Knowledge Graph Language

Final Report

Due Friday, Dec 22 , 2015

Name	UNI	Role
Bingyan Hu	bh2447	Project Manager
Cheng Huang	ch2994	Language Guru
Ruoxin Jiang	rj2394	System Architect
Nicholas Mariconda	nm2812	Verification & Validation

Contents

1 Introduction

1.1 Motivation

Almost everything in the world is connected together through some complex web of relationships. As such, building, expressing and traversing graphs is one of 0the most essential applications of computer science. However, it is common knowledge that implementing graphs in traditional languages is no trivial task. Many past projects have addressed this problem by designing graph-based languages that make building graphs easier. However, such projects were limited by having single, fixed relationships between nodes. Often, algorithms that operate on real-world data – such as social network and information retrieval algorithms – are too obfuscated to be represented by a graph with one-dimensional relationships.

1.2 Overview

Knowledge Graph Language (KGL) is a domain-specific graphing language that supports multiple user-defined relationships between nodes. Edges, nodes, and graphs are built-in types of the language; however, two nodes can be connected by multiple edges, with each edge being identified by a unidirectional, user-defined relationship. KGL reaps many of the benefits of a graphing domain-specific language – users can build, express and traverse complex graphs succinctly – while also providing a means for users to query their graphs directly. This is the main thrust of the language – by providing the users with a mechanism for defining their own relationships between nodes, they can extract a more robust collection of data through graph queries.

2 Language Tutorial

2.1 Program Execution

Go to `src` folder, type `make` to create the kgl-to-c++ compiler. Then use the script `./kgl_run.sh` to compile and execute the source code. Note that the script `./kgl_run.sh` combines translating kgl to c++, compiling c++ code, and executing the c++ binary code into one step. Below is an example to run `helloworld.kgl`

Input: `./kgl_run.sh helloword.kgl`
Output:

```
graph g = {|
"Derrick"--("favorite")-->"Bikes";
"Sara"--("favorite")-->"Cats";
"Sara"--("favorite")-->"Bikes";
"Jill"--("favorite")-->"Bikes"
```

```
|}
```

Listing 1: helloworld.kgl

```
func void main(){
    ##initializes a graph with edge
    graph g = {|
    "Derrick"--("favorite")-->"Bikes";
    "Sara"--("favorite")-->"Cats";
    "Sara"--("favorite")-->"Bikes";
    "Jill"--("favorite")-->"Bikes"
    |};

    print(g);
}
```

2.2 Data Manipulation

2.2.1 Primitive Types

Listing 2: primitives.kgl

```
int a = 3+5;
float b = 3+5.0; #implicit conversion from int to float
char = 'a';
boolean c = true/false;
string d = "a" + "b";
```

2.2.2 List and Dictionary

Listing 3: listanddict.kgl

```
#List Initialization
list<char> h = [|'a','b','c'|]; #Initialize a list with literal
list<int> l;#Initialize a list with the bracket operator
l[0] = 1;l[1] = 2;

#List Manipulation
h[0] = 'a'; h[1] = 'b';#Access an element from list
list<int> l = [1,2,3]; list<int> h = [1,4]; #Adds two lists
l+=h  ----> [1,2,3,1,4];

#Dictionary Initialization
```

```
dict<string,int> d = (|"one":1,"two":2,"three":3,"four":4|); #Initialize a
        ↪ dictionary with literal
dict<string,int> d;
d["one"] = 1; d["two"] = 2; #Initialize a dictionary with the bracket operator

#Dictionary Manipulation
dict<string,int> d;d["one"] = 1; d["two"] = 2;#Access elements from dictionary
dict<string,int> e;e["three"] = 3; e["four"] = 4;
d+=e ----> (|"one":1,"two":2,"three":3,"four":4|); #Merges two dictionary
```

2.2.3 Node and Graph

Listing 4: nodegraph.kgl

```
node n = g["Jack"];#gets a node from the graph
## adds attributes to the node
n["DOB"] = "04/09/1993";
n["gender"] = "male";

##initializes a graph with edge
graph g = {|
    "Derrick"--("favorite")-->"Bikes";
    "Sara"--("favorite")-->"Cats";
    "Sara"--("favorite")-->"Bikes";
    "Jill"--("favorite")-->"Bikes"
|};

#merges two graphs with "+="
graph g = {|
    "Derrick"--("favorite")-->"Bikes";
    "Sara"--("favorite")-->"Cats";
|};

g+={|
    "Sara"--("favorite")-->"Bikes";
    "Jill"--("favorite")-->"Bikes"
|};

/# graph g is now:
{|
    "Derrick"--("favorite")-->"Bikes";
    "Sara"--("favorite")-->"Cats";
    "Sara"--("favorite")-->"Bikes";
    "Jill"--("favorite")-->"Bikes"
```

```
|};
#/

graph g = {|
    "Derrick"--("own")-->"Bikes";
    "Sara"--("favorite")-->"Cats";
    "Sara"--("favorite")-->"Bikes";
    "Jill"--("favorite")-->"Bikes"
|};

#Node Built-in Functions
getOutNeighbors(g["Sara"],"favorites"); ----> "Bikes" "Cats"
getInNeighbors(g["Bikes"],"favorites"); ----> "Sara" "Jill"

#Graph Built-in Funcitons
getNodes(g); ----> "Derrick" "Bikes" "Sara" "Jill" "Cats"
getLabels(g); ----> "facorite" "own"

#checks if node "Jack"      #checks is graph g has
#has "DOB" attribute        #node n
node n = g["Jack"];         graph g;
string attr;                node n;
if("DOB" in n){};           for(n in g){};
                            if(n in g){};
```

2.3 Control Flow

Listing 5: ctrlflow.kgl

```
int a = 5;
graph g;
node n;
#if-else block
#if graph g contains node
if(n in g){
    #do task
}
else{
    #do task
}

#while loop
while(a!=10){
    a=a+1;
```

```
}

#for loop
#regular for loop
int i;
for(i=0;i<a;i=i+1){
    #do task
}

#enhanced for loop
#traverses all the nodes in a graph
node n;
for(n in g){
}
```

2.4 Function Calls

Listing 6: func.kgl

```
func void main()
{
    #defines a graph
    graph g = {|
    "Derrick"--("favorite")-->"Bikes";
    "Sara"--("favorite")-->"Cats";
    "Sara"--("favorite")-->"Bikes";
    "Jill"--("favorite")-->"Bikes"
    |};

    #examples of function calls
    list<node> nodes = getNodes(g);#returns all the nodes in graph g
    list<string> nodes = getLabels(g);#returns all the relations in g
}
```

3 Language Manual

3.1 Lexical Conventions

3.1.1 Comments

The characters ## starts a single-line comment, which terminates at the end of the line. The characters /# introduces a multi-line comment, which terminates with characters #/. Comments do not nest.

3.1.2 Identifiers

An identifier is a combination of letters , numbers and underscores _. The first character must be a letter. Upper and lower case letters are different.

3.1.3 Keywords

The following is a list of reserved keywords in the language. They cannot be used as variable or names:

int	float	char	boolean	string
graph	node	list	dict	return
for	while	continue	break	in
if	else	true	false	null
func	void			

3.1.4 Literals

3.1.4.1 Integer Literal

An integer literal consists of an optional minus sign, followed by a sequence of digits. If the digit sequence has more than one digit, the first may not be zero. The literals are interpreted as base-10 (decimal) numbers. Examples: `76 -65 43445 0 -9090`

3.1.4.2 Float Literal

A float literal consists of a signed integer part, a decimal part and a fraction part. The integer and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing. Examples: `-1.36 67.0 .67 8.9`

3.1.4.3 Boolean Literal

A boolean literal has two values: `true`, `false`

3.1.4.4 Char Literal

A char literal is a single character enclosed in single quotes. The following escape sequences may be used: \n, \t, \\. Examples: `'a' ' ' 'H' '7'`

3.1.4.5 String Literal

A string literal consists of a sequence of characters enclosed in double quotes. The following characters can be escaped inside of strings with a backslash: \n, \t, \\. Examples: `"hello" "" "367"`

3.1.5 Punctuation

Colon :
separator of a key : value pair in dictionary

Semicolon ;
end of statement
separator of edge list in graph

Parenthesis ()
expression precedence
conditional parameters
function arguments

Brackets [ ]
node access
list/dictionary access

Curly braces {}
statement blocks
function body

Angle brackets <>
element type of derived types (list/dict)

Comma ,
separator of function arguments
separator of elements in list and key value pairs in dictionary

Edge brackets −( )−>
edge expression

List brackets [| |]
list expression

Dict bracket (| |)
dictionary expression

Graph brackets {| |}
graph expression

Quotes ' '
character literal declaration

Double quotes " "
string literal declaration

3.1.6 Operators

The following table lists the precedence and associativity of the operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	()	Function call	
1	[ ]	list/dictionary/graph access, attribute access in node	
2	!	Logical NOT	
3	* / %	Multiplication, division, remainder	
4	+ -	Addition, Subtraction	Left-to-right
5	>>= <<=	Relational greater, greater equal, less, less equal	
6	== !=	Relational equal, not equal	
7	in	Membership	
8	&&	Logical AND	
9	||	Logical OR	
10	=		Right-to-left

3.1.7 Whitespace

Symbols that are considered to be whitespace are blank, tab, and newline characters.

3.2 Types

3.2.1 Primitive Types

int	a signed 32-bit integer
float	a signed single precision floating point type, with a size of 32 bits
char	an 8-byte data type used to store ASCII characters
boolean	a 1-byte data type that only accepts true or false
string	a sequence of chars

3.2.2 Graph-related Types

3.2.2.1 Node

Nodes are building blocks of a graph. Each node in a graph contains an unique id (different from other node ids in the same graph) and a dictionary of user-defined attributes. A node can only exist in a graph; there is no isolated node. A node in a graph could only be created when a new edge(relationship) of this node is added in the graph.

The `node` type is only a pointer to a real node in a graph. It can either point to a node in graph or be `null`. A variable of type `node` can access and modify the real node it points to in graph.

For example:

```
## a points to the node with id "Nick" in graph g
node a = g["Nick"];
## returns the value of key "age" in the node's attribute dictionary
a["age"];
## sets the value of key "age" to 22 in the node's attribute dictionary
a["age"] = 22;
```

3.2.2.2 Graph

A graph is defined by a set of nodes and their unidirectional relationships between each other. An unidirectional relationship between two nodes is also called an edge, defined as such:

```
source_id--(label)-->target_id
## the relationship between the source node and the target node is of value label

"Nike"--("knows")-->"Mike"
## represents the "knows" relationship from node "Nike" to node "Mike"
```

3.2.3 Derived Types

Besides the primitive types, `graph` and `node`, there is a conceptually infinite class of derived types constructed from any type:

list<vtype >	A list is an ordered sequence of elements of a given type (vtype)
dict<ktype, vtype >	A dictionary is a collection of key value pairs. Duplicated keys are not permitted.

3.2.4 Void

void can only be used as function return types to indicate that a function has no return value.

3.3 Expressions

3.3.1 Primary expressions

Primary expressions are literals, identifiers, graph expressions, list expression, dict expressions or expressions in parentheses.

```
primary-expression:
    literal
    identifier
    null
    (expression)
```

3.3.1.1 Literals

```
literal:
    int-literal
    float-literal
    boolean-literal
    char-literal
    string-literal
    list-expression
    dict-expression
    graph-expression
```

3.3.1.2 List/Dictionary expressions

A list expression is an possibly-empty ordered sequence of expressions.

```
list-expression:
    [| expression-list-opt |]
```

A dictionary expression is possibly-empty collection of key/value pairs. Each key/value pair contains two expressions, the key and the value, separated by a colon.

```
dict-expression:
    (| kv-pairs-opt |)

kv-pair:
    expression : expression
```

3.3.1.3 Graph expressions

A graph expression is a possibly-empty collection of edges(relationships). Each edge contains three expressions: the id string of source node, the label string of this relationship, and the id string of target node

```
graph-expression:
    (| edge-list-opt |)

edge:
    expression--(expression)-->expression
```

3.3.2 Postfix expressions

Postfix expressions are primary expressions, function calls, and list/dict/graph references

```
postfix-expression:
    primary-expression
    postfix-expression (argument_list_opt}
    postfix-expression [expression]
```

3.3.2.1 References

A posfix expression followed by an expression in square brackets is a postfix expression denoting a reference to list element/graph node /dictionary value/ node attribute.

Reference to	Syntax	Description
list element	list_var[3]	the 4th element in list list_var
graph node	graph_var["Nick"]	the node with id "Nick" in graph graph_var
dictionary value	dict_var["k"]	the value of key "k" in dictionary dict_var
node attribute	node_var["age"]	the value of attribute "age" in node node_var

3.3.2.2 Function calls

A function call is an function identifier, followed by parentheses with a possibly empty, comma-separated argument list.

3.3.3 Operator expressions

Unary Operator

```
unary-expression:
      postfix-expression
      ! expression
```

Multiplicative Operators

```
multiplicative-expression:
       unary-expression
       multiplicative-expression * mulplicative-expression
       multiplicative-expression \ mulplicative-expression
       multiplicative-expression % mulplicative-expression
```

Additive Operators

```
additive-expression:
       multiplicative-expression
       additive-expression + additive-expression
       additive-expression - additive-expression
```

Relational Operators

```
relational-expression:
       additive-expression
       relational-expression < relational-expression
       relational-expression > relational-expression
       relational-expression <= relational-expression
       relational-expression => relational-expression
```

Equality Operators

```
equality-expression:
       relational-expression
       equality-expression == equality-expression
       equality-expression != equality-expression
```

Logical AND Operator

```
logical-AND-expression:
       equality-expression
       logical-AND-expression && logical-AND-expression
```

Logical OR Operator

```
logical-OR-expression:
       logical-AND-expression
       logical-OR-expression || logical-OR-expression
```

Assignment expressions

```
assignment-expression:
       logical-OR-expression
       assignment-expression = assignment-expression
       assignment-expression += assignment-expression
       assignment-expression -= assignment-expression
```

3.3.3.1 List and Dict

Only the following operators have meanings for list-expression and dict-expression: ! +/+= == != =

Operand	Operator	Operand	Description
	!	list/dict-expression	true if list/dict is empty, false otherwise
list-expression	+/+=	list-expression	concatenation of two lists
dict-expression	+/+=	dict-expression	union of two dictionaries
list/dict-expression	==	list/dict-expression	true if they contain the same elements, false otherwise
list/dict expression	!=	list/dict expression	true if they do not contain the same elements, true otherwise

3.3.3.2 Graph and Node

Only the following operators have meanings for graph-expression and node: ! + - == != =

Operand	Operator	Operand	Description
	!	node	true if node is null, false otherwise
node	==	node	true if they point to the same node in the same graph
node	!=	node	false if they point to the same node in the same graph
	!	graph-expression	true if graph is empty, false otherwise
graph-expression	+/+=	graph-expression	union of two graphs
graph-expression	-/-=	graph-expression	graph of edges in the first but not in the second
graph-expression	==	graph-expression	true if they contain the same nodes and relationships
graph-expression	!=	graph-expression	false if they contain the same nodes and relationships

Examples:

```
## add nodes and relationships to a graph
graph g = {||};
## adds two new nodes and a relationship to g
g +={|"Nick"--("friends")-->"Mike"|};
## adds a new relationship to g
g +={|"Nick"--("knows")-->"Jack"|};
## add an attribute to node
string key = "gender"; string value = "male";
g["Nick"][key] = value;
## remove nodes / relationship from graph
g -= {|"Nick"--("knows")-->"Jack"|};
## node "Jake" which is connected to no other nodes in g
## is deleted from the graph as well.
```

3.4 Declarations

Declarations defines the types of identifiers and initializes their values.

```
declaration:
    type-specifier identifier
    type-specifier identifier = initializer
```

3.4.1 Type Specifiers & Initializer

The type-specifiers are

```
type-specifier:                 initializer:
    void                            N/A (void only used for func return type)
    int                             expression evaluated to int
    float                           expression evaluated to float
    boolean                         expression evaluated to boolean
    char                            expression evaluated to char
    string                          expression evaluated to string
    graph                           graph-expression
    node                            reference to node in graph
    dict<type-specifier>
    <type-specifier>                dict-expression
    list<type-specifier>            list-expression
```

3.4.2 Graph and Node

The following examples shows the declaration and initialization of graphs and nodes.

```
/# Syntax of graph declaration and initialization
graph g = {| sourceID--(edgeLabel)-->targetID;
             sourceID--(edgeLabel)-->targetID;...|};
#/

## declare a graph g1
graph g1;

## declare a graph g2 and initialize it with two relationships
graph g2 = {|
             "Nick"--("friends")-->"Mike";
             "Mike"--("knows")-->"Jake"
            |};

## declare and initialize another graph g3
string rel = "knows";
string sourceID = "Nick"; string targetID = "Mike";
graph g3 = {| sourceID--(rel)-->targetID |};

## declares a node
node n1;

## declare and initialize nodes
node n2 = g2["Nick"];
node n3 = g3[targetID];
```

3.4.3 List and Dict

```
list<int> l1 = [||] ## declare an empty integer list
list<float> l2 = [|1.0, 2.0, 3.0 |] ## declare an float list with three elements
dict<string, int> d = (||) ## declare an empty dictionary
dict<string, int> d = (|"num": 1, "age": 22, "friends": 5|) ## declare a dict wit
```

3.5 Statements

```
statement:
    block-statement
    expression-statement
    conditional-statement
    loop-statement
    jump-statement
```

3.5.1 Block statements

```
block-statement:
    { statement-list-opt }
```

3.5.2 Expression statements

Expression statements are mostly used as assignments or function calls.

```
expression-statement:
    expression;
```

3.5.3 Conditional statements

A conditional statement is used to express decisions.

```
conditional-statement:
    if (expression-opt) statement
    if (expression-opt) statement else statement
```

3.5.4 Loop statements

Loops are control statements that specify iteration, which allow a block of code ("sub-statement") to be executed multiple times. KGL supports two types of loops: the `for` loop and the `while` loop.

```
loop-statement:
    for (expression-opt; expression-opt; expression-opt) statement
    for (expression in expression) statement
    while (expression) statement
```

3.5.4.1 For loop

The `for` loop supports two separate usages. The first is the standard usage in which the for loop takes three expressions: an initialization expression, a test expression, and an update expression.

