COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Circline

Circline - An Easy Graph Language
Final Report

Haikuo Liu	hl3023
Jia Zhang	jz2784
Qing Lan	ql2282
Zehao Song	zs2324

Dec 19, 2016

Abstract

Circline is an easy syntax language that natively support Node, Edge and Graph definition. The optimal goal of this language is to provide a clear view on the Tree or graph relationship in reality. User could use this language to build his/her own graph and do some common algorithm such as Dijkstra, Breadth-First Search and Depth-First Search. It contains scanner, parser, organizer, ast, cast, semantic, code generation and external C libraries. These are the core components of Circline. This report generally explained the usage and implementation of the Circline.

Contents

1. Introduction

1.1. Background

Graph is an important data structure in computer science widely used to present a complex relationship between events and resources, such as tree, S-T flow and even Neural Networks. However, current programming language could only provide limited functionalities in building a graph and the relationship between different nodes would not be clearly shown. Hence, there is a need to create a comprehensive programming language that designed specifically for graph to present the relationship among objects.

1.2. Aims and Objectives

To facilitate the use of graph, Circline, a Graph-Oriented language, was created. It should be designed in a user-friendly way and the graph should present all necessary information such as the relationship between nodes, the type of edge and relationship between graphs and nodes. Moreover, Circline should be easy to understand and as fast as possible. It should allow user to use the basic data structure such as Array, List and Dictionary. User could also use this language to build his/her own functions. If all functionalities indicated before could be achieved, the user will be able to build up complicated graph algorithm to make full use of graph in their calculation.

Circline has easier syntax for describing a graph closed to script language. It allows user to define a variable, functions anywhere and allow nested functions. The function definition of Circline is in a Java style and the usage of them is closed to Python. The highlighted feature of Circline is its native support for Node, Edge and Graph definition. The followings would describe the usage of them.

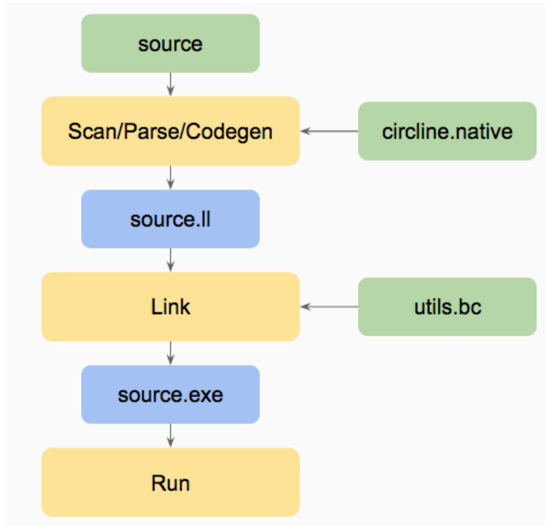
As an example, to create a graph with three nodes (a, b, c), where a is root node and link with node b and c. We could easily define it like:

```
node a = node(); node b = node(); node c = node(); graph gh = a -- [b, c];
```

In above example, we first define three nodes and then link them using the symbol "--", which defines edges linking node a and node b, c. As you can see, Circline use special syntax like "--" to define the edge, which is more straightforward and convenient.

This report would firstly introduce the Setup and Usage for this language through a step-by-step tutorial. Then, the full language reference manual would be provided to form a comprehensive overview of the language feature. Project Plan and Language Architecture would describe the timeline and detailed implementation of Circline. Finally, performance analysis and testing suite would be introduced.

2. Tutorial



2.1. Hello World

To compile the compiler, open circline folder in the terminal and run **make all**. To test that everything is okay, run **make test**. Then, a series of test cases would run and show success in the terminal.
Here is a simple program in our language, **hello**.

```
print( “Hello World!” );
```

To run this program, follows the steps shown in the previous illustration graph or just run

```
$ sh circline.sh hello
Hello World!
```

2.2. Graph Creation

First, you should define all nodes.

```
node a = node("a");
node b = node("b");
node c = node("c");
node d = node("d");
node e = node("e");
```

Note: the value of the node could be one of the int, bool, float and string.

Attention: In the entire reference document, single letters are reference to node defined as above.

Second, define the graph.

```
graph g = a -> b -> c;
```

Then, you could show the graph by

```
print(g);
```

The output would be:

```
--------------------------------------
#Nodes: 3  Root Node: 4
node  3: b
node  2: c
node  4: a
#Edges: 2
edge  3->  2
edge  4->  3
--------------------------------------
```

Note: Each node has been assigned a global unique id.

You could merge graphs into a more complicated graph:

`graph g1 = a->b->c;`
 	
`graph g2 = c->d;`
 	
`print(g1 + g2);`
	`--------------------------------------`
	`#Nodes: 4  Root Node: 4`
	`node  3: b`
	`node  2: c`
	`node  4: a`
	`node  1: d`
	`#Edges: 3`
	`edge  3->  2`
	`edge  4->  3`
	`edge  2->  1`
	`--------------------------------------`

You could manipulate the graph by removing nodes:

`graph g1 = (g-a).get(0);`
`print( g1 );`
	`--------------------------------------`
	`#Nodes: 2  Root Node: 3`
	`node  3: b`

	`node  2: c`
	`#Edges: 1`
	`edge  3->  2`
	`--------------------------------------`

Note: (g-a) returns a list<graph> object, list.get(0) return the first graph.

You could traverse the graph by getting the neighbors of a particular node:

```
graph g = a -> [b, c, d];
list<node> l = g@a;
print( l );
```

```
list:[node  3: b
node  2: c
node  1: d
]
```

For more information on tutorials, please refer the testing example in Testing section for complicated implementation and code generation test cases.

3. Language Reference Manual

3.1. Types and Literals

3.1.1. Primitive Types

Name	Prefix	Description
Boolean	bool	true | false Example: true false
Integer	int	Possible value: 32-bit signed Integer (-2147483638 ~ 2147483647) Example: -123 43 0
Floating point	float	Possible value: A IEEE 754 double-precision (64-bit) numbers Example: 0.356 3.4e-16 1.

String	string	Possible value: A sequence of ASCII enclosed by double quotes Example: “I’m Haikuo! Talent Guy!” “” “Hello world!\n”
Null	null	A type represent ‘nothing’ Example: null

3.1.2. Node

Node is used to define a point in the graph, it could be linked to other nodes or graph. Each node could only store a single value. The stored node value could be one of the following types: **int**, **float**, **bool** and **string**.

```
node( 1 )
node( true )
node( 4.5 )
node( “Hello world!” )
```

To retrieve the value of node, there are two options:

1. print

`print( node(1) );`	`=> node  0: 1`
`print( node(true) );`	`=> node  2: true`
`print( node(1.2) );`	`=> node  1: 1.200000`
`print( node("Node") );`	`=> node  0: Node`

2. cast

`int( node( 1 ) )`	`=> 1`
`bool( node( true ) )`	`=> true`
`float( node( 4.5 ) )`	`=> 4.5`
`string( node( “Hello world!” ) )`	`=> “Hello world!”`

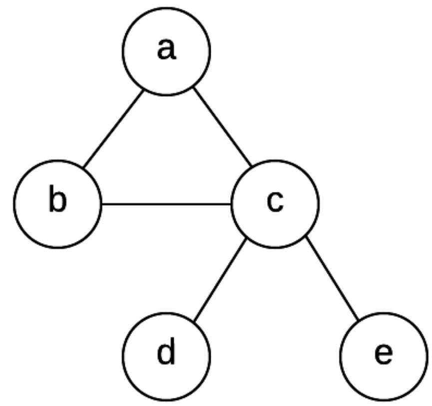
The cast functions could retrieve the value in the node.

3.1.3. Graph

Graph is a set of linked nodes, it’s like a component in the union-find problem, which means that two unlinked graph can’t be represented by one graph without operation. A graph variable keeps the following information:

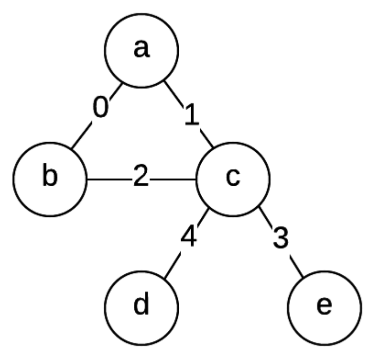
1. All nodes shown in the graph.
2. The Edge (direction, value), by which the nodes are connected.

3.1.3.1. Undirected Graph

Define undirected graph without edge value.	
`graph gh = a -- b -- c -- [a,d,e];` `print( gh.nodes() );`	
Output	

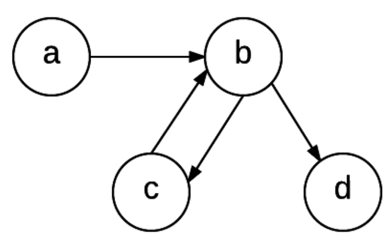
`list:[node  3: c`
`node  5: a`
`node  2: d`
`node  1: e`
`node  4: b`
`]`	

Define undirected graph with edge value.	

`graph gh= a--0&b--3&c--[1&a, 4&d, 3&e];`
`print( gh@(d,c) );`
`print( gh@(c,d) );`
	

Output	
`3` `3`	

3.1.3.2. Directed Graph

Define a simple directed graph, which could be defined by a single statement.	
`graph gh = a -> b -> [ c -> b, d ];` `print( gh );`	
Output	

`-------------------------------------`
`#Nodes: 4  Root Node: 5`
`node  4: b`
`node  3: c`
`node  2: d`
`node  5: a`
`#Edges: 4`	

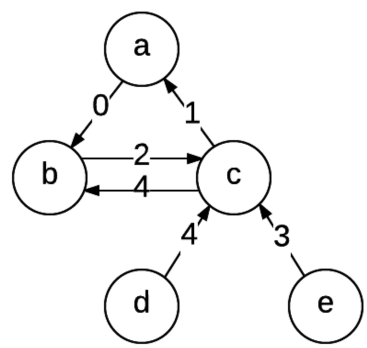
```
edge  3->  4
edge  4->  3
edge  4->  2
edge  5->  4
---------------------------------------
```

Define a complicated directed graph with edge values, which cannot be defined by a single statement, through graph merging.

```
graph gh =
a ->0& b ->2& c ->[1&a, 4&b] +
d ->4& c + e ->3& c;

print( gh );
```

Output

```
---------------------------------------
#Nodes: 5  Root Node: 5
node  3: c
node  5: a
node  4: b
node  2: d
node  1: e
#Edges: 6
edge  3->  5: 1
edge  3->  4: 4
edge  4->  3: 2
edge  5->  4: 0
edge  2->  3: 4
edge  1->  3: 3
---------------------------------------
```

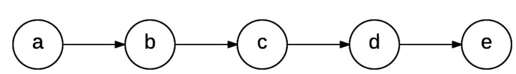


Define a linked list.

```
graph gh= a->b->c->d->e;
print( gh );
```

Output

```
---------------------------------------
#Nodes: 5  Root Node: 5
node  2: d
node  1: e
node  3: c
node  4: b
node  5: a
```



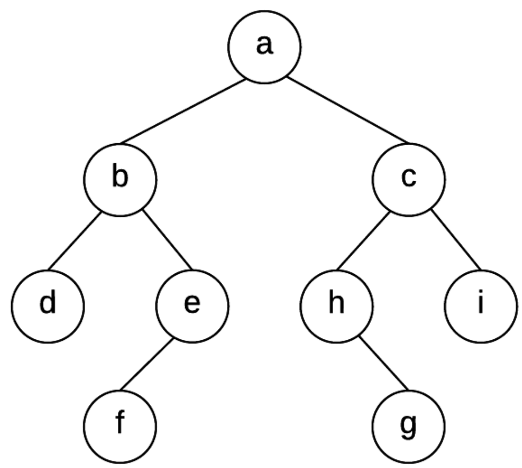
```
#Edges: 4
edge  2->  1
edge  3->  2
edge  4->  3
edge  5->  4
--------------------------------------
```

Define a binary tree Since the edge of BST has direction, assign a direction value for each edge..

```
bool l = true;
bool r = false;
graph gh = a -> [
      l&b -> [ l&d, r&e -> l&f],
      r&c -> [ l&h -> r&g, r&i ]
];
print( gh );
```

Output

```
--------------------------------------
#Nodes: 9  Root Node: 8
node  8: a
node  7: b
node  5: d
node  4: e
node  3: f
node  6: c
node  1: h
node  2: g
node  0: i
#Edges: 8
node  7->  5: true
node  4->  3: true
node  7->  4: false
node  8->  7: true
node  1->  2: false
node  6->  1: true
node  6->  0: false
node  8->  6: false
--------------------------------------
```



3.1.4. List

List literals are a sequence of literals enclosed in square braces. The items are separated by commas or semicolons.

`print( [1, 2, 3, 4] );`	`=> list:[1, 2, 3, 4]`

```
print( [1.2, -3.4] );            => list:[1.200000, -3.400000]
print( ["a", "ab", "abc"] );     => list:[a, ab, abc]
print( [true, false, 1>2] );     => list:[true, false, false]
```

This is not allowed: [1, “apple”, 3]

To assign the list to a variable, the type of the element must be declared.

```
list<int> li = [1, 2, 3, 4];
print(li);                       => list:[1, 2, 3, 4]
list<float> lf = [1.2, -3.4];
print(lf);                       => list:[1.200000, -3.400000]
list<string> ls = ["a", "ab"];
print(ls);                       => list:[a, ab]
list<bool> lb = [1<2, 1>2];
print(lb);                       => list:[true, false]
```

The list in our language support the following methods: size(), pop(), push(), add(), set(), remove(), get(). Details of these methods will be discussed in the List paragraph.

Attention: Lists are strongly typed — all elements of an list must be of the same type. Besides, <int> would be automatically converted to <float>, <node> would be automatically converted to <graph> when necessary.

```
print([1, 2, 3.]);               => list:[1.000000, 2.000000, 3.000000]

list<graph> lgh = [a, b->c ];

print( lgh.get(0).size() );      => 1
print( lgh.get(1).size() );      => 2
```

3.1.5. Dict

Dict defines a list of key-value pairs. The key-value pairs in it is are enclosed by curly braces, and are separated by commas.
Supported key types: int, string and node.
Supported value types: int, float, string, node, graph
The value should be subject to the convention of the identifiers, and it should be unique. For a given dict, both key and value could be only one type, which can be chosen from all types in Circline, and we should declare it. A key-value pair should contain both key and value, no single key or single value is allowed.
Here are some examples:

```
dict<string> = {“name1”: “circle”, “name2”: “heha”};
dict<int> = {1: “circle”, 2: 1};
```

```
dict<string> = {“name1”:1, 1:”name2”}; /* error */
```

3.2. Operators and Expressions

3.2.1. Comments

Only one type of the comment is accepted, enclosed by “/*” and “*/”, such as

```
/* write an Ocaml a line */
```

```
/* Write an Ocaml a line,
     Keep you happy a day */
```

The elements in the middle are automatically ignored.

3.2.2. Identifiers

Identifiers are sequence combination of letters (both upper and lower case), digits, and underscores. The function identifier and variable identifier is the same type and the starting element of a function name must be a letter.

Valid names:

```
Merge_sort
apple
a3b_a21
a12
I_Like_Ocaml
```

Invalid names:

```
_bash
1st
3a5
I-Like-ocaml
```

3.2.3. Arithmetic Operators

The arithmetic operators are +, -, *, /, %.

The + and - operators have same precedence, * and / have same precedence. * and / have a higher precedence than + and -. We support automatic promotion of int and float, which we can have int + int, float + float and int + float, same as other arithmetic operators. But other primary types are not allowed as for the arithmetic operators.

Valid input:

```
1 + 2
1 - 2
1 * 2
1 / 2
1.0 + 2.0
1.0 * 2, 1 / 2.0
1002 % 2
```

Invalid Input:

```
1.0 + true
 1 + “”
 1 + {}
1 + [].
```

3.2.4. Logical and Relational Operators

Relational Operator >, <, >=, <= and == are in the same precedence, which is lower than round bracket and higher than and, or, not. For example:

```
if(a==1 and b <=2)
```

means both a equals to 1 and b smaller or equals to 2.

```
if(c or (a and b))
```

means boolean variable a, b and c are judged by c or result of (a and b). If a = true, b = true, c = true the result will be true.

3.2.5. List Operators

3.2.5.1. .size()

Size operator can return the length of a list, it’s used as aList.size().

`list<int> aList= [1, 2, 3, 4];` `aList.size();`	`=> 4`

3.2.5.2. .get(int index)

A specific element can be selected and used by using the index of the element:

`list<int> aList= [1, 2, 3, 4];`	

`aList.get(1);`	`=> 2`

Besides, user can get the last element of a list by setting the index as -1.

`aList.get(-1)` `aList.get(-2)`	`=> 4` `=> 3`

3.2.5.3. .set(int index, list.type value)

List can change the value of an element in the list by calling set() function:

`[1,2,3,4,5].set(1,3);` `[“Dog”, “is”, “a”, “nerd”].set(0,”Pig”);`	`=> [1,3,3,4,5]` `=> [“Pig”, “is”, “a”, “nerd”]`

3.2.5.4. .add/push(list.type value)

List can easily append new elements using +. What is need to be careful is that, the element to be appended should be the same type as the list, and it should be only one element, to append another list, see the section Concat.

`[1].add(2);`
`[“str1”].add(“str2”);`
`[1].add( “str”);`
	`=> [1,2]`
	`=> [“str1”,”str2”]`
	`=> Error`

3.2.5.5. .pop()

List can use pop to remove the last element from the list.

`[1,2,3].pop()`	`=> [1,2]`

3.2.5.6. Concat (+)

As mentioned in Append, two list can be concatenated together also using +. Similarly, the two list should be same type, otherwise error is reported.

`[1,2,3] + [4,5,6]`
`[“str1”,”str2”,”str3”] + [“str4”,”str5”,”str6”]`
[“str1”,”str2”,”str3”] + [1,2,3]
	`=> [1,2,3,4,5,6]`
	`=> [“str1”, ”str2”, ”str3”, “str4”, ”str5”, ”str6”]`
	`=> ERROR`

3.2.5.7. .remove(int index)

User can delete a specific element of the list using .remove(index), for example:

`string[] aList = [“big”, “fat”, “cat”, “cat”];` `aList.remove(0);`	`=> [“fat”, “cat”, “cat”]`

`aList.remove(-1);`	`=> ["fat", "cat"]`

Example:

`list<int> li = [1, 2, 3];`
`print(li);`
`li.add(4);`
`print(li);`
`print(li.get(0));`
`li.set(0, 4);`
`print(li);`
`li.remove(0);`
`print(li);`
`print(li.size());`
`print(li.pop());`
`print(li);`
`print(li.push(5));`
	
	`=> list:[1, 2, 3]`
	
	`=> list:[1, 2, 3, 4]`
	`=> 1`
	
	`=> list:[4, 2, 3, 4]`
	
	`=> list:[2, 3, 4]`
	`=> 3`
	`=> 4`
	`=> list:[2, 3]`
	`=> list:[2, 3, 5]`

3.2.6. Dict Operators

3.2.6.1. .get(dict.keytype key)

The value could be easily obtained from dict by calling get function. The current keytype: Node, String and Integer.

`dict<int> aDict = {"pig": 123}`
`aDict.gete("pig")`
`dict<string> bDict = {10: "Dog"}`
`bDict.get(10)`
	
	`=> 123`
	
	`=> "Dog"`

3.2.6.2. .remove(dict.keytype key)

aDict.remove(keyname) allows user to remove a specific key-pair value. However, the keyname should exist, otherwise there will be an error.

`dict<string> d = {`
`    "pig": "some",`
`    "dog": "sam"`
`}`
`d.remove("pig");`
`d.remove("pig");`
	
	`=> { "dog": "sam" }`
	`=> Error`

3.2.6.3. .put(dict.keytype key, dict.valuetype value)

The put function can insert a new key-value pair to the existing dictionary.
aDict.put(“hey”, “buddy”); /* aDict={“hey” : “buddy” }*/

3.2.6.4. .size()

Return the size of Dictionary

3.2.6.5. .keys()

Return the List of keys.
{10: “hi”, 20 : “aa”, 30: “bb”}.keys(); /* return [10,20,30] */

Example:

`dict<int> d_int =`	
`    {1: 11, 2: 22, 3: 33};`	
`print(d_int);`	`=> {2: 22, 1: 11, 3: 33}`
`print(d_int.get(1));`	`=> 11`
`print(d_int.put(4, 44));`	`=> {2: 22, 1: 11, 3: 33, 4: 44}`
`print(d_int.remove(2));`	`=> {1: 11, 3: 33, 4: 44}`
`print(d_int.size());`	`=> 3`
`list<int> l_int =`	
`d_int.keys();`	
`print(l_int);`	`=> list:[1, 3, 4]`
`print(d_int.has(2));`	`=> false`
`print(d_int.has(3));`	`=> true`

3.2.7. Graph

3.2.7.1. Definition

There are three link operators:
1. “--” Double Link
2. “->” Right Link
3. “<-” Left Link

3.2.7.1.1. <node> <op> <Edge Value> & <node> => <graph>

Link two nodes together with specified edge value, and return a graph, whose root is the first node.

3.2.7.1.2. <node> <op> <Edge Value> & <graph> => <graph>

Attention: The following statements are equal to each other.
1. a -- 1&b-- 1&c
2. a -- 1& (b--1&c)

3.2.7.1.3. <node> <op> <Edge Value> & [<node/graph>] => <graph>

3.2.7.1.4. <node> <op> [<Edge Value> & <node/graph>] => <graph>

If the second operand is of type graph[], link the first node with a list of graphs by connecting the node and roots of graphs with the same edge value.

a -- 1& [b--2&c, d--3&e] =>
a -- [1& (b -- 2&c), 1& (d -- 3&c)] =>
a -- [1&b -- 2&c, 1&d -- 3&c]

If the second operand is of type node[], link the first node with all nodes in the list with the same edge value.

a -- 2&[b, c, d] => a -- [2&b, 2&c, 2&d]

Attention 1: If both node and graph are existed in the same list, all nodes will be automatically converted to graphs with single node.

a -- 1&[b, c --2&d] =>
a -- [1&b, 1&(c -- 2&d)] =>
a -- [1&b, 1&c -- 2&d]

Attention 2: If the edge value are not of the same, must use the full definition.
a -- [1&b, 2&c, 3&d --4&e]

a -- [1&b, 2&c -- 3&[d, e]] =>
a -- [1&b, 2&c -- [3&d, 3&e]]

3.2.7.2. Methods

3.2.7.2.1. .root()

Return the root of a graph.

(a -- b -- c).root() => a

3.2.7.2.2. .size()

Return the size of the graph (number of nodes).

(a -- b -- c).size() => 3

3.2.7.2.3. .nodes()

Return a list of nodes in the graph with random order.

