };

fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        Leo: Node = actor["Leo", 20];

        neo: Rel = actedIn["Neo"];
        jordan: Rel = actedIn["Jordan"];
        nelson: Rel = actedIn["Nelson"];

        matrix: Node = movie["Matrix", 1999];
        wolf: Node = movie["Wolf", 1994];
        sweet_nov: Node = movie["Sweet November",
           2000];

        Cast: Graph = <
                Keanu neo matrix,
                Leo jordan wolf,
                Keanu nelson sweet_nov
        >;

        node_lit: List<Node>;
        node_lit = Cast.find_many(actor["Keanu", _]);

        print(node_lit);
        print("\n");
```

```
}
```

8.8.21 graph.n2n

```
actor: Node = { name: String, age: Int };
actedIn: Rel = { role: String };
movie: Node = { title: String, year: Int };

fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        Leo: Node = actor["Leo", 20];

        neo: Rel = actedIn["Neo"];
        jordan: Rel = actedIn["Jordan"];

        matrix: Node = movie["Matrix", 1999];
        wolf: Node = movie["Wolf", 1994];

        Cast: Graph = <
                Keanu neo matrix,
                Leo jordan wolf
        >;

        print(Cast);

}
```

8.8.22 graph2.n2n

```
actor: Node = { name: String, age: Int };
actedIn: Rel = { role: String };
movie: Node = { title: String, year: Int };

fn main() -> Void {
```

```
        Cast: Graph = <
                actor["Keanu", 35] actedIn["Neo"] movie
                    ["Matrix", 1999],
                actor["Leo", 20] actedIn["Jordan"]
                    movie["Wolf", 1994]
        >;

        print(Cast);
}
```

8.8.23 graphInsDel.n2n

```
actor: Node = { name: String, age: Int };
actedIn: Rel = { role: String };
movie: Node = { title: String, year: Int };

fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        Leo: Node = actor["Leo", 20];

        neo: Rel = actedIn["Neo"];
        jordan: Rel = actedIn["Jordan"];

        matrix: Node = movie["Matrix", 1999];
        wolf: Node = movie["Wolf", 1994];

        Cast: Graph = <
                Keanu neo matrix,
                Leo jordan wolf
        >;

        print(Cast);
        print("\n");
        print("\n");
```

```
        Cast ^+ (Keanu actedIn["Nelson"] movie["Sweet␣
           November", 2000]);
        print(Cast);
        print("\n");
        print("\n");

        Cast ^- (Keanu actedIn["Nelson"] movie["Sweet␣
           November", 2000]);
        print(Cast);
        print("\n");
        print("\n");

}
```

8.8.24 if.n2n

```
fn main() -> Void {
        if (true) {
                print("success");
                print("\n");
        }
        else{

        }

        if(4<5){
                print(6);
                print("\n");
                if(0>1){
                        print(7);
                }
                else{
                        print("else");
                        print("\n");
                }
        }
}
```

8.8.25 map.n2n

```
actor: Node = { name: String, age: Int };
actedIn: Rel = { role: String };
movie: Node = { title: String, year: Int };

fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        Leo: Node = actor["Leo", 20];

        neo: Rel = actedIn["Neo"];
        jordan: Rel = actedIn["Jordan"];

        matrix: Node = movie["Matrix", 1999];
        wolf: Node = movie["Wolf", 1994];

        Cast: Graph = <
                Keanu neo matrix,
                Leo jordan wolf
        >;

        visited_node: List<Node>;
        mutated_graph: Graph = Cast.map(node in {node
           [+] visited:Bool;});

        Keanu.visited = true;

        ;;Need to be considered;;

        print(Keanu.visited);
        print(matrix.visited);
        print(Leo.visited);
        print(wolf.visited);
```

```
}
```

8.8.26 neighbor.n2n

```
actor: Node = { name: String, age: Int };
actedIn: Rel = { role: String };
movie: Node = { title: String, year: Int };

fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        Leo: Node = actor["Leo", 20];

        neo: Rel = actedIn["Neo"];
        jordan: Rel = actedIn["Jordan"];

        matrix: Node = movie["Matrix", 1999];
        wolf: Node = movie["Wolf", 1994];

        Cast: Graph = <
                Keanu neo matrix,
                Leo jordan wolf
        >;

        Keanu_movies: List<Node>;
        Keanu_movies = Cast.neighbors(Keanu);

        print(Keanu_movies);
        print("\n");

}
```

8.8.27 node.n2n

```
actor: Node = { name: String, age: Int };
```

```
fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        print(Keanu);
        print("\n");

}
```

8.8.28 nodeAccess.n2n

```
actor: Node = { name: String, age: Int };

fn main() -> Void {

        Keanu: Node = actor["Keanu", 35];
        Keanu.name = "Reeves";
        print(Keanu.name);
        print("\n");

}
```

8.8.29 nodeInsDel.n2n

```
actor: Node = { name: String, age: Int };

fn main() -> Void {
        Keanu: Node = actor["Keanu", 35];
        print(Keanu);
        print("\n");

        Keanu [+] visited: Bool;
        Keanu.visited = true;
        print(Keanu);
        print("\n");

        Keanu [-] visited: Bool;
```

```
        print(Keanu);
        print("\n");

}
```

8.8.30 rel.n2n

```
```

8.8.31 relAccess.n2n

```
actedIn: Rel = { role: String };

fn main() -> Void {

        neoRole: Rel = actedIn["Neo"];
        print(neoRole);
        print("\n");

}
```

8.8.32 simple.n2n

```
a:Int = 5;
b:String = "test";
c:Node = {name:String, age:Int};
fn t()->Void {}
;;comment;;
fn main()->Void {
        a:Int = 5;
        print(5);
}
```

8.8.33 simple2.n2n

```
fn main()->Void {
        a:Int = 5;
}
```