The second usage of the `for` loop resembles the Pythonic implementation – it executes the substatement for each element in a given collection. The first expression is a user-specified label that will serve as a reference to the current element in the collection, and the second expression is the handle for the collection being iterated through. Supported collections include lists, dictionaries and graphs.

3.5.4.2 While loop

The `while` loop is the `for` loop stripped of the initialization and update expressions. It contains the test expression and statement. The statement is executed repeatedly until the test expression is evaluated to a boolean false. The expression is reevaluated before each iteration of the loop.

3.5.5 Jump Statements

```
jump-statement:
    continue;
    break;
    return expression_opt;
```

3.5.5.1 Continue

`continue` is used to skip some statements in the iteration loops and cause control to pass to the loop-continuation portion of the smallest enclosing such statement. For example:

```
while (expr) {
    ...
    if (expr) { continue; }
    ...
}
```

3.5.5.2 Break

`break` statement is used to terminate all iteration loops. For example

```
while (expr) {
    ...
    if (expr) { break; }
    ...
}
```

3.5.5.3 Return statements

The `return` statement terminates the execution of a function and returns control to the calling function. When `return` is followed by an expression, the value is returned to the caller of the function.

3.6 Built-in Functions

3.6.1 Node

- `getName(node v)`: return the name of node v
- `getOutNeighbors(node v, string label)`: returns the list of all nodes pointed from node v by the relationship 'label'
- `getInNeighbors(node v,string label)`: returns the list of all nodes pointing to node v by the relationship 'label'
- `getOutNeighbors(node v)`: returns the list of all labels pointing from this node
- `getInNeighbors(node v)`: returns the list of all labels pointing to this node
- `getAttributes(node v)`: return the dictionary of attributes of node v

3.6.2 Graph

- `getNodes(graph g)`: return list of nodes in graph g
- `getLabels(graph g)`: returns the list of all relationship labels in graph g

3.6.3 Graph

- `getSize(list l)`: return the length of list l

4 Project Plan

4.1 Process

4.1.1 Planning and Developing

Our team met weekly to discuss and work on the project. We often started with recapping last week's progress, setting up weekly goals for each member and bringing up any problems we encountered at current stage so we could discuss with our TA Aquila Khanam. By meeting and working as a whole group weekly, we maintained a highly communicative working environment to make sure that all members dedicate complete focus on each stage and cooperate together while staying ahead of all mandatory deadlines. After getting our proposal and LRM back, we had a hard time revising our plan. We tended to focus on the data structure and possible features users could implement using our language, so we had a harder and longer time working on our semantic checking and code generation to best meet our goals. We often referred back to our LRM and made modification to make sure that we are in align with our plan.

4.1.2 Testing

We developed a complete test suite during our compiler development. The test suite is a mixed of unit, regression and integration testings to fully test the functionality of our language. As we wrote each component, we created unit test to evaluate each piece of new code and found out how it reacted to valid or invalid inputs. By doing so, we adopted and accumulated a regression suite of unit tests to detect unexpected changes by either modifying our existing module or adding new components. We also conducted integration test combing components together to verify the communication and functionality are working properly end-to-end.More details about our test suites are included in next section.

4.2 Team Responsibilities

Team Member	Responsibility
Bingyan Hu	Language Development and Documentation
Ruoxin Jiang	Compiler Front-end and Code Generation
Cheng Huang	Code Generation and Testing
Nicholas Maricond	Code Generation and Testing

As the table showed above, the main responsibilities were assigned roughly between our four members; however, there was no strict division of responsibilities as the team actually met and worked together weekly. Roles were overlapped as we coded in pairs and helped with each other to fix bugs.

4.3 Program Timeline

Time	Task Completed
September 18	Brainstorm
September 26	Finished Project Proposal
October 23	Finished Lexer and Parser
October 25	Finished Language Reference Manual; AST
November 16	Hello World Demo
December 10	Finished Semantic Checking
December 18	Finished Code Generation
December 19	Finished Testing
December 22	Project Presentation; Submitted Final Project Report

4.4 Software Development Environment

Version Control: Bitbucket-Hosted Git Repository

Development System: Mac OS & Linux

Programming Language: Ocaml 4.02.1; Ocamlyacc & Ocamllex; C++

Tools: Sublime, iTerm, Terminal, Vim, Emacs.

4.5 Project Log

See Appendix 8.1

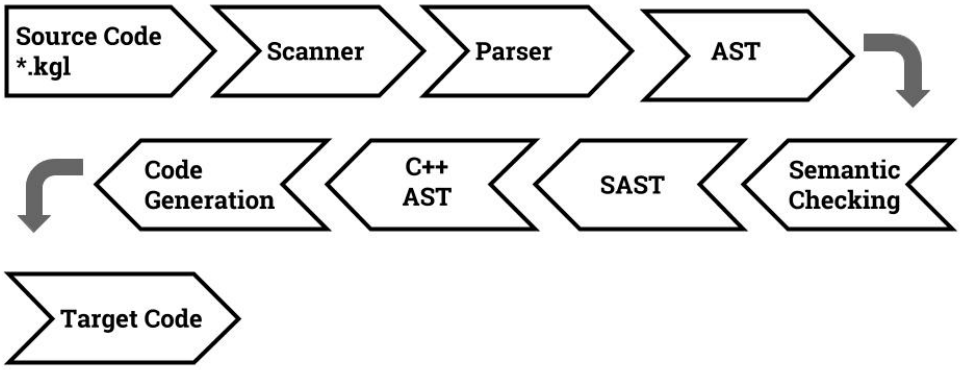

Figure 1: Architecture Design Flow Chart

5 Architecture Design

As is shown in the diagram, the compiler pipleine of KGL language goes through five sections: the Scanner, Parser, Semantic Checker, C++ AST Generator, Code Generator.

5.1 Scanner (scanner.mll)

The Scanner takes in a *.kgl source program and produces a tokenized output. The Scanner discards all whitespaces and comments and throws an error if any invalid characters are encountered. The Scanner was written for `ocamlyacc`.

5.2 Parser (parser.mly, ast.ml)

The Parser inputs the token stream and outputs an abstract syntax tree (AST). The structure of AST is definied in `ast.ml` and the AST generation part is in `parser.mly` The Parser was written for `ocamllex`.

5.3 Semantic Checker (semantic.ml, sast.ml)

The Semantic Checker recursively traverses the AST and checks the type and semantics of each node. It binds type information to raw AST to generate an extended SAST. The structure of the new SAST is defined in `sast.ml` and the semantic checking and SAST generation is in `semantic.ml`.

5.4 C++ AST Generator (cast.ml)

The C++ AST Generator parses the SAST and produces the corresponding C++ AST. A node in SAST will be transformed into different C++ AST nodes depending on the type information. For example, a plus operator node in SAST will be transformed into different C++ calls depending on the types of its operands (polymorphism of list/dictionary/graph types).

5.5 Code Generator (cast.ml)

`cast.ml` also generates C++ code from C++ AST. It walks through all nodes in C++ AST and prints the corresponding code.

5.6 C++ Library

5.6.1 list.h

Contains a set of functions to perform the following tasks.

- checks if the list contains a given element
- returns the size of this list
- adds an element to a specified position in the list
- removes a given list from this list
- merges two lists

5.6.2 dict.h

Contains a set of functions to perform the following tasks.

- removes a key-value pair from the dictionary
- checks if the dictionary contains a given key
- removes an attribute from a given node
- removes a given list from this list
- merges two dictionaries

5.6.3 graph.h

- defines a node type and associated operations
- defines a graph type and associated operations

6 Test Plan

6.1 Demo Code

See Appendix 8.4

6.2 Test Suites

We implemented full regression testing throughout the entire development of our language, from the parser all the way to the semantic checker. We began by piping the scanner's output from some basic KGL files into menhir, and finished with a robust test suite that contains over 100 tests to ensure our language implements all the features discussed in the manual. We organized our test suite into many subdirectories, each containing tests for a specific feature of our language, such as tests for the "for" loop or tests for the "list" object. Our testing script was designed to recursively traverse this directory and display the results of all the tests.

6.3 Test Case Explanation

Our general strategy in designing test cases was to make each test as short and focused as possible, so that each tests a small, specific aspect of our language. This allowed us to easily determine the problem within our language whenever a test failed.

We performed two types of tests: tests that are meant to pass compilation and tests that are meant to fail compiliation. The former pass by generating output that matches the corresponding output file in the 'outputs' directory in our test suite. If no corresponding output file is found, one is automatically generated and added to the directory. The testing script prints a flag when this occurs, so the tester can check to ensure the correct output file was created. This allows new tests to be effortlessly added to our test suite, and builds an inherent form of regression testing.

Tests that are designed to fail compilation begin with the characters "x_". They are marked as a success when they fail to compile. We paired each fail case with a corresponding success case, changing only the part that should cause the failure. This minimized the possibility of a fail case failing to compile for the wrong reason.

7 Lessons Learned

7.1 Bingyan Hu

Start early and always keep starting early! As a project manager, I was responsible for planning and organizing the project and the development of our language. Our team was able to work on writing the compiler quite early initially because we came up with a relatively clear goal of what our language would be like and we set up weekly meeting to pair

program together. The feedback from helloworld demo pushed us further to rethink about our language features and we came to focus on the possible algorithms we want to show in our final demo. The revision was hard when coming into details even though we had a large picture ahead. At the late half stage, we underestimated the amount of work implementing specific features and lacked the detailed plan of our language structure. We was working extensively and intensively the last week to finish the most of the work. One thing I wish to do differently was to keep our initial good pace at the late half stage so we could get our basic compiler working before adding more sophisticated features to enrich our language.

7.2 Ruoxin Jiang

One lesson I learned from my system architect role is initializing a reasonable language reference manual with very detailed but not too fancy features. These features should be carefully planned with our implementation methods(syntactics, semantics, grammar and etc) so we could have an easier time writing our compiler at later stage. Besides the hard deadlines for our milestones, it would be better to set up mini deadlines for each milestone and stick with them.

7.3 Cheng Huang

For this project my tasks mainly focused on the code generation part. Specifically, I was responsible for creating the c++ libraries. One thing I learned was to have a clear design of the program architecture before implementation. When I was implementing the program, I did not have a clear plan on which functions to implements and what data structures to use. The consequence was that during implementation phase, I had to modify or, in the worst case, rewrite the existing codes whenever the design changed. Skipping to implementation with a clear plan proved to be a time-consuming mistake.

7.4 Nicholas Mariconda

The initial test strategy was to write large and complex programs that included all features of our language. This, strategy, however, turned out to be less efficient since it was hard to trace the origins which caused the error and it was difficult to come up with programs that contains all features of the language. Later, a new strategy was developed. It contains a large number of small and simple test cases which are designed to test specific features listed in the language requirement manual. Being a tester gave me the chance to learn how to develop comprehensive test cases by dividing the problem into smaller parts and conquering each separately.

8 Appendix

8.1 Project Log

```
f8b25a9 2015-12-22 added comment and cleaned up code | Ruoxin Jiang (rj2394)
02bb340 2015-12-22 changed _deleteGraph to _minusGraph | Cheng Huang
1f8e971 2015-12-22 added two _deleteGraph() functions | Cheng Huang
f2c09b1 2015-12-21 removed redundant test cases | Ruoxin Jiang (rj2394)
d6bbbdc 2015-12-21 added support for char; deleted redundant test cases | Ruoxin Jiang (
4e9e452 2015-12-20 revised test cases | Ruoxin Jiang (rj2394)
19f984c 2015-12-20 clean up | Ruoxin Jiang (rj2394)
d0b972d 2015-12-20 add attribute demo | Ruoxin Jiang (rj2394)
d40a492 2015-12-20 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
8aa8ea2 2015-12-20 added recommend_friend | Cheng Huang
d9777a3 2015-12-20 add built-in function getName | Ruoxin Jiang (rj2394)
3d0d3ff 2015-12-20 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
6a8ca76 2015-12-20 find_path demo works | Ruoxin Jiang (rj2394)
7d6cef8 2015-12-20 testing | Nicholas Mariconda
daa51b3 2015-12-20 why it didn't work | Ruoxin Jiang (rj2394)
4264330 2015-12-20 friend_of_friend works | Ruoxin Jiang (rj2394)
0f7fde1 2015-12-20 friend_of_friend test | Ruoxin Jiang (rj2394)
4e39af5 2015-12-20 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
7eac246 2015-12-20 changed map to unorderde_map in dict.h | Cheng Huang
f2a65d7 2015-12-20 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
218ad10 2015-12-20 added wrapper function to get the neighbors of a node | Cheng Huang
8a0ac57 2015-12-20 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
e223781 2015-12-20 added wrapper function to create a graph | Cheng Huang
eee673e 2015-12-20 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
738f56d 2015-12-20 revised graph | Ruoxin Jiang (rj2394)
2b10929 2015-12-20 added wrapper for getOutNeighbor() | Cheng Huang
b401941 2015-12-20 added warpper for insertEdge | Cheng Huang
953deea 2015-12-20 add test | Ruoxin Jiang (rj2394)
e88ef2d 2015-12-20 add newGraph func | Ruoxin Jiang (rj2394)
bf6e465 2015-12-20 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
1bc2c48 2015-12-20 added wrapper functions | Cheng Huang
41c56b7 2015-12-20 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
85def7e 2015-12-20 added test option in test script | Ruoxin Jiang (rj2394)
4a54e97 2015-12-20 compiled cgen cast | Ruoxin Jiang (rj2394)
3909fc1 2015-12-20 updates on library files | Cheng Huang
7900e67 2015-12-20 compiled cast | Ruoxin Jiang (rj2394)
e0a057c 2015-12-20 added recommend_friend.cpp | Cheng Huang
800f1d3 2015-12-20 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
```

```
42dfb0a 2015-12-20 modified two library files | Cheng Huang
c268520 2015-12-19 partially work cast | Ruoxin Jiang (rj2394)
81f090b 2015-12-19 added c++ ast; gcd passed | Ruoxin Jiang (rj2394)
5207690 2015-12-19 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
859ef20 2015-12-19 finished c++ library files. Needs to test | Cheng Huang
7b3eb15 2015-12-19 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
1b8bc22 2015-12-19 added part of c++ ast | Ruoxin Jiang (rj2394)
412b7b2 2015-12-19 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
be5395d 2015-12-19 added notes to fancyPanda | Cheng Huang
ab551d7 2015-12-19 revised demos | Ruoxin Jiang (rj2394)
56f9230 2015-12-19 add new demo: find path | Cheng Huang
ed68608 2015-12-19 added two demo programs | Ruoxin Jiang (rj2394)
63b6911 2015-12-19 new features | Ruoxin Jiang (rj2394)
6cdc240 2015-12-18 revised cgen | Ruoxin Jiang (rj2394)
d27a131 2015-12-05 added list.h | Cheng Huang
86b2ed1 2015-12-05 added functions to graph.h and created list.h | Cheng Huang
18468f8 2015-12-05 conclude merge | Cheng Huang
a47291e 2015-12-05 added semantic checks (TODO: Dict) | Ruoxin Jiang (rj2394)
3fbb578 2015-12-05 added test case | Ruoxin Jiang (rj2394)
cd19023 2015-12-05 added semantic checks | Ruoxin Jiang (rj2394)
85abb41 2015-12-05 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
6b352c5 2015-12-05 added test cases for semantic checking | Ruoxin Jiang (rj2394)
4d18341 2015-12-05 demo v1.1 | Bingyan
698d50a 2015-11-16 remove error msg from test script | Cheng Huang
9a2034d 2015-11-16 added test case | Ruoxin Jiang (rj2394)
b0abea6 2015-11-16 revised cgen semantic and test header | Ruoxin Jiang (rj2394)
635972c 2015-11-16 add c++11 extension when comile | Ruoxin Jiang (rj2394)
230e834 2015-11-16 added the test folders | Cheng Huang
7e0089c 2015-11-16 added function gotest.sh to automatically generate c++ executable | C
bbe176f 2015-11-16 renamed kpl.ml | Ruoxin Jiang (rj2394)
5e4e98e 2015-11-16 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
07d109b 2015-11-16 solved the print char error | Ruoxin Jiang (rj2394)
7e22c18 2015-11-15 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
7681537 2015-11-15 fixed typos in ask.ml | Cheng Huang
c0d5e6d 2015-11-15 test C++ library header for print functions | Ruoxin Jiang (rj2394)
d86eac4 2015-11-15 unfinished semantic checking and code generation | Ruoxin Jiang (rj23
28a745c 2015-11-15 revised Makefile | Ruoxin Jiang (rj2394)
30418dc 2015-11-15 changed kpl | Ruoxin Jiang (rj2394)
7bf6752 2015-11-15 compiled yet unfinished semantic checker | Ruoxin Jiang (rj2394)
28fde40 2015-11-15 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
38574a5 2015-11-15 revised parser | Ruoxin Jiang (rj2394)
0c762b1 2015-11-15 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
```

```
df5e159 2015-11-15 added test file test ast tree | Cheng Huang
734f014 2015-11-15 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
8fd90d6 2015-11-15 add print.h | Bingyan
e0f9efd 2015-11-15 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
a8f3ae0 2015-11-15 added printing ast functionality | Cheng Huang
2c4373f 2015-11-15 add main.kgl | Bingyan
9b8d9a8 2015-11-14 Merge branch 'master' of https://bitbucket.org/pltteam2015/plt_projec
c56515c 2015-11-14 added test script | Cheng Huang
0b17a46 2015-11-14 add file{graph.h}: graph library template | Bingyan
ddee360 2015-11-13 added print ast statements | Cheng Huang
e88b827 2015-10-25 changed elif_list rules | Ruoxin Jiang (rj2394)
d03ea08 2015-10-25 add support for list, break, continue, and node attribute assign | Ru
cc85d29 2015-10-25 revised scanner type | Ruoxin Jiang (rj2394)
793381e 2015-10-25 revised scanner | Ruoxin Jiang (rj2394)
750860b 2015-10-25 added list type | Ruoxin Jiang (rj2394)
5c7cd9a 2015-10-25 compiled scanner/parser/ast | Ruoxin Jiang (rj2394)
7ae7814 2015-10-24 compiled ast | Ruoxin Jiang (rj2394)
1cdadb4 2015-10-24 Compiled scanner and parser (need testing | Ruoxin Jiang (rj2394)
7fe07f4 2015-10-24 Revised Makefile | Ruoxin Jiang (rj2394)
61099f3 2015-10-23 add graph/dict expression to parser | Ruoxin Jiang (rj2394)
def3664 2015-10-23 added parser.mly | Ruoxin Jiang (rj2394)
ba690bc 2015-10-23 added unfinished Makefile | Ruoxin Jiang (rj2394)
fb5bd89 2015-10-23 compiled scanner.mll | Ruoxin Jiang (rj2394)
```