(a -- b -- [c, d]).nodes() => [a, b, c, d]

Exemples:

graph gh = a->b->c	=> /* Define a new graph */
gh.root()	=> a
gh.size()	=> 3 /* Num of nodes */
gh.nodes()	=> [b, c, a] /* List of nodes */
(d<-e).root()	=> d
(a--[b,c]).root()	=> a
((a--[b,c])~c).root()	=> c
(a->[b->c, d<-e]).size()	=> 5
(a->[b->c, d<-e]).nodes()	=> [a, b, c, d, e]

3.2.7.3. Operator

3.2.7.3.1. Reset Root: <graph> ~ <node>=> <graph>

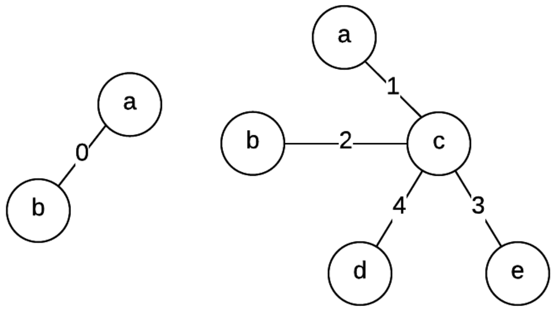
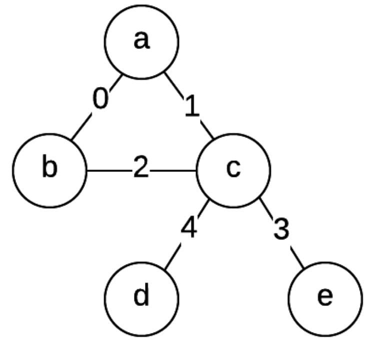
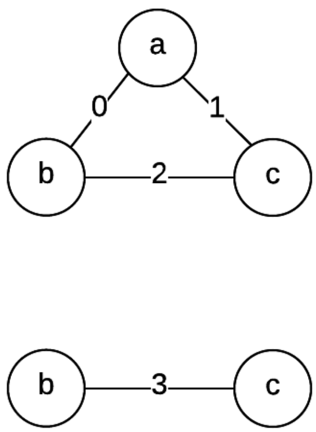
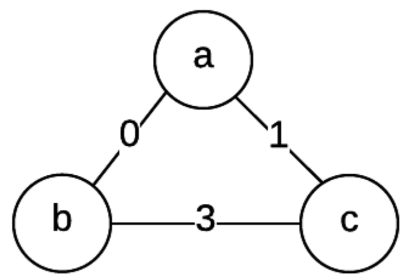
Change the root of a graph to the specific node, and return a new graph. If the node is not existed in the graph, throws an error.

graph g1 = a->b->c	=> /* Define a new graph*/
g1.root()	=> a
g2 = gh~b	=> /* return a new graph */
g1.root()	=> a /* old graph's root remains unchanged */
g2.root()	=> b /* new graph with different root */
((a -- b -- c) ~ b).root()	=> b

3.2.7.3.2. Merge Graph: <graph> + <graph>=> <graph>

Merge the nodes and edges of the two graphs, if there is a conflict in the edge, use the edge value in the second graph. The root of the returned graph is the same as the first graph.
Attention:
The two graphs should have shared nodes. Otherwise, return a new graph which is exactly the same the first graph.

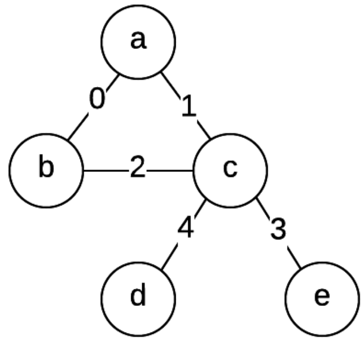
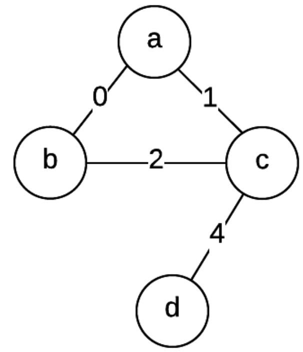
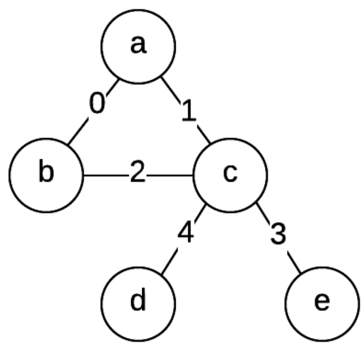
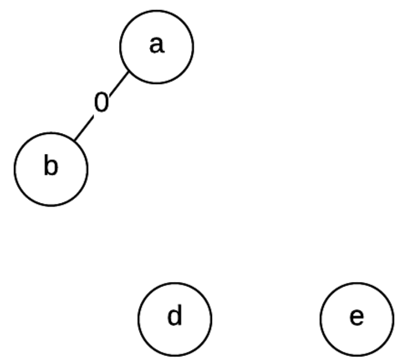
a -- 0&b + c -- [1&a, b&2, 4&d, 3&e]=> [a -- 0&b -- 2&c -- [1&a, 4&d, 3&e]]	
Original Graph	Return

	
a -- 0&b -- 2&c -- 1&a + b -- 3&c => a -- 0&b -- c&3 -- 1&a	
Original Graph	Return
	

3.2.7.3.3. Remove Nodes: <graph> - <node>=> <graph>[]

The only operator available here is the delete "-", which would remove the specific nodes as well as all connected edges from the graph and return a list of remaining graphs. The root of the first graph in the list is guaranteed to be the original root, unless the node got deleted is the root itself, in which case the root is randomly assigned. For the graphs other than the first in the return list, the root node is randomly assigned.

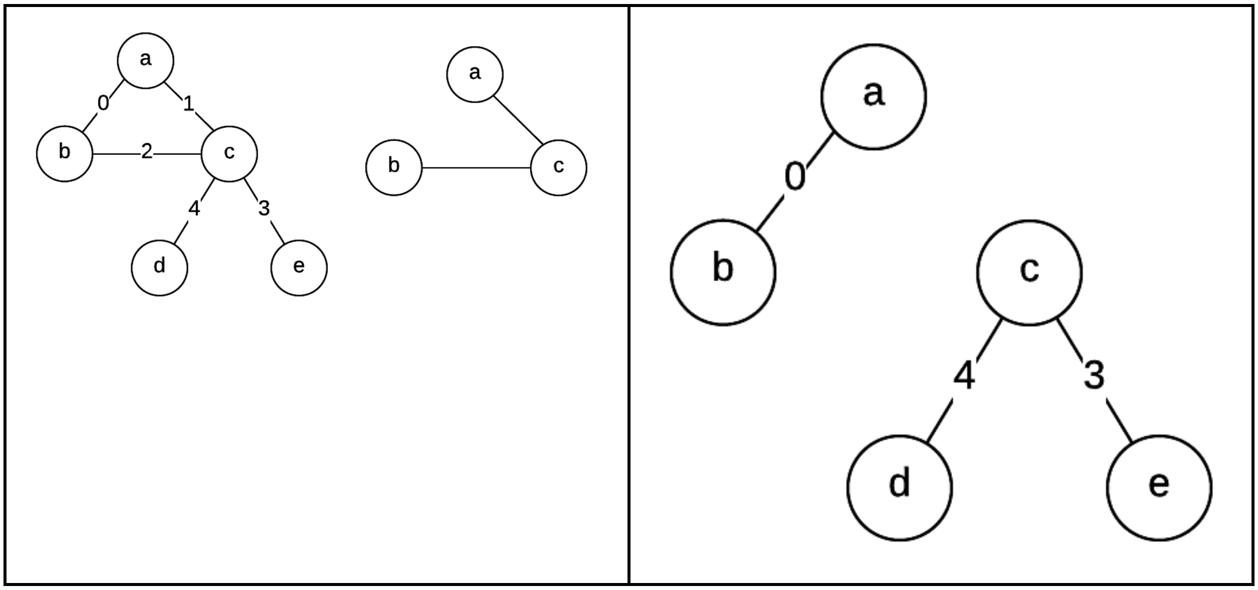
a -- 0&b -- 2&c -- [1&a, 3&d, 4&e] - e => [a -- 0&b -- 2&c -- [1&a, 4&d]]	
Original Graph	Return

<table>
<tr><td>
</td><td>
</td></tr>
<tr><td colspan="2">a -- 0&b -- 2&c -- [1&a, 3&d, 4&e] - c =>
[a -- 0&b, d, e]</td></tr>
<tr><td>Original Graph</td><td>Return</td></tr>
<tr><td>
</td><td>
</td></tr>
</table>

3.2.7.3.4. Remove Edges: <graph> - <graph>=> <graph>[]

Remove the edges from the first graph if the edge is existed in the second graph, regardless of the edge value. The return value is a list of graphs. The first graph in the list share the same root with the original first graph. For other graphs in the list, the root is randomly assigned.

<table>
<tr><td colspan="2">a -- 0&b -- 2&c -- [1&a, 4&d, 3&e] - a -- c -- b =>
[a -- 0&b, d -- 4&c -- 3&e]</td></tr>
<tr><td>Original Graph</td><td>Return</td></tr>
</table>



3.3. Control Flow

3.3.1. Loops

<table>
<tr><th colspan="2">while loops</th></tr>
<tr><td><pre>dict<int> di = {0:0};
int i = 1;
while (i<3) {
      di.put(i, i);
      i = i + 1;
}
print(di);</pre></td><td>=> {2: 2, 1: 1, 0: 0}</td></tr>
<tr><th colspan="2">for loops</th></tr>
<tr><td><pre>list<int> li = [0];
int i;
for (i=1; i<5; i=i+1) {
      li.add(i);
}
print(li);</pre></td><td>=> list:[0, 1, 2, 3, 4]</td></tr>
</table>

3.3.2. Conditionals

There are only one form of conditional expressions in our language:

if (boolean expression) {

statement

} else {
statement
}

Here is a simple example:

`int a = 2;`
`if(a>3) {`
`      print(10);`
`}`
`else {`
`      print("True");`
`}`
	`True`

3.4. Program Structure

3.4.1. Functions

3.4.1.1. Default Functions

void print(<T> args ...)
Arguments Type: One or more arguments of the following type <T> = <int> <float> <bool> <string> <list> <dict> <node> <graph> <null>
Single Argument Example: print(1) => 1 print(-1.2) => -1.20000 print(1>2) => false print(true) => true print(“Hello World!”) => Hello World! print([1,2,3]) => list: [1, 2, 3] print({“a”: 1}) => {a: 1} print(node(“a”)) => node 0: a **Multi Arguments Example:** print(1,true,3) => 1 true 3
void printf(<string>, <T> args...)
Arguments Type: The first argument should be a format string (%d, %f, %s). One or more arguments of the following type

<T> = <int> <float> <string>

int a = 1;
float b = 1.2;
string d = "What!";
 	
printf("%d--\n%.2f--\n%s\n", a, b , d) =>
1--
1.20--
What!

int int(<T> arg)
Arguments Type: <T> = <int> <node> <edge>

int(1) => 1
int(node(12)) => 12 // Get the node value
int((a->2&b)@(a,b)) => 2 // Get the edge value

float float(<T> arg)
Arguments Type: <T> = <int> <float> <node> <edge>

float(1.2) => 1.200000
float(1) => 1.000000
float(node(1.2)) => 1.200000 // Get Node Value
float((a -- (3.2)& b)@(a,b)) => 3.200000 // Get Edge Value

bool bool(<T> arg)
Arguments Type: <T> = <bool> <node> <edge>

bool(1>3) => false
bool(1<3) => true
bool(node(2>3)) => false // Get Node Value
bool((a -- (2<3)& b)@(a,b)) => true // Get Edge Value

int int(<T> arg)
Arguments Type: <T> = <int> <node> <edge>

int(1) => 1
int(node(12)) => 12 // Get the node value
int((a->2&b)@(a,b)) => 2 // Get the edge value

3.4.1.2. Customized Functions

Functions are defined by normal C style, as shown in the following example.

```
string hello() {
    return “Hello world!”;
}

/* Return value will be cast to return type if possible */
float somefunction( int a ) {
    return 1 + a;
}

/* Usage of null */
node printNode( node a ) {
    print(a);
    return null;
}
```

3.4.2. Scoping & Nested functions

The outest scope is the whole program, which is also called global scope. Inside the program, you could create local scope such as functions. Local scope could access the value of outer scope. When the program looks for a variable, it first find the variable in local scope. If not found, it will look at the outer scope until global scope. If it could not find the variable in any scope, the program will raise exception. That’s to say, you could access the variable of outer scope.

```
int d = 1;
int b(int c) {
  int d = 2;
  int a() {
    return d + c;
  }
  return a();
}
print(b(3));     /* Output 5 */
print(d);        /* Output 1 */
```

4. Project Plan

The group met two to three times per week (Wednesday after class and Monday before meeting TA). After several group discussion at the beginning, a draft timeline table was set down at the beginning of the semester. The timeline table indicate the major milestones of the project and its corresponding person. The responsible member is the person in charge of the target and all the other member should generally follow the plan that responsible member decided. All the members should follow code, test and push three steps. The implementation of circline could be generally categorized into three steps.

The first two deadline is considerable a good practise on teamwork. During these time, each of the member found their personal talent in this project and ready to tackle the challenge. The implementation of the language is the most funny, boring and hard time. Since the start of scanner, parser, the team stick on thinking of LRM and implementing them really fast. However, since the start of code generation and semantic check, the team realize, an ignorable design error were occurred at the beginning. The actual LLVM would only support a C flavored grammar.

Several discussion were focusing on the syntax of the language. The team agreed to keep the original syntax and do transformation of the token from the parser to the semantic and code generation. The name of this operation is called Second Step implementation. During this great expedition, the organizer were created to translate the language and BFS search were applied in Ocaml to rearrange the function and variables. After the implementation of it. The problem on code generating was solved.

The Third Step implementation is to appeal the need of this language, supporting on List, Dict, Graph and Node operation. At this stage, the team decided to write the C library from scratch to implement all the complex data structure. Then, the Code generation would call these data structures and semantic could start checking on them. On 14th Dec 2016, the overall Circline body was completed and it could do BFS and DFS search. On 18th Dec 2016, the support on Dijkstra Algorithm indicate that the Circline is fully functional. The team is moving on to final error checking procedure. The automated testing suite implemented at the beginning was helpful to accelerate the testing speed and the warning were solved in each stage of Circline.

4.1. Ocaml Style

• Indent with two spaces.
• Indent to indicate scope.
• Never Wrap lines.
• Comments are not required, but should be included for confusing or weird-looking code.
• Pattern match all cases.
• Use a pipe character | with all match cases, including the first one.
• Be as specific as possible when throwing exceptions.
• Do not repeat code — refactor if possible.
• Be descriptive and consistent in naming everywhere.
• Use lowercase letters and underscores in naming.

4.2. Circline Style

• Indent with four spaces.
• Indent to indicate scope.

• No wrap lines.
• Array items should be by default be single space separated, with a space separating the open and closing brackets/braces.
• The closing brace/bracket/parenthesis on multi-line constructs should be lined up under the first character of the line that starts the multi-line construct.
• Use camelCase for both variable names & function names.
• Writing multiple statements on the same line is discouraged.
• All variables should be initialized in declaration.
• Declare all variables at the beginning of the functions.

4.3. Project Timeline

Time	Achievement
10.22	Finish Scanner and Ast
10.23	Finish Tokenize and test files
10.25	Established Travis CI online testing
10.31	Build Parser and parserize file for testing
11.1	Makefile Linking the whole project
11.6	Parser second step implementation, start code gen and semantic
11.13	Finish parser, add cast and start on Organizer
11.19	Organizer first step complete
11.20	Hello World to Circline
11.26	Organizer BFS complete and Semantic second step complete
11.30	Code gen second step complete
12.1	Start on C library linking
12.6	Graph and Node C library finished
12.15	List and Dict C library C library finished
12.17	Major bug fix, Semantic third step complete
12.18	Dijkstra Algorithm complete, maintenance on data structure APIs
12.19	Final Checking

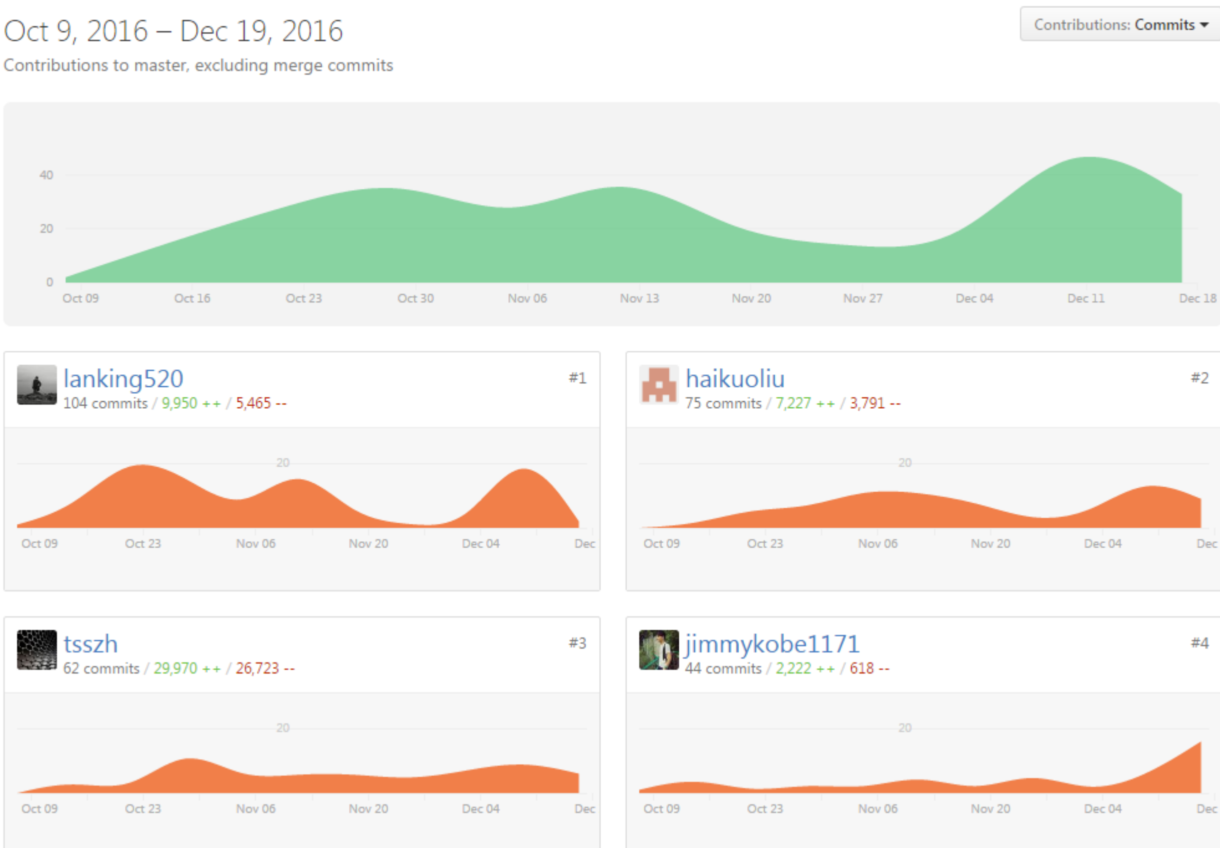
For more information, please check https://github.com/jimmykobe1171/circline/commits/master

4.4. Roles and Responsibilities

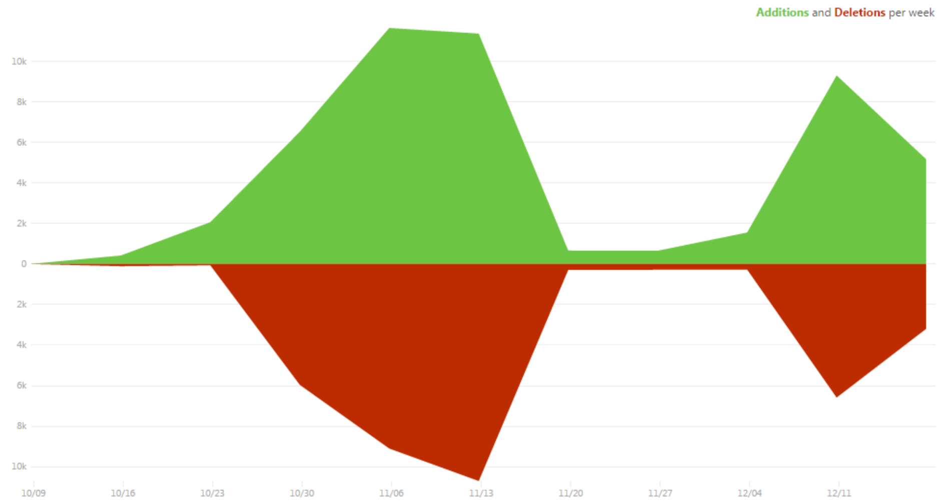
Target	Responsible Member
The purpose and usage of language	Everyone
Preliminary Project Plan	Jia Zhang
Language Definition and Syntax	Zehao Song
Implementation Procedure and goals for each step	Haikuo Liu
Testing plan and automated testing suite design	Qing Lan
Scanner design and implementation	Zehao Song
ast file and tokenize testing suite	Jia Zhang
Parser design and first step implementation	Haikuo Liu
Mid term project summary and plan for next term	Jia Zhang
Parser design second step Level adding	Qing Lan
Semantic Checking Plan and implementation	Jia Zhang
Code Generation First step implementation	Zehao Song
The need of organizer! First step implementation	Qing Lan
Maintenance of Parser and Scanner for current plan	Haikuo Liu
Code Generation Second step implementation	Zehao Song, Haikuo Liu
Testing suite maintenance for semantic check and code gen	Qing Lan
Semantic Check Second Step implementation	Jia Zhang
Organizer Modification and Code Generation Replanning	Haikuo Liu, Qing Lan
C External Library Implementation - Dict & List	Qing Lan, Haikuo Liu
C External Library Implementation - Graph	Zehao Song
Code Generation Third Step implementation	Haikuo Liu, Zehao Song

Code generation and Semantic Checking sync	Zehao Song, Jia Zhang
Finalizing system design	Jia Zhang