8.2 Source Code

/src/scanner.mll

```
(*
 * KGL scanner
 *)


{ open Parser }

let digit = ['0'-'9']
let letter = ['a'-'z' 'A'-'Z']
let flt = '-'? digit+ '.' digit* | '-'? '.' digit+
let identifier = letter (letter | digit | '_')*

rule token = parse

  (* Whitespace *)
  | [' ' '\t' '\n' '\r'] { token lexbuf }

  (* Comments *)
  | "/#" { comment lexbuf }
  | "##" { comment_line lexbuf }

  (* Puncutation *)
  | '('  { LPAREN }     | ')'   { RPAREN }
  | '{'  { LBRACE }     | '}'   { RBRACE }
  | '['  { LBRACK }     | ']'   { RBRACK }
  | ';'  { SEMI  }      | ':'   { COLON }
  | ','  { COMMA }
  | "[|" { LLIST }      | "|]"  { RLIST }
  | "(|" { LSET  }      | "|)"  { RSET  }
  | "{|" { LGRAPH }     | "|}"  { RGRAPH }
  | "--(" { LEDGE }     | ")-->" { REDGE }


  (* Arithmetic operators *)
  | '+'  { PLUS }       | '-' { MINUS }
  | '*'  { MULTI }      | '/' { DIVIDE }
  | '%'  { MOD  }

  (* Logical operators *)
  | "&&" { AND }
  | "||" { OR  }
  | "!"  { NOT }
```

```
43
44  (* Assignment operator *)
45  | "="  { ASSIGN }
46  | "+=" { ADD }
47  | "-=" { SUB }
48
49  (* Relational operators *)
50  | "==" { EQUAL }      | "!=" { NEQ }
51  | ">=" { GEQ }        | "<=" { LEQ }
52  | '>'  { GT }         | '<' { LT }
53
54  (* Loop keywords *)
55  | "for"      { FOR }
56  | "while"    { WHILE }
57  | "continue" { CONTINUE }
58  | "break"    { BREAK }
59
60  (* Conditional keywords *)
61  | "if"       { IF }
62  | "else"     { ELSE }
63
64  (* Function keywords *)
65  | "func"     { FUNC }
66  | "return"   { RETURN }
67
68  (* Type keywords *)
69  | "int"      { INT  }
70  | "float"    { FLOAT }
71  | "char"     { CHAR }
72  | "boolean"  { BOOL }
73  | "string"   { STR  }
74  | "graph"    { GRAPH }
75  | "node"     { NODE }
76  | "list"     { LIST }
77  | "dict"     { DICT }
78
79
80  (* Other keywords *)
81  | "in"       { IN }
82  | "null"     { NULL }
83  | "void"     { VOID }
84
85  (* Literals *)
86  | "true"            { BOOL_LIT(true) }
87  | "false"           { BOOL_LIT(false) }
```

```
  | identifier as lit { ID(lit) }
  | ('0' | '-'? ['1'-'9']digit*) as lit
                    { INT_LIT(int_of_string lit)}
  | flt as lit      { FLOAT_LIT(float_of_string lit) }
  | '"' (([' '-'!' '#'-'[' ']'-'~'] | '\\' ['\\' '"' 'n' 't'])* as lit) '"'
                    { STR_LIT(lit) }
  | '\'' ([^'\''] as lit) '\''
                    { CHAR_LIT(lit) }
  | "\'\\\\\'"      { CHAR_LIT('\\') }
  | "\'\\t\'"       { CHAR_LIT('\t') }
  | "\'\\n\'"       { CHAR_LIT('\n') }

  (* Illegal characters *)
  | _ as c { raise (Failure("Illegal character: " ^ Char.escaped c)) }

  (* End of file *)
  | eof { EOF }

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

and comment_line = parse
  | '\n' { token lexbuf }
  | eof { EOF }
  | _   {comment_line lexbuf}
```

/src/ast.ml

```
(*
 * KGL Abstract Syntax Tree types
 *
 *)


(* Binary operators *)
type op = Plus | Minus | Multi | Divide | Mod
        | Equal | Nequal | Less | Leq  | Greater | Geq
        | In    | And   | Or

(* Assignment operators *)
type asnop = Add | Sub | Asn


(* Expressions *)
type expr =
```

```
18      IntLit of int
19    | FloatLit of float
20    | BoolLit of bool
21    | CharLit of char
22    | StrLit of string
23    | Id of string
24    | Assign of expr * asnop * expr (* Assignment: expr1 =/+=/-= expr2 *)
25    | Not of expr
26    | Binop of expr * op * expr
27    | Call of string * expr list  (* Function call: f(param1, param2,...) *)
28    | Value of expr * expr        (* Reference: expr1[expr2] *)
29    | ListLit of expr list        (* List literal: [| e1, e2, e3, ... |] *)
30    | DictLit of kv list          (* Dict literal: (| k1:v1, k2:v2, ... |) *)
31    | GraphLit of edge list       (* Graph literal: {|src--(label)-->dest; ...|}*)
32    | Null
33    | Noexpr
34
35  and kv = {
36          key: expr;
37          value: expr;
38  }
39
40  and edge = {
41          src: expr;
42          dest: expr;
43          label: expr;
44  }
45
46
47  (* Variable Types and Declaration *)
48  type var_type =
49            Int   | Float | Bool | Char | Str
50          | Graph | Node
51          | List of var_type
52          | Dict of var_type * var_type
53          | Void
54
55  type var_decl = {
56          v_type: var_type;
57          v_name: string;
58          v_init: expr;
59  }
60
61  (* Statements *)
62  type stmt =
```

```
        | Block of stmt list
        | Variable of var_decl
        | Expr of expr
        | Return of expr
        | If of expr * stmt * stmt
        | For of expr * expr * expr * stmt
        | Foreach of expr * stmt
        | While of expr * stmt
        | Continue
        | Break


(* Function Declaration *)
type func_decl = {
        fname: string;
        formals: var_decl list;
        rtype: var_type;
        body: stmt list;
}

type program = var_decl list * func_decl list

exception Variable_of_void



(*Prints Out the AST*)
let string_of_binop = function
    Plus   -> "+"
  | Minus -> "-"
  | Multi -> "*"
  | Divide -> "/"
  | Mod    -> "%"
  | Equal -> "=="
  | Nequal -> "!="
  | Less   -> "<"
  | Leq    -> "<="
  | Greater -> ">"
  | Geq    -> ">="
  | In     -> "in"
  | And    -> "&&"
  | Or     -> "||"

let string_of_asnop = function
    Add    -> "+="
```

```
108   | Sub   -> "-="
109   | Asn   -> "="
110
111 let rec string_of_expr = function
112     IntLit(lit)  -> "IntLit( " ^ string_of_int lit ^ " )"
113   | FloatLit(lit) -> "FloatLit( " ^ string_of_float lit ^ " )"
114   | BoolLit(lit) -> "BoolLit( " ^ string_of_bool lit ^ " )"
115   | CharLit(lit) -> "CharLit( " ^ Char.escaped lit ^ " )"
116   | StrLit(lit)  -> "StrLit( " ^ lit ^ " )"
117   | Id(id)       -> "Id( " ^ id ^ " )"
118   | Assign(e1, asnop, e2) -> "Assign( " ^ string_of_expr e1 ^ " " ^
119                              string_of_asnop asnop ^ " " ^ string_of_expr e2 ^ " )"
120   | Not(expr)    -> "Not( " ^ string_of_expr expr ^ " )"
121   | Binop(expr1,op,expr2) -> "Binop( " ^ string_of_expr expr1 ^ " " ^
          ↪ string_of_binop op ^ " " ^ string_of_expr expr2 ^ " )"
122   | Call(fname,param) -> "Call( " ^ fname ^ ", " ^ String.concat "; " (List.map
          ↪ string_of_expr param) ^ " )"
123   | Value(expr1,expr2) -> "Value( " ^ string_of_expr expr1 ^ ", " ^
          ↪ string_of_expr expr2 ^ " )"
124   | ListLit(elements) -> "ListLit( [" ^ String.concat "; " (List.map
          ↪ string_of_expr elements) ^ "] )"
125   | DictLit(kvpairs) -> "DictLit( [" ^ String.concat ";" (List.map
          ↪ string_of_dict kvpairs) ^ "] )"
126   | GraphLit(edges) -> "GraphLit( [" ^ String.concat ";" (List.map
          ↪ string_of_edge edges) ^ "] )"
127   | Null -> "Null"
128   | Noexpr -> "Nonexpr"
129
130 and string_of_edge edge = string_of_expr edge.src ^ "-- (" ^ string_of_expr
        ↪ edge.label ^ ") -->" ^ string_of_expr edge.dest
131
132 and string_of_dict kvpair = string_of_expr kvpair.key ^ ":" ^ string_of_expr
        ↪ kvpair.value
133
134 let rec string_of_var_type = function
135   Int    -> "int"
136 | Float  -> "float"
137 | Bool   -> "boolean"
138 | Char   -> "char"
139 | Str    -> "string"
140 | Graph  -> "graph"
141 | Node   -> "node"
142 | List(var_type) -> "list(" ^ string_of_var_type var_type ^ ")"
143 | Void   -> "void"
144 | Dict(t1, t2) -> "dict(" ^ string_of_var_type t1 ^ "," ^ string_of_var_type t2
```

```
        ↪ ^ ")"
145
146 let string_of_var_decl vardecl = "Var( name: " ^ vardecl.v_name ^ "; " ^
        ↪ string_of_var_type vardecl.v_type ^ "; " ^ string_of_expr
        ↪ vardecl.v_init ^ " )"
147
148 let rec string_of_stmt = function
149     Block(stmt_list) -> (String.concat "\n" (List.map string_of_stmt stmt_list))
            ↪ ^ "\n"
150   | Variable(var)    -> "Variable( " ^ (string_of_var_decl var) ^ " )"
151   | Expr(expr)       -> "Expr( " ^ (string_of_expr expr) ^ " )"
152   | Return(expr)     -> "Return( " ^ (string_of_expr expr) ^ ")"
153   | If(expr,stmt1,stmt2) -> "if (" ^ (string_of_expr expr) ^ ")\n" ^
        ↪ (string_of_stmt stmt1) ^ (string_of_stmt stmt2) ^ ")"
154   | For(init, test, after, stmt) -> "for (" ^ string_of_expr init ^ ", " ^
        ↪ string_of_expr test ^ ", " ^ string_of_expr after ^ ") {\n" ^
        ↪ string_of_stmt stmt ^ "\n}"
155   | Foreach(expr,stmt) -> "Foreach (" ^ (string_of_expr expr) ^ ") {\n" ^
        ↪ (string_of_stmt stmt) ^ "\n}"
156   | While(test, stmt) -> "while( " ^ (string_of_expr test) ^ " ) {\n" ^
        ↪ (string_of_stmt stmt) ^ "\n}"
157   | Continue         -> "continue"
158   | Break            -> "break"
159
160 let string_of_func_decl funcdecl = "Function( type: (" ^ string_of_var_type
        ↪ funcdecl.rtype ^ ") name: \"" ^ funcdecl.fname ^ "\" formals: "
161   ^ (String.concat ", " (List.map string_of_var_decl funcdecl.formals))
162   ^ ") {\n" ^ String.concat "\n\n" (List.map string_of_stmt funcdecl.body) ^
        ↪ "\n}"
163
164 let string_of_program (var_list,func_list)= "Program_START\n" ^ (String.concat
        ↪ "\n" (List.map string_of_var_decl var_list)) ^ "\n\n" ^
165   (String.concat "\n\n" (List.map string_of_func_decl (List.rev func_list) )) ^
        ↪ "\nProgram_END\n"
```

/src/sast.ml

```
1 (* KGL Semantically Checked ASt *)
2
3
4 open Ast
5
6 (* Expressions *)
7 type s_expr =
8     S_IntLit of int
9   | S_FloatLit of float
```

```
  | S_BoolLit of bool
  | S_CharLit of char
  | S_StrLit of string
  | S_Id of string * var_type
  | S_Assign of s_expr * asnop * s_expr * var_type
  | S_Not of s_expr * var_type
  | S_Binop of s_expr * op * s_expr * var_type
  | S_In of s_expr * s_expr * var_type * var_type
  | S_Call of string * s_expr list * var_type
  | S_Value of s_expr * s_expr * var_type * var_type
  | S_ListLit of s_expr list * var_type
  | S_DictLit of (s_expr * s_expr) list * var_type * var_type
  | S_GraphLit of (s_expr * s_expr * s_expr) list
  | S_Null
  | S_Noexpr

(* Variable declaration *)
type s_var_decl = {
  s_vtype: var_type;
  s_vname: string;
  s_vinit: s_expr;
}

(* Statement *)
and s_stmt =
    S_Block of symbol_table * s_stmt list
  | S_Variable of s_var_decl
  | S_Expr of s_expr * var_type
  | S_Return of s_expr
  | S_If of s_expr * s_stmt * s_stmt
  | S_For of s_expr * s_expr * s_expr * s_stmt
  | S_Foreach of s_expr * s_expr * var_type * s_stmt
  | S_While of s_expr * s_stmt
  | S_Continue
  | S_Break

(* Function declaration *)
and s_func_decl = {
  s_fname: string;
  s_formals: (var_type * string) list;
  s_rtype: var_type;
  s_body: s_stmt list;
}

(* Program *)
```

```
and s_program = {
  s_var_decls: s_var_decl list;
  s_func_decls: s_func_decl list;
  s_symbols : symbol_table;
}

(* Symbol table *)
and symbol_table = {
  parent: symbol_table option; (* parent symbol table *)
  mutable variables: (string * var_type) list;
}

(* Environment for each function /statment block *)
type environment = {
  scope: symbol_table;
  mutable funcs: (var_type * string * var_type list) list;
  return_type: var_type;
  in_loop: bool;              (* if in loop, to check break/continue *)
}
```

/src/cast.ml

```
(*
 * KGL C++ AST and Code Generation
 *)

open Ast
open Sast
exception Error of string


(* Types *)
type c_var_type =
       Int | Double | Bool | Char | Str
     | GraphPtr (* Reference to graph *)
     | NodePtr (* Pointer to node *)
     | Vector of c_var_type
     | Map of c_var_type * c_var_type
     | Auto
     | Void


(* SAST Type -> CAST Type *)
let rec to_c_var_type (t : Ast.var_type) : c_var_type =
  match t with
    Ast.Int -> Int
```

```
  | Ast.Float -> Double
  | Ast.Bool -> Bool
  | Ast.Char -> Char
  | Ast.Str -> Str
  | Ast.List(v) -> Vector(to_c_var_type v)
  | Ast.Dict(v1, v2) -> Map(to_c_var_type v1, to_c_var_type v2)
  | Ast.Node -> NodePtr
  | Ast.Graph -> GraphPtr
  | Ast.Void -> Void

let rec string_c_var_type (vt : c_var_type) = match vt with
    Int -> "int"
  | Double -> "double"
  | Char -> "char"
  | Str -> "string"
  | Bool -> "bool"
  | GraphPtr -> "Graph &"
  | NodePtr -> "Node *"
  | Vector(v) -> "vector<" ^ string_c_var_type v ^ ">"
  | Map(kt, vt) ->
      "unordered_map<" ^ string_c_var_type kt ^ "," ^ string_c_var_type vt ^ ">"
  | Auto -> "auto"
  | Void -> "void"


(* CAST variable declaration *)
type c_var_decl = {
  c_vtype: c_var_type;
  c_vname: string;
  c_vinit: c_expr;
}

(* CAST expr *)
and c_expr =
    C_IntLit of int
  | C_DoubleLit of float
  | C_BoolLit of bool
  | C_CharLit of char
  | C_StrLit of string
  | C_Id of string
  | C_Assign of c_expr * asnop * c_expr
  | C_Not of c_expr
  | C_Binop of c_expr * op * c_expr
  | C_Call of string * (c_expr list)
  | C_Value of c_expr * c_expr
```

```
  | C_Null
  | C_Noexpr
  | C_Exprstmt of c_stmt list
  | C_ListLit of c_expr list
  | C_DictLit of (c_expr * c_expr) list

(* CAST stmt *)
and c_stmt =
    C_Block of c_stmt list
  | C_Variable of c_var_decl
  | C_Expr of c_expr
  | C_Return of c_expr
  | C_If of c_expr * c_stmt * c_stmt
  | C_For of c_expr * c_expr * c_expr * c_stmt
  | C_Foreach of c_expr * c_expr * c_stmt
  | C_While of c_expr * c_stmt
  | C_Continue
  | C_Break

(* CAST function declaration *)
type c_func_decl = {
  c_fname: string;
  c_formals: (c_var_type * string) list;
  c_rtype: c_var_type;
  c_body: c_stmt list;
}

(* CAST program *)
type c_program = {
  c_global_vars: c_var_decl list;
  c_func_decls: c_func_decl list;
  headers: string list;
}

let rec to_c_var_decl (var: Sast.s_var_decl) : c_var_decl =
  let typ = to_c_var_type var.s_vtype and init = to_c_expr var.s_vinit in
  {
    c_vtype = typ;
    c_vname = var.s_vname;
    c_vinit =
      (if typ = GraphPtr && init = C_Noexpr then
        (* Initialize with an empty graph *)
        C_Call("_newGraph", [])
      else
        init);
```

```
115   }
116
117
118 (* TO CAST In expression: element in list, key in dict, attribute in node *)
119 and to_c_in_expr (e1: Sast.s_expr) (e2: Sast.s_expr) (t1: Ast.var_type) (t2:
120   Ast.var_type) =
121   (let e1 = to_c_expr e1 and e2 = to_c_expr e2 in
122   let t1 = to_c_var_type t1 and t2 = to_c_var_type t2 in
123   match t2 with
124     Vector(vt) -> C_Call("_contains<" ^ string_c_var_type vt ^ ">", [e2; e1])
125   | Map(kt, vt) -> C_Call("_containsKey<" ^ string_c_var_type kt ^ ", "
126                           ^ string_c_var_type vt ^ ">", [e2; e1])
127   | NodePtr -> C_Call("_hasAttribute", [e2; e1]))
128
129 (* To CAST Reference: node[attribute], graph[node], dict[key], list[index]*)
130 and to_c_value (e1: Sast.s_expr) (e2: Sast.s_expr) (t: var_type) =
131   let e1 = (to_c_expr e1) and e2 = (to_c_expr e2) in
132     if t = Node then
133       C_Value(C_Call("_getAttributes", [e1]), e2)
134     else if t = Graph then
135       C_Call("_getNode", [e1; e2])
136     else
137       C_Value(e1, e2)
138
139 (* TO CAST expression *)
140 and to_c_expr (e: Sast.s_expr) : c_expr = match e with
141     S_IntLit(lit) -> C_IntLit(lit)
142   | S_FloatLit(lit) -> C_DoubleLit(lit)
143   | S_BoolLit(lit) -> C_BoolLit(lit)
144   | S_CharLit(lit) -> C_CharLit(lit)
145   | S_StrLit(lit) -> C_StrLit(lit)
146   | S_Id(id, _) -> C_Id(id)
147   | S_Assign(e1, asnop, e2, t) -> to_c_assign e1 asnop e2 t
148   | S_Not(e1, t1) -> to_c_not_expr e1 t1
149   | S_Binop(e1, op, e2, t) ->
150       to_c_binop_expr e1 op e2 t
151   | S_In(e1, e2, t1, t2) -> to_c_in_expr e1 e2 t1 t2
152   | S_Call(fname, params, t) ->
153       C_Call(fname, List.map (fun p -> to_c_expr p) params)
154   | S_Value(e1, e2, _, t) -> to_c_value e1 e2 t
155   | S_ListLit(el, t) -> to_c_list_lit el t
156   | S_DictLit(kvl, kt, vt) -> to_c_dict_lit kvl kt vt
157   | S_GraphLit(edgel) -> to_c_graph_lit edgel
158   | S_Null -> C_Null
159   | S_Noexpr -> C_Noexpr
```

```
160
161
162 (* To CAST Not expression *)
163 and to_c_not_expr (e1: Sast.s_expr) (t1 : var_type) : c_expr =
164   (* Not empty list/dict -> false *)
165   if t1 = Void || t1 = List(Void) || t1 = Dict(Void, Void) then
166     C_BoolLit(true)
167   else
168     C_Not(to_c_expr e1)
169
170
171 (* To CAST binary expression *)
172 and to_c_binop_expr (e1: Sast.s_expr) (op: Ast.op) (e2: Sast.s_expr) (t:
173   var_type): c_expr =
174   let e1 = to_c_expr e1 and e2 = to_c_expr e2 and t = to_c_var_type t in
175   if op = Plus then
176     (match t with
177       GraphPtr -> C_Call("_plus", [e1; e2])
178     | Vector(vt) -> C_Call("_plus<" ^ string_c_var_type vt ^ ">", [e1; e2])
179     | Map(kt, vt) -> C_Call("_plus<" ^ string_c_var_type kt ^ ", " ^
180                     string_c_var_type vt ^ ">", [e1; e2])
181     | _ -> C_Binop(e1, op, e2))
182   else (
183     if op = Minus && t = GraphPtr then
184       C_Call("_minusGraph", [e1; e2])
185     else
186       C_Binop(e1, op, e2) )
187
188 (* TO CAST assign expression *)
189 and to_c_assign (e1: Sast.s_expr) (asnop: Ast.asnop) (e2: Sast.s_expr)
190 (t: Ast.var_type) : c_expr =
191   let e1 = to_c_expr e1 and e2 = to_c_expr e2 and t = to_c_var_type t in
192   match t with
193   | Vector(vt) ->
194       (match asnop with
195         Asn -> C_Assign(e1, asnop, e2)
196       | Add -> C_Call("_plusequal<" ^ string_c_var_type vt ^ ">", [e1; e2]))
197   | Map(kt, vt) ->
198       (match asnop with
199         Asn -> C_Assign(e1, asnop, e2)
200       | Add -> C_Call("_plusequal<" ^ string_c_var_type vt ^ ", " ^
201               string_c_var_type kt ^ ">", [e1; e2]) )
202   | GraphPtr ->
203       (match asnop with
204         Asn -> C_Assign(e1, asnop, e2)
```

```
205       | Add -> C_Call("_plusequal", [e1; e2])
206       | Sub -> C_Call("_minusEqualGraph", [e1; e2]))
207   | _  -> C_Assign(e1, asnop, e2)
208
209
210 (* TO CAST list literal:
211   * [|1, 2, 3|] --> ({ vector<int> _tmp = {1, 2, 3}; _tmp; })
212   *)
213 and to_c_list_lit (el: Sast.s_expr list) (t : var_type) =
214   let tmp = {
215     c_vtype = Vector(to_c_var_type t);
216     c_vname = "_tmp";
217     c_vinit = C_ListLit(List.map to_c_expr el);
218   } in C_Exprstmt([C_Variable(tmp); C_Expr(C_Id("_tmp"))])
219
220
221 (* TO CAST list literal:
222  * [|1, 2, 3|] --> ({ vector<int> _tmp = {1, 2, 3}; _tmp; })
223  *)
224 and to_c_dict_lit (kvl: (Sast.s_expr * Sast.s_expr) list) (kt : var_type) (vt:
225   var_type) =
226   let tmp = {
227     c_vtype = to_c_var_type (Ast.Dict(kt, vt));
228     c_vname = "_tmp";
229     c_vinit = C_DictLit(List.map (fun kv -> to_c_expr (fst kv), to_c_expr
230     (snd kv)) kvl);
231   } in C_Exprstmt([C_Variable(tmp); C_Expr(C_Id("_tmp"))])
232
233
234 (* TO CAST graph literal:
235  * (| s1--(l1)-->d1; s2--(l2)-->d2 |) ->
236  *  _createGraph(2, _createEdge(s1, l1, d1), _createEdge(s2, l2, d2));
237  *)
238 and to_c_graph_lit (edgel: (Sast.s_expr * Sast.s_expr * Sast.s_expr) list) =
239   let el =
240     List.map (fun (src, dest, label) ->
241       C_Call("_createEdge",[to_c_expr src; to_c_expr label; to_c_expr dest]))
              ↪ edgel
242   in
243     C_Call("_createGraph", C_IntLit(List.length el) :: el)
244
245
246 (* TO CAST statement *)
247 and to_c_stmt (s: Sast.s_stmt) = match s with
248     S_Block(_, sl) -> C_Block(List.map to_c_stmt sl)
```

```
249   | S_Variable(var) -> C_Variable(to_c_var_decl var)
250   | S_Expr(expr, _) -> C_Expr(to_c_expr expr)
251   | S_Return(expr) -> C_Return(to_c_expr expr)
252   | S_If(expr, s1, s2) -> C_If(to_c_expr expr, to_c_stmt s1, to_c_stmt s2)
253   | S_For(init, test, after, stmt) ->
254       C_For(to_c_expr init, to_c_expr test, to_c_expr after, to_c_stmt stmt)
255   | S_Foreach(e1, e2, t, stmt) ->
256       if t = Graph then
257         (* for (node in graph) ->
258          * for (auto _tmp : graph) { node = _tmp ... }
259          *)
260         C_Foreach(C_Id("_tmp"), C_Call("getNodes", [to_c_expr e2]),
261         C_Block([C_Expr(C_Assign(to_c_expr e1, Ast.Asn, C_Id("_tmp")));
262         to_c_stmt stmt]))
263       else
264       C_Foreach(C_Id("_tmp"), to_c_expr e2,
265       C_Block([C_Expr(C_Assign(to_c_expr e1, Ast.Asn,
266       (match t with
267          List(_) -> C_Id("_tmp")
268         (* for (element in list) ->
269          * for (auto _tmp : list) { element = _tmp ... }
270          *)
271        | Dict(_, _) -> C_Id("_tmp.first")
272         (* for (key in dict) ->
273          * for (auto _tmp : dict) { key = _tmp ... }
274          *)
275       ))) ; to_c_stmt stmt]))
276   | S_While(test, stmt) -> C_While(to_c_expr test, to_c_stmt stmt)
277   | S_Continue -> C_Continue
278   | S_Break -> C_Break
279
280
281 (* TO CAST function declaration *)
282 and to_c_func_decl (f: Sast.s_func_decl) =
283   { c_fname = f.s_fname;
284     c_formals = List.map (fun f -> (to_c_var_type (fst f), snd f)) f.s_formals;
285     c_rtype = to_c_var_type f.s_rtype;
286     c_body = List.map to_c_stmt f.s_body; }
287
288 (* TO CAST program *)
289 and to_c_program (program: Sast.s_program) : c_program =
290   { c_global_vars = (List.map to_c_var_decl program.s_var_decls);
291     c_func_decls = (List.map to_c_func_decl program.s_func_decls);
292     headers = ["<iostream>"; "<string>"; "<vector>"; "<unordered_map>";
293                "\"graph.h\""; "\"list.h\""; "\"dict.h\""];
```

```
294   }
295
296
297
298
299 (*******************************************************
300  * C++ Code Generation
301  *******************************************************)
302
303 let rec string_c_expr (e: c_expr) = match e with
304     C_IntLit(lit) -> string_of_int lit
305   | C_DoubleLit(lit) -> string_of_float lit
306   | C_BoolLit(lit) -> string_of_bool lit
307   | C_CharLit(lit) -> "'" ^ Char.escaped lit ^ "'"
308   | C_StrLit(lit) -> "\"" ^ lit ^ "\""
309   | C_Id(id) -> id
310   | C_Assign(e1, asnop, e2) ->
311       string_c_expr e1 ^ " " ^ string_of_asnop asnop ^ string_c_expr e2
312   | C_Not(e) -> "!(" ^ string_c_expr e ^ ")"
313   | C_Binop(e1, op, e2) ->
314       string_c_expr e1 ^ " " ^ string_of_binop op ^ " " ^ string_c_expr e2
315   | C_Call(fname, params) ->
316       fname ^ "(" ^ (String.concat ", " (List.map string_c_expr params)) ^ ")"
317   | C_Value(e1, e2) -> string_c_expr e1 ^ "[" ^ string_c_expr e2 ^ "]"
318   | C_Null -> "null"
319   | C_Noexpr -> "Noexpr"
320   | C_Exprstmt(sl) ->
321       "({" ^ (String.concat " " (List.map string_c_stmt sl)) ^ "})"
322   | C_ListLit(el) ->
323       "{" ^ (String.concat ", " (List.map string_c_expr el)) ^ "}"
324   | C_DictLit(kvl) ->
325       "{" ^ (String.concat ", " (List.map (fun kv -> "{" ^ string_c_expr
326       (fst kv) ^ ", " ^ string_c_expr (snd kv) ^ "}") kvl)) ^ "}"
327
328
329 and string_c_var_decl (v: c_var_decl) =
330   let str = string_c_var_type v.c_vtype ^ " " ^ v.c_vname in
331   str ^ (if v.c_vinit = C_Noexpr then ""
332          else " = " ^ string_c_expr v.c_vinit) ^ ";"
333
334 and string_c_if (e: c_expr) (s1: c_stmt) (s2: c_stmt) =
335   "if (" ^ string_c_expr e ^ ") " ^ string_c_stmt s1
336   ^ (if s2 <> C_Expr(C_Noexpr)
337   then " else " ^ string_c_stmt s2 else "")
338
```

```
and string_c_stmt (s: c_stmt) = match s with
    C_Block(sl) -> "{\n" ^ (String.concat "\n" (List.map string_c_stmt sl)) ^
            ↪ "\n}"
  | C_Variable(var) -> string_c_var_decl var
  | C_Expr(e) -> string_c_expr e ^ ";"
  | C_Return(e) -> "return " ^ string_c_expr e ^ ";"
  | C_If(e, s1, s2) -> string_c_if e s1 s2
  | C_For (e1, e2, e3, s1) ->
      "for (" ^ string_c_expr e1 ^ "; " ^ string_c_expr e2 ^ "; "
      ^ string_c_expr e3 ^ ") " ^ string_c_stmt s1
  | C_Foreach(e1, e2, s1) ->
      "for (auto " ^ string_c_expr e1 ^ " : " ^ string_c_expr e2 ^ ") "
      ^ string_c_stmt s1
  | C_While(e1, s1) ->
      "while (" ^ string_c_expr e1 ^ ") " ^ string_c_stmt s1
  | C_Continue -> "continue;"
  | C_Break -> "break;"


let string_c_func_decl (f: c_func_decl) =
  string_c_var_type f.c_rtype ^ " " ^ f.c_fname ^ "("
  ^ (String.concat ", " (List.map (fun f -> (string_c_var_type (fst f))
  ^ " " ^ (snd f)) f.c_formals))
  ^ ") {\n"
  ^ String.concat "\n" (List.map string_c_stmt f.c_body)
  ^ "\n}"

let string_c_program (program: c_program) =
  (String.concat "\n" (List.map (fun h -> "#include " ^ h) program.headers))
  ^ "\nusing namespace std;\n\n"
  ^ (String.concat "\n" (List.map string_c_var_decl program.c_global_vars))
  ^ "\n\n"
  ^ (String.concat "\n\n" (List.map string_c_func_decl program.c_func_decls))
  ^ "\n\nint main() { __main(); return 0; }"
```