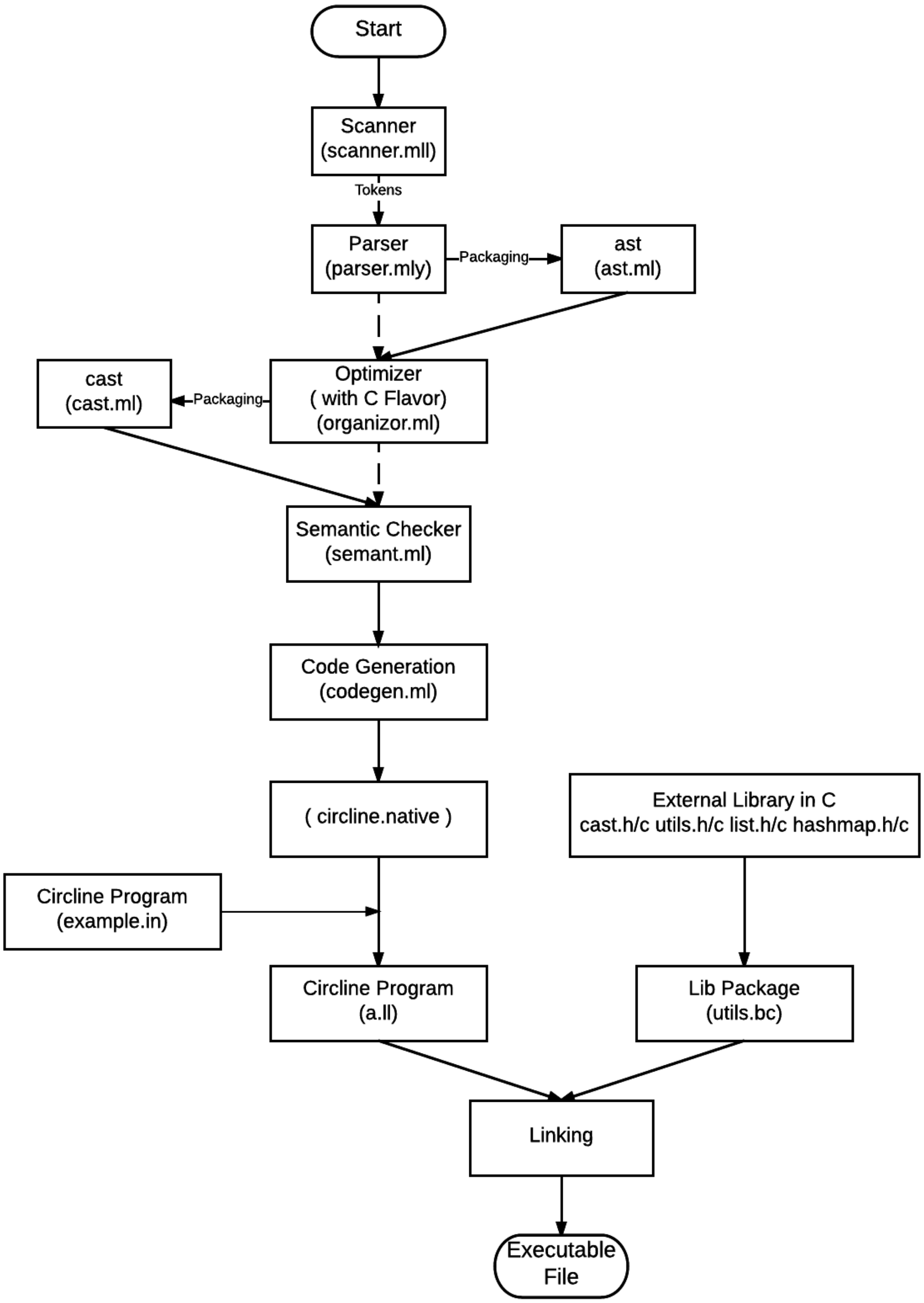
4.5. Commits



4.6. Code Frequency



5. Language Architecture



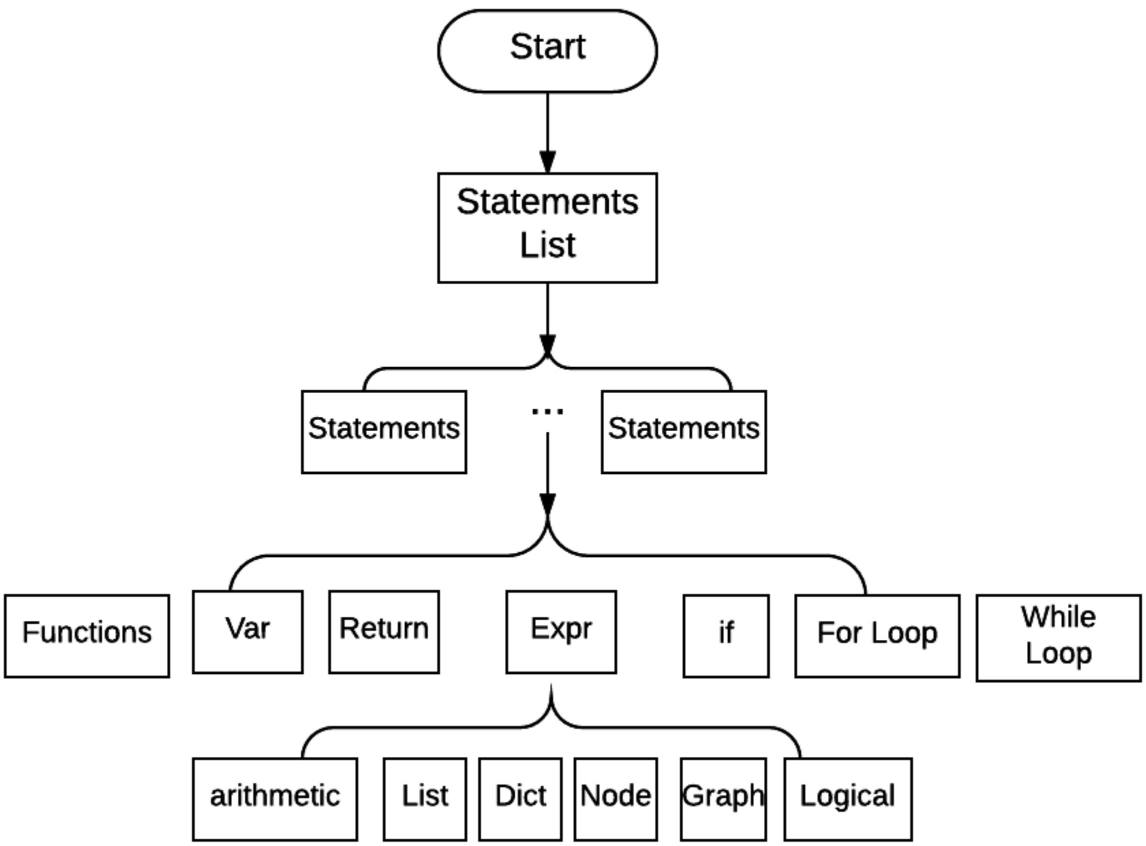
5.1. Scanner (scanner.mll)

The scanner generates tokens, which are keywords, arithmetic operators, graph operators, logical operators, primary types, literals, and symbols. Apart from matching regular expressions, its tasks are:

1. Converting escape sequences into string literals.
2. Discarding whitespace and characters that are no longer needed.
3. Removing comments (for each /* comments */).

5.2. Parser (parser.mly)

The parser takes in the tokens passed to it from scanner and produce an abstract syntax tree (AST) based on the definitions provided and the input tokens. The top level of the AST is a list containing all statements. Then we parse the statements list into functions, variables, etc. The layout of the Parser can be represented as following:



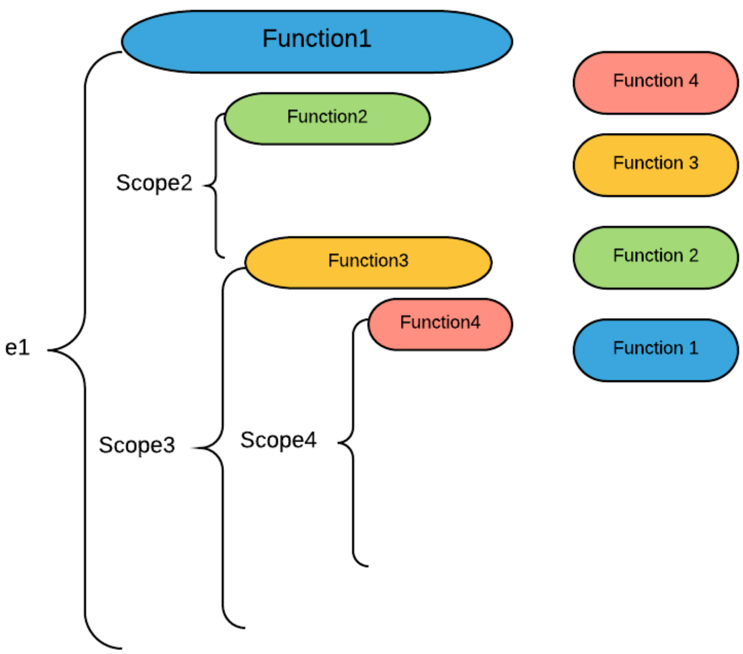
5.3. Optimizer (organizer.ml)

The major need of the organizer is to reconstruct the Circline to be closed to C. The way it works is similar to Cython to Python. The Script-Language grammar would work well in Circline however not ideal to LLVM. The major job could be concluded into two parts:

1. Lift all Variable declarations to the front of its scope.
2. Rearrange the order among all function declarations and variable

For the first part, organizer would store all variables in the 'locals' field of its corresponding function scope. These variable declaration would be placed directly after the declaration of the function. For the case such as int apple = 10; The statement is splitted into two parts: Keep apple = 10 at it original position and move int apple; to the front. This would ensure the variable would not be assigned a value until the original position. A function label would add to the front of variable, which provide an ease for semantic check used to identify the variables has the same name however from different scopes.

The second problem needs a complex algorithm to solve, which could be described as shown in the graph below:



The nested function could be in any types of data structures. However, these function could only be declared at the beginning in c in order to use them. Hence, a Breadth-First Search would be applied in here. All the functions were arranged inside out. Similarly, we label the function name with their parent names and leave the usage of function right at the position and drag the declaration at the front. After these operation, the final tokens would be necessary to be parsed in C. In brief, Organizer establish a bridge between Circline and C.

5.4. Semantic Check (semant.ml)

The input of semantic check module is the output of Optimizer discussed above, which is a list of function objects defined in cast.ml.

For each function, we need to check its returnType, args, locals and body. For example, the function body consists of a list statements, so we need to check every statement in function body. For a statement, it consist of expression, so we need to check every expression. This is kind of performing DFS on the AST tree.

There is one thing that need to be mentioned. Since we support nested function, we need to check the situation that accessing a variable that is defined in the scope of its parent function. As described in Optimizer, we store the parent name for every function. When we encounter an expression that evaluate a variable, we first loop up the variable in local scope. If we could not find it in local scope, we will use the parent name stored in function object to access the parent scope and find the variable there. Such process continues until we reach the main scope, which is the outest scope and acts like global scope. If we could not find the variable at the end, we will raise exception.

5.5. Code Generator (codegen.ml)

The input of the code generator is the semantic-checked AST. The output of the code generator is a **.ll** file in LLVM syntax.
The structure of the code generator is very similar to that of **microC**.

1. Declare the context, module and linked C library.
2. Declare all types (int, float, bool, string, list, dict, node, graph).
3. Declare all external functions
4. Declare the name-function map & name-variable map
5. Translate each function in the program
    a. Declare all variables and stored into name-variable map
    b. Translate each statement
        i. Translate each expression

Compared to **microC**, statements in **circline** are the same. The expressions are not the same.
For LLVM primary types, including int, float, bool, the operation are directly built in ocaml.
For complicated struct type, all operations are handled by C Library. The basic idea is pass the struct pointers between each declared C functions.

5.6. C library

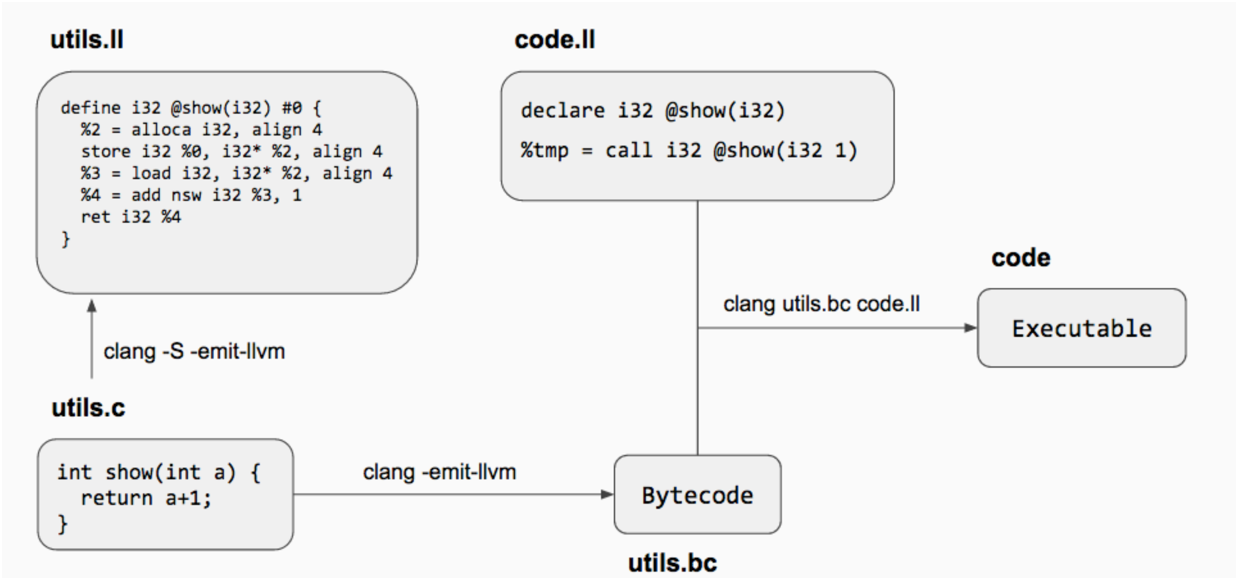
Most of the complicated functions are implemented by C, including:

1. List Operation
    a. create list
    b. concat list

c. list add/push elements
d. list get elements
e. list setelements
f. list remove/pop elements
g. get list size
h. print out list
2. Dict Operation
   a. create dict
   b. dict put key-value pairs
   c. dict remove key-value pairs
   d. dict get value
   e. check key existence in dict
   f. get dict size
   g. get list of keys
   h. print out dict
3. Node Operation
   a. create node
   b. get node value
4. Graph Operation
   a. create graph
   b. add nodes / edges
   c. remove nodes
   d. graph merge
   e. graph subtraction
   f. graph get / set root
   g. graph get all nodes
   h. get neighbors of node in the graph
   i. graph get edges
   j. print out graph

How to use the C library in ocaml code generator? Here is a brief illustration.

First, write the C functions in utils.c, eg. **show()** functions.
Second, compile the utils.c file into LLVM IR by the command **clang -S -emit-llvm**. As shown in the graph (**utils.ll**), a function with name **show** is defined.
Third, in the **codegen.ml**, an external function should be declared and called when necessary, as shown in the graph (**code.ll**)
Finally, combine / link **code.ll** and **utils.bc** together to generate the final executable file **code**.



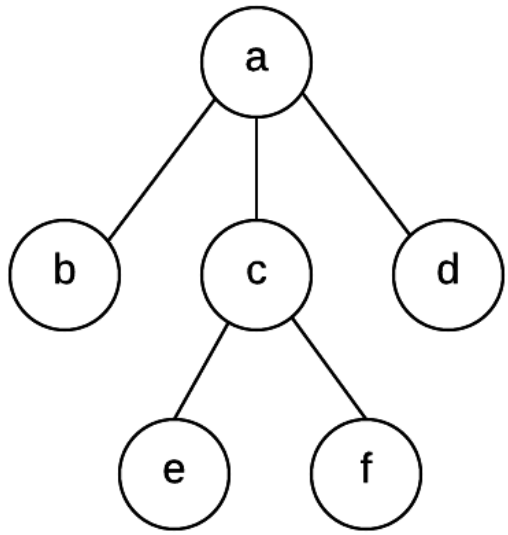
6. Testing

6.1.1. Examples

6.1.1.1. > Breadth-first search (BFS)

```
list<node> bfs(graph gh, node r) {
  if (gh == null or gh.size() == 0) { return null; }

  int i; node curr; node tmp_n; list<node> children;
  dict<node> set = { r: r };
  list<node> res = null;

  list<node> queue = [ r ];
  while (queue.size() > 0) {
    curr = queue.get(0); queue.remove(0);
    if (res == null) { res = [curr]; } else { res.add(curr); }

    children = gh@curr;
    for (i=0; i<children.size(); i=i+1) {
      tmp_n = children.get(i);
      if (not set.has( tmp_n )) {
        set.put( tmp_n, tmp_n );
        queue.add(tmp_n);
      }
    }
  }
```

```
  return res;
}
```
Here are examples:



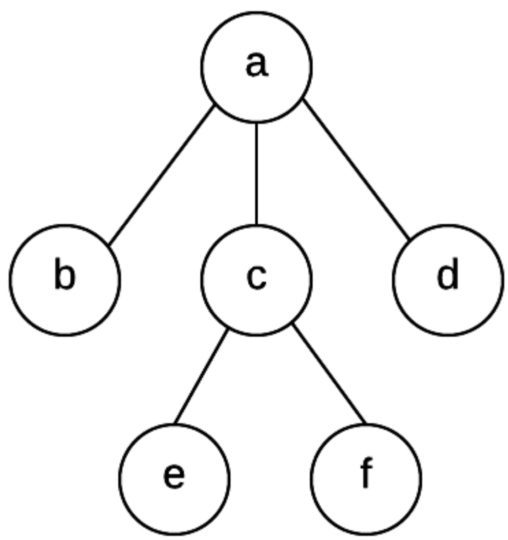
bfs(gh, a)	=> [a, b, c, d, e, f]
bfs(gh, b)	=> [b, a, c, d, e, f]
bfs(gh, c)	=> [c, e, f, a, b, d]
bfs(gh, d)	=> [d, a, b, c, e, f]
bfs(gh, e)	=> [e, c, f, a, b, d]
bfs(gh, f)	=> [f, c, e, a, b, d]

6.1.1.2. > Depth First Search (DFS)

```
list<node> dfs(graph gh, node r) {
  if (gh == null or gh.size() == 0) { return null; }

  int i; node curr; node tmp_n; list<node> children;
  bool found;
  dict<int> set = { r: 0 };
  list<node> res = [r];

  list<node> stack = [ r ];
  while (stack.size() > 0) {
    curr = stack.get( stack.size() - 1 );
    set.put(curr, 1);

    children = gh@curr;
```

```
    found = false;
    for (i=0; (not found) and (i<children.size()); i=i+1) {
      tmp_n = children.get(i);
      if (not set.has( tmp_n )) { set.put( tmp_n, 0 ); }
      if (set.get(tmp_n) == 0) {
        stack.push(tmp_n);
        res.add(tmp_n);
        found = true;
      }
    }
    if (not found) {
      set.put(r, 2);
      stack.pop();
    }
  }

  return res;
}
```

Here are examples:



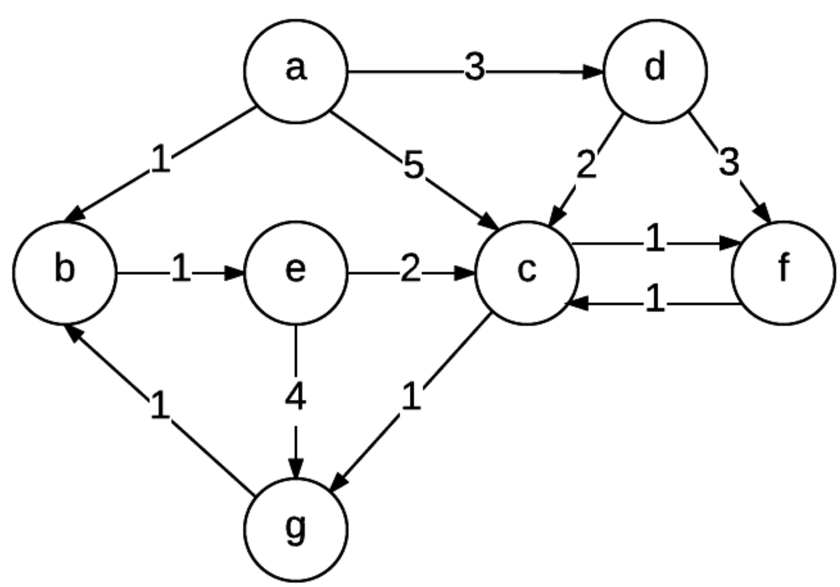
dfs(gh, a)	=> [a, b, c, e, f, d]
dfs(gh, b)	=> [b, a, c, e, f, d]
dfs(gh, c)	=> [c, e, f, a, b, d]
dfs(gh, d)	=> [d, a, b, c, e, f]
dfs(gh, e)	=> [e, c, f, a, b, d]
dfs(gh, f)	=> [f, c, e, a, b, d]

6.1.1.3. > Dijkstra’s Algorithm for Shortest Path

```
void dijkstra(graph gh, node sour) {
      dict<int> distance = { sour: 0 };
      list<node> queue = gh.nodes();
      dict<node> parent = {sour: sour};
      int i;
      for (i=0; i<queue.size(); i=i+1) {
            distance.put(queue.get(i), 2147483647);
            parent.put(queue.get(i), null);
      }
      distance.put(sour, 0);

      while (queue.size() > 0) {
            updateDistance( findMin() );
      }
      queue = gh.nodes();
      for (i=0; i<queue.size(); i=i+1) {
            showRes(queue.get(i));
      }

      node findMin() {
            node minNode = queue.get(0);
            int minDis = distance.get(minNode);
            int minIndex = 0;

            int i; node tmp;
            for (i = 1; i < queue.size(); i=i+1) {
                  tmp = queue.get(i);
                  if ( distance.get(tmp) < minDis ) {
                        minNode = tmp;
                        minDis = distance.get(tmp);
                        minIndex = i;
                  }
            }
            queue.remove(minIndex);
            return minNode;
      }

      void updateDistance(node u) {
            int i; int dv; int dis; node v;
            list<node> neighs = gh@u;
            int du = distance.get(u);
            for (i = 0; i<neighs.size(); i=i+1) {
                  v = neighs.get(i);
                  dv = distance.get(v);
                  dis = int( gh@(u, v) );
                  if ((dis + du) < dv) {
```

```
                        distance.put(v, dis+du);
                        parent.put(v, u);
                  }
            }
      }

      void showRes(node dest) {
            list<node> res = [dest];
            node tmp = parent.get(dest);
            while (tmp != null) {
                  res.add( tmp );
                  tmp = parent.get(tmp);
            }
            int i;
            printf("%s -> %s : %d [ ", string(sour), string(dest),
distance.get(dest) );
            for (i=res.size()-1; i > 0; i=i-1) {
                  printf("%s, ", string( res.get(i) ));
            }
            if (i == 0) {
                  printf("%s ]\n", string( res.get(i) ));
            } else {
                  print("]");
            }
      }
}
```

Here are examples:



```
Dijkstra Results:
a -> a : 0 [ a ]
a -> e : 2 [ a, b, e ]
a -> g : 5 [ a, b, e, c, g ]
a -> b : 1 [ a, b ]
a -> c : 4 [ a, b, e, c ]
a -> f : 5 [ a, b, e, c, f ]
a -> d : 3 [ a, d ]
```

6.1.2. Automated Test Suite

By simply calling **make test**, the functionalities of scanner, parser, semantic checker would be tested. Test cases for each file were created in the tests folder. Before the implementation of each component of circline, the test cases would be built for future verification. In order to compare the input and output, several helper Ocaml file were created. For Scanner, a tokenize.ml file were created to convert the output of scanner to string. Parser use parserize.ml to printout the corresponding ast function called. Semantic check used semantic_check.ml to print out the error information.

```
aList = ["str1","str2","str3"];
int i;
for(i = 0; i<= 5; i=i+1){
    if (str == "str2"){
        3+2;
    }
    else{
        /* do something */
    }
}
```

```
Expr(Assign(aList, List(String_Lit(str1), String_Lit(str2), String_Lit(str3))));
Var_dec(Local(int, i, Noexpression));
For(Assign(i, Num_Lit(0));Binop(Id(i), Leq, Num_Lit(5));Assign(i, Binop(Id(i), Add,
Num_Lit(1)))){
    If(Binop(Id(str), Equal, String_Lit(str2))){
        Expr(Binop(Num_Lit(3), Add, Num_Lit(2)));
    }
    Else{
    }
}
```

In order to test it automatically, all of the test file were linked with makefile.

6.1.2.1. Scanner Test Cases

```
bash ./test_scanner.sh
Running scanner tests...
  - checking scanner/_arithmetic.in...                    SUCCESS
  - checking scanner/_boolean_operation.in...             SUCCESS
  - checking scanner/_bracket.in...                       SUCCESS
  - checking scanner/_comment.in...                       SUCCESS
  - checking scanner/_comparator.in...                    SUCCESS
  - checking scanner/_graph_operator.in...                SUCCESS
  - checking scanner/_integer_float.in...                 SUCCESS
  - checking scanner/_logic_opearation.in...              SUCCESS
  - checking scanner/_primary_type.in...                  SUCCESS
  - checking scanner/_quote.in...                         SUCCESS
  - checking scanner/_separator.in...                     SUCCESS
```

6.1.2.2. Parser Test Cases

```
bash ./test_parser.sh
Running Parser tests...
  - checking parser/_arithmetic.in...                    SUCCESS
  - checking parser/_conditional.in...                   SUCCESS
  - checking parser/_dict.in...                          SUCCESS
  - checking parser/_function.in...                      SUCCESS
  - checking parser/_graph.in...                         SUCCESS
  - checking parser/_list.in...                          SUCCESS
  - checking parser/_literals.in...                      SUCCESS
  - checking parser/_node.in...                          SUCCESS
  - checking parser/_relational.in...                    SUCCESS
  - checking parser/_type_dec.in...                      SUCCESS
```

6.1.2.3. Semantic Check Test Cases

```
bash ./test_semantic.sh
Running Semantic Check tests...
  - checking semantic_check/_access_outer_func_variable.in...     SUCCESS
  - checking semantic_check/_illegal_assignment.in...             SUCCESS
  - checking semantic_check/_illegal_binary_operation1.in...      SUCCESS
  - checking semantic_check/_illegal_binary_operation2.in...      SUCCESS
  - checking semantic_check/_illegal_binary_operation3.in...      SUCCESS
  - checking semantic_check/_illegal_binary_operation4.in...      SUCCESS
  - checking semantic_check/_illegal_binary_operation5.in...      SUCCESS
  - checking semantic_check/_illegal_unary_operation1.in...       SUCCESS
  - checking semantic_check/_illegal_unary_operation2.in...       SUCCESS
  - checking semantic_check/_incompatible_func_arg_type.in...     SUCCESS
  - checking semantic_check/_inconsistent_dict_element_type.in... SUCCESS
  - checking semantic_check/_inconsistent_list_element_type.in... SUCCESS
  - checking semantic_check/_invalid_dict_get1.in...              SUCCESS
  - checking semantic_check/_invalid_dict_get2.in...              SUCCESS
  - checking semantic_check/_invalid_dict_keys1.in...             SUCCESS
  - checking semantic_check/_invalid_dict_keys2.in...             SUCCESS
  - checking semantic_check/_invalid_dict_put1.in...              SUCCESS
  - checking semantic_check/_invalid_dict_put2.in...              SUCCESS
  - checking semantic_check/_invalid_dict_put3.in...              SUCCESS
  - checking semantic_check/_invalid_dict_remove1.in...           SUCCESS
  - checking semantic_check/_invalid_dict_remove2.in...           SUCCESS
  - checking semantic_check/_invalid_dict_size.in...              SUCCESS
  - checking semantic_check/_invalid_dict_type1.in...             SUCCESS
  - checking semantic_check/_invalid_empty_dict.in...             SUCCESS
  - checking semantic_check/_invalid_empty_list.in...             SUCCESS
  - checking semantic_check/_invalid_expr_after_return.in...      SUCCESS
  - checking semantic_check/_invalid_graph_edge_at.in...          SUCCESS
  - checking semantic_check/_invalid_graph_edges.in...            SUCCESS
  - checking semantic_check/_invalid_graph_link.in...             SUCCESS
  - checking semantic_check/_invalid_graph_list_node_at.in...     SUCCESS
  - checking semantic_check/_invalid_graph_nodes.in...            SUCCESS
  - checking semantic_check/_invalid_graph_root.in...             SUCCESS
  - checking semantic_check/_invalid_graph_root_as.in...          SUCCESS
  - checking semantic_check/_invalid_graph_size.in...             SUCCESS
```

```
 - checking semantic_check/_invalid_list_add1.in...            SUCCESS
 - checking semantic_check/_invalid_list_add2.in...            SUCCESS
 - checking semantic_check/_invalid_list_get1.in...            SUCCESS
 - checking semantic_check/_invalid_list_get2.in...            SUCCESS
 - checking semantic_check/_invalid_list_pop.in...             SUCCESS
 - checking semantic_check/_invalid_list_push1.in...           SUCCESS
 - checking semantic_check/_invalid_list_push2.in...           SUCCESS
 - checking semantic_check/_invalid_list_remove1.in...         SUCCESS
 - checking semantic_check/_invalid_list_remove2.in...         SUCCESS
 - checking semantic_check/_invalid_list_set1.in...            SUCCESS
 - checking semantic_check/_invalid_list_set2.in...            SUCCESS
 - checking semantic_check/_invalid_list_set3.in...            SUCCESS
 - checking semantic_check/_invalid_list_size.in...            SUCCESS
 - checking semantic_check/_invalid_list_type1.in...           SUCCESS
 - checking semantic_check/_legal_binary_operation.in...       SUCCESS
 - checking semantic_check/_legal_unary_operation.in...        SUCCESS
 - checking semantic_check/_redefine_print_func.in...          SUCCESS
 - checking semantic_check/_support_default_funcs.in...        SUCCESS
 - checking semantic_check/_undeclared_variable.in...          SUCCESS
 - checking semantic_check/_unmatched_func_arg_len.in...       SUCCESS
 - checking semantic_check/_unsupport_graph_list_edge_at.in... SUCCESS
 - checking semantic_check/_valid_dict_operation.in...         SUCCESS
 - checking semantic_check/_valid_graph_operation.in...        SUCCESS
 - checking semantic_check/_valid_list_operation.in...         SUCCESS
```

6.1.2.4. Code Generator Test Cases

```
bash ./test_code_gen.sh
Running code_gen tests...
  - checking code_gen/_cast.in...                              SUCCESS
  - checking code_gen/_dict.in...                              SUCCESS
  - checking code_gen/_dict_node.in...                         SUCCESS
  - checking code_gen/_graph_direct_def.in...                  SUCCESS
  - checking code_gen/_graph_edge.in...                        SUCCESS
  - checking code_gen/_graph_merge.in...                       SUCCESS
  - checking code_gen/_graph_method.in...                      SUCCESS
  - checking code_gen/_graph_sub_graph.in...                   SUCCESS
  - checking code_gen/_graph_sub_node.in...                    SUCCESS
  - checking code_gen/_id_defalut_assign.in...                 SUCCESS
  - checking code_gen/_list.in...                              SUCCESS
  - checking code_gen/_list_automatic_conversion.in...         SUCCESS
  - checking code_gen/_node_var_type.in...                     SUCCESS
  - checking code_gen/_print_test.in...                        SUCCESS
  - checking code_gen/_test.in...                              SUCCESS
  - checking code_gen/example_bfs.in...                        SUCCESS
  - checking code_gen/example_dfs.in...                        SUCCESS
  - checking code_gen/example_dijkstra.in...                   SUCCESS
  - checking code_gen/test_arith.in...                         SUCCESS
  - checking code_gen/test_if.in...                            SUCCESS
  - checking code_gen/test_inner_var_access.in...              SUCCESS
  - checking code_gen/test_while.in...                         SUCCESS
```

This project is using Travis CI to lively test all code in every commit to the github.

7. Conclusion

Circline is almost complete now. The project started from September, a concept thinking of building a graph, to a comprehensive language that support much more. Step by steps, a reasonable Language reference Manual was created. Referenced by the manual, scanner and parser were built. Although several difficulties were tackled since the LLVM appeared not friendly to Circline's syntax, the team found the solution to converted into C style. Makefile played a significant role in the build of the program. It reduce the total amount of time to compile and test all of the test files. Finally, with several example implemented, the Circline programming language was proved to be useful.

Special thanks to Alexandra Francine Medway, the TA of Circline team, for her patience and continuous support on the Circline Language design and implementation.

8. Lessons Learned

The importance of the structure of the program file and automation test: a good structure of the program files improve our efficiency a lot, thus we can have a strong "make" command to compile all the source files and run the automatic test easily. This makes our life much easier. Thanks to the tester, a good test plan were created in the very early of the semester. This provided the team an ease to debug, and convenient for members to coordinate with each other. TravisCI helps the team a lot at the beginning. Unfortunately, when LLVM were applied to use, the team experienced the hardship on it, caused by the virtual machine provided by TravisCi not support LLVM. Team's TA suggested that every test document should be prepared before every commit so that there would be no concern for TravisCI to compile the source code and it solved this problem.

Use C or any other language as library. Initially, the team did not know that llvm support C library addition. Once this method was found, team rapidly implemented the essential data structure. It was much more convenient to code in C and build corresponding APIs.

Ocaml is a hard language to hands on quickly, however improves the overall speed of developing circline: When we first learn Ocaml, we can hardly write down even ten lines of code, because it's not easy to use as Java. But when we are getting familiar with it, we find it enable correctness guarantees that are impossible in imperative languages. Because once you fix all the compile errors, you can almost conduct no run time error. This save us a tons of time to debug our code.

Variable type is not as easy as what we thought. Variable type causes us a lot of troubles when we were writing the code code generation, it's much harder than what we thought it would be. Our language could support a lot of types when we designed it, and it didn't bother us before the code generation. But when we were writing the code generation, we found that type support in LLVM was not very strong, and we had to spend a lot of time dealing with types. So if we could start again, we may choose to encapsulate all types in the C struct, or allow less types in our language so that we can have more time focusing on other interesting part of our language.

9. Appendix

For complete Code with test cases. Please find it in the code files

9.1. Scanner.mll

```
{
  open Parser
  let unescape s =
       Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
}
let digit = ['0'-'9']
let letter = ['a'-'z' 'A'-'Z']
let variable = letter (letter | digit | '_') *
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
rule token =
parse [' ' '\t' '\r' '\n'] { token lexbuf }
(* comment *)
| "/*" { comment lexbuf }
(* calculation *)
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '%' { MOD }
(* separator *)
| ';' { SEMICOLUMN }
| ',' { SEQUENCE }
| '=' { ASSIGN }
| ':' { COLUMN }
| '.' { DOT }
(* logical operation *)
| "and" { AND }
| "or" { OR }
| "not" { NOT }
| "if" { IF }
| "else" { ELSE }
| "for" { FOR }
| "while" { WHILE}
| "break" { BREAK }
| "continue" { CONTINUE }
| "in" { IN }
| "return" {RETURN}
(* comparator *)
| '>' { GREATER }
| ">=" { GREATEREQUAL }
| '<' { SMALLER }
| "<=" { SMALLEREQUAL }
| "==" { EQUAL}
| "!=" { NOTEQUAL}
(* graph operator *)
| "--" { LINK }
| "->" { RIGHTLINK }
| "<-" { LEFTLINK }
| '@' { AT }
| '&' { AMPERSAND }
```

```
| '~' { SIMILARITY }
(* primary type *)
| "void" { VOID }
| "int" { INT }
| "float" { FLOAT }
| "string" { STRING }
| "bool" { BOOL }
| "node" { NODE }
| "graph" { GRAPH }
| "list" { LIST }
| "dict" { DICT }
| "null" { NULL }
(* integer and float *)
| digit+ as lit { INT_LITERAL(int_of_string lit) }
| digit+'.'digit* as lit { FLOAT_LITERAL(float_of_string lit) }
| '"' ((ascii | escape)* as lit) '"' { STRING_LITERAL(unescape lit) }
(* quote *)
| '"'  { QUOTE }
(* boolean operation *)
| "true" | "false" as boollit { BOOL_LITERAL(bool_of_string boollit)}
(* bracket *)
| '[' { LEFTBRACKET }
| ']' { RIGHTBRACKET }
| '{' { LEFTCURLYBRACKET }
| '}' { RIGHTCURLYBRACKET }
| '(' { LEFTROUNDBRACKET }
| ')' { RIGHTROUNDBRACKET }
(* id *)
| variable as id { ID(id) }
| eof { EOF }

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

9.2. Parser.mli

```
type token =
  | PLUS
  | MINUS
  | TIMES
  | DIVIDE
  | MOD
  | SEMICOLUMN
  | SEQUENCE
  | ASSIGN
  | COLUMN
  | DOT
  | GREATER
  | GREATEREQUAL
  | SMALLER
  | SMALLEREQUAL
  | EQUAL
  | NOTEQUAL
  | AND
  | OR
  | NOT
```

```
  | IF
  | ELSE
  | FOR
  | WHILE
  | BREAK
  | CONTINUE
  | IN
  | RETURN
  | LINK
  | RIGHTLINK
  | LEFTLINK
  | SIMILARITY
  | AT
  | AMPERSAND
  | INT
  | FLOAT
  | STRING
  | BOOL
  | NODE
  | GRAPH
  | LIST
  | DICT
  | NULL
  | VOID
  | QUOTE
  | LEFTBRACKET
  | RIGHTBRACKET
  | LEFTCURLYBRACKET
  | RIGHTCURLYBRACKET
  | LEFTROUNDBRACKET
  | RIGHTROUNDBRACKET
  | EOF
  | ID of (string)
  | INT_LITERAL of (int)
  | STRING_LITERAL of (string)
  | FLOAT_LITERAL of (float)
  | BOOL_LITERAL of (bool)

val program :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf ->  Ast.program
```

9.3. Parser.mly

```
%{ open Ast %}

/* Arithmetic Operators */
%token PLUS MINUS TIMES DIVIDE MOD

/* Separator */
%token SEMICOLUMN SEQUENCE ASSIGN COLUMN DOT

/* Relational Operators */
%token GREATER GREATEREQUAL SMALLER SMALLEREQUAL EQUAL NOTEQUAL

/* Logical Operators & Keywords*/
%token AND OR NOT IF ELSE FOR WHILE BREAK CONTINUE IN RETURN

/* Graph operator */
```

```
%token LINK RIGHTLINK LEFTLINK SIMILARITY AT AMPERSAND

/* Primary Type */
%token INT FLOAT STRING BOOL NODE GRAPH LIST DICT NULL VOID

/* Quote */
%token QUOTE

/* Bracket */
%token LEFTBRACKET RIGHTBRACKET LEFTCURLYBRACKET RIGHTCURLYBRACKET LEFTROUNDBRACKET
RIGHTROUNDBRACKET
/* EOF */
%token EOF

/* Identifiers */
%token <string> ID

/* Literals */
%token <int> INT_LITERAL
%token <string> STRING_LITERAL
%token <float> FLOAT_LITERAL
%token <bool> BOOL_LITERAL

/* Order */
%right ASSIGN
%left AND OR
%left EQUAL NOTEQUAL
%left GREATER SMALLER GREATEREQUAL SMALLEREQUAL
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT
%right LINK RIGHTLINK LEFTLINK AMPERSAND
%left SIMILARITY AT
%right LEFTROUNDBRACKET
%left  RIGHTROUNDBRACKET
%right COLUMN
%right DOT

%start program
%type < Ast.program> program

%%

/* Program flow */
program:
| stmt_list EOF                         { List.rev $1 }

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

stmt:
| expr SEMICOLUMN                       { Expr($1) }
| func_decl                             { Func($1) }
| RETURN SEMICOLUMN                { Return(Noexpr) }
| RETURN expr SEMICOLUMN                { Return($2) }
| FOR LEFTROUNDBRACKET for_expr SEMICOLUMN expr SEMICOLUMN for_expr RIGHTROUNDBRACKET
LEFTCURLYBRACKET stmt_list RIGHTCURLYBRACKET
  {For($3, $5, $7, List.rev $10)}
| IF LEFTROUNDBRACKET expr RIGHTROUNDBRACKET LEFTCURLYBRACKET stmt_list RIGHTCURLYBRACKET
ELSE LEFTCURLYBRACKET stmt_list RIGHTCURLYBRACKET
```

```
  {If($3,List.rev $6,List.rev $10)}
| IF LEFTROUNDBRACKET expr RIGHTROUNDBRACKET LEFTCURLYBRACKET stmt_list RIGHTCURLYBRACKET
  {If($3,List.rev $6,[])}
| WHILE LEFTROUNDBRACKET expr RIGHTROUNDBRACKET LEFTCURLYBRACKET stmt_list RIGHTCURLYBRACKET
  {While($3, List.rev $6)}
| var_decl SEMICOLUMN                   { Var_dec($1)}

var_decl:
| var_type ID             { Local($1, $2, Noexpr) }
| var_type ID ASSIGN expr  { Local($1, $2, $4) }

var_type:
| VOID                  {Void_t}
| NULL                  {Null_t}
| INT                                                         {Int_t}
| FLOAT                                                            {Float_t}
| STRING                                                           {String_t}
| BOOL {Bool_t}
| NODE {Node_t}
| GRAPH {Graph_t}
| DICT SMALLER INT GREATER {Dict_Int_t}
| DICT SMALLER FLOAT GREATER {Dict_Float_t}
| DICT SMALLER STRING GREATER {Dict_String_t}
| DICT SMALLER NODE GREATER {Dict_Node_t}
| DICT SMALLER GRAPH GREATER {Dict_Graph_t}
| LIST SMALLER INT GREATER {List_Int_t}
| LIST SMALLER FLOAT GREATER {List_Float_t}
| LIST SMALLER STRING GREATER {List_String_t}
| LIST SMALLER BOOL GREATER {List_Bool_t}
| LIST SMALLER NODE GREATER {List_Node_t}
| LIST SMALLER GRAPH GREATER {List_Graph_t}

formal_list:
| /* nothing */               { [] }
| formal                      { [$1] }
| formal_list SEQUENCE formal { $3 :: $1 }


formal:
| var_type ID           { Formal($1, $2) }

func_decl:
| var_type ID LEFTROUNDBRACKET formal_list RIGHTROUNDBRACKET LEFTCURLYBRACKET stmt_list
RIGHTCURLYBRACKET {
  {
    returnType = $1;
    name = $2;
    args = List.rev $4;
    body = List.rev $7;
  }
}


/* For loop decl*/
for_expr:
| /* nothing */                         { Noexpr }
| expr                                  { $1 }

expr:
  literals {$1}
| NULL                            { Null }
```

```
| arith_ops                        { $1 }
| graph_ops                        { $1 }
| NODE LEFTROUNDBRACKET expr RIGHTROUNDBRACKET { Node($3) }
| ID                                                     { Id($1) }
| ID ASSIGN expr                                           { Assign($1, $3) }
| expr AT LEFTROUNDBRACKET expr SEQUENCE expr RIGHTROUNDBRACKET { EdgeAt($1, $4, $6) }
| LEFTBRACKET list RIGHTBRACKET                    { ListP(List.rev $2) }
| LEFTCURLYBRACKET dict RIGHTCURLYBRACKET   { DictP(List.rev $2) }
| LEFTROUNDBRACKET expr RIGHTROUNDBRACKET   { $2 }
| ID LEFTROUNDBRACKET list RIGHTROUNDBRACKET              { Call($1, List.rev $3) }
| INT LEFTROUNDBRACKET list RIGHTROUNDBRACKET              { Call("int", List.rev $3) }
| FLOAT LEFTROUNDBRACKET list RIGHTROUNDBRACKET              { Call("float", List.rev $3) }
| BOOL LEFTROUNDBRACKET list RIGHTROUNDBRACKET              { Call("bool", List.rev $3) }
| STRING LEFTROUNDBRACKET list RIGHTROUNDBRACKET              { Call("string", List.rev
$3) }
| expr DOT ID LEFTROUNDBRACKET list RIGHTROUNDBRACKET   {CallDefault($1, $3, List.rev $5)}

/* Lists */
list:
| /* nothing */                      { [] }
| expr                               { [$1] }
| list SEQUENCE expr                 { $3 :: $1 }

list_graph:
| expr AMPERSAND expr         { { graphs = [$3]; edges = [$1] } }
| list_graph SEQUENCE expr AMPERSAND expr
    { { graphs = $5 :: ($1).graphs; edges = $3 :: ($1).edges } }

list_graph_literal:
| LEFTBRACKET list_graph RIGHTBRACKET   {
  { graphs = List.rev ($2).graphs; edges = List.rev ($2).edges }
}

dict_key_value:
| expr COLUMN expr { ($1, $3) }

/* dict */
dict:
| /* nothing */                  { [] }
| dict_key_value                                                        { [$1] }
| dict SEQUENCE dict_key_value                    {$3 :: $1}

arith_ops:
| expr PLUS        expr                                  { Binop($1, Add,   $3) }
| expr MINUS       expr                                  { Binop($1, Sub,   $3) }
| expr TIMES       expr                                  { Binop($1, Mult,  $3) }
| expr DIVIDE      expr                                  { Binop($1, Div,   $3) }
| expr EQUAL       expr                                  { Binop($1, Equal, $3) }
| expr NOTEQUAL    expr                                  { Binop($1, Neq,   $3) }
| expr SMALLER     expr                                  { Binop($1, Less,  $3) }
| expr SMALLEREQUAL expr                                 { Binop($1, Leq,   $3) }
| expr GREATER     expr                                  { Binop($1, Greater,  $3) }
| expr GREATEREQUAL expr                                 { Binop($1, Geq,   $3) }
| expr AND         expr                                  { Binop($1, And,   $3) }
| expr MOD         expr                                  { Binop($1, Mod,   $3) }
| expr OR     expr                 { Binop($1, Or,    $3) }
| NOT  expr                                                      { Unop (Not,   $2) }
| MINUS expr                                                     { Unop (Neg, $2) }
| expr SIMILARITY expr             { Binop($1, RootAs, $3) }
| expr AT AT expr                  { Binop($1, ListEdgesAt, $4) }
| expr AT expr                     { Binop($1, ListNodesAt, $3) }
```

```
graph_ops:
| expr LINK expr                       { Graph_Link($1, Double_Link, $3, Null) }
| expr LINK list_graph_literal         { Graph_Link($1, Double_Link, ListP(($3).graphs),
ListP(($3).edges)) }
| expr LINK expr AMPERSAND expr        { Graph_Link($1, Double_Link, $5, $3) }
| expr RIGHTLINK expr                  { Graph_Link($1, Right_Link, $3, Null) }
| expr RIGHTLINK list_graph_literal    { Graph_Link($1, Right_Link, ListP(($3).graphs),
ListP(($3).edges)) }
| expr RIGHTLINK expr AMPERSAND expr   { Graph_Link($1, Right_Link, $5, $3) }
| expr LEFTLINK expr                   { Graph_Link($1, Left_Link, $3, Null) }
| expr LEFTLINK list_graph_literal     { Graph_Link($1, Left_Link, ListP(($3).graphs),
ListP(($3).edges)) }
| expr LEFTLINK expr AMPERSAND expr    { Graph_Link($1, Left_Link, $5, $3) }

literals:
  INT_LITERAL            {Num_Lit( Num_Int($1) )}
| FLOAT_LITERAL          {Num_Lit( Num_Float($1) )}
| STRING_LITERAL     {String_Lit($1) }
| BOOL_LITERAL       {Bool_lit($1) }
```

9.4. Ast.ml

```
(* Binary Operators *)
type binop =
  Add         (* + *)
| Sub         (* - *)
| Mult        (* * *)
| Div         (* / *)
| Mod         (* % *)
| Equal       (* == *)
| Neq         (* != *)
| Less        (* < *)
| Leq         (* <= *)
| Greater     (* > *)
| Geq         (* >= *)
| And         (* and *)
| Or          (* or *)
(* Graph Only *)
| ListNodesAt        (* <graph> @  <node> *)
| ListEdgesAt        (* <graph> @@ <node> *)
| RootAs             (* <graph> ~  <node> *)

(* Unary Operators *)
type unop =
  Neg         (* - *)
| Not         (* not *)

(* Numbers int | float *)
type num =
  Num_Int of int          (* 514 *)
| Num_Float of float      (* 3.1415 *)

(* Variable Type *)
type var_type =
  Int_t                   (* int *)
| Float_t                 (* float *)
```

```
| String_t                (* string *)
| Bool_t
| Node_t
| Graph_t
| Dict_Int_t
| Dict_Float_t
| Dict_String_t
| Dict_Node_t
| Dict_Graph_t
| List_Int_t
| List_Float_t
| List_String_t
| List_Bool_t
| List_Node_t
| List_Graph_t
| Void_t
| Null_t

(* Type Declaration *)
type formal =
| Formal of var_type * string   (* int aNum *)

type graph_op =
| Right_Link
| Left_Link
| Double_Link

type expr =
    Num_Lit of num
|   Null
|   String_Lit of string
|   Bool_lit of bool
|   Node of expr
|       Graph_Link of expr * graph_op * expr * expr
|   EdgeAt of expr * expr * expr
|       Binop of expr * binop * expr
|       Unop of unop * expr
|   Id of string
|   Assign of string * expr
|   Noexpr
|   ListP of expr list
|   DictP of (expr * expr) list
|   Call of string * expr list    (* function call *)
|   CallDefault of expr * string * expr list

and edge_graph_list = {
  graphs: expr list;
  edges: expr list;
}

type var_decl =
| Local of var_type * string * expr

(* Statements *)
type stmt =
  Expr of expr     (* set foo = bar + 3 *)
| Return of expr
| For of expr * expr * expr * stmt list
| If of expr * stmt list * stmt list
| While of expr * stmt list
| Var_dec of var_decl
```

```
| Func of func_decl

(* Function Declaration *)
and func_decl = {
  returnType: var_type;
  name: string;
  args: formal list;
  body: stmt list;
}