/src/semantic.ml

```
(*
 * KGL Semantic analysis: type checking and generation of SAST
 *)

open Ast
open Sast
exception Error of string


```

```
let rec find_variable (scope: Sast.symbol_table) name =
  try
    List.find (fun (var, _) -> var = name) scope.variables
  with Not_found ->
    match scope.parent with
      Some(parent) -> find_variable parent name
    | _ -> raise Not_found


let find_func (fname: string) (opts: var_type list)
             (funcs: (var_type * string * var_type list) list) : (var_type)=
  let fl = (List.filter (fun (_, f, params) -> f = fname && params = opts)
          ↪ funcs) in
  if (List.length fl) = 0 then
    raise Not_found
  else
    let (ret_type, _, _) = List.hd fl in ret_type


let rec check_expr (e: Ast.expr) (env: environment) = match e with
    IntLit(lit) -> S_IntLit(lit), Int
  | FloatLit(lit) -> S_FloatLit(lit), Float
  | BoolLit(lit) -> S_BoolLit(lit), Bool
  | CharLit(lit) -> S_CharLit(lit), Char
  | StrLit(lit) -> S_StrLit(lit), Str
  | Id(v) -> let (_, typ) = try
               find_variable env.scope v
             with Not_found ->
               raise (Error("Undeclared identifier " ^ v));
             in S_Id(v, typ), typ
  | Assign(e1, asnop, e2) -> check_assign e1 asnop e2 env
  | Not(e) ->
      let (e1, t1) = check_expr e env in
        if not (check_bool_valued t1) then
          raise (Error("Type mismatch is operator ! " ^ string_of_var_type t1));
        S_Not(e1, t1), Bool
  | Binop(e1, op, e2) when op <> In ->
      let (e1, t1) = check_expr e1 env and (e2, t2) = check_expr e2 env in
      let typ = check_binop_type t1 op t2 env in
      S_Binop(e1, op, e2, typ), typ
  | Binop(e1, In, e2) ->
      let (e1, t1) = check_expr e1 env and (e2, t2) = check_expr e2 env in
      if not (check_in_expr t1 t2) then
        raise (Error("Type mismatch in operator " ^ Ast.string_of_binop In));
      S_In(e1, e2, t1, t2), Bool
```

```
54   | Call(f, el) -> check_call f el env
55   | Value(e1, e2) -> check_access_value e1 e2 env
56   | ListLit(el) -> check_list_lit el env
57   | DictLit(kvl) -> check_dict_lit kvl env
58   | GraphLit(edgel) -> check_graph_lit edgel env
59   | Null -> S_Null, Node
60   | Noexpr -> S_Noexpr, Void
61
62
63 (* Check list literal: all elements must be of the same type *)
64 and check_list_lit (el: Ast.expr list) (env: environment) =
65   if (List.length el) = 0 then
66     S_ListLit([], Void), List(Void)
67   else
68     let (_, t1) = check_expr (List.hd el) env in
69     let s_el = List.map (fun e ->
70       let (e2, t2) = check_expr e env in
71         if t2 <> t1 then
72           raise (Error("Elements in a list must be" ^ " of the same type."));
73       e2) el in
74     S_ListLit(s_el, t1), List(t1)
75
76 (* Check if a type is boolean valued *)
77 and check_bool_valued (t: Ast.var_type) = match t with
78     Bool | Int | Str
79   | List(_) | Graph | Dict(_) | Node -> true
80   | Float | Char | Void -> false
81
82
83 and check_binop_type (t1: Ast.var_type) (op : Ast.op) (t2: Ast.var_type) (env:
84   environment) : (Ast.var_type) =
85   match op with
86     Plus ->
87       (try
88          type_of_math_expr t1 t2
89        with Not_found ->
90         match (t1, t2) with
91            (Str, Str) -> Str
92          | (Graph, Graph) -> Graph
93          | (List(t1), List(t2)) when t1 = t2 -> List(t1)
94          (* [|1, 2, 3 |] + [| |] empty list OK *)
95          | (List(Void), List(typ)) | (List(typ), List(Void)) -> List(typ)
96          | (Dict(t1, t2), Dict(t3, t4)) when t1 = t3 && t2 = t4 -> Dict(t1, t2)
97          (* (|1:1|) + (| |) empty dict OK *)
98          | (Dict(Void, Void), Dict(t1, t2)) | (Dict(t1, t2), Dict(Void, Void))
```

```
99          -> Dict(t1, t2)
100         | (_, _) -> raise (Error("Type mismatch in operator "
101                           ^ Ast.string_of_binop op)))
102
103   | Minus ->
104       (try
105          type_of_math_expr t1 t2
106        with Not_found ->
107          if t1 <> Graph || t2 <> Graph then
108            raise (Error("Type mismatch in operator " ^ Ast.string_of_binop op));
109          Graph)
110
111   | Multi -> type_of_math_expr t1 t2
112   | Divide -> type_of_math_expr t1 t2
113   | Mod -> type_of_math_expr t1 t2
114   | Equal | Nequal ->
115       (try
116          type_of_equal_expr t1 t2
117        with Not_found ->
118          if t1 <> t2 then
119            raise (Error("Type mismatch in operator "
120            ^ Ast.string_of_binop op));
121          Bool)
122   | Less | Leq | Greater | Geq ->
123       (try
124          type_of_equal_expr t1 t2
125        with Not_found ->
126          match (t1, t2) with
127            (Char, Char) -> Bool
128          | (Str, Str) -> Bool
129          | (_, _) -> raise (Error("Type mismatch in operator "
130                           ^ Ast.string_of_binop op)))
131    | And | Or ->
132       if not (check_bool_valued t1) || not (check_bool_valued t2) then
133         raise (Error("Type mismatch in operator " ^ Ast.string_of_binop op));
134       Bool
135
136
137 (* Check if t1 and t2 can be operands of arithmetic operations *)
138 and type_of_math_expr (t1: var_type) (t2: var_type) : (var_type) =
139   match (t1, t2) with
140     (Int, Int) -> Int
141   | (Float, Float) -> Float
142   | (Int, Float) -> Float
143   | (Float, Int) -> Float
```

```
144   | (_, _) -> raise Not_found
145
146
147
148 (* Check if t1 and t2 can be operands of equal *)
149 and type_of_equal_expr (t1: var_type) (t2: var_type) : (var_type) =
150   match (t1, t2) with
151     (Int, Int) -> Bool
152   | (Float, Float) -> Bool
153   | (Int, Float) -> Bool
154   | (Float, Int) -> Bool
155   | (_, _) -> raise Not_found
156
157
158 (* Check if t1 and t2 can be operands of in *)
159 and check_in_expr (t1: var_type) (t2: var_type) : bool =
160   match (t1, t2) with
161     (typ, List(vtyp)) when vtyp = typ -> true (* element in list *)
162   | (Str, Graph) -> true                    (* node in graph *)
163   | (typ, Dict(kt, _)) when typ = kt -> true (* key in dict *)
164   | (Str, Node) -> true                     (* attributes in node *)
165   | (_, _) -> false
166
167
168 and  check_assign (e1: Ast.expr) (asnop: Ast.asnop) (e2: Ast.expr) (env:
169   environment) =
170   let (e1, t1) = (check_expr e1 env) and (e2, t2) = (check_expr e2 env) in
171   if assign_type_conversion t1 asnop t2 then
172     S_Assign(e1, asnop, e2, t1), t1
173   else
174     raise (Error("Cannot assign(" ^ string_of_asnop asnop ^ ") "
175           ^  Ast.string_of_var_type t2 ^ " to type "
176     ^ Ast.string_of_var_type t1))
177
178
179 (* The callee must exist before the caller *)
180 and  check_call f (el: expr list) (env: environment) =
181   let (el, el_t) = List.fold_right (fun e r ->
182     let (e, t) = check_expr e env in
183     (e :: fst r, t :: snd r )) el ([], []) in
184   let ret_type =
185     try
186       find_func f el_t env.funcs
187     with Not_found ->
188       let s = "Call( " ^ f ^ ", " ^ String.concat "; " (List.map
```

```
189       string_of_var_type el_t) ^ ")"
190       in raise (Error("Function " ^ s ^ " is no found"))
191   in
192   S_Call(f, el, ret_type), ret_type
193
194 (* Check refernce: list[index], graph[node], dict[key], or node[attribute] *)
195 and check_access_value (e1: expr) (e2: expr) (env: environment) =
196   let (e1, t1) = check_expr e1 env and (e2, t2) = check_expr e2 env in
197   let typ = (match (t1, t2) with
198               (List(vtyp), Int) -> vtyp
199             | (Graph, Str) -> Node
200             | (Dict(k1, vtyp), k2) when k1 = k2 -> vtyp
201             | (Node, Str) -> Str
202             | (_, _) -> raise (Error("Type mismatch in reference")))
203   in
204   S_Value(e1, e2, typ, t1), typ
205
206 (* Check Dict literal: all keys / all values must be of the same types *)
207 and check_dict_lit (kvl: kv list) (env: environment) =
208   if (List.length kvl) = 0 then
209     S_DictLit([], Void, Void), Dict(Void, Void)
210   else
211     let k_t = snd (check_expr (List.hd kvl).key env)
212     and v_t = snd (check_expr (List.hd kvl).value env) in
213     let el = List.map (
214         fun kv ->
215           let (e1, t1) = check_expr kv.key env and (e2, t2) = check_expr
216           kv.value env in
217              if t1 <> k_t then raise (Error("Keys in a dict must be"
218                                      ^ " of the same type."));
219              if t2 <> v_t then raise (Error("Values in a dict must be"
220                                      ^ " of the same type."));
221           (e1, e2)) kvl
222     in
223     S_DictLit(el, k_t, v_t), Dict(k_t, v_t)
224
225 (* Check Graph literal: source/label/dest must be string valued *)
226 and check_graph_lit (edgel: edge list) (env: environment) =
227   let edgel_t = List.map (fun e -> (check_expr e.src env, check_expr e.dest env,
228   check_expr e.label env)) edgel in
229   if (List.length (List.filter (fun (src, dest, label) -> snd src <> Str ||
230   snd dest <> Str || snd label <> Str) edgel_t)) > 0 then
231     raise (Error("Edges and nodes in a graph must be strings."))
232   else
233     S_GraphLit(List.map (fun ((e1,_), (e2,_), (e3,_)) -> (e1, e2, e3)) edgel_t)
```

```
234     , Graph
235
236 (* Check return type *)
237 and check_return (e: Ast.expr) (env: environment) : s_stmt =
238   let (ret_e, ret_t) = check_expr e env in
239   if ret_t = env.return_type then
240     S_Return(ret_e)
241   else
242     raise (Error("Cannot convert " ^ Ast.string_of_var_type ret_t ^ " to return
                ↪ type "
243     ^ Ast.string_of_var_type env.return_type))
244
245
246 (* Check for each loop: element in list, key in dict, node in graph *)
247 and check_foreach (e: Ast.expr) (s: Ast.stmt) (env: environment) =
248   let Binop(e1, op, e2) = e in
249     if op <> In then
250       raise (Error("Condition of a for-each loop must be (expr1 in expr2)"));
251     (match e1 with
252       Id(_) ->
253         let scope1 = {parent = Some(env.scope); variables = [];} in
254         let env1 = { env with scope = scope1; in_loop = true} in
255         let (e1, t1) = check_expr e1 env1 and (e2, t2) = check_expr e2 env1 in
256         (match t2 with
257           List(t) -> if t <> t1 then
258                       raise (Error("Cannot convert type " ^ string_of_var_type
259                       t ^ " to type " ^ string_of_var_type t1 ));
260                     S_Foreach(e1, e2, t2, check_stmt s env1)
261         | Dict(t, _) ->
262             if t <> t1 then
263                raise (Error("Cannot convert type " ^ string_of_var_type t
264                ^ " to type " ^ string_of_var_type t1 ));
265             S_Foreach(e1, e2, t2, check_stmt s env1)
266         | Graph -> S_Foreach(e1, e2, t2, check_stmt s env1)
267
268         | _ -> raise (Error("For-each loop must iterate a list/dict/graph")))
269      | _ -> raise (Error("First expression of a for-each loop must be an
                ↪ identifier")))
270
271
272 (* Check value declaration *)
273 and check_var_decl (v: Ast.var_decl) (env: environment) : (s_var_decl) =
274   let exist = List.exists (fun (name, _) -> name = v.v_name) env.scope.variables
275   in
276   if exist then
```

```
277     raise (Error("Indentifier already declared"))
278   else
279     env.scope.variables <- (v.v_name, v.v_type) :: env.scope.variables;
280     let (s_vinit, t) = check_expr v.v_init env in
281       if assign_type_conversion v.v_type Asn t then
282         {s_vtype = v.v_type; s_vname = v.v_name;
283          s_vinit = s_vinit; }
284       else
285         raise (Error("Variable initialization: cannot convert "
286         ^ Ast.string_of_var_type t
287         ^ " to type " ^ Ast.string_of_var_type v.v_type))
288
289
290 (* Check if t2 type can be assigned to t1 type *)
291 and assign_type_conversion (t1: Ast.var_type) (asnop: Ast.asnop) (t2:
        ↪ Ast.var_type) : bool =
292   match asnop with
293     Asn ->
294       (if t1 = t2 then
295         true
296       else
297         match (t1, t2) with
298           (_, Void) -> true
299         | (List(_), List(Void)) -> true         (* empty list *)
300         | (Dict(_, _), Dict(Void, Void)) -> true (* empty dict *)
301         | (Float, Int) -> true                  (* float a = 3 OK *)
302         | _ -> false)
303   | Add ->
304       (if t1 = Char || t1 = Bool then
305         false
306       else if t1 = t2 then
307         true
308       else
309         match (t1, t2) with
310           (List(_), List(Void)) -> true         (* empty list *)
311         | (Dict(_, _), Dict(Void, Void)) -> true (* empty dict *)
312         | (Float, Int) -> true                  (* float a += 3*)
313         | _ -> false)
314   | Sub ->
315         (match (t1, t2) with
316           (Int, Int) -> true
317         | (Float, Float) -> true
318         | (Float, Int) -> true
319         | (Graph, Graph) -> true                (* graph -= graph *)
320         | _ -> false)
```

```
321
322 (* Check statements *)
323 and check_stmt (s: Ast.stmt) (env: environment) = match s with
324     Block(slist) ->
325       let scope1 = {parent = Some(env.scope); variables = [];} in
326       let env1 = { env with scope = scope1} in
327       let slist = List.map (fun s -> check_stmt s env1) slist in
328       scope1.variables <- List.rev scope1.variables;
329       S_Block(scope1, slist)
330   | Variable(v) -> S_Variable(check_var_decl v env)
331   | Expr(e) -> let (e1, t1) = check_expr e env in S_Expr(e1, t1)
332   | Return(e) -> check_return e env
333   | If(e, s1, s2) ->
334       let (e, t) = (check_expr e env) in
335         if not (check_bool_valued t) then
336           raise (Error("Condition must be boolean-valued"));
337       S_If(e, check_stmt s1 env, check_stmt s2 env)
338   | For(e1, e2, e3, s) ->
339       let scope1 = {parent = Some(env.scope); variables = [];} in
340       let env1 = { env with scope = scope1; in_loop = true} in
341       let (e2, t) = check_expr e2 env1 in
342         if not (check_bool_valued t) then
343           raise (Error("Condition must be boolean-valued"));
344       S_For(fst (check_expr e1 env), e2, fst (check_expr e3 env), check_stmt s
                 ↪ env)
345   | Foreach(e, s) -> check_foreach e s env
346   | While(e, s) ->
347       let scope1 = {parent = Some(env.scope); variables = [];} in
348       let env1 = { env with scope = scope1; in_loop = true} in
349       let (e, t) = (check_expr e env1) in
350         if not (check_bool_valued t) then
351           raise (Error("Condition must be boolean-valued"));
352       S_While(e, check_stmt s env1)
353   | Continue ->
354       if env.in_loop then
355         S_Continue
356       else
357         raise (Error("Continue statement must be in loop"))
358   | Break ->
359       if env.in_loop then
360         S_Break
361       else
362         raise (Error("Break statement must be in loop"))
363
364 (* Check function declaraion *)
```

```
let check_func_decl (func: Ast.func_decl) (env: environment) : (s_func_decl) =
  let scope1 = {parent = Some(env.scope); variables = [];} in
  let env1 = {env with scope = scope1; return_type = func.rtype } in
  let fname = (if func.fname = "main" then "__main" else func.fname) in
  let formals_t = List.map (fun v -> (check_var_decl v env1).s_vtype)
          ↪ func.formals
  in
  try
    ignore(find_func fname formals_t env.funcs);
    raise (Error("Duplicate function definition for " ^ fname));
  with Not_found ->
    let fdecl = {
      s_fname = fname;
      s_formals = (List.map (fun var -> (var.v_type, var.v_name)) func.formals);
      s_body = (List.map (fun stmt -> check_stmt stmt env1) func.body);
      s_rtype = func.rtype;
    } in
    env.funcs <- (fdecl.s_rtype, fdecl.s_fname, formals_t) ::env.funcs; fdecl
  | _ -> raise (Error("Duplicate function definition for " ^ fname))

(* Initial environment *)
let init_env : (environment) =
  let std_funcs = [
                  (* Print *)
                  (Void, "print", [Str]);
                  (Void, "print", [Int]);
                  (Void, "print", [Float]);
                  (Void, "print", [Bool]);
                  (Void, "print", [Char]);
                  (Void, "print", [Graph]);
                  (Void, "print", [Node]);

                  (* Node *)
                  (Str, "getName", [Node]);
                  (List(Node), "getOutNeighbors", [Node; Str]);
                  (List(Node), "getOutNeighbors", [Node]);
                  (List(Node), "getInNeighbors", [Node; Str]);
                  (List(Node), "getInNeighbors", [Node]);
                  (Dict(Str, Str), "getAttributes", [Node]);

                  (* Graph *)
                  (List(Str), "getLabels", [Graph]);
                  (List(Node), "getNodes", [Graph]);

                  (* List *)
```

```
                  (Int, "getSize", [List(Int)]);
                  (Int, "getSize", [List(Float)]);
                  (Int, "getSize", [List(Char)]);
                  (Int, "getSize", [List(Str)]);
                  (Int, "getSize", [List(Bool)]);
                  (Int, "getSize", [List(Node)]);
                ] in

  let init_scope = { parent = None; variables = []; } in
  {
    scope = init_scope;
    funcs = std_funcs;
    return_type = Void;
    in_loop = false;
  }

let check_program (program: Ast.program) : (Sast.s_program) =
  let env = init_env in
  let (vdecls, fdecls) = program in
  {
    s_var_decls = List.map (fun var -> check_var_decl var env) (List.rev vdecls);
    s_func_decls = List.map (fun func -> check_func_decl func env) (List.rev
    fdecls);
    s_symbols = env.scope
  }
```