(* Program entry point *)
type program = stmt list
```

9.5. Cast.ml

```
(* Binary Operators *)
type binop =
  Add         (* + *)
| Sub         (* - *)
| Mult        (* * *)
| Div         (* / *)
| Mod         (* % *)
| Equal       (* == *)
| Neq         (* != *)
| Less        (* < *)
| Leq         (* <= *)
| Greater     (* > *)
| Geq         (* >= *)
| And         (* and *)
| Or          (* or *)
(* Graph Only *)
| ListNodesAt       (* <graph> @  <node> *)
| ListEdgesAt       (* <graph> @@ <node> *)
| RootAs            (* <graph> ~  <node> *)

(* Unary Operators *)
type unop =
  Neg         (* - *)
| Not         (* not *)

(* Numbers int | float *)
type num =
  Num_Int of int          (* 514 *)
| Num_Float of float      (* 3.1415 *)

(* Variable Type *)
type var_type =
  Int_t                   (* int *)
| Float_t                 (* float *)
| String_t                (* string *)
| Bool_t
| Node_t
| Edge_t
| Graph_t
| Dict_Int_t
| Dict_Float_t
```

```
| Dict_String_t
| Dict_Node_t
| Dict_Graph_t
| List_Int_t
| List_Float_t
| List_Bool_t
| List_String_t
| List_Node_t
| List_Graph_t
| List_Null_t
| Void_t
| Null_t

(* Type Declaration *)
type formal =
| Formal of var_type * string   (* int aNum *)

type graph_op =
| Right_Link
| Left_Link
| Double_Link

type expr =
    Num_Lit of num
|   Null
|   String_Lit of string
|   Bool_lit of bool
|   Node of int * expr
|   Graph_Link of expr * graph_op * expr * expr
|   EdgeAt of expr * expr * expr
|   Binop of expr * binop * expr
|   Unop of unop * expr
|   Id of string
|   Assign of string * expr
|   Noexpr
|   ListP of expr list
|   DictP of (expr * expr) list
|   Call of string * expr list    (* function call *)
|   CallDefault of expr * string * expr list

and edge_graph_list = {
  graphs: expr list;
  edges: expr list;
}

type var_decl =
| Local of var_type * string * expr

(* Statements *)
type stmt =
  Expr of expr     (* set foo = bar + 3 *)
| Return of expr
| For of expr * expr * expr * stmt list
| If of expr * stmt list * stmt list
| While of expr * stmt list

(* Function Declaration *)
and func_decl = {
  returnType: var_type;
  name: string;
  args: formal list;
```

```
  body: stmt list;
  locals: formal list;
  pname: string; (* parent func name *)
}


(* Program entry point *)
type program = func_decl list
```

9.6. Organizer.ml

```
module A = Ast
module C = Cast

module StringMap = Map.Make(String)
let node_num = ref 0

let convert_binop = function
    A.Add -> C.Add
  | A.Sub -> C.Sub
  | A.Mult -> C.Mult
  | A.Div -> C.Div
  | A.Mod -> C.Mod
  | A.Equal -> C.Equal
  | A.Neq -> C.Neq
  | A.Less -> C.Less
  | A.Leq -> C.Leq
  | A.Greater -> C.Greater
  | A.Geq -> C.Geq
  | A.And -> C.And
  | A.Or -> C.Or
  | A.ListNodesAt -> C.ListNodesAt
  | A.ListEdgesAt -> C.ListEdgesAt
  | A.RootAs -> C.RootAs

let convert_unop = function
  A.Neg -> C.Neg
| A.Not -> C.Not

let convert_num = function
    A.Num_Int(a) -> C.Num_Int(a)
  | A.Num_Float(a) -> C.Num_Float(a)

let convert_var_type = function
    A.Int_t -> C.Int_t
  | A.Float_t -> C.Float_t
  | A.String_t -> C.String_t
  | A.Bool_t -> C.Bool_t
  | A.Node_t -> C.Node_t
  | A.Graph_t -> C.Graph_t
  | A.List_Int_t -> C.List_Int_t
  | A.List_Float_t -> C.List_Float_t
  | A.List_String_t -> C.List_String_t
  | A.List_Node_t -> C.List_Node_t
  | A.List_Graph_t -> C.List_Graph_t
  | A.List_Bool_t -> C.List_Bool_t
```

```
  | A.Dict_Int_t -> C.Dict_Int_t
  | A.Dict_Float_t -> C.Dict_Float_t
  | A.Dict_String_t -> C.Dict_String_t
  | A.Dict_Node_t -> C.Dict_Node_t
  | A.Dict_Graph_t -> C.Dict_Graph_t
  | A.Void_t -> C.Void_t
  | A.Null_t -> C.Null_t

let convert_graph_op = function
| A.Right_Link -> C.Right_Link
| A.Left_Link -> C.Left_Link
| A.Double_Link -> C.Double_Link

let rec get_entire_name m aux cur_name =
  if (StringMap.mem cur_name m) then
    let aux = (StringMap.find cur_name m) ^ "." ^ aux in
    (get_entire_name m aux (StringMap.find cur_name m))
  else aux

let increase_node_num =
  let node_num = ref(!node_num) in
  !(node_num) - 1

let rec convert_expr m = function
    A.Num_Lit(a) -> C.Num_Lit(convert_num a)
|   A.Null -> C.Null
|   A.String_Lit(a) -> C.String_Lit(a)
|   A.Bool_lit(a) -> C.Bool_lit(a)
|   A.Node(a) -> node_num := (!node_num + 1); C.Node(!node_num - 1, convert_expr m a)
|   A.Graph_Link(a,b,c,d) -> C.Graph_Link(
      convert_expr m a,
      convert_graph_op b,
      convert_expr m c,
      (match (c,d) with
        | (A.ListP(_), A.ListP(_))
        | (A.ListP(_), A.Noexpr)
        | (A.ListP(_), A.Null) -> convert_expr m d
        | (A.ListP(_), _) -> C.ListP([convert_expr m d])
        | _ -> convert_expr m d
      ))
|   A.EdgeAt(a,b,c) -> C.EdgeAt(convert_expr m a, convert_expr m b, convert_expr m c)
|   A.Binop(a,b,c) -> C.Binop(convert_expr m a, convert_binop b, convert_expr m c)
|   A.Unop(a,b) -> C.Unop(convert_unop a, convert_expr m b)
|   A.Id(a) -> C.Id(a)
|   A.Assign(a,b) -> C.Assign(a, convert_expr m b)
|   A.Noexpr -> C.Noexpr
|   A.ListP(a) -> C.ListP(convert_expr_list m a)
|   A.DictP(a) -> C.DictP(convert_dict_list m a)
|   A.Call(a,b) -> C.Call(get_entire_name m a a, convert_expr_list m b)
|   A.CallDefault(a,b,c) -> C.CallDefault(convert_expr m a, b, convert_expr_list m c)

and convert_expr_list m = function
    [] -> []
  | [x] -> [convert_expr m x]
  | _ as l -> (List.map (convert_expr m) l)

and convert_dict m = function
  (c,d) -> (convert_expr m c, convert_expr m d)

and convert_dict_list m = function
    [] -> []
```

```
  | [x] -> [convert_dict m x]
  | _ as l -> (List.map (convert_dict m) l)

let convert_edge_graph_list m = function
  {A.graphs = g; A.edges = e} -> {C.graphs = convert_expr_list m g; C.edges =
convert_expr_list m e}

let convert_formal = function
  | A.Formal(v, s) -> C.Formal(convert_var_type v, s)

let convert_formal_list = function
    [] -> []
  | [x] -> [convert_formal x]
  | _ as l -> (List.map convert_formal l)

(* create a main funcition outside of the whole statement list *)
let createMain stmts = A.Func({
    A.returnType = A.Int_t;
    A.name = "main";
    A.args = [];
    A.body = stmts;
  })

let rec get_funcs_from_body_a = function
    [] -> []
  | A.Func(_) as x::tl -> x :: (get_funcs_from_body_a tl)
  | _::tl -> get_funcs_from_body_a tl

let rec get_body_from_body_a = function
    [] -> []
  | A.Func(_)::tl -> get_body_from_body_a tl
  | _ as x::tl -> x :: (get_body_from_body_a tl)

let rec mapper parent map = function
   [] -> map
 | A.Func{A.name = n; _}::tl ->
    mapper parent (StringMap.add n parent map) tl
 | _-> map

let convert_bfs_insider my_map = function
    A.Func{A.name = n; A.body = b; _}->
    let curr = get_funcs_from_body_a b in
      let my_map = mapper n my_map curr in
    (curr,my_map)
  | _->([],my_map)

let rec bfser m result = function
    [] ->(List.rev result, m)
  | A.Func{A.returnType = r; A.name = n; A.args = args; A.body = b} as a ::tl -> let result1
= convert_bfs_insider m a in
    let latterlist = tl @ (fst result1) in
    let m = (snd result1) in
    let addedFunc = A.Func({
      A.returnType = r; A.name = n; A.args = args; A.body = get_body_from_body_a b
    }) in
    let result = result @ [addedFunc] in
     bfser m result latterlist
  | _->([], m)

(* convert stament in A to C, except those Var_dec and Func, we will convert them separately
*)
```

```
let rec convert_stmt m = function
    A.Expr(a) -> C.Expr(convert_expr m a)
  | A.Return(a) -> C.Return(convert_expr m a)
  | A.For(e1, e2, e3, stls) -> C.For(convert_expr m e1, convert_expr m e2, convert_expr m
e3, List.map (convert_stmt m) stls)
  | A.If(e, stls1, stls2) -> C.If(convert_expr m e, List.map (convert_stmt m) stls1,
List.map (convert_stmt m) stls2)
  | A.While(e, stls) -> C.While(convert_expr m e, List.map (convert_stmt m) stls)
  | _ -> C.Expr(C.Noexpr)


let rec get_body_from_body_c m = function
    [] -> []
  | A.Var_dec(A.Local(_, name, v))::tl when v <> A.Noexpr -> C.Expr(C.Assign(name,
convert_expr m v)) :: (get_body_from_body_c m tl)
  | A.Var_dec(A.Local(_, _, v))::tl when v = A.Noexpr -> (get_body_from_body_c m tl)
  | _ as x::tl -> (convert_stmt m x) :: (get_body_from_body_c m tl)

let rec get_local_from_body_c = function
    [] -> []
  | A.Var_dec(A.Local(typ, name, _))::tl -> C.Formal(convert_var_type typ, name) ::
(get_local_from_body_c tl)
  | _::tl -> get_local_from_body_c tl

(* convert the horizental level function list in A to C *)
let rec convert_func_list_c m = function
    [] -> []
  | A.Func{A.returnType = r; A.name = n; A.args = a; A.body = b} :: tl -> {
    C.returnType = convert_var_type r;
    C.name = get_entire_name m n n;
    C.args = convert_formal_list a;
    C.body = get_body_from_body_c m b;
    C.locals = get_local_from_body_c b;
    C.pname = if n = "main" then "main" else get_entire_name m (StringMap.find n m)
(StringMap.find n m)
  } :: (convert_func_list_c m tl)
  | _::tl -> convert_func_list_c m tl

(* entry point *)
let convert stmts =
  let funcs = createMain stmts in
  let horizen_funcs_m = bfser StringMap.empty [] [funcs] in
  convert_func_list_c (snd horizen_funcs_m) (fst horizen_funcs_m)
```

9.7. Semant.ml

```
open Cast
open Printf


module StringMap = Map.Make(String)

(* Pretty-printing functions *)
let string_of_typ = function
    Int_t -> "int"
  | Float_t -> "float"
  | String_t -> "string"
```

```
  | Bool_t -> "bool"
  | Node_t -> "node"
  | Graph_t -> "graph"
  | List_Int_t -> "list<int>"
  | List_Float_t -> "list<float>"
  | List_String_t -> "list<string>"
  | List_Node_t -> "list<node>"
  | List_Graph_t -> "list<graph>"
  | List_Bool_t -> "list<bool>"
  | List_Null_t -> "list<null>"
  | Dict_Int_t -> "dict<int>"
  | Dict_Float_t -> "dict<float>"
  | Dict_String_t -> "dict<string>"
  | Dict_Node_t -> "dict<node>"
  | Dict_Graph_t -> "dict<graph>"
  | Void_t -> "void"
  | Null_t -> "null"
  | Edge_t -> "edge"

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "and"
  | Or -> "or"
  | ListNodesAt -> "@"
  | ListEdgesAt -> "@@"
  | RootAs -> "~"

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

let string_of_graph_op = function
    Right_Link -> "->"
  | Left_Link -> "<-"
  | Double_Link -> "--"

let rec string_of_expr = function
    Num_Lit(Num_Int(l)) -> string_of_int l
  | Num_Lit(Num_Float(l)) -> string_of_float l
  | Null -> "null"
  | String_Lit(l) -> l
  | Bool_lit(true) -> "true"
  | Bool_lit(false) -> "false"
  | Node(_, e) -> "node(" ^ string_of_expr e ^ ")"
  | EdgeAt(e, n1, n2) -> string_of_expr e ^ "@" ^ "(" ^ string_of_expr n1 ^ "," ^
string_of_expr n2 ^ ")"
  | Graph_Link(e1, op, e2, e3) ->
      "graph_link(" ^ string_of_expr e1 ^ " " ^ string_of_graph_op op ^ " " ^ string_of_expr
e2 ^ " " ^ string_of_expr e3 ^ ")"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
```

```
  | Unop(o, e) -> string_of_uop o ^ " " ^ string_of_expr e
  | Id(s) -> s
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Noexpr -> ""
  (* TODO: maybe revise to a more meaningful name *)
  | ListP(_) -> "list"
  | DictP(_) -> "dict"
  | Call(n, _) -> "function call " ^ n
  | CallDefault(e, n, _) -> "function call " ^ string_of_expr e ^ "." ^ n


exception SemanticError of string

(* error message functions *)
let undeclared_function_error name =
    let msg = sprintf "undeclared function %s" name in
    raise (SemanticError msg)

let duplicate_formal_decl_error func name =
    let msg = sprintf "duplicate formal %s in %s" name func.name in
    raise (SemanticError msg)

let duplicate_local_decl_error func name =
    let msg = sprintf "duplicate local %s in %s" name func.name in
    raise (SemanticError msg)

let undeclared_identifier_error name =
    let msg = sprintf "undeclared identifier %s" name in
    raise (SemanticError msg)

let illegal_assignment_error lvaluet rvaluet ex =
    let msg = sprintf "illegal assignment %s = %s in %s" lvaluet rvaluet ex in
    raise (SemanticError msg)

let illegal_binary_operation_error typ1 typ2 op ex =
    let msg = sprintf "illegal binary operator %s %s %s in %s" typ1 op typ2 ex in
    raise (SemanticError msg)

let illegal_unary_operation_error typ op ex =
    let msg = sprintf "illegal unary operator %s %s in %s" op typ ex in
    raise (SemanticError msg)

let invaid_list_type_error typ =
    let msg = sprintf "invalid list type: %s" typ in
    raise (SemanticError msg)

let invaid_dict_type_error typ =
    let msg = sprintf "invalid dict type: %s" typ in
    raise (SemanticError msg)

let inconsistent_list_element_type_error typ1 typ2 =
    let msg = sprintf "list can not contain objects of different types: %s and %s" typ1 typ2
in
    raise (SemanticError msg)

let inconsistent_dict_element_type_error typ1 typ2 =
    let msg = sprintf "dict can not contain objects of different types: %s and %s" typ1 typ2
in
    raise (SemanticError msg)

let unmatched_func_arg_len_error name =
```

```
    let msg = sprintf "args length not match in function call: %s" name in
    raise (SemanticError msg)

let incompatible_func_arg_type_error typ1 typ2 =
    let msg = sprintf "incompatible argument type %s, but %s is expected" typ1 typ2 in
    raise (SemanticError msg)

let invalid_expr_after_return_error _ =
    let msg = sprintf "nothing may follow a return" in
    raise (SemanticError msg)

let redefine_print_func_error _ =
    let msg = sprintf "function print may not be defined" in
    raise (SemanticError msg)

let duplicate_func_error name =
    let msg = sprintf "duplicate function declaration: %s" name in
    raise (SemanticError msg)

let unsupport_operation_error typ name =
    let msg = sprintf "unsupport operation on type %s: %s" typ name in
    raise (SemanticError msg)

let invalid_list_size_method_error ex =
    let msg = sprintf "list size method do not take arguments: %s" ex in
    raise (SemanticError msg)

let invalid_list_pop_method_error ex =
    let msg = sprintf "list pop method do not take arguments: %s" ex in
    raise (SemanticError msg)

let invalid_list_get_method_error ex =
    let msg = sprintf "list get method should only take one argument of type int: %s" ex in
    raise (SemanticError msg)

let invalid_list_add_method_error typ ex =
    let msg = sprintf "list add method should only take one argument of type %s: %s" typ ex
in
    raise (SemanticError msg)

let invalid_list_push_method_error typ ex =
    let msg = sprintf "list push method should only take one argument of type %s: %s" typ ex
in
    raise (SemanticError msg)

let invalid_list_remove_method_error ex =
    let msg = sprintf "list remove method should only take one argument of type int: %s" ex
in
    raise (SemanticError msg)

let invalid_list_set_method_error typ ex =
    let msg = sprintf "list set method should only take two argument of type int and %s: %s"
typ ex in
    raise (SemanticError msg)

let invalid_empty_list_decl_error ex =
    let msg = sprintf "invalid empty list declaration: %s" ex in
    raise (SemanticError msg)

let invalid_dict_get_method_error ex =
```

```
    let msg = sprintf "dict get method should only take one argument of type int, string or
node: %s" ex in
    raise (SemanticError msg)

let invalid_dict_remove_method_error ex =
    let msg = sprintf "dict remove method should only take one argument of type int, string
or node: %s" ex in
    raise (SemanticError msg)

let invalid_dict_size_method_error ex =
    let msg = sprintf "dict size method do not take arguments: %s" ex in
      raise (SemanticError msg)

let invalid_dict_keys_method_error ex =
    let msg = sprintf "dict keys method do not take arguments: %s" ex in
      raise (SemanticError msg)

let invalid_dict_put_method_error typ ex =
    let msg = sprintf "dict put method should only take two argument of type (int, string or
node) and %s: %s" typ ex in
      raise (SemanticError msg)

let invalid_empty_dict_decl_error ex =
    let msg = sprintf "invalid empty dict declaration: %s" ex in
    raise (SemanticError msg)

let invalid_graph_root_method_error ex =
    let msg = sprintf "graph root method do not take arguments: %s" ex in
    raise (SemanticError msg)

let invalid_graph_size_method_error ex =
    let msg = sprintf "graph size method do not take arguments: %s" ex in
    raise (SemanticError msg)

let invalid_graph_nodes_method_error ex =
    let msg = sprintf "graph nodes method do not take arguments: %s" ex in
    raise (SemanticError msg)

let invalid_graph_edges_method_error ex =
    let msg = sprintf "graph edges method do not take arguments: %s" ex in
    raise (SemanticError msg)

let invalid_graph_link_error ex =
    let msg = sprintf "left side of graph link should be node type: %s" ex in
    raise (SemanticError msg)

let invalid_graph_edge_at_error ex =
    let msg = sprintf "invalid graph edge at: %s" ex in
    raise (SemanticError msg)

let invalid_graph_list_node_at_error ex =
    let msg = sprintf "invalid graph list node at: %s" ex in
    raise (SemanticError msg)

let unsupport_graph_list_edge_at_error ex =
    let msg = sprintf "unsupport graph list edge at: %s" ex in
    raise (SemanticError msg)

let invalid_graph_root_as_error ex =
    let msg = sprintf "invalid graph root as: %s" ex in
    raise (SemanticError msg)
```

```
let wrong_func_return_type_error typ1 typ2 =
    let msg = sprintf "wrong function return type: %s, expect %s" typ1 typ2 in
    raise (SemanticError msg)


let  match_list_type = function
  Int_t -> List_Int_t
| Float_t -> List_Float_t
| String_t -> List_String_t
| Node_t -> List_Node_t
| Graph_t -> List_Graph_t
| Bool_t -> List_Bool_t
| _ as t-> invaid_list_type_error (string_of_typ t)

let  reverse_match_list_type = function
  List_Int_t -> Int_t
| List_Float_t -> Float_t
| List_String_t -> String_t
| List_Node_t -> Node_t
| List_Graph_t -> Graph_t
| List_Bool_t -> Bool_t
| _ as t-> invaid_list_type_error (string_of_typ t)

let  match_dict_type = function
  Int_t -> Dict_Int_t
| Float_t -> Dict_Float_t
| String_t -> Dict_String_t
| Node_t -> Dict_Node_t
| Graph_t -> Dict_Graph_t
| _ as t-> invaid_dict_type_error (string_of_typ t)

let  reverse_match_dict_type = function
  Dict_Int_t -> Int_t
| Dict_Float_t ->  Float_t
| Dict_String_t -> String_t
| Dict_Node_t -> Node_t
| Dict_Graph_t -> Graph_t
| _ as t-> invaid_dict_type_error (string_of_typ t)


(* list check helper function  *)
let check_valid_list_type typ =
    if typ = List_Int_t || typ = List_Float_t || typ = List_String_t || typ = List_Node_t ||
typ = List_Graph_t || typ = List_Bool_t then typ
    else invaid_list_type_error (string_of_typ typ)

let check_list_size_method ex es =
    match es with
    [] -> ()
    | _ -> invalid_list_size_method_error (string_of_expr ex)

let check_list_pop_method ex es =
    match es with
    [] -> ()
    | _ -> invalid_list_pop_method_error (string_of_expr ex)

(* dict check helper function  *)
let check_valid_dict_type typ =
    if typ = Dict_Int_t || typ = Dict_Float_t || typ = Dict_String_t || typ = Dict_Node_t ||
typ = Dict_Graph_t then typ
```

```
    else invaid_dict_type_error (string_of_typ typ)

let check_dict_size_method ex es =
    match es with
    [] -> ()
    | _ -> invalid_dict_size_method_error (string_of_expr ex)

let check_dict_keys_method ex es =
    match es with
    [] -> ()
    | _ -> invalid_dict_keys_method_error (string_of_expr ex)

(* graph check helper function  *)
let check_graph_root_method ex es =
    match es with
    [] -> ()
    | _ -> invalid_graph_root_method_error (string_of_expr ex)

let check_graph_size_method ex es =
    match es with
    [] -> ()
    | _ -> invalid_graph_size_method_error (string_of_expr ex)

let check_graph_nodes_method ex es =
    match es with
    [] -> ()
    | _ -> invalid_graph_nodes_method_error (string_of_expr ex)

let check_graph_edges_method ex es =
    match es with
    [] -> ()
    | _ -> invalid_graph_edges_method_error (string_of_expr ex)

let check_graph_list_node_at ex lt rt =
    if lt = Graph_t && rt = Node_t then () else
    invalid_graph_list_node_at_error (string_of_expr ex)

let check_graph_root_as ex lt rt =
    if lt = Graph_t && rt = Node_t then () else
    invalid_graph_root_as_error (string_of_expr ex)

let check_return_type func typ =
    let lvaluet = func.returnType and rvaluet = typ in
    match lvaluet with
        Float_t when rvaluet = Int_t -> ()
        | String_t when rvaluet = Null_t -> ()
        | Node_t when rvaluet = Null_t -> ()
        | Graph_t when rvaluet = Null_t -> ()
        | List_Int_t | List_String_t | List_Float_t | List_Node_t | List_Graph_t |
List_Bool_t when rvaluet = Null_t -> ()
        | Dict_Int_t | Dict_String_t | Dict_Float_t | Dict_Node_t | Dict_Graph_t when
rvaluet = Null_t -> ()
        (* for dict.keys() *)
        | List_Int_t | List_String_t | List_Node_t when rvaluet = List_Null_t -> ()
        | _ -> if lvaluet == rvaluet then () else
          wrong_func_return_type_error (string_of_typ rvaluet) (string_of_typ lvaluet)


(* get function obj from func_map, if not found, raise error *)
let get_func_obj name func_map =
    try StringMap.find name func_map
```

```
    with Not_found -> undeclared_function_error name


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

(* check function *)
let check_function func_map func =
    (* check duplicate formals *)
    let args = List.map (fun (Formal(_, n)) -> n) func.args in
    report_duplicate (duplicate_formal_decl_error func) args;

    (* check duplicate locals *)
    let locals = List.map (fun (Formal(_, n)) -> n) func.locals in
    report_duplicate (duplicate_local_decl_error func) locals;


    (* search locally, if not found, then recursively search parent environment *)
    let rec type_of_identifier func s =
        let symbols = List.fold_left (fun m (Formal(t, n)) -> StringMap.add n t m)
            StringMap.empty (func.args @ func.locals )
        in
        try StringMap.find s symbols
        with Not_found ->
            if func.name = "main" then undeclared_identifier_error s else
            (* recursively search parent environment *)
            type_of_identifier (StringMap.find func.pname func_map) s
    in
    (* Raise an exception of the given rvalue type cannot be assigned to
    he given lvalue type, noted that int could be assinged to float type variable *)
    let check_assign lvaluet rvaluet ex = match lvaluet with
          Float_t when rvaluet = Int_t -> lvaluet
        | String_t when rvaluet = Null_t -> lvaluet
        | Node_t when rvaluet = Null_t -> lvaluet
        | Graph_t when rvaluet = Null_t -> lvaluet
        | List_Int_t | List_String_t | List_Float_t | List_Node_t | List_Graph_t |
List_Bool_t when rvaluet = Null_t -> lvaluet
        | Dict_Int_t | Dict_String_t | Dict_Float_t | Dict_Node_t | Dict_Graph_t when
rvaluet = Null_t -> lvaluet
        | List_Int_t | List_String_t | List_Node_t when rvaluet = List_Null_t -> lvaluet
        | _ -> if lvaluet == rvaluet then lvaluet else
            illegal_assignment_error (string_of_typ lvaluet) (string_of_typ rvaluet)
(string_of_expr ex)
    in
    (* Return the type of an expression or throw an exception *)
    let rec expr = function
          Num_Lit(Num_Int _) -> Int_t
        | Num_Lit(Num_Float _) -> Float_t
        | Null -> Null_t
        | String_Lit _ -> String_t
        | Bool_lit _ -> Bool_t
        (* check node and graph *)
        | Node(_, _) -> Node_t
        | Graph_Link(e1, _, _, _) ->
            let check_graph_link e1 =
                let typ = expr e1 in
```

```
                match typ with
                Node_t -> ()
                |_ -> invalid_graph_link_error (string_of_expr e1)
            in
            ignore(check_graph_link e1); Graph_t
        | EdgeAt(e, n1, n2) ->
            let check_edge_at e n1 n2 =
                if (expr e) = Graph_t && (expr n1) = Node_t && (expr n2) = Node_t then ()
                else invalid_graph_edge_at_error (string_of_expr e)
            in
            ignore(check_edge_at e n1 n2); Edge_t
        | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
            (match op with
            (* +,-,*,/ *)
            Add | Sub | Mult | Div when t1 = Int_t && t2 = Int_t -> Int_t
            |Add | Sub | Mult | Div when t1 = Float_t && t2 = Float_t -> Float_t
            |Add | Sub | Mult | Div when t1 = Int_t && t2 = Float_t -> Float_t
            |Add | Sub | Mult | Div when t1 = Float_t && t2 = Int_t -> Float_t
            (* + - for graph *)
            | Add when t1 = Graph_t && t2 = Graph_t -> Graph_t
            | Sub when t1 = Graph_t && t2 = Graph_t -> List_Graph_t
            | Sub when t1 = Graph_t && t2 = Node_t -> List_Graph_t
            (* ==, != *)
            | Equal | Neq when t1 = t2 -> Bool_t
            (* <, <=, >, >= *)
            | Less | Leq | Greater | Geq when (t1 = Int_t || t1 = Float_t) && (t2 = Int_t ||
t2 = Float_t) -> Bool_t
            (* and, or *)
            | And | Or when t1 = Bool_t && t2 = Bool_t -> Bool_t
            (* mode *)
            | Mod when t1 = Int_t && t2 = Int_t -> Int_t
            | ListNodesAt -> ignore(check_graph_list_node_at e t1 t2); List_Node_t;
            | ListEdgesAt -> unsupport_graph_list_edge_at_error (string_of_expr e)
            | RootAs -> ignore(check_graph_root_as e t1 t2); Graph_t;
            | _ -> illegal_binary_operation_error (string_of_typ t1) (string_of_typ t2)
(string_of_op op) (string_of_expr e)
            )
        | Unop(op, e) as ex -> let t = expr e in
            (match op with
            Neg when t = Int_t -> Int_t
            |Neg when t = Float_t -> Float_t
            | Not when t = Bool_t -> Bool_t
            | _ -> illegal_unary_operation_error (string_of_typ t) (string_of_uop op)
(string_of_expr ex)
            )
        | Id s -> type_of_identifier func s
        | Assign(var, e) as ex -> let lt = type_of_identifier func var and rt = expr e in
            check_assign lt rt ex
        | Noexpr -> Void_t
        | ListP([]) as ex -> invalid_empty_list_decl_error (string_of_expr ex)
        | ListP(es) ->
            let element_type =
              let determine_element_type ss = List.fold_left
                (fun l e -> (match l with
                  [] -> [expr e]
                | t :: _ when t = (expr e) -> [t]
                | t :: _ when (t = Graph_t && (expr e) = Node_t) || (t = Node_t && (expr e)
= Graph_t) -> [Graph_t]
                | t :: _ when (t = Float_t && (expr e) = Int_t) || (t = Int_t && (expr e) =
Float_t) -> [Float_t]
```

```
                | t :: _ -> inconsistent_list_element_type_error (string_of_typ t)
(string_of_typ (expr e))
                )) [] ss
              in
              List.hd (determine_element_type es)
            in
            match_list_type element_type
        | DictP([]) as ex -> invalid_empty_dict_decl_error (string_of_expr ex)
        | DictP(es) ->
            let element_type =
              let determine_element_type ss = List.fold_left
                (fun l (_, e) -> (match l with
                  [] -> [expr e]
                | t :: _  when t = (expr e) -> [t]
                | t :: _ -> inconsistent_dict_element_type_error (string_of_typ t)
(string_of_typ (expr e))
                )) [] ss
              in
              List.hd (determine_element_type es)
            in
            match_dict_type element_type
        | Call(n, args) -> let func_obj = get_func_obj n func_map in
              (* check function call such as the args length, args type *)
              let check_funciton_call func args =
                  let check_args_length l_arg r_arg = if (List.length l_arg) = (List.length
r_arg)
                      then () else (unmatched_func_arg_len_error func.name)
                  in
                  if List.mem func.name ["printb"; "print"; "printf"; "string"; "float";
"int"; "bool"] then ()
                  else check_args_length func.args args;
                  (* l_arg is a list of Formal(typ, name), r_arg is a list of expr *)
                  let check_args_type l_arg r_arg =
                      List.iter2
                          (fun (Formal(t, _)) r -> let r_typ = expr r in if t = r_typ then
() else
                            incompatible_func_arg_type_error (string_of_typ r_typ)
(string_of_typ t)
                          )
                          l_arg r_arg
                  in
                  (* do not check args type of function print, do conversion in codegen *)
                  if List.mem func.name ["printb"; "print"; "printf"; "string"; "float";
"int"; "bool"] then ()
                  else check_args_type func.args args
              in
              ignore(check_funciton_call func_obj args); func_obj.returnType
              (* TODO: implement call default *)
        | CallDefault(e, n, es) -> let typ = expr e in
              (* should not put it here, but we need function expr, so we cann't put outside
*)
              let check_list_get_method ex es =
                  match es with
                  [x] when (expr x) = Int_t -> ()
                  | _ -> invalid_list_get_method_error (string_of_expr ex)
              in
              let check_list_add_method typ ex es =
                  match es with
                  [x] when (expr x) = (reverse_match_list_type typ) -> ()
                  | _ -> invalid_list_add_method_error (string_of_typ
(reverse_match_list_type typ)) (string_of_expr ex)
```

```
              in
              let check_list_push_method typ ex es =
                  match es with
                  [x] when (expr x) = (reverse_match_list_type typ) -> ()
                  | _ -> invalid_list_push_method_error (string_of_typ
(reverse_match_list_type typ)) (string_of_expr ex)
              in
              let check_list_remove_method ex es =
                  match es with
                  [x] when (expr x) = Int_t -> ()
                  | _ -> invalid_list_remove_method_error (string_of_expr ex)
              in
              let check_list_set_method typ ex es =
                  match es with
                  [index; value] when (expr index) = Int_t && (expr value) =
(reverse_match_list_type typ) -> ()
                  | _ -> invalid_list_set_method_error (string_of_typ
(reverse_match_list_type typ)) (string_of_expr ex)
              in
              let check_dict_get_method ex es =
                  match es with
                  [x] when List.mem (expr x) [Int_t; String_t; Node_t] -> ()
                  | _ -> invalid_dict_get_method_error (string_of_expr ex)
              in
              let check_dict_remove_method ex es =
                  match es with
                  [x] when List.mem (expr x) [Int_t; String_t; Node_t] -> ()
                  | _ -> invalid_dict_remove_method_error (string_of_expr ex)
              in
              let check_dict_put_method typ ex es =
                  match es with
                  [key; value] when List.mem (expr key) [Int_t; String_t; Node_t]
                                  && ((expr value) = (reverse_match_dict_type typ) ||
(expr value) = Null_t) -> ()
                  | _ -> invalid_dict_put_method_error (string_of_typ
(reverse_match_dict_type typ)) (string_of_expr ex)
              in
              match typ with
                  List_Int_t | List_Float_t | List_String_t | List_Node_t | List_Graph_t |
List_Bool_t ->
                    (match n with
                      "add" -> ignore(check_list_add_method typ e es); typ
                      | "push"  -> ignore(check_list_push_method typ e es); typ
                      | "remove" -> ignore(check_list_remove_method e es); typ
                      | "set" -> ignore(check_list_set_method typ e es); typ
                      (* | "concat" ->  *)
                      | "pop" -> ignore(check_list_pop_method e es); reverse_match_list_type
typ
                      | "get" -> ignore(check_list_get_method e es); reverse_match_list_type
typ
                      | "size" -> ignore(check_list_size_method e es); Int_t
                      | _ -> unsupport_operation_error (string_of_typ typ) n
                    )
                  | Dict_Int_t | Dict_Float_t | Dict_String_t | Dict_Node_t | Dict_Graph_t
->
                    (* key support type node, string, int *)
                    (match n with
                      "put" -> ignore(check_dict_put_method typ e es); typ
                      | "get" -> ignore(check_dict_get_method e es); reverse_match_dict_type
typ
                      | "remove" -> ignore(check_dict_remove_method e es); typ
```

```
                    | "size" -> ignore(check_dict_size_method e es); Int_t
                    (* return List_Null_t here to bypass the semantic check *)
                    | "keys" -> ignore(check_dict_keys_method e es); List_Null_t
                    | _ -> unsupport_operation_error (string_of_typ typ) n
                  )
                | Graph_t ->
                  (match n with
                    "root" -> ignore(check_graph_root_method e es); Node_t
                    | "size" -> ignore(check_graph_size_method e es); Int_t
                    | "nodes" -> ignore(check_graph_nodes_method e es); List_Node_t
                    | "edges" -> ignore(check_graph_edges_method e es); List_Int_t
                    | _ -> unsupport_operation_error (string_of_typ typ) n
                  )
                | _ -> unsupport_operation_error (string_of_typ typ) n
    in
    (* check statement *)
    let rec stmt = function
            Expr(e) -> ignore (expr e)
            | Return e -> ignore (check_return_type func (expr e))
            | For(e1, e2, e3, stls) ->
                ignore (expr e1); ignore (expr e2); ignore (expr e3); ignore(stmt_list stls)
            | If(e, stls1, stls2) -> ignore(e); ignore(stmt_list stls1); ignore(stmt_list
stls2)
            | While(e, stls) -> ignore(e); ignore(stmt_list stls)
    and
    (* check statement list *)
    stmt_list = function
            Return _ :: ss when ss <> [] -> invalid_expr_after_return_error ss
            | s::ss -> stmt s ; stmt_list ss
            | [] -> ()

    in
    stmt_list func.body

(* program here is a list of functions *)
let check program =
    let end_with s1 s2 =
        let len1 = String.length s1 and len2 = String.length s2 in
        if len1 < len2 then false
        else
            let last = String.sub s1 (len1-len2) len2 in
            if last = s2 then true else false
    in
    if List.mem true (List.map (fun f -> end_with f.name "print") program)
    then redefine_print_func_error "_" else ();
    (* check duplicate function *)
    let m = StringMap.empty in
    ignore(List.map (fun f ->
        if StringMap.mem f.name m
        then (duplicate_func_error f.name)
        else StringMap.add f.name true m) program);
    (* Function declaration for a named function *)
    let built_in_funcs =
      let funcs = [
          (
            "print",
           { returnType = Void_t; name = "print"; args = [Formal(String_t, "x")];
             locals = []; body = []; pname = "main"}
          );
          (
            "printb",
```

```
           { returnType = Void_t; name = "printb"; args = [Formal(Bool_t, "x")];
             locals = []; body = []; pname = "main"}
          );
          (
          "printf",
           { returnType = Void_t; name = "printf"; args = [Formal(String_t, "x")];
             locals = []; body = []; pname = "main"}
          );
          (
          "string",
           { returnType = String_t; name = "string"; args = [Formal(String_t, "x")];
             locals = []; body = []; pname = "main"}
          );
          (
          "int",
           { returnType = Int_t; name = "int"; args = [Formal(String_t, "x")];
             locals = []; body = []; pname = "main"}
          );
          (
          "float",
           { returnType = Float_t; name = "float"; args = [Formal(String_t, "x")];
             locals = []; body = []; pname = "main"}
          );
          (
          "bool",
           { returnType = Bool_t; name = "bool"; args = [Formal(String_t, "x")];
             locals = []; body = []; pname = "main"}
          )
      ]
      in
      let add_func funcs m =
          List.fold_left (fun m (n, func) -> StringMap.add n func m) m funcs
      in
      add_func funcs StringMap.empty
    in
    (* collect all functions and store in map with key=name, value=function *)
    let func_map = List.fold_left (fun m f -> StringMap.add f.name f m) built_in_funcs
program in
    let check_function_wrapper func m =
        func m
    in
    (**** Checking functions ****)
    List.iter (check_function_wrapper check_function func_map) program
```

9.8. Codegen.ml

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

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

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

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
```

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

*)

module L = Llvm
module A = Cast

module StringMap = Map.Make(String)

let context = L.global_context ()
let llctx = L.global_context ()
let customM = L.MemoryBuffer.of_file "utils.bc"
let llm = Llvm_bitreader.parse_bitcode llctx customM
let the_module = L.create_module context "Circline"

let i32_t  = L.i32_type  context
and f_t  = L.double_type context
and i8_t   = L.i8_type   context
and i1_t   = L.i1_type   context
and str_t  = L.pointer_type (L.i8_type context)
and void_t = L.void_type context
and void_ptr_t = L.pointer_type (L.i8_type context)

let node_t = L.pointer_type (match L.type_by_name llm "struct.Node" with
    None -> raise (Failure "struct.Node doesn't defined.")
  | Some x -> x)

let edge_t = L.pointer_type (match L.type_by_name llm "struct.Edge" with
    None -> raise (Failure "struct.Edge doesn't defined.")
  | Some x -> x)

let graph_t = L.pointer_type (match L.type_by_name llm "struct.Graph" with
    None -> raise (Failure "struct.Graph doesn't defined.")
  | Some x -> x)

let dict_t = L.pointer_type (match L.type_by_name llm "struct.hashmap_map" with
    None -> raise (Failure "struct.hashmap_map doesn't defined.")
  | Some x -> x)

let list_t = L.pointer_type (match L.type_by_name llm "struct.List" with
    None -> raise (Failure "struct.List doesn't defined.")
  | Some x -> x)

let ltype_of_typ = function
    A.Int_t -> i32_t
  | A.Float_t -> f_t
  | A.Bool_t -> i1_t
  | A.String_t -> str_t
  | A.Void_t -> void_t
  | A.Node_t -> node_t
  | A.Edge_t -> edge_t
  | A.List_Int_t -> list_t
  | A.List_Float_t -> list_t
  | A.List_String_t -> list_t
  | A.List_Node_t -> list_t
  | A.List_Graph_t -> list_t
  | A.List_Bool_t -> list_t
  | A.Dict_Int_t -> dict_t
  | A.Dict_Float_t -> dict_t
  | A.Dict_String_t -> dict_t
  | A.Dict_Node_t -> dict_t
```

```
  | A.Dict_Graph_t -> dict_t
  | A.Graph_t -> graph_t
  | _ -> raise (Failure ("[Error] Type Not Found for ltype_of_typ."))

let type_of_list_type = function
    A.List_Int_t -> A.Int_t
  | A.List_Float_t -> A.Float_t
  | A.List_String_t -> A.String_t
  | A.List_Node_t -> A.Node_t
  | A.List_Graph_t -> A.Graph_t
  | A.List_Bool_t -> A.Bool_t
  | _ -> raise (Failure ("[Error] Type Not Found for type_of_list_type."))

let type_of_dict_type = function
    A.Dict_Int_t -> A.Int_t
  | A.Dict_Float_t -> A.Float_t
  | A.Dict_String_t -> A.String_t
  | A.Dict_Node_t -> A.Node_t
  | A.Dict_Graph_t -> A.Graph_t
  | _ -> raise (Failure ("[Error] Type Not Found for type_of_dict_type."))

let lconst_of_typ = function
    A.Int_t -> L.const_int i32_t 0
  | A.Float_t -> L.const_int i32_t 1
  | A.Bool_t -> L.const_int i32_t 2
  | A.String_t -> L.const_int i32_t 3
  | A.Node_t -> L.const_int i32_t 4
  | A.Graph_t -> L.const_int i32_t 5
  | A.Edge_t -> L.const_int i32_t 8
  (* | A.List_Int_t -> list_t
  | A.Dict_String_t -> dict_t *)
  | _ -> raise (Failure ("[Error] Type Not Found for lconst_of_typ."))

let int_zero = L.const_int i32_t 0
and float_zero = L.const_float f_t 0.
and bool_false = L.const_int i1_t 0
and bool_true = L.const_int i1_t 1
and const_null = L.const_int i32_t 0
and str_null = L.const_null str_t
and node_null = L.const_null node_t
and graph_null = L.const_null graph_t
and list_null = L.const_null list_t
and dict_null = L.const_null dict_t

let get_null_value_of_type = function
  | A.String_t -> str_null
  | A.Node_t -> node_null
  | A.Graph_t -> graph_null
  | A.List_Int_t
  | A.List_Float_t
  | A.List_String_t
  | A.List_Node_t
  | A.List_Graph_t
  | A.List_Bool_t -> list_null
  | A.Dict_Int_t
  | A.Dict_Float_t
  | A.Dict_String_t
  | A.Dict_Node_t
  | A.Dict_Graph_t -> dict_null
  | _ -> raise (Failure ("[Error] Type Not Found for get_null_value_of_type."))
```

```
let get_default_value_of_type = function
  | A.Int_t as t -> L.const_int (ltype_of_typ t) 0
  | A.Bool_t as t -> L.const_int (ltype_of_typ t) 0
  | A.Float_t as t-> L.const_float (ltype_of_typ t) 0.
  | t-> L.const_null (ltype_of_typ t)

(*
================================================================
  Casting
================================================================
*)

let int_to_float llbuilder v = L.build_sitofp v f_t "tmp" llbuilder

let void_to_int_t  = L.function_type i32_t [| L.pointer_type i8_t |]
let void_to_int_f  = L.declare_function "VoidtoInt" void_to_int_t the_module
let void_to_int void_ptr llbuilder =
  let actuals = [| void_ptr |] in
    L.build_call void_to_int_f actuals "VoidtoInt" llbuilder

let void_to_float_t  = L.function_type f_t [| L.pointer_type i8_t |]
let void_to_float_f  = L.declare_function "VoidtoFloat" void_to_float_t the_module
let void_to_float void_ptr llbuilder =
  let actuals = [| void_ptr |] in
    L.build_call void_to_float_f actuals "VoidtoFloat" llbuilder

let void_to_bool_t  = L.function_type i1_t [| L.pointer_type i8_t |]
let void_to_bool_f  = L.declare_function "VoidtoBool" void_to_bool_t the_module
let void_to_bool void_ptr llbuilder =
  let actuals = [| void_ptr |] in
    L.build_call void_to_bool_f actuals "VoidtoBool" llbuilder

let void_to_string_t  = L.function_type str_t [| L.pointer_type i8_t |]
let void_to_string_f  = L.declare_function "VoidtoString" void_to_string_t the_module
let void_to_string void_ptr llbuilder =
  let actuals = [| void_ptr |] in
    L.build_call void_to_string_f actuals "VoidtoString" llbuilder

let void_to_node_t  = L.function_type node_t [| L.pointer_type i8_t |]
let void_to_node_f  = L.declare_function "VoidtoNode" void_to_node_t the_module
let void_to_node void_ptr llbuilder =
  let actuals = [| void_ptr |] in
    L.build_call void_to_node_f actuals "VoidtoNode" llbuilder

let void_to_graph_t  = L.function_type graph_t [| L.pointer_type i8_t |]
let void_to_graph_f  = L.declare_function "VoidtoGraph" void_to_graph_t the_module
let void_to_graph void_ptr llbuilder =
  let actuals = [| void_ptr |] in
    L.build_call void_to_graph_f actuals "VoidtoGraph" llbuilder

let void_start_to_tpy value_void_ptr llbuilder = function
    A.Int_t -> void_to_int value_void_ptr llbuilder
  | A.Float_t -> void_to_float value_void_ptr llbuilder
  | A.Bool_t -> void_to_bool value_void_ptr llbuilder
  | A.String_t -> void_to_string value_void_ptr llbuilder
  | A.Node_t -> void_to_node value_void_ptr llbuilder
  | A.Graph_t -> void_to_graph value_void_ptr llbuilder
  | _ -> raise (Failure("[Error] Unsupported value type."))