/src/kgl.ml

```
open Printf
type action = Ast | Sast | Cgen

let _ =
  let action =
    if Array.length Sys.argv > 1 then
      List.assoc Sys.argv.(1) [ ("-a", Ast); ("-s", Sast); ("-c", Cgen); ]
    else
      Cgen
  in
  let filename =
    if action == Cgen && Array.length Sys.argv > 2 then Sys.argv.(2)
    else "kgl_generated.cc"
  in

  let lexbuf = Lexing.from_channel stdin in
  let prog = Parser.test Scanner.token lexbuf in
  match action with
```

```
      Ast -> print_string (Ast.string_of_program prog)
    | Sast -> print_string "SAST"
    | Cgen -> let sast = Semantic.check_program prog in
            let cast = Cast.to_c_program sast in
            let c_code = Cast.string_c_program cast in
            let file = open_out (filename) in
            fprintf file "%s" c_code
```

/src/cgen.ml

```
open Ast
open Sast

let rec c_var_type (typ: Ast.var_type) = match typ with
    Int -> "int"
  | Float -> "double"
  | Bool -> "bool"
  | Char -> "char"
  | Str -> "string"
  | Graph -> "Graph"
  | Node -> "Node"
  | List(vtype) -> "List<" ^ c_var_type vtype ^ ">"
  | Dict(v1, v2) -> "Dict<" ^ c_var_type v1 ^ "," ^ c_var_type v2 ^ ">"
  | Void -> "void"


and c_var_decl (v: Sast.s_var_decl) =
  let str = c_var_type v.s_vtype ^ " " ^ v.s_vname in
  if v.s_vinit.e = Noexpr then
    str
  else
    str ^ " = " ^ c_expr v.s_vinit

and c_expr (e: Sast.s_expr) = c_ast_expr e.e

and c_ast_expr (e: Ast.expr) = match e with
    IntLit(lit)  -> string_of_int lit
  | FloatLit(lit) -> string_of_float lit
  | BoolLit(lit) -> string_of_bool lit
  (* TODO *)
  | CharLit(lit) -> "'" ^ Char.escaped lit ^ "'"
  | StrLit(lit)  -> "\"" ^ lit ^ "\""
  | Id(id)       -> id
  | Assign(e1, asnop, e2) -> "(" ^ c_ast_expr e1 ^ string_of_asnop asnop ^
        ↪ c_ast_expr e2 ^ ")"
  | Not(expr)    -> "(!" ^ c_ast_expr expr ^ ")"
```

```
36   | Binop(expr1,op,expr2) -> "(" ^ c_ast_expr expr1 ^ " " ^ string_of_binop op ^
             ↪ " "
37                            ^ c_ast_expr expr2 ^ ")"
38   | Call(fname,param) -> fname ^ "(" ^
39                          (String.concat ", " (List.map c_ast_expr param)) ^ ")"
40   (* TODO *)
41   | Value(expr1,expr2) -> "(" ^ c_ast_expr expr1 ^ "[" ^ c_ast_expr expr2 ^ "])"
42   | ListLit(elements) -> "ListLit( [" ^ String.concat "; " (List.map c_ast_expr
             ↪ elements) ^ "] )"
43   | SetLit(elements) -> "SetLit( [" ^ String.concat ", " (List.map c_ast_expr
             ↪ elements) ^ "] )"
44   | DictLit(kvpairs) -> "DictLit( [ TODO ] )"
45   | GraphLit(edges) ->
46       "({Graph _tmp; " ^ (String.concat "; "
47       (List.map (fun edge -> "_tmp.insertEdge(" ^ c_ast_expr edge.src ^ ", "
48                              ^ c_ast_expr edge.label ^ ", " ^ c_ast_expr
49                              edge.dest ^ ")" ) edges)) ^ "; _tmp;})"
50   | Null -> "null"
51   | Noexpr -> ""
52
53
54 and c_stmt (s: Sast.s_stmt) = match s with
55     S_Block(_,sl) -> "{\n" ^ (String.concat ";\n" (List.map c_stmt sl)) ^ ";\n}\n"
56   | S_Variable(var) -> c_var_decl var
57   | S_Expr(expr) -> "(" ^ (c_expr expr) ^ ")"
58   | S_Return(expr) -> "return " ^ (c_expr expr)
59   | S_If(expr, s1, s2) -> "if (" ^ c_expr expr ^ ")\n"
60                           ^ c_stmt s1 ^ ";\nelse" ^ c_stmt s2
61   | S_For(init, test, after, stmt) -> "for (" ^ c_expr init ^ "; " ^ c_expr test
62                                       ^ "; " ^ c_expr after ^ ") " ^ c_stmt stmt
63   | S_Foreach(e1, e2, stmt) -> "for (" ^ c_expr e1 ^ " : " ^ c_expr e2 ^ ") " ^
             ↪ c_stmt stmt
64   | S_While(test, stmt) -> "while ( " ^ c_expr test ^ " )" ^ c_stmt stmt
65   | S_Continue        -> "continue"
66   | S_Break           -> "break"
67
68
69
70 let c_func_decl (f: Sast.s_func_decl) =
71   let fname = f.s_fname in
72   c_var_type f.s_rtype ^ " " ^ fname ^ "("
73   ^ (String.concat ", " (List.map (fun f -> (c_var_type (fst f)) ^ " " ^ snd f)
             ↪ f.s_formals))
74   ^ ") {\n" ^ (String.concat ";\n" (List.map c_stmt f.s_body)) ^ ";\n}"
75
```

```
76
77 let c_program program =
78   let global_vars = (List.map c_var_decl program.s_var_decls) in
79   let func_decls = (List.map c_func_decl program.s_func_decls) in
80   "#include <iostream>\n#include <string>\n#include \"test.h\"\n"
81   ^ "\nusing namespace std;\n\n"
82   ^ (String.concat ";\n" global_vars) ^ "\n"
83   ^ (String.concat "\n" func_decls)
84   ^ "\nint main() { __main(); return 0; }"
```

/src/list.h

```
1  #ifndef __LIST_H_
2  #define __LIST_H_
3
4  #include <vector>
5  #include <algorithm>
6  using namespace std;
7
8  template<typename T>
9  bool _contains(const vector<T> &vec, T &element)
10 {
11     if(find(vec.begin(),vec.end(),element) != vec.end()){
12         return true;
13     }
14     return false;
15 }
16
17 template<typename T>
18 int getSize(const vector<T> &vec)
19 {
20     return vec.size();
21 }
22
23 template<typename T>
24 void _add(vector<T> &vec, T element, int index)
25 {
26     vec.insert(vec.begin()+index,element);
27 }
28
29 //removes all elments in v2 from v1
30 template<typename T>
31 void _removeAll(vector<T> &v1,vector<T> &v2)
32 {
33     for(int i=0;i<v2.size();++i)
34     {
```

```
        v1.erase(remove(v1.begin(),v1.end(),v2[i]),v1.end());
    }
}

template<typename T>
void _plusequal(vector<T>& v1, const vector<T>& v2)
{
    v1.insert(v1.end(),v2.begin(),v2.end());
}

template<typename T>
vector<T> _plus(const vector<T>& v1, const vector<T>& v2)
{
        vector<T> result;
        result.reserve(v1.size() + v2.size());
        result.insert(result.end(), v1.begin(), v1.end());
        result.insert(result.end(), v2.begin(), v2.end());
    return result;
}



#endif
```

/src/dict.h

```
#ifndef __DICT_H_
#define __DICT_H_

#include <unordered_map>
class Node;
using namespace std;

template<typename K, typename V>
void _removeKey(unordered_map<K, V> &d, K e)
{
    d.erase(e);
}

template<typename K, typename V>
bool _containsKey(unordered_map<K,V> &d, K key)
{
    return d.find(key)!=d.end();
}

template<typename K,typename V>
```

```
void _removeKey(Node& node, K key)
{
    node.getAttributes().erase(key);
}


template<typename K, typename V>
void _plusequal(unordered_map<K, V> &d1, const unordered_map<K, V> &d2)
{
    d1.insert(d2.begin(), d2.end());
}

template<typename K, typename V>
unordered_map<K, V> _plus(const unordered_map<K, V> &d1, const unordered_map<K,
        ↪ V> &d2)
{
    unordered_map<K, V> ret_unordered_map;
    ret_unordered_map.insert(d1.begin(), d1.end());
    ret_unordered_map.insert(d2.begin(), d2.end());
    return ret_unordered_map;
}

#endif
```

/src/graph.h

```
#ifndef __GRAPH_H_
#define __GRAPH_H_


#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>
#include <map>
#include <algorithm>

using namespace std;

class Graph;

//defines a Node type
class Node {
public:
  //initializes a node with the given string as the node's name
  //and the graph where this node belongs to
```

```
21  Node(string s,Graph *g)
22  {
23    this -> name = s;
24    this -> g = g;
25    neighborSize=0;
26  }
27
28  //returns the size of this neighbors which is the sum of
29  //outNeighbors and incomingNeighbors
30  int getNeighborSize()
31  {
32    return neighborSize;
33  }
34
35  //returns the neighbors that are connected by the given
36  //relationship label where this node is the source
37  vector<Node*> getOutNeighbors(string label){
38    if(outNeighbors.find(label)==outNeighbors.end())
39    {
40      return vector<Node*>();
41    }
42    return outNeighbors[label];
43  }
44
45  //returns all neighbor nodes where this node is the source
46  vector<Node*> getOutNeighbors(){
47    vector<Node*> result;
48    for(auto it=outNeighbors.begin();it!=outNeighbors.end();it++){
49      vector<Node*> curr = it->second;
50      for(int i=0;i<curr.size();++i)
51      {
52        if(find(result.begin(),result.end(),curr[i])==result.end())
53        {
54          result.push_back(curr[i]);
55        }
56      }
57    }
58    return result;
59  }
60
61  unordered_map<string,vector<Node*> >& getOutNeighborsMap()
62  {
63    return outNeighbors;
64  }
65
```

```
66  //returns the neighbors that are connected by the given
67  //relationship label where this node is the target
68  vector<Node*> getInNeighbors(string label)
69  {
70    if(inNeighbors.find(label)==inNeighbors.end())
71    {
72      return vector<Node*>();
73    }
74    return inNeighbors[label];
75  }
76
77  unordered_map<string,vector<Node*> >& getInNeighborsMap()
78  {
79    return inNeighbors;
80  }
81
82  //returns all neighbor nodes where this node is the source
83  vector<Node*> getInNeighbors(){
84    vector<Node*> result;
85    for(auto it=inNeighbors.begin();it!=inNeighbors.end();it++){
86      vector<Node*> curr = it->second;
87      for(int i=0;i<curr.size();++i)
88      {
89        if(find(result.begin(),result.end(),curr[i])==result.end())
90        {
91          result.push_back(curr[i]);
92        }
93      }
94    }
95    return result;
96  }
97
98  //addes a neighbor to this node
99  void addNeighbor(Node &node,string label){
100   addToOutNeighbors(node,label);
101   node.addToInNeighbors(*this,label);
102 }
103
104 void addToOutNeighbors(Node &node, string label)
105 {
106   if(outNeighbors.find(label)==outNeighbors.end()){
107     vector<Node*> curr;
108     curr.push_back(&node);
109     outNeighbors.emplace(label,curr);
110   }
```

```
111     else{
112       outNeighbors[label].push_back(&node);
113     }
114     neighborSize+=1;
115   }
116
117   void addToInNeighbors(Node &node, string label)
118   {
119     if(inNeighbors.find(label)==inNeighbors.end()){
120       vector<Node*> curr;
121       curr.push_back(&node);
122       inNeighbors.emplace(label,curr);
123     }
124     else{
125       inNeighbors[label].push_back(&node);
126     }
127     neighborSize+=1;
128   }
129
130   //removes a neighbor from this node
131   void removeNeighbor(Node& node)
132   {
133     removeOutNeighbor(node);
134     node.removeInNeighbor(*this);
135   }
136
137   void removeOutNeighbor(Node& node)
138   {
139     for(auto it=outNeighbors.begin();it!=outNeighbors.end();++it)
140     {
141       vector<Node*>& curr = it->second;
142       auto iter = find(curr.begin(),curr.end(),&node);
143       if(iter!=curr.end())
144       {
145         curr.erase(iter);
146       }
147     }
148     neighborSize-=1;
149   }
150
151   void removeInNeighbor(Node& node)
152   {
153     cout << "removeInNeighbor : " << node.getName() << endl;
154     for(auto it=inNeighbors.begin();it!=inNeighbors.end();++it)
155     {
```

```
156       vector<Node*>& curr = it->second;
157       cout<< curr.size() << endl;
158       auto iter = find(curr.begin(),curr.end(),&node);
159       if(iter!=curr.end())
160       {
161         cout << "removed" << endl;
162         curr.erase(iter);
163       }
164       cout << curr.size() <<endl;
165     }
166     neighborSize-=1;
167   }
168
169   //adds attribute to this node
170   void addAttribute(string key, string value)
171   {
172     attributes[key]=value;
173   }
174
175   //removes a attribute from this node
176   void removeAttribute(string key)
177   {
178     if(attributes.find(key)!=attributes.end())
179     {
180       attributes.erase(key);
181     }
182   }
183
184   //returns the name of this node
185   string getName() const
186   {
187     return name;
188   }
189
190   //returns the graph where this node belongs to
191   const Graph* getGraph() const
192   {
193     return g;
194   }
195
196   bool operator==(const Node& other) const
197   {
198     return g==other.getGraph() && name == other.getName();
199   }
200
```

```
201   unordered_map<string,string>& getAttributes()
202   {
203     return attributes;
204   }
205
206   friend ostream &operator<<(ostream &os, Node &node);
207   friend bool _hasAttribute(const Node *n,const string &key);
208   friend unordered_map<string,string>& _getAttributes(Node *n);
209   friend vector<Node*> getOutNeighbors(Node *n,string label);
210   friend vector<Node*> getInNeighbors(Node *n,string label);
211
212 private:
213 string name;
214 const Graph *g;
215  unordered_map<string,vector<Node*> > outNeighbors;
216  unordered_map<string,vector<Node*> > inNeighbors;
217  unordered_map<string,string> attributes;
218  int neighborSize;
219 };
220
221 bool _hasAttribute(Node *n,const string &key)
222 {
223   unordered_map<string,string>& attributes = n->getAttributes();
224   return attributes.find(key) != attributes.end();
225 }
226
227 unordered_map<string,string>& _getAttributes(Node *n)
228 {
229   return n->getAttributes();
230 }
231
232 vector<Node*> getOutNeighbors(Node *n,string label)
233 {
234   return n->getOutNeighbors(label);
235 }
236
237 vector<Node*> getInNeighbors(Node *n,string label)
238 {
239   return n->getInNeighbors(label);
240 }
241
242 //prints the attributes, outgoing neighbors, and incoming neighbors of this node
243 ostream &operator<<(ostream &os, Node &node) {
244   os << "Node : " + node.getName() + "\n";
245   os << "----------------------------\n";
```

```
246   unordered_map<string,vector<Node*> > &oNeighbors = node.getOutNeighborsMap();
247   for(auto it = oNeighbors.begin();it!=oNeighbors.end();it++)
248   {
249     os << "Outgoing Relation: " + it->first + "\n";
250     vector<Node*>& neighbors = it->second;
251     for(int i=0;i<neighbors.size();i++)
252     {
253       os<< neighbors[i]->getName() + "\n";
254     }
255   }
256   unordered_map<string,vector<Node*> > &iNeighbors = node.getInNeighborsMap();
257   os << "------------------------------\n";
258   for(auto it = iNeighbors.begin();it!=iNeighbors.end();it++)
259   {
260     os << "Incoming Relation: " + it->first + "\n";
261     vector<Node*>& neighbors = it->second;
262     for(int i=0;i<neighbors.size();i++)
263     {
264       os<< neighbors[i]->getName() + "\n";
265     }
266   }
267   unordered_map<string,string>& attrs = node.getAttributes();
268   os << "------------------------------\n";
269   os << "Attributes:\n ";
270   for(auto it = attrs.begin();it!=attrs.end();it++)
271   {
272     os<< it->first + " : " + it->second + "\n";
273   }
274   return os;
275 }
276
277 namespace std
278 {
279   template<>
280   struct hash<Node>
281   {
282     size_t operator()(const Node& k) const
283     {
284       return hash<string>()(k.getName());
285     }
286   };
287 }
288
289 //defines a graph type
290 class Graph {
```

```
public:
  Graph(){}

  ~Graph() {
    for(auto it = nodes.begin();it!=nodes.end();it++)
    {
      delete(it->second);
    }
  }

  static Graph getGraph() {return Graph(); }

  //inserts an edge to the graph by creating nodes with given names
  //as source and dest
  void insertEdge(string source, string label, string dest) {
    if (nodes.find(source) == nodes.end()) {
      // new Node ()
      // --> create NodeRef -> vector
      Node *sNode = new Node(source,this);
      nodes[source]=sNode;
    }
    if (nodes.find(dest) == nodes.end()) {
      Node *tNode = new Node(dest,this);
      nodes[dest] = tNode;
    }

    Node *sourceNode = nodes[source];
    Node *targetNode = nodes[dest];
    sourceNode->addNeighbor(*targetNode,label);
    edges[make_pair(source, dest)].push_back(label);
  }

  Node& getNode(const string& name){
    return *nodes[name];
  }

  unordered_map<string,Node*>& getAllNodes()
  {
    return nodes;
  }

  //returns all the relations between n1 and n2
  vector<string> getLabels(Node& n1,Node& n2)
  {
    string node1 = n1.getName();
```

```
336   string node2 = n2.getName();
337   pair<string,string> p1(node1,node2);
338   for(auto it = edges.begin();it!=edges.end();it++)
339   {
340     pair<string,string> key = it -> first;
341     if(key==p1)
342     {
343       return it->second;
344     }
345   }
346   return vector<string>();
347 }
348
349 map<pair<string,string>,vector<string> >& getEdges()
350 {
351   return edges;
352 }
353
354 //merges this graph with the given graph g
355 void mergeGraph(Graph &g)
356 {
357   unordered_map<string,Node*>& newNodes = g.getAllNodes();
358   for(auto it=newNodes.begin();it!=newNodes.end();it++)
359   {
360     if(nodes.find(it->first)==nodes.end())
361     {
362       nodes[it->first]=it->second;
363     }
364   }
365   newNodes.clear();
366
367   map<pair<string,string>,vector<string>>& newEdges =
          ↪ (map<pair<string,string>,vector<string>>&)g.getEdges();
368   for(auto it=newEdges.begin();it!=newEdges.end();it++)
369   {
370     pair<string,string>& curr =(pair<string,string>&) it->first;
371     if(edges.find(curr) == edges.end())
372     {
373       edges.emplace(curr,it->second);
374       vector<string>& newLabels = it->second;
375       for(auto it = newLabels.begin();it!=newLabels.end();it++)
376       {
377           getNode(curr.first).addNeighbor(getNode(curr.second),*it);
378       }
379     }
```

```
380       else
381       {
382         vector<string>& newLabels = it->second;
383         vector<string>& currLabels = edges[curr];
384         for(auto it = newLabels.begin();it!=newLabels.end();it++)
385         {
386           if(find(currLabels.begin(),currLabels.end(),*it)==currLabels.end())
387           {
388             currLabels.push_back(*it);
389             getNode(curr.first).addNeighbor(getNode(curr.second),*it);
390           }
391         }
392       }
393     }
394   }
395
396   //deletes the given graph g from this graph
397   void deleteGraph(Graph& g)
398   {
399     map<pair<string,string>,vector<string> >& subgraphEdges
              ↪ =(map<pair<string,string>,vector<string> >&)g.getEdges();
400     for(auto it = subgraphEdges.begin();it!=subgraphEdges.end();it++)
401     {
402       pair<string,string>& curr = (pair<string,string>&)it->first;
403       if(edges.find(curr)!=edges.end())
404       {
405         edges.erase(curr);
406         Node *source = &getNode(curr.first);
407         Node *target = &getNode(curr.second);
408         source->removeNeighbor(getNode(curr.second));
409         if(source->getNeighborSize()==0)
410         {
411           nodes.erase(curr.first);
412           delete(source);
413         }
414         if(target->getNeighborSize()==0)
415         {
416           nodes.erase(curr.second);
417           delete(target);
418         }
419       }
420     }
421   }
422
423   friend ostream &operator<<(ostream &os, Graph const &g);
```

```
424   friend Node* _getNode(Graph& g,const string& name);
425   friend Graph& _plusequal(Graph& g,Graph& h);
426   friend Graph& _plus(Graph &g, Graph& h);
427   friend void _insertEdge(Graph &g,string source,string label,string dest);
428   friend vector<Node*> getNodes(Graph &g);
429   friend vector<string> getLabels(Graph &g);
430   friend void _minusEqualGraph(Graph& g, Graph &h);
431   friend Graph& _minusGraph(Graph& g,Graph &h);
432
433 private:
434   map<pair<string, string>, vector<string> > edges;
435   unordered_map<string,Node*> nodes;
436 };
437
438 void _insertEdge(Graph& g,string source,string label,string dest)
439 {
440   g.insertEdge(source,label,dest);
441 }
442
443 void _minusEqualGraph(Graph &g, Graph &h)
444 {
445   g.deleteGraph(h);
446 }
447
448 Graph& _minusGraph(Graph &g,Graph &h)
449 {
450   Graph *result = new Graph();
451   result->deleteGraph(h);
452   return *result;
453 }
454
455 vector<Node*> getNodes(Graph &g)
456 {
457   vector<Node*> result;
458   for(auto it=g.nodes.begin();it!=g.nodes.end();it++)
459   {
460     result.push_back(it->second);
461   }
462   return result;
463 }
464
465 vector<string> getLabels(Graph &g)
466 {
467   vector<string> result;
468   for(auto it=g.edges.begin();it!=g.edges.end();it++)
```

```
469   {
470     vector<string>& curr = it->second;
471     for(int i=0;i<curr.size();i++)
472     {
473       result.push_back(curr[i]);
474     }
475   }
476   return result;
477 }
478 
479 Node* _getNode(Graph& g,const string& name)
480 {
481   return &(g.getNode(name));
482 }
483 
484 Graph& _plusequal(Graph &g, Graph& h)
485 {
486   g.mergeGraph(h);
487   return g;
488 }
489 
490 Graph& _plus(Graph &g, Graph &h)
491 {
492   Graph *result = new Graph();
493   map<pair<string, string>, vector<string> >& edges = g.getEdges();
494   unordered_map<string,Node*>& nodes = g.getAllNodes();
495   for(auto it = edges.begin();it!=edges.end();it++)
496   {
497     vector<string>& labels = it->second;
498     for(int i=0;i<labels.size();i++)
499     {
500       result->insertEdge(it->first.first,labels[i],it->first.second);
501     }
502   }
503   result->mergeGraph(h);
504   return *result;
505 }
506 
507 Graph& _newGraph()
508 {
509   return *(new Graph());
510 }
511 
512 ostream &operator<<(ostream &os, const Graph &g) {
513   os << "Graph {\n";
```

```
514   //for (auto it = g.getEdges().begin(); it != g.getEdges().end(); it++) {
515   const map<pair<string, string>, vector<string>> & es = g.edges;
516   for (auto it = es.begin(); it != es.end(); it++) {
517     for (int i = 0; i < it->second.size(); i++) {
518       os << "Node(" << it->first.first << ") --" << it->second[i];
519       os << "--> Node(" << it->first.second << ")\n";
520     }
521   }
522   os << "}";
523   return os;
524 }
525
526
527 struct edge {
528   string source, label, dest;
529   edge(const string &s, const string &l, const string &d): source(s),
530   label(l), dest(d) {}
531 };
532
533 edge* _createEdge(string src, string label, string dest) {
534   return new edge(src, label, dest);
535 }
536
537 Graph& _createGraph(int num, ...) {
538   va_list edges;
539   va_start(edges, num);
540   Graph *g = new Graph();
541   for (int i = 0; i < num; ++i) {
542     edge *s = va_arg(edges, edge*);
543     g->insertEdge(s->source, s->label, s->dest);
544     delete s;
545   }
546   va_end(edges);
547   return *g;
548 }
549
550 /* Built-in functions */
551 void print(int t) {
552   cout << t;
553 }
554
555 void print(double t) {
556   cout << t;
557 }
558
```

```
559 void print(char t) {
560   cout << t;
561 }
562
563 void print(const string &t) {
564   cout << t;
565 }
566
567 void print(const Graph &t) {
568   cout << t;
569 }
570
571 void print(const Node *t) {
572   cout << t->getName();
573 }
574
575 string getName(const Node *t) {
576   return t->getName();
577 }
578
579
580
581
582 #endif
```

/src/Makefile

```
1  OBJS = ast.cmo semantic.cmo cast.cmo parser.cmo scanner.cmo kgl.cmo
2
3
4  kgl: $(OBJS)
5      ocamlc -g -o kgl $(OBJS)
6
7  scanner.ml: scanner.mll
8      ocamllex scanner.mll
9
10 parser.ml parser.mli: parser.mly
11     ocamlyacc parser.mly
12
13 %.cmo: %.ml
14     ocamlc -w A -c $<
15
16 %.cmi: %.mli
17     ocamlc -w A -c $<
18
19 .PHONY: clean
```

```
clean:
    rm -f kgl parser.ml parser.mli parser.output scanner.ml *.cmo *.cmi *~ *.cc

.PHONY: all
all: clean kgl
# Generated by ocamldep *.ml *.mli
ast.cmo :
ast.cmx :
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
parser.cmi : ast.cmo
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
kpl.cmo : scanner.cmo parser.cmi semantic.cmo cgen.cmo ast.cmo
kpl.cmx : scanner.cmx parser.cmx semantic.cmx cgen.cmx ast.cmx
sast.cmo : ast.cmo
sast.cmx : ast.cmx
semantic.cmo : sast.cmo ast.cmo
semantic.cmx : sast.cmx ast.cmx
```

8.3 Test Suites

gotest.sh

```
#!/bin/bash
KGL="src/kgl"
successes=0
failures=0
creates=0

Check_rec()
{
    fileName=$(basename $1 | cut -f 1 -d '.')

    eval "$KGL -c src/kgl_tmp.cc <$1 2>/dev/null"
    eval "g++ -std=c++0x src/kgl_tmp.cc > /dev/null 2>&1"
    if [ $? -ne 0 ] ; then
        expectedFail=$( echo $fileName | cut -f 1 -d '_')
        if [ "$expectedFail" == "x" ]; then
            successes=$((successes+1))
            echo "Success"
        else
            failures=$((failures+1))
            echo "Failed to compile"
        fi
    else
        eval "./a.out > output.txt"

        if [ -f "test_suite/outputs/$fileName.out" ]; then
            eval "diff -B --strip-trailing-cr output.txt
                    ↪ test_suite/outputs//$fileName.out >/dev/null" && isSame=1
                    ↪ || isSame=0
            if [ $isSame -eq 1 ]; then
                successes=$((successes+1))
                echo "Success"
            else
                echo "Failed should not happen"
                failures=$((failures+1))
            fi
        else
            eval "cp output.txt test_suite/outputs/$fileName.out"
            creates=$((creates+1))
            echo "new output created"
        fi
    fi
}
```

```
41
42 Check()
43 {
44     for test in $1;
45     do
46         if [ -f $test ]; then
47             echo "------------ $test ------------"
48             Check_rec $test
49             eval "rm -f src/kgl_tmp.cc a.out"
50
51         elif [ -d $test ]; then
52             append="/*"
53             Check "$test$append"
54         fi
55     done
56 }
57
58 getopts ":s:l:p" opt;
59 case $opt in
60 s)
61     tests="test_suite/tests/semantic/*"
62     ;;
63 l)
64     tests="test_suite/tests/lexer/*"
65     ;;
66 p)
67     tests="test_suite/tests/parser/*"
68     ;;
69 *)
70     tests="test_suite/tests/*"
71     ;;
72 esac
73
74 Check "$tests" "-----------------"
75
76 echo "$successes passed"
77 echo "$failures failed"
78 echo "$creates file/s created"
```

/test suite/lexer/

```
1 func void main(){
2   print("hello word!");
3 }
```

```
func void main(){

    print(42);
}
```

```
func void main(){
    print(2.45);
}
```

```
func void main(){
    print(.45);
}
```

```
func void main(){
    print(" hi hi _ \n");
}
```

```
func void main() {
  graph g = {| "Mike"--("friends")-->"Nick";
               "Jack"--("friends")-->"Nick" |};
  print(g);
}
```

```
func void main(){
    print(k);
}
```

/test suite/parser/literals/char/

```
func void main(){
    print('\t');
}
```

```
func void main(){
    print('\n');
}
```

```
func void main(){
    print('\\');
}
```

```
func void main(){
    print('7');
}
```

```
func void main(){
    print(' ');
}
```

```
func void main(){
    print('\ty');
}
```

```
func void main(){
    print('');
}
```

/test suite/parser/literals/double

```
func void main(){
    print(2.45);
}
```

```
func void main(){
    print(.45);
}
```

```
func void main(){
    print(-.45);
}
```

```
func void main(){
    print(-45.);
}
```

```
func void main(){
    print(2.4.5);
}
```

```
func void main(){
    print(.);
}
```

```
func void main(){
    print(-.4-5);
}
```

```
func void main(){
    print(-.45.);
}
```

/test suite/parser/literals/int

```
func void main(){

    print(42);
}
```

```
func void main(){

    print(-42);
}
```

```
func void main(){

    print(4-2);
}
```

/test suite/parser/literals/string

```
func void main(){
    print("hello word!");
}
```

```
func void main(){
    print(" hi hi _ \n");
}
```

```
func void main(){
    print("");
}
```

```
func void main(){
    print(" ");
}
```

```
func void main(){
    print(" hi hi" _ \n");
}
```

/test suite/parser/newline

```
func void main(){
    string s = "newline\nme"
    print(s);
}
```

```
func void main(){

    print("new line here >\n<");
}
```

```
func void main(){
    print(\n);
}
```

/test suite/parser/slash

```
func void main(){
    string s = "slash\\me"
    print(s);
}
```

```
func void main(){

    print("slash here >\\<");
}
```

```
func void main(){
    print(\\);
}
```

/test suite/parser/tab

```
func void main(){
    string s = "tab\tme"
    print(s);
}
```

```
func void main(){

    print("tab here >\t<");
}
```

```
func void main(){
    print(\t);
}
```

/test suite/semantic/break

```
func void main() {
  int i;
  for (i in [|1, 2, 3|])
  {
    print(i);
    break;
    print("bad");
  }
}
```

```
func void main() {
  int i;
  for (i in [|1, 2, 3|])
     print(i);
  break;
}
```

/test suite/semantic/continue

```
func void main() {
  int i;
  for (i in [|1, 2, 3|])
  {
    continue;
    print(i);
  }
  print("just this");
}
```

```
func void main() {
  int i;
```

```
  continue;
}
```