(*
================================================================
```

```
  Declare printf(), which the print built-in function will call
================================================================
*)
let printf_t = L.var_arg_function_type i32_t [| str_t |]
let printf_func = L.declare_function "printf" printf_t the_module
let codegen_print llbuilder el =
  L.build_call printf_func (Array.of_list el) "printf" llbuilder

let print_bool_t = L.function_type i32_t [| i1_t |]
let print_bool_f = L.declare_function "printBool" print_bool_t the_module
let print_bool e llbuilder =
  L.build_call print_bool_f [| e |] "print_bool" llbuilder

let codegen_string_lit s llbuilder =
  L.build_global_stringptr s "str_tmp" llbuilder

(*
================================================================
  Node & Edge
================================================================
*)
let create_node_t  = L.var_arg_function_type node_t [| i32_t; i32_t |]
let create_node_f  = L.declare_function "createNode" create_node_t the_module
let create_node (id, typ, nval) llbuilder =
  let actuals = [| id; lconst_of_typ typ; nval |] in
    L.build_call create_node_f actuals "node" llbuilder

let node_get_value_t = L.function_type void_ptr_t [| node_t; i32_t |]
let node_get_value_f = L.declare_function "nodeGetValue" node_get_value_t the_module
let node_get_value node typ llbuilder =
  let actuals = [| node; lconst_of_typ typ |] in
  let ret = L.build_call node_get_value_f actuals "nodeValue" llbuilder in
  (   match typ with
    | A.Int_t -> void_to_int ret llbuilder
    | A.Float_t -> void_to_float ret llbuilder
    | A.Bool_t -> void_to_bool ret llbuilder
    | A.String_t -> void_to_string ret llbuilder
    | _ -> raise (Failure("[Error] Unsupported node value type."))
  )

let edge_get_value_t = L.function_type void_ptr_t [| edge_t; i32_t |]
let edge_get_value_f = L.declare_function "edgeGetValue" edge_get_value_t the_module
let edge_get_value edge typ llbuilder =
  let actuals = [| edge; lconst_of_typ typ |] in
  let ret = L.build_call edge_get_value_f actuals "edgeValue" llbuilder in
  (   match typ with
    | A.Int_t -> void_to_int ret llbuilder
    | A.Float_t -> void_to_float ret llbuilder
    | A.Bool_t -> void_to_bool ret llbuilder
    | A.String_t -> void_to_string ret llbuilder
    | _ -> raise (Failure("[Error] Unsupported edge value type."))
  )

let print_node_t  = L.function_type i32_t [| node_t |]
  let print_node_f  = L.declare_function "printNode" print_node_t the_module
  let print_node node llbuilder =
   L.build_call print_node_f [| node |] "printNode" llbuilder

let print_edge_t  = L.function_type i32_t [| edge_t |]
  let print_edge_f  = L.declare_function "printEdgeValue" print_edge_t the_module
  let print_edge edge llbuilder =
```

```
    L.build_call print_edge_f [| edge |] "printEdge" llbuilder


(*
================================================================
  Dict
================================================================
*)
let create_dict_t = L.var_arg_function_type dict_t [| i32_t; i32_t |]
let create_dict_f = L.declare_function "hashmap_new" create_dict_t the_module
let create_dict fst_typ snd_typ llbuilder =
    L.build_call create_dict_f [| fst_typ; snd_typ |] "hashmap" llbuilder
    (* L.build_call create_dict_f [| L.const_int i32_t 3; L.const_int i32_t 3 |]
"hashmap_new" llbuilder *)

let put_dict_t = L.var_arg_function_type dict_t [| dict_t |]
let put_dict_f = L.declare_function "hashmap_put" put_dict_t the_module
let put_dict d key v llbuilder =
    let actuals = [| d; key; v |] in
    ignore (L.build_call put_dict_f actuals "hashmap_put" llbuilder); d

let get_dict_t = L.var_arg_function_type (L.pointer_type i8_t) [| dict_t |]
let get_dict_f = L.declare_function "hashmap_get" get_dict_t the_module
let get_dict dict_ptr key llbuilder v_typ =
    let actuals = [| dict_ptr; key |] in
    let value_void_ptr = L.build_call get_dict_f actuals "hashmap_get" llbuilder in
    void_start_to_tpy value_void_ptr llbuilder v_typ

let remove_dict_t = L.var_arg_function_type dict_t [| dict_t |]
let remove_dict_f = L.declare_function "hashmap_remove" remove_dict_t the_module
let remove_dict dict_ptr key llbuilder =
    let actuals = [| dict_ptr; key |] in
    L.build_call remove_dict_f actuals "hashmap_remove" llbuilder

let size_dict_t = L.var_arg_function_type i32_t [| dict_t |]
let size_dict_f = L.declare_function "hashmap_length" size_dict_t the_module
let size_dict dict_ptr llbuilder =
    let actuals = [| dict_ptr |] in
    L.build_call size_dict_f actuals "hashmap_length" llbuilder

let keys_dict_t = L.var_arg_function_type list_t [| dict_t |]
let keys_dict_f = L.declare_function "hashmap_keys" keys_dict_t the_module
let keys_dict dict_ptr llbuilder =
    let actuals = [| dict_ptr |] in
    L.build_call keys_dict_f actuals "hashmap_keys" llbuilder

let key_type_dict_t = L.var_arg_function_type i32_t [| dict_t |]
let key_type_dict_f = L.declare_function "hashmap_keytype" key_type_dict_t the_module
let key_type_dict dict_ptr llbuilder =
    let actuals = [| dict_ptr |] in
    L.build_call key_type_dict_f actuals "hashmap_keytype" llbuilder

let print_dict_t  = L.function_type i32_t [| dict_t |]
let print_dict_f  = L.declare_function "hashmap_print" print_dict_t the_module
let print_dict d llbuilder =
  L.build_call print_dict_f [| d |] "hashmap_print" llbuilder

let haskey_dict_t = L.var_arg_function_type i1_t [| dict_t |]
let haskey_dict_f = L.declare_function "hashmap_haskey" haskey_dict_t the_module
let haskey_dict dict_ptr key llbuilder =
    let actuals = [| dict_ptr; key |] in
```

```
    L.build_call haskey_dict_f actuals "hashmap_haskey" llbuilder

let rec put_multi_kvs_dict dict_ptr llbuilder = function
  | [] -> dict_ptr
  | hd :: tl -> ignore(put_dict dict_ptr (fst hd) (snd hd) llbuilder); put_multi_kvs_dict
dict_ptr llbuilder tl

let dict_call_default_main builder dict_ptr params_list v_typ = function
  | "get" -> (get_dict dict_ptr (List.hd params_list) builder (type_of_dict_type v_typ)),
(type_of_dict_type v_typ)
  | "put" -> (put_dict dict_ptr (List.hd params_list) (List.nth params_list 1) builder),
v_typ
  | "remove" -> (remove_dict dict_ptr (List.hd params_list) builder), v_typ
  | "size" -> (size_dict dict_ptr builder), A.Int_t
  | "keys" -> (keys_dict dict_ptr builder), A.List_Null_t
  | "has" -> (haskey_dict dict_ptr (List.hd params_list) builder), A.Bool_t
  | _ as name -> raise (Failure ("[Error] Unsupported default call for dict." ^ name))

(*
================================================================
  List
================================================================
*)

let create_list_t  = L.function_type list_t [| i32_t |]
let create_list_f  = L.declare_function "createList" create_list_t the_module
let create_list typ llbuilder =
  let actuals = [|lconst_of_typ typ|]in (
    L.build_call create_list_f actuals "createList" llbuilder
  )

let add_list_t  = L.var_arg_function_type list_t [| list_t |]
let add_list_f  = L.declare_function "addList" add_list_t the_module
let add_list data l_ptr llbuilder =
  let actuals = [| l_ptr; data|] in
    (L.build_call add_list_f actuals "addList" llbuilder)

let set_list_t  = L.var_arg_function_type i32_t [| list_t; i32_t |]
let set_list_f  = L.declare_function "setList" set_list_t the_module
let set_list l_ptr index data llbuilder =
  let actuals = [| l_ptr; index; data |] in
    ignore(L.build_call set_list_f actuals "setList" llbuilder);
    l_ptr

let remove_list_t  = L.var_arg_function_type i32_t [| list_t; i32_t |]
let remove_list_f  = L.declare_function "removeList" remove_list_t the_module
let remove_list l_ptr index llbuilder =
  let actuals = [| l_ptr; index |] in
    ignore(L.build_call remove_list_f actuals "removeList" llbuilder);
    l_ptr

let size_list_t  = L.var_arg_function_type i32_t [| list_t |]
let size_list_f  = L.declare_function "getListSize" size_list_t the_module
let size_list l_ptr llbuilder =
  let actuals = [| l_ptr |] in
    L.build_call size_list_f actuals "getListSize" llbuilder

let pop_list_t  = L.var_arg_function_type (L.pointer_type i8_t) [| list_t |]
let pop_list_f  = L.declare_function "popList" pop_list_t the_module
let pop_list l_ptr typ llbuilder =
  let actuals = [| l_ptr |] in
```

```
  let value_void_ptr = L.build_call pop_list_f actuals "popList" llbuilder in
  void_start_to_tpy value_void_ptr llbuilder typ

let get_list_t  = L.var_arg_function_type (L.pointer_type i8_t) [| list_t; i32_t|]
let get_list_f  = L.declare_function "getList" get_list_t the_module
let get_list l_ptr index typ llbuilder =
  let actuals = [| l_ptr; index|] in
  let value_void_ptr = L.build_call get_list_f actuals "getList" llbuilder in
  void_start_to_tpy value_void_ptr llbuilder typ

let concat_list_t  = L.var_arg_function_type list_t [| list_t; list_t |]
let concat_list_f  = L.declare_function "concatList" concat_list_t the_module
let concat_list l_ptr1 l_ptr2 llbuilder =
  let actuals = [| l_ptr1; l_ptr2 |] in
    L.build_call concat_list_f actuals "concatList" llbuilder

let cast_float data typ builder = if typ == A.Float_t then int_to_float builder data else
data

let rec add_multi_elements_list l_ptr typ llbuilder = function
  | [] -> l_ptr
  | h :: tl -> add_multi_elements_list (add_list (cast_float h typ llbuilder) l_ptr
llbuilder) typ llbuilder tl

let print_list_t  = L.function_type i32_t [| list_t |]
let print_list_f  = L.declare_function "printList" print_list_t the_module
let print_list l llbuilder =
  L.build_call print_list_f [| l |] "printList" llbuilder

let list_call_default_main builder list_ptr params_list expr_tpy = function
    "add" -> (add_list (List.hd params_list) list_ptr builder), expr_tpy
  | "get" -> (get_list list_ptr (List.hd params_list) (type_of_list_type expr_tpy) builder),
(type_of_list_type expr_tpy)
  | "set" -> (set_list list_ptr (List.hd params_list) (List.nth params_list 1) builder),
expr_tpy
  | "remove" -> (remove_list list_ptr (List.hd params_list) builder) ,expr_tpy
  | "size" -> (size_list list_ptr builder), A.Int_t
  | "pop" -> (pop_list list_ptr (type_of_list_type expr_tpy) builder), (type_of_list_type
expr_tpy)
  | "push" -> (add_list (List.hd params_list) list_ptr builder), expr_tpy
  | _ -> raise (Failure ("[Error] Unsupported default call for list."))
(*
================================================================
  Graph
================================================================
*)
(* Create a new empty grpah *)
let create_graph_t  = L.function_type graph_t [| |]
let create_graph_f  = L.declare_function "createGraph" create_graph_t the_module
let create_graph llbuilder =
  L.build_call create_graph_f [| |] "graph" llbuilder

(* Get the number of nodes in a graph *)
let graph_num_of_nodes_t  = L.function_type i32_t [| graph_t |]
let graph_num_of_nodes_f  = L.declare_function "graphNumOfNodes" graph_num_of_nodes_t
the_module
let graph_num_of_nodes g llbuilder =
  L.build_call graph_num_of_nodes_f [| g |] "graphNodeSize" llbuilder

(* Get the number of edges in a graph *)
let graph_num_of_edges_t  = L.function_type i32_t [| graph_t |]
```

```
let graph_num_of_edges_f  = L.declare_function "graphNumOfEdges" graph_num_of_edges_t
the_module
let graph_num_of_edges g llbuilder =
  L.build_call graph_num_of_edges_f [| g |] "graphEdgeSize" llbuilder

(* Create a copy of origianl grpah *)
let copy_graph_t  = L.function_type graph_t [| graph_t |]
let copy_graph_f  = L.declare_function "copyGraph" copy_graph_t the_module
let copy_graph g llbuilder =
  L.build_call copy_graph_f [| g |] "graph" llbuilder

(* Merge two graphs into a single graph *)
let merge_graph_t = L.function_type graph_t [| graph_t; graph_t |]
let merge_graph_f = L.declare_function "mergeGraph" merge_graph_t the_module
let merge_graph g1 g2 llbuilder =
  L.build_call merge_graph_f [| g1; g2 |] "graph" llbuilder

(* Get the root node of the graph *)
let graph_get_root_t  = L.function_type node_t [| graph_t |]
let graph_get_root_f  = L.declare_function "graphGetRoot" graph_get_root_t the_module
let graph_get_root g llbuilder =
  L.build_call graph_get_root_f [| g |] "rootNode" llbuilder

(* Set the root node of the graph *)
let graph_set_root_t  = L.function_type graph_t [| graph_t; node_t |]
let graph_set_root_f  = L.declare_function "graphSetRoot" graph_set_root_t the_module
let graph_set_root graph node llbuilder = (
      ignore(L.build_call graph_set_root_f [| graph; node |] "setRootRes" llbuilder);
      graph
  )

(* Add a list of Nodes or Graphs to graph *)
let graph_add_list_t = L.function_type i32_t [| graph_t; i32_t; list_t; list_t |]
let graph_add_list_f = L.declare_function "graphAddList" graph_add_list_t the_module
let graph_add_list graph vals (edges, etyp) dir llbuilder =
  let edges = (
    match etyp with
    | A.List_Int_t | A.List_Float_t | A.List_String_t
    | A.List_Node_t | A.List_Graph_t | A.List_Bool_t -> edges
    | _ -> list_null
  ) in
  let direction = (
    match dir with
    | A.Right_Link -> L.const_int i32_t 0
    | A.Left_Link -> L.const_int i32_t 1
    | A.Double_Link -> L.const_int i32_t 2
  ) in
  L.build_call graph_add_list_f [| graph; direction; vals; edges |] "graphAddList" llbuilder

(* Add a new node to graph *)
let graph_add_node_t = L.function_type i32_t [| graph_t; node_t |]
let graph_add_node_f = L.declare_function "graphAddNode" graph_add_node_t the_module
let graph_add_node graph node llbuilder =
  L.build_call graph_add_node_f [| graph; node |] "addNodeRes" llbuilder

(* Add a new edge to graph *)
let graph_add_edge_t = L.function_type i32_t
  [| graph_t; node_t; node_t; i32_t; i32_t; f_t; i1_t; str_t |]
let graph_add_edge_f = L.declare_function "graphAddEdge" graph_add_edge_t the_module
let graph_add_edge graph (sour, dest) op (typ, vals) llbuilder =
```

```
  let actuals = [| graph; sour; dest; int_zero; int_zero; float_zero; bool_false; str_null
|] in
  let actuals_r = [| graph; dest; sour; int_zero; int_zero; float_zero; bool_false; str_null
|] in
  let (typ_val, loc) = (match typ with
    | A.Int_t -> (0, 4)
    | A.Float_t -> (1, 5)
    | A.Bool_t -> (2, 6)
    | A.String_t -> (3, 7)
    | A.Void_t | A.Null_t -> (-1, 4)
    | _ -> raise (Failure "[Error] Unsupported edge value type.")
  ) in (
    ignore( actuals.(3) <- (L.const_int i32_t typ_val) );
    ignore( actuals_r.(3) <- (L.const_int i32_t typ_val) );
    ignore( actuals.(loc) <- vals );
    ignore( actuals_r.(loc) <- vals );
    match op with
    | A.Right_Link -> L.build_call graph_add_edge_f actuals "addRightEdgeRes" llbuilder
    | A.Left_Link -> L.build_call graph_add_edge_f actuals_r "addLeftEdgeRes" llbuilder
    | A.Double_Link -> (
        ignore(L.build_call graph_add_edge_f actuals "addRightEdgeRes" llbuilder);
        L.build_call graph_add_edge_f actuals_r "addLeftEdgeRes" llbuilder
      )
  )

let graph_edge_exist_t = L.function_type i1_t [| graph_t; node_t; node_t |]
let graph_edge_exist_f = L.declare_function "graphEdgeExist" graph_edge_exist_t the_module
let graph_edge_exist graph sour dest llbuilder =
  L.build_call graph_edge_exist_f [| graph; sour; dest |] "boolValue" llbuilder

let graph_get_edge_t = L.function_type edge_t [| graph_t; node_t; node_t |]
let graph_get_edge_f = L.declare_function "graphGetEdge" graph_get_edge_t the_module
let graph_get_edge graph sour dest llbuilder =
  L.build_call graph_get_edge_f [| graph; sour; dest |] "edgeValue" llbuilder

(* Print out the graph *)
let print_graph_t  = L.function_type i32_t [| graph_t |]
let print_graph_f  = L.declare_function "printGraph" print_graph_t the_module
let print_graph graph llbuilder =
  L.build_call print_graph_f [| graph |] "printGraph" llbuilder

(* Get all neighbor nodes of the specific node which a graph *)
let graph_get_child_nodes_t  = L.function_type list_t [| graph_t; node_t |]
let graph_get_child_nodes_f  = L.declare_function "graphGetChildNodes"
graph_get_child_nodes_t the_module
let graph_get_child_nodes graph root llbuilder =
  L.build_call graph_get_child_nodes_f [| graph; root |] "childNodes" llbuilder

(* Get all nodes of the graph *)
let graph_get_all_nodes_t  = L.function_type list_t [| graph_t |]
let graph_get_all_nodes_f  = L.declare_function "graphGetAllNodes" graph_get_all_nodes_t
the_module
let graph_get_all_nodes graph llbuilder =
  L.build_call graph_get_all_nodes_f [| graph |] "nodesList" llbuilder

(* Remove a particular node of the graph *)
let graph_remove_node_t = L.function_type list_t [| graph_t; node_t |]
let graph_remove_node_f = L.declare_function "graphRemoveNode" graph_remove_node_t
the_module
let graph_remove_node graph node llbuilder =
  L.build_call graph_remove_node_f [| graph; node |] "listOfSubGraphs" llbuilder
```

```
(* Remove a particular node of the graph *)
let graph_sub_graph_t = L.function_type list_t [| graph_t; graph_t |]
let graph_sub_graph_f = L.declare_function "subGraph" graph_sub_graph_t the_module
let graph_sub_graph g1 g2 llbuilder =
  L.build_call graph_sub_graph_f [| g1; g2 |] "listOfSubGraphs" llbuilder

let graph_call_default_main llbuilder gh = function
  | "root" -> graph_get_root gh llbuilder , A.Node_t
  | "size" -> graph_num_of_nodes gh llbuilder, A.Int_t
  | "nodes" -> graph_get_all_nodes gh llbuilder, A.List_Node_t
  | _ as name -> raise (Failure("[Error] Unsupported graph methods: " ^ name ))

(*
================================================================
        context_funcs_vars
================================================================
*)
let context_funcs_vars = Hashtbl.create 50
let print_hashtbl tb =
  print_endline (Hashtbl.fold (fun k _ m -> (k^", "^m)) tb "")

(*
================================================================
        Main Codegen Function
================================================================
*)
let translate program =
  (* Define each function (arguments and return type) so we can call it *)
  let function_decls =
    let function_decl m fdecl =
      let name = fdecl.A.name
      and formal_types =
              Array.of_list (List.map (fun (A.Formal(t, _)) -> ltype_of_typ t) fdecl.A.args)
      in
      let ftype = L.var_arg_function_type (ltype_of_typ fdecl.A.returnType) formal_types in
      StringMap.add name (L.define_function name ftype the_module, fdecl) m in
    List.fold_left function_decl StringMap.empty program in

  (* Fill in the body of the given function *)
  let build_function_body fdecl =
    let get_var_name fname n = (fname ^ "." ^ n) in
    let (the_function, _) = StringMap.find fdecl.A.name function_decls in
    (* let bb = L.append_block context "entry" the_function in *)
    let builder = L.builder_at_end context (L.entry_block the_function) in

    (* Construct the function's "locals": formal arguments and locally
       declared variables.  Allocate each on the stack, initialize their
       value, if appropriate, and remember their values in the "locals" map *)
    let _ =
      let add_to_context locals =
        ignore(Hashtbl.add context_funcs_vars fdecl.A.name locals);
        (* ignore(print_hashtbl context_funcs_vars); *)
        locals
      in
      let add_formal m (A.Formal(t, n)) p =
        let n' = get_var_name fdecl.A.name n in
        let local = L.define_global n' (get_default_value_of_type t) the_module in
          if L.is_null p then () else ignore (L.build_store p local builder);
         StringMap.add n' (local, t) m
      in
```

```
    let add_local m (A.Formal(t, n)) =
      let n' = get_var_name fdecl.A.name n in
     let local_var = L.define_global n' (get_default_value_of_type t) the_module in
      StringMap.add n' (local_var, t) m
    in

    let formals = List.fold_left2 add_formal StringMap.empty fdecl.A.args
        (Array.to_list (L.params the_function)) in
    add_to_context (List.fold_left add_local formals fdecl.A.locals)
  in

  (* Return the value for a variable or formal argument *)
  (* let lookup n = StringMap.find n local_vars
  in *)
  let lookup n =
    let get_parent_func_name fname =
      let (_, fdecl) = StringMap.find fname function_decls in
      fdecl.A.pname
    in
    let rec aux n fname = (
      try StringMap.find (get_var_name fname n) (Hashtbl.find context_funcs_vars fname)
      with Not_found -> (
        if fname = "main" then
          (raise (Failure("[Error] Local Variable not found.")))
        else
          (aux n (get_parent_func_name fname))
      )
    ) in
    aux n fdecl.A.name
  in

  (* Construct code for an expression; return its value *)
  let handle_binop e1 op e2 dtype llbuilder =
    (* Generate llvalues from e1 and e2 *)

    let float_ops op e1 e2 =
      match op with
        A.Add     -> L.build_fadd e1 e2 "flt_addtmp" llbuilder
      | A.Sub     -> L.build_fsub e1 e2 "flt_subtmp" llbuilder
      | A.Mult    -> L.build_fmul e1 e2 "flt_multmp" llbuilder
      | A.Div     -> L.build_fdiv e1 e2 "flt_divtmp" llbuilder
      | A.Mod     -> L.build_frem e1 e2 "flt_sremtmp" llbuilder
      | A.Equal   -> L.build_fcmp L.Fcmp.Oeq e1 e2 "flt_eqtmp" llbuilder
      | A.Neq     -> L.build_fcmp L.Fcmp.One e1 e2 "flt_neqtmp" llbuilder
      | A.Less    -> L.build_fcmp L.Fcmp.Ult e1 e2 "flt_lesstmp" llbuilder
      | A.Leq     -> L.build_fcmp L.Fcmp.Ole e1 e2 "flt_leqtmp" llbuilder
      | A.Greater -> L.build_fcmp L.Fcmp.Ogt e1 e2 "flt_sgttmp" llbuilder
      | A.Geq     -> L.build_fcmp L.Fcmp.Oge e1 e2 "flt_sgetmp" llbuilder
      | _ -> raise (Failure("[Error] Unrecognized float binop opreation."))
    in

    (* chars are considered ints, so they will use int_ops as well*)
    let int_ops op e1 e2 =
      match op with
        A.Add     -> L.build_add e1 e2 "addtmp" llbuilder
      | A.Sub     -> L.build_sub e1 e2 "subtmp" llbuilder
      | A.Mult    -> L.build_mul e1 e2 "multmp" llbuilder
      | A.Div     -> L.build_sdiv e1 e2 "divtmp" llbuilder
      | A.Mod     -> L.build_srem e1 e2 "sremtmp" llbuilder
      | A.Equal   -> L.build_icmp L.Icmp.Eq e1 e2 "eqtmp" llbuilder
```

```
        | A.Neq     -> L.build_icmp L.Icmp.Ne e1 e2 "neqtmp" llbuilder
        | A.Less    -> L.build_icmp L.Icmp.Slt e1 e2 "lesstmp" llbuilder
        | A.Leq     -> L.build_icmp L.Icmp.Sle e1 e2 "leqtmp" llbuilder
        | A.Greater -> L.build_icmp L.Icmp.Sgt e1 e2 "sgttmp" llbuilder
        | A.Geq     -> L.build_icmp L.Icmp.Sge e1 e2 "sgetmp" llbuilder
        | A.And     -> L.build_and e1 e2 "andtmp" llbuilder
        | A.Or      -> L.build_or  e1 e2 "ortmp" llbuilder
        | _ -> raise (Failure("[Error] Unrecognized int binop opreation."))
        in
      let type_handler d = match d with
        | A.Float_t -> float_ops op e1 e2
        | A.Bool_t
        | A.Int_t -> int_ops op e1 e2
        | _ -> raise (Failure("[Error] Unrecognized binop data type."))
      in (type_handler dtype,
        match op with
        | A.Add | A.Sub | A.Mult | A.Div | A.Mod -> dtype
        | _ -> A.Bool_t
      )
    in

    let rec expr builder = function
            A.Num_Lit(A.Num_Int i) -> (L.const_int i32_t i, A.Int_t)
      | A.Num_Lit(A.Num_Float f) -> (L.const_float f_t f, A.Float_t)
      | A.Bool_lit b -> (L.const_int i1_t (if b then 1 else 0), A.Bool_t)
      | A.String_Lit s -> (codegen_string_lit s builder, A.String_t)
      | A.Noexpr -> (L.const_int i32_t 0, A.Void_t)
      | A.Null -> (const_null, A.Null_t)
      | A.Id s ->
          let (var, typ) = lookup s in
          (L.build_load var s builder, typ)
      | A.Node(id, e) ->
          let (nval, typ) = expr builder e in
          (create_node (L.const_int i32_t id, typ, nval) builder, A.Node_t)
      | A.EdgeAt(e0, e1, e2) ->
          let (gh_val, gh_typ) = expr builder e0 in
          let (n1_val, n1_typ) = expr builder e1 in
          let (n2_val, n2_typ) = expr builder e2 in (
                match (gh_typ, n1_typ, n2_typ) with
              | (A.Graph_t, A.Node_t, A.Node_t) -> (
                  (graph_get_edge gh_val n1_val n2_val builder, A.Edge_t)
                )
              | _ -> raise (Failure("[Error] Unsupported EdgeAt() Expr."))
          )
      | A.ListP(ls) ->
          let from_expr_typ_to_list_typ = function
              A.Int_t -> A.List_Int_t
            | A.Float_t -> A.List_Float_t
            | A.String_t -> A.List_String_t
            | A.Node_t -> A.List_Node_t
            | A.Graph_t -> A.List_Graph_t
            | A.Bool_t -> A.List_Bool_t
            | _ -> A.List_Int_t
          in
          (* get the list typ by its first element *)
          let rec check_float_typ = function
            [] -> A.Int_t
          | hd::ls -> if (snd(expr builder hd)) == A.Float_t then A.Float_t else
check_float_typ ls in
          let rec check_graph_typ = function
              [] -> A.Node_t
```

```
          | hd::ls -> if (snd(expr builder hd)) == A.Graph_t then A.Graph_t else
check_graph_typ ls in
        let list_typ = snd (expr builder (List.hd ls)) in
        let list_typ = if list_typ == A.Int_t then check_float_typ ls else list_typ in
        let list_typ = if list_typ == A.Node_t then check_graph_typ ls else list_typ in

        let list_conversion el =
          let (e_val, e_typ) = expr builder el in
          (   match e_typ with
            | A.Node_t when list_typ = A.Graph_t -> (
              let gh = create_graph builder in (
                  ignore(graph_add_node gh e_val builder);
                  (gh, A.Graph_t)
              )
            )
            | _ -> (e_val, e_typ)
          )
        in

        (* create a new list first *)
        let l_ptr_type = (create_list list_typ builder, from_expr_typ_to_list_typ
list_typ) in
        (* then add all initial values to the list *)
          add_multi_elements_list (fst l_ptr_type) list_typ builder (List.map fst
(List.map list_conversion ls)), (snd l_ptr_type)
    | A.DictP(expr_list) ->
        let from_type_to_dict_typ = function
            A.Int_t -> A.Dict_Int_t
          | A.String_t -> A.Dict_String_t
          | A.Node_t -> A.Dict_Node_t
          | A.Float_t -> A.Dict_Float_t
          | A.Graph_t -> A.Dict_Graph_t
          | _ -> raise (Failure "[Error] Unsupported key type for dict.")
        in
        let first_expr_kv = List.hd expr_list in
        (* get type of key and value *)
        let first_typ = lconst_of_typ (snd (expr builder (fst first_expr_kv))) in
        let second_typ = lconst_of_typ (snd (expr builder (snd first_expr_kv))) in
        let return_typ = from_type_to_dict_typ (snd (expr builder (snd first_expr_kv))) in
        let dict_ptr = create_dict first_typ second_typ builder in
        ignore(put_multi_kvs_dict dict_ptr builder
              (List.map (fun (key, v) -> fst(expr builder key), fst(expr builder v))
expr_list), return_typ);
              (dict_ptr, return_typ)

    | A.Graph_Link(left, op, right, edges) ->
        let (ln, ln_type) = expr builder left in
        let (rn, rn_type) = expr builder right in
        let (el, el_type) = expr builder edges in (
          match (ln_type, rn_type, el_type) with
          | (A.Node_t, A.Null_t, _) -> (
              let gh = create_graph builder in (
                  ignore(graph_add_node gh ln builder);
                  (gh, A.Graph_t)
              )
            )
          | (A.Node_t, A.Node_t, _) -> (
              let gh = create_graph builder in (
                  ignore(graph_add_node gh ln builder); (* Also set the root *)
                  ignore(graph_add_node gh rn builder);
                  ignore(graph_add_edge gh (ln, rn) op (el_type, el) builder);
```

```
                (gh, A.Graph_t)
              )
            )
          | (A.Node_t, A.Graph_t, _) -> (
              let gh = copy_graph rn builder in
              let rt = graph_get_root rn builder in (
                  ignore(graph_add_node gh ln builder);
                  ignore(graph_set_root gh ln builder);
                  ignore(graph_add_edge gh (ln, rt) op (el_type, el) builder);
                  (gh, A.Graph_t)
              )
            )
          | (A.Node_t, A.List_Graph_t, _)
          | (A.Node_t, A.List_Node_t, _) -> (
              let gh = create_graph builder in (
                ignore(graph_add_node gh ln builder); (* Also set the root *)
                ignore(graph_add_list gh rn (el, el_type) op builder);
                (gh, A.Graph_t)
              )
            )
          | _ -> raise (Failure "[Error] Graph Link Under build.")
        )
    | A.Binop (e1, op, e2) ->
      let (e1', t1) = expr builder e1
      and (e2', t2) = expr builder e2 in
      (* Handle Automatic Binop Type Converstion *)
      (match (t1, t2) with
        | (A.List_Int_t, A.List_Int_t)
        | (A.List_Float_t, A.List_Float_t)
        | (A.List_Bool_t, A.List_Bool_t)
        | (A.List_String_t, A.List_String_t)
        | (A.List_Node_t, A.List_Node_t)
        | (A.List_Graph_t, A.List_Graph_t) -> (
              match op with
              | A.Add -> (concat_list e1' e2' builder, t1)
              | _ -> raise (Failure ("[Error] Unsuported Binop Type On List."))
            )
        | ( A.Graph_t, A.Graph_t) -> (
              match op with
              | A.Add -> (merge_graph e1' e2' builder, A.Graph_t)
              | A.Sub -> (graph_sub_graph e1' e2' builder, A.List_Graph_t)
              | _ -> raise (Failure ("[Error] Unsuported Binop Type On Graph."))
            )
        | ( A.Graph_t, A.Node_t ) -> (
              match  op with
              | A.RootAs ->
                  let gh = copy_graph e1' builder in
                    (graph_set_root gh e2' builder, A.Graph_t)
              | A.ListNodesAt -> (graph_get_child_nodes e1' e2' builder, A.List_Node_t)
              | A.Sub -> (graph_remove_node e1' e2' builder, A.List_Graph_t)
              | _ -> raise (Failure ("[Error] Unsuported Binop Type On Graph * Node."))
          )
        | ( _, A.Null_t ) -> (
                match op with
              | A.Equal -> (L.build_is_null e1' "isNull" builder, A.Bool_t)
              | A.Neq -> (L.build_is_not_null e1' "isNull" builder, A.Bool_t)
              | _ -> raise (Failure("[Error] Unsupported Null Type Operation."))
          )
        | ( A.Null_t, _ ) -> (
                match op with
              | A.Equal -> (L.build_is_null e2' "isNotNull" builder, A.Bool_t)
```

```
                | A.Neq -> (L.build_is_not_null e2' "isNotNull" builder, A.Bool_t)
                | _ -> raise (Failure("[Error] Unsupported Null Type Operation."))
            )
          | ( t1, t2) when t1 = t2 -> handle_binop e1' op e2' t1 builder
          | ( A.Int_t, A.Float_t) ->
             handle_binop (int_to_float builder e1') op e2' A.Float_t builder
          | ( A.Float_t, A.Int_t ) ->
             handle_binop e1' op (int_to_float builder e2') A.Float_t builder
          | _ -> raise (Failure ("[Error] Unsuported Binop Type."))
        )
      | A.Unop(op, e) ->
         let (e', typ) = expr builder e in
         ((match op with
           A.Neg     -> if typ = A.Int_t then L.build_neg else L.build_fneg
          | A.Not     -> L.build_not) e' "tmp" builder, typ)
      | A.Assign (s, e) ->
          let (e', etyp) = expr builder e in
          let (var, typ) = lookup s in
          (( match (etyp, typ) with
            | (t1, t2) when t1 = t2 -> ignore (L.build_store e' var builder); e'
            | (A.List_Null_t, _) -> ignore (L.build_store e' var builder); e'
            | (A.Null_t, _) -> ignore (L.build_store (get_null_value_of_type typ) var
builder); (get_null_value_of_type typ)
            | (A.Int_t, A.Float_t) -> let e' = (int_to_float builder e') in ignore
(L.build_store e' var builder); e'
            | _ -> raise (Failure("[Error] Assign Type inconsist."))
          ), typ)
      | A.Call ("print", el) ->
          let print_expr e =
            let (eval, etyp) = expr builder e in (
              match etyp with
              | A.Int_t -> ignore(codegen_print builder [(codegen_string_lit "%d\n"
builder); eval])
              | A.Null_t -> ignore(codegen_print builder [(codegen_string_lit "null\n"
builder)])
              | A.Bool_t -> ignore(print_bool eval builder)
              | A.Float_t -> ignore(codegen_print builder [(codegen_string_lit "%f\n"
builder); eval])
              | A.String_t -> ignore(codegen_print builder [(codegen_string_lit "%s\n"
builder); eval])
              | A.Node_t -> ignore(print_node eval builder)
              | A.Edge_t -> ignore(print_edge eval builder)
              | A.Dict_Int_t | A.Dict_Float_t | A.Dict_String_t | A.Dict_Node_t
              | A.Dict_Graph_t -> ignore(print_dict eval builder)
              | A.List_Int_t -> ignore(print_list eval builder)
              | A.List_Float_t -> ignore(print_list eval builder)
              | A.List_Bool_t -> ignore(print_list eval builder)
              | A.List_String_t -> ignore(print_list eval builder)
              | A.List_Node_t -> ignore(print_list eval builder)
              | A.List_Graph_t -> ignore(print_list eval builder)
              | A.Graph_t -> ignore(print_graph eval builder)
              | _ -> raise (Failure("[Error] Unsupported type for print."))
          ) in List.iter print_expr el; (L.const_int i32_t 0, A.Void_t)
      | A.Call ("printf", el) ->
          (codegen_print builder (List.map
            (fun e -> (let (eval, _) = expr builder e in eval))
            el), A.Void_t)
      | A.Call ("int", el) ->
            let (eval, etyp) = expr builder (List.hd el) in
            ((  match etyp with
              | A.Int_t -> eval
```

```
              | A.Node_t -> node_get_value eval A.Int_t builder
              | A.Edge_t -> edge_get_value eval A.Int_t builder
              | _ -> raise (Failure("[Error] Can't convert to int."))
              ), A.Int_t)
      | A.Call ("float", el) ->
            let (eval, etyp) = expr builder (List.hd el) in
            ((  match etyp with
              | A.Int_t -> int_to_float builder eval
              | A.Float_t -> eval
              | A.Node_t -> node_get_value eval A.Float_t builder
              | A.Edge_t -> edge_get_value eval A.Float_t builder
              | _ -> raise (Failure("[Error] Can't convert to float."))
              ), A.Float_t)
      | A.Call ("bool", el) ->
            let (eval, etyp) = expr builder (List.hd el) in
            ((  match etyp with
              | A.Bool_t -> eval
              | A.Node_t -> node_get_value eval A.Bool_t builder
              | A.Edge_t -> edge_get_value eval A.Bool_t builder
              | _ -> raise (Failure("[Error] Can't convert to bool."))
              ), A.Bool_t)
      | A.Call ("string", el) ->
            let (eval, etyp) = expr builder (List.hd el) in
            ((  match etyp with
              | A.String_t -> eval
              | A.Node_t -> node_get_value eval A.String_t builder
              | A.Edge_t -> edge_get_value eval A.String_t builder
              | _ -> raise (Failure("[Error] Can't convert to string."))
              ), A.String_t)
      | A.Call (f, act) ->
         let (fdef, fdecl) = StringMap.find f function_decls in
        let actuals = List.rev (List.map
          (fun e -> (let (eval, _) = expr builder e in eval)) (List.rev act)) in
        let result = (match fdecl.A.returnType with A.Void_t -> ""
                                                     | _ -> f ^ "_result") in
         (L.build_call fdef (Array.of_list actuals) result builder, fdecl.A.returnType)

      (* default get operator of dict *)
      | A.CallDefault(val_name, default_func_name, params_list) ->
        (* get caller tpye *)
        let (id_val, expr_tpy) = (expr builder val_name) in
        let assign_func_by_typ builder = function
          (* deal with list *)
          | A.List_Int_t | A.List_Float_t | A.List_String_t | A.List_Bool_t
          | A.List_Node_t | A.List_Graph_t ->
              list_call_default_main builder id_val (List.map (fun e -> fst (expr builder
e)) params_list) expr_tpy default_func_name
          | A.Dict_Int_t | A.Dict_Float_t | A.Dict_String_t | A.Dict_Node_t | A.Dict_Graph_t
->
              dict_call_default_main builder id_val (List.map (fun e -> fst (expr builder
e)) params_list) expr_tpy default_func_name
          | A.Graph_t ->
              graph_call_default_main builder id_val default_func_name
          | _ -> raise (Failure ("[Error] Default function not support."))
          in
            assign_func_by_typ builder expr_tpy

(*       | _ -> (L.const_int i32_t 0, A.Void_t)
 *)    in

    (* Invoke "f builder" if the current block doesn't already
```

```
     have a terminal (e.g., a branch). *)
  let add_terminal builder f =
    match L.block_terminator (L.insertion_block builder) with
           Some _ -> ()
    | None -> ignore (f builder) in

  (* Build the code for the given statement; return the builder for
     the statement's successor *)
  let rec stmt builder = function
    | A.Expr e -> ignore (expr builder e); builder
    | A.Return e ->
        ignore (
          let (ev, et) = expr builder e in
          match (fdecl.A.returnType, et) with
               (A.Void_t, _) -> L.build_ret_void builder
             | (t1, t2) when t1 = t2 -> L.build_ret ev builder
          | (A.Float_t, A.Int_t) -> L.build_ret (int_to_float builder ev) builder
          | (t1, A.Null_t) -> L.build_ret (get_default_value_of_type t1) builder
          | _ -> raise (Failure("[Error] Return type doesn't match."))
        ); builder
    | A.If (predicate, then_stmt, else_stmt) ->
       let (bool_val, _) = expr builder predicate in
      let merge_bb = L.append_block context "merge" the_function in

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

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

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

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

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

       let pred_builder = L.builder_at_end context pred_bb in
       let (bool_val, _) = expr pred_builder predicate in

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

    | A.For (e1, e2, e3, body) -> List.fold_left stmt builder
          ( [A.Expr e1 ; A.While (e2, body @ [A.Expr e3]) ] )
  in

  (* Build the code for each statement in the function *)
  let builder = List.fold_left stmt builder fdecl.A.body in

  (* Add a return if the last block falls off the end *)
```

```
    add_terminal builder (match fdecl.A.returnType with
        A.Void_t -> L.build_ret_void
      | A.Int_t as t -> L.build_ret (L.const_int (ltype_of_typ t) 0)
      | A.Bool_t as t -> L.build_ret (L.const_int (ltype_of_typ t) 0)
      | A.Float_t as t-> L.build_ret (L.const_float (ltype_of_typ t) 0.)
      | t-> L.build_ret (L.const_null (ltype_of_typ t))
    )
  in

  List.iter build_function_body (List.rev program);

  the_module
```

9.9. Circline.ml

```
(* Top-level of the MicroC compiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module *)

type action = LLVM_IR | Compile (* | AST *)

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [
                            ("-l", LLVM_IR);  (* Generate LLVM, don't check *)
                            ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  let cast = Organizer.convert ast in
  Semant.check cast;
  match action with
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate cast))
  | Compile -> let m = Codegen.translate cast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

9.10. Circline.sh

```
# Check whether the file "utils.bc" exist
file="utils.bc"
if [ ! -e "$file" ]
then
       clang -emit-llvm -o utils.bc -c lib/utils.c
fi

if [ $# -eq 1 ]
then
    ./circline.native <$1 >a.ll
else
    ./circline.native $1 <$2 >a.ll
fi
clang -Wno-override-module utils.bc a.ll -o $1.exe
./$1.exe
rm a.ll
```

```
rm ./$1.exe

# /usr/local/opt/llvm38/bin/clang-3.8
```

9.11. Parserize_cast.ml

```
open Cast
open Printf

(* Unary operators *)
let txt_of_unop = function
  | Not -> "Not"
  | Neg -> "Sub"

(* Binary operators *)
let txt_of_binop = function
  (* Arithmetic *)
  | Add -> "Add"
  | Sub -> "Sub"
  | Mult -> "Mult"
  | Div -> "Div"
  | Mod -> "Mod"
  (* Boolean *)
  | Or -> "Or"
  | And -> "And"
  | Equal -> "Equal"
  | Neq -> "Neq"
  | Less -> "Less"
  | Leq -> "Leq"
  | Greater -> "Greater"
  | Geq -> "Geq"
  (* Graph *)
  | ListNodesAt -> "Child_Nodes_At"
  | ListEdgesAt -> "Child_Nodes&Edges_At"
  | RootAs -> "Root_As"

let txt_of_graph_op = function
  | Right_Link -> "RLink"
  | Left_Link -> "LLink"
  | Double_Link -> "DLink"

let txt_of_var_type = function
  | Void_t -> "void"
  | Null_t -> "null"
  | Int_t -> "int"
  | Float_t -> "float"
  | String_t -> "string"
  | Bool_t -> "bool"
  | Node_t -> "node"
  | Graph_t -> "graph"
  | Dict_Int_t -> "dict<int>"
  | Dict_Float_t -> "dict<float>"
  | Dict_String_t -> "dict<string>"
  | Dict_Node_t -> "dict<node>"
  | Dict_Graph_t -> "dict<graph>"
  | List_Int_t -> "list<int>"
  | List_Float_t -> "list<float>"
  | List_String_t -> "list<string>"
```

```
  | List_Node_t -> "list<node>"
  | List_Graph_t -> "list<graph>"

let txt_of_formal = function
| Formal(vtype, name) -> sprintf "Formal(%s, %s)" (txt_of_var_type vtype) name


let txt_of_formal_list formals =
  let rec aux acc = function
    | [] -> sprintf "%s" (String.concat ", " (List.rev acc))
    | fml :: tl -> aux (txt_of_formal fml :: acc) tl
  in aux [] formals

let txt_of_num = function
  | Num_Int(x) -> string_of_int x
  | Num_Float(x) -> string_of_float x

(* Expressions *)
let rec txt_of_expr = function
  | Num_Lit(x) -> sprintf "Num_Lit(%s)" (txt_of_num x)
  | Bool_lit(x) -> sprintf "Bool_lit(%s)" (string_of_bool x)
  | String_Lit(x) -> sprintf "String_Lit(%s)" x
  | Null -> sprintf "Null"
  | Node(node_num, x) -> sprintf "Node(%s, %s)" (string_of_int node_num) (txt_of_expr x)
  | Unop(op, e) -> sprintf "Unop(%s, %s)" (txt_of_unop op) (txt_of_expr e)
  | Binop(e1, op, e2) -> sprintf "Binop(%s, %s, %s)"
      (txt_of_expr e1) (txt_of_binop op) (txt_of_expr e2)
  | Graph_Link(e1, op1, e2, e3) -> sprintf "Graph_Link(%s, %s, %s, WithEdge, %s)"
      (txt_of_expr e1) (txt_of_graph_op op1) (txt_of_expr e2) (txt_of_expr e3)
  | Id(x) -> sprintf "Id(%s)" x
  | Assign(e1, e2) -> sprintf "Assign(%s, %s)" e1 (txt_of_expr e2)
  | Noexpr -> sprintf "Noexpression"
  | ListP(l) -> sprintf "List(%s)" (txt_of_list l)
  | DictP(d) -> sprintf "Dict(%s)" (txt_of_dict d)
  | Call(f, args) -> sprintf "Call(%s, [%s])" (f) (txt_of_list args)
  | CallDefault(e, f, args) -> sprintf "CallDefault(%s, %s, [%s])" (txt_of_expr e) f
(txt_of_list args)

(*Variable Declaration*)
and txt_of_var_decl = function
  | Local(var, name, e1) -> sprintf "Local(%s, %s, %s)"
    (txt_of_var_type var) name (txt_of_expr e1)

(* Lists *)
and txt_of_list = function
  | [] -> ""
  | [x] -> txt_of_expr x
  | _ as l -> String.concat ", " (List.map txt_of_expr l)

(* Dict *)
and txt_of_dict_key_value = function
  | (key, value) -> sprintf "key:%s,value:%s" (txt_of_expr key) (txt_of_expr value)

and txt_of_dict = function
  | [] -> ""
  | [x] -> txt_of_dict_key_value x
  | _ as d -> String.concat ", " (List.map txt_of_dict_key_value d)

(* Functions Declaration *)
and txt_of_func_decl f =
  sprintf "returnType(%s) name(%s) args(%s) body{%s} locals{%s} parent(%s)"
```

```
    (txt_of_var_type f.returnType) f.name (txt_of_formal_list f.args) (txt_of_stmts f.body)
(txt_of_formal_list f.locals) f.pname

(* Statements *)
and txt_of_stmt = function
  | Expr(expr) -> sprintf "Expr(%s);" (txt_of_expr expr)
  | Return(expr) -> sprintf "Return(%s);" (txt_of_expr expr)
  | For(e1,e2,e3,s) -> sprintf "For(%s;%s;%s){%s}"
    (txt_of_expr e1) (txt_of_expr e2) (txt_of_expr e3) (txt_of_stmts s)
  | If(e1,s1,s2) -> sprintf "If(%s){%s} Else{%s}"
    (txt_of_expr e1) (txt_of_stmts s1) (txt_of_stmts s2)
  | While(e1, s) -> sprintf "While(%s){%s}"
    (txt_of_expr e1) (txt_of_stmts s)

and txt_of_stmts = function
  | [] -> ""
  | [x] -> txt_of_stmt x
  | _ as s -> String.concat ", " (List.map txt_of_stmt s)

and txt_of_funcs funcs =
  let rec aux acc = function
      | [] -> sprintf "%s" (String.concat "\n" (List.rev acc))
      | f :: tl -> aux (txt_of_func_decl f :: acc) tl
  in aux [] funcs

(* Program entry point *)
let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Organizer.convert (Parser.program Scanner.token lexbuf) in
  let result = txt_of_funcs program in
  print_endline result
```

9.12. compiler.Makefile

```
# OBJS = parser.cmo scanner.cmo semant.cmo

default: circline.native

.PHONY : circline.native

circline.native :
	ocamlbuild -use-ocamlfind -pkgs
llvm,llvm.analysis,llvm.linker,llvm.bitreader,llvm.irreader -cflags -w,+a-4 \
		circline.native;
	clang -emit-llvm -o utils.bc -c lib/utils.c -Wno-varargs

OBJS = organizer.cmx cast.cmx ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx
circline.cmx parserize.cmx

circline: $(OBJS)
	ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis $(OBJS) -o circline

all:
	cd ..; make all

scanner.ml : scanner.mll
	ocamllex scanner.mll
```

```
parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -c $<

%.cmi : %.