/test suite/semantic/dict

```
func void main() {
  dict<string, int> candidate;
  candidate["hi"] = 5;
  print(candidate["hi"]);
}
```

```
func void main() {
  dict<boolean, string> candidate;
  candidate[false] = "true";
  print(candidate[true]);
}
```

```
func void main() {
  char x = 'x';
  dict<char, char> candidate;
  candidate['c'] = x;
  print(candidate['c']);
}
```

```
func void main() {
  dict<string, int> d1 =(| "one" : 1, "two" : 2, "three" : 3 |);
  dict<string, int> d2 =(| "four" : 4, "five" : 5, "six" : 6 |);
  dict<string, int> d3 = d1 + d2;
  print("15\n");
  print(d3["one"]);
  print(d3["four"]);
}
```

```
func void main() {
  dict<string, int> d1 =(| "one" : 1, "two" : 2, "three" : 3 |);
  dict<string, int> d2 =(| "four" : 4, "five" : 5, "six" : 6 |);
  d2 = d2 + d1;
  print(d2["one"]);
  print(d2["four"]);
}
```

```
func void main() {
  string x = "x";
```

```
  dict<char, char> candidate;
  candidate['c'] = x;
  print(candidate['c']);
}
```

```
func void main() {
  dict<char, char> candidate;
  candidate['c'] = 54;
  print(candidate['c']);
}
```

```
func void main() {
  dict<string, int> d1 =(| "one" : 1, "two" : 2, "three" : 3 |);
  dict<string, char> d2 =(| "4" : '4', "5" : '5', "6" : '6' |);
  dict<string, int> d3 = d1 + d2;
  print("15\n");
  print(d3["one"]);
  print(d3["four"]);
}
```

/test suite/semantic/foreach

```
func void main() {
  int i;
  for (i in [|1, 2, 3|]) {
    print(i);
  }
}
```

```
func void main() {
  char c;
  for (c in ['a','b','c'|])
    print(c);

}
```

```
func void main() {
  string i;
  list<string> j = [|"this", "should", "work"|];
  for (i in j)
    print(i);
}
```

```
func void main() {
  float i;
  for (i in [|1, 2, 3|]) {
    print(i);
  }
}
```

```
func void main() {
  int i;
  for (i in [|'a', 'b', 'c'|])
    print(i);

}
```

```
func void main() {
  string i;
  string j;
  for (i in j)
    print(i);

}
```

/test suite/semantic/func˙call

```
func void test(int a) {print(a);}
func void main() {
  test(1);
}
```

```
func void test(int a, graph g) {print(a); print(g);}
func void main() {
  test(1, {||});
}
```

```
func void test() {}
func void main() {
  test(1);
}
```

```
func void test(int a, graph g) {}
func void main() {
  test({||}, 1);
```

```
}
```

/test suite/semantic/graph

```
func void main() {
  graph g = {||};
  print("success");
}
```

```
func void main() {
  graph g = {||};
  g = g + {|"Nick"--("brother")-->"Mike"|};
  print(g);
}
```

```
func void main() {
  graph g = {||};
  g = g + {|"Nick"--("brother")-->"Mike"|};
  graph g2 = {|"John"--("brother")-->"Greg"|};
  graph g3 = g + g2;
  print(g3);
}
```

```
func void main() {
  string a = "Mike";
  string b = "Nick";
  string r = "brother";
  graph g = {|a--(r)-->b|};
  print(g);
}
```

```
func void main() {
  string a = "Mike";
  string b = "Nick";
  string r = "brother";
  graph g1 = {|a--(r)-->b|};
  graph g2 = {|"Jeff"--("likes")-->"nothing"|};
  print(g1);
  print("\n");
  print(g2);
}
```

```
func void main() {
```

```
  graph g = {|};
  print("success");
}
```

```
func void main() {
  graph g = {||};
  g = g + {|"Nick"--("brother"-->"Mike"|);
  print(g);
}
```

```
func void main() {
  graph g = {||};
  g = g + {|"Nick"--("brother")-->"Mike"|);
  graph g2 = {|"John"--("brother")-->"Greg"|};
  g3 = g += g2;
  print(g3);
}
```

```
func void main() {
  string a = "Mike";
  string b = "Nick";
  int r = 3;
  graph g = {|a--(r)-->b|};
}
```

/test suite/semantic/if

```
func void main() {
  boolean go = false;

  if (true) {
    print("true");
  }
  else {
    print("false");
  }
}
```

```
func void main() {
  if (5) {
  }
}
```

/test suite/semantic/in

```
func void main() {
  graph g;
  1 in g;
}
```

```
func void main() {
  list<int> l;
  3.3 in l;
}
```

```
func void main() {
  dict d;
  1.3 in d;
}
```

/test suite/semantic/list

```
func void main() {
  list<boolean> l = [|true, false|];
  boolean darkness = l[1];
  print(darkness);
}
```

```
func void main() {
  list<int> l = [|3, 13|];
  int num = l[1];
  print(num);
}
```

```
func void main() {
  list<int> l1 = [|3, 13|];
  list<int> l2 = [|4, 2|];
  l1[0] = l2[1];
  print(l1[0]);
}
```

```
func void main() {
  print([|true, false|][0]);
}
```

```
func void main() {
  [|-9.6, 62., 7.88|];
}
```

```
func void main() {
  list<int> l = [|3, 13|];
  float num = l[1];
}
```

```
func void main() {
  list<int> l = [|3, 13|];
  string num = l[1];
}
```

```
func void main() {
  list<int> l1 = [|3, 13|];
  list<string> l2 = [|"hi", "bye"|];
  l2[0] = l1[1];
}
```

```
func void main() {
  [|1, 2.3, "123"|];
}
```

```
func void main() {
  [|1, 2.3, 2.9|];
}
```

```
func void main() {
  [|"9", "2.3", '2'|];
}
```

```
func void main() {
  [|true, false, 0|];
}
```

/test suite/semantic/logical

```
func void main() {
  boolean yes = true;
  boolean no = false;
  if (yes || no) { print("good");}
```

```
}
```

```
func void main() {
  boolean no = false;
  while (true && no) {print("bad");}
}
```

```
func void main() {
  if ([|1,2,3|] && 3.3) {}
}
```

```
void main() {
  if (3.3 || {||} ) {}
}
```

```
func void main() {
  while ([|1,2,3|] && 3.3) {}
}
```

```
func void main() {
  if (92 && false) {}
}
```

/test suite/semantic/misc

```
func void test (graph g, int i) {
}


func void test (graph h, int j) {
}

func void main () {
}
```

/test suite/semantic/node

```
func void test(graph g) {
  node a = g["nick"];
}

func void main() {
  print("success");
```

```
}
```

```
func void main() {
  graph g = {| "nick"--("friend")--> "jack" |};
  node a = g["nick"];
  g["nick"]["a"] = "strong";
  print(g["nick"]["a"]);
}
```

```
func void main() {
  graph g = {| "nick"--("friend")--> "jack" |};
  node a = g["nick"];
  node b = a;
  if(a == b)
    print("success");
  else
    print("fail");
}
```

```
func void main() {
  graph g = {| "nick"--("friend")--> "mike" |};
  node a = g["nick"];
  node b = g["mike"];
  a["nick"]["a"] = "strong";
  b["mike"]["b"] = "bad";
  // getAttributes(a) + getAttributes(b)
  node brothers = a + b;
  print(brothers["nick"]);
  print(brothers["mike"]);

}
```

```
func void main() {
  graph g = {| "nick"--("friend")--> "jack" |};
  node a = g["nick"]];
  print("success");
}
```

```
func void main() {
  graph g = {| "nick"--("friend")--> "jack" |};
  node a = g["nick"];
  g["nick"] = "strong";
```

```
  print(g["!not_nick!"]);
}
```

```
func void main() {
  graph g = {| "nick"--("friend")--> "jack" |};
  node a = g["nick"];
  node b == a;
  if(a == b)
    print("success");
  else
    print("fail");
}
```

```
func void main() {
  graph g = {| "nick"--("friend")--> "jack" |};
  node a = g["nick"];
  node b = g["mike"];
  a["nick"] = "strong";
  b["mike"] = "bad";
  node brothers = a + b;
  print(brothers["nick"]);
  print(brothers["tom"]);

}
```

/test suite/semantic/not

```
func void main()
{
  boolean tru = true;
  if (!tru) {
    print("bad");
  }
  else{
    print("good");
  }
}
```

```
func void main()
{
  if (!false) {
    print("good");
  }
}
```

```
func void main()
{
  if (!3.3) {
    print(3.3);
  }
}
```

```
func void main() {
  if (!'3') {}
}
```

/test suite/semantic/plus˙operator

```
func void main() {
  int d = 7 + 3;
  print(d);
}
```

```
func void main() {
  list<int> l = [|1, 2, 3|] + [|7, 0, 98|];
  print(l[3]);
}
```

```
func void main() {
  int a = 3 + "313";
}
```

```
func void main() {
  [|1, 2, 3|] + [|"1", "2", "3"|];
}
```

/test suite/semantic/relational

```
func void main() {
  int hi = 1;
  int bye = 1;
  if (bye == hi)
    print("good");
}
```

```
func void main() {
  string nut = "";
```

```
  string nutty = "n";
  if (nut != nutty)
    print("good");
  else
    print("bad");
}
```

```
func void main() {
  int five = 5;
  int five_minus_two = 3;
  if (five == five_minus_two + 2)
    print("good");
  else
    print("bad");
}
```

```
func void main() {
  if (3.3 != "3.3")
    print(3.3);
}
```

```
void main() {
  graph g;
  list<int> l;
  if (g == l)
    print(g);
}
```

```
void main() {
  graph g;
  graph h;
  if (g > h)
    print(g);
}
```

/test suite/semantic/return

```

func int test() {
  int x = 5;
  if (true)
  {
    x = 9;
```

```
  }
  return x;
}


func void main() {
}
```

```

func list<string> test() {
  list<string> l = [|"this", "should", "work"|];
  return l;
}


func void main() {
  print(test()[2]);
}
```

```

func int test() {
  int x = 5;
  return x;
}


func void main() {
  int x = 5;
  x = x + test();
  print("10?\n");
  print(x);
}
```

```

func int test() {
  if (true)
  {
    return false;
  }
  return 1;
}


func void main() {
```

```
}
```

```

func int test() {
  return 1.2;
}

func void main() {
}
```

```

func int test() {
  boolean never = true;
  return never;
}

func void main() {
}
```

/test suite/semantic/type˙match

```
func void main() {
  int g = 4;
  int m;
  m = g;
  print(m);
}
```

```
func void main() {
  double g = 4.5;
  int m;
  m = g;
}
```

```
func void main() {
  string g = "4.5";
  char m;
  m = g;
}
```

```
func void main() {
  string g = "true";
  boolean m;
```

```
  m = g;
}
```

```
func void main() {
  int g = 4;
  char m;
  m = g;
}
```

```
func void main() {
  string g = 'g';
}
```

/test suite/semantic/var˙decl

```
func void main() {
  int a = 3;
  int b = 22;
  {
    print(a);
    print(b);
  }
}
```

```
func void main() {
  int a = 3;
  {
    print(a);
    print(b);
  }
}
```

```
func void main() {
  int g = 1;
  boolean g = false;
}
```

/test suite/semantic/var˙match

```
func void main() {
  int g = 4;
  int m;
  m = g;
  print(m);
```

```
}
```

```
func void main() {
  char v = '4';
  char m;
  m = v;
  print(m);
}
```

```
func void main() {
  double g = 4.5;
  int m;
  m = g;
}
```

```
func void main() {
  string g = "4.5";
  char m;
  m = g;
}
```

```
func void main() {
  string g = "true";
  boolean m;
  m = g;
}
```

```
func void main() {
  int g = 4;
  char m;
  m = g;
}
```

8.4 Demo Code

/demo/gcd.kgl

```
func int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}

func void main() {
  print(gcd(42, 35));
}
```

/demo/find˙path.kgl

```

## Find path between node s and node t
func list<node> find_path(graph g, string s, string t, string relation) {

  list<node> queue;
  dict<node, node> backPointer;
  node source = g[s];
  node dest = g[t];
  queue += [| source |];
  backPointer[source] = source;
  int i = 0;
  boolean found = false;
  while (!found && i < getSize(queue)) {
    node curr = queue[i];
    i += 1;
    list<node> neighbors = getOutNeighbors(curr, relation);
    node n;
    for (n in neighbors) {
      if (!(n in backPointer)) {
         backPointer[n] = curr;
         queue += [| n |];
      }
      if (n == dest) {
         found = true;
         break;
      }
```

```
    }
  }

  list<node> result;
  node curr = dest;
  while (curr != source) {
     result = [| curr |] + result;
     curr = backPointer[curr];
  }
  result = [| source |] + result;
  return result;
}

func void main() {
  graph g = {|"Joe"--("knows")-->"Bill";
              "Joe"--("knows")-->"Sara";
              "Sara"--("knows")-->"Jill";
              "Sara"--("knows")-->"Bill"|};

  g += {|"Sara"--("knows")-->"Ian";
         "Bill"--("knows")-->"Derrick";
         "Bill"--("knows")-->"Ian"|};

  list<node> result = find_path(g, "Joe", "Ian", "knows");
  node curr;
  for (curr in result) {
    print(curr); print(" ");
  }
  print("\n");
}
```

/demo/friend_of_friend.kgl

```
func void main() {
  graph g = {|"Joe"--("knows")-->"Bill";
              "Joe"--("knows")-->"Sara";
              "Sara"--("knows")-->"Jill";
              "Sara"--("knows")-->"Bill"|};

  g += {|"Sara"--("knows")-->"Ian";
         "Bill"--("knows")-->"Derrick";
         "Bill"--("knows")-->"Ian"|};


  list<node> friends = getOutNeighbors(g["Joe"], "knows");
  dict<node, int> candidates;
```

```
 node f;
 node p;
 for (f in friends) {
   for (p in getOutNeighbors(f, "knows")) {
     if (!(p in candidates))
       candidates[p] = 0;
     candidates[p] += 1;
   }
 }

 ## print(g);

 for (f in candidates) {
   if (!(f in friends) && (f != g["Joe"])) {
     print(f);
     print(" : ");
     print(candidates[f]);
     print("\n");
   }
 }
}
```

/demo/recommend˙friend.kgl

```
func dict<node,int> recommendFriends(graph g,string source,string relation)
{
    dict<node,int> result;
    list<node> outNeighbors = getOutNeighbors(g[source],relation);
    node n;
    for(n in outNeighbors)
    {
        list<node> inNeighbors = getInNeighbors(n,relation);
        node curr;
        for(curr in inNeighbors)
        {
            if((curr!=g[source]) && !(curr in
                    ↪ getOutNeighbors(g[source],"friend")))
            {
                if(!(curr in result))
                {
                    result[curr] = 1;
                }
                else
                {
                    result[curr] += 1;
                }
```

```
            }
        }
    }
    return result;
}

func void main()
{
    graph g = {|
        "Joe"--("friend")-->"Sara";
        "Joe"--("favorite")-->"Cats";
        "Joe"--("favorite")-->"Bikes";
        "Derrick"--("favorite")-->"Cats";
        "Derrick"--("favorite")-->"Bikes";
        "Sara"--("favorite")-->"Cats";
        "Sara"--("favorite")-->"Bikes";
        "Jill"--("favorite")-->"Bikes"
    |};

    dict<node,int> result = recommendFriends(g,"Joe","favorite");
    node curr;
    for(curr in result)
    {
        print(curr); print(" : "); print(result[curr]); print("\n");
    }
}
```

/demo/attributes.kgl

```
func dict<node,int> find_friends_similar_favorites(graph g, string source,string
        ↪ relation) {
  dict<node,int> result;
  list<node> friends = getOutNeighbors(g[source], "friend");
  node n;
  for (n in getOutNeighbors(g[source], relation)) {
    node curr;
    for (curr in getInNeighbors(n, relation)) {
      if (curr == g[source] || curr in friends) continue;
      if ("gender" in curr && curr["gender"] != g[source]["gender"]) {
    if (!(curr in result))
     result[curr] = 1;
        else
     result[curr] += 1;
      }
    }
  }
```

```
  return result;
}


func void main() {
  graph g = {| "Joe"--("friend")-->"Sara";
        "Joe"--("favorite")-->"Cats";
        "Joe"--("favorite")-->"Bikes";
        "Derrick"--("favorite")-->"Cats";
        "Derrick"--("favorite")-->"Bikes";
        "Sara"--("favorite")-->"Cats";
        "Sara"--("favorite")-->"Bikes";
        "Jill"--("favorite")-->"Bikes" |};


  g["Joe"]["gender"] = "male";
  g["Derrick"]["gender"] = "male";
  g["Sara"]["gender"] = "female";
  g["Jill"]["gender"] = "female";

  dict<node,int> result = find_friends_similar_favorites(g,"Joe","favorite");
  node curr;
  for(curr in result) {
    print(curr); print(" : "); print(result[curr]); print("\n");
  }
}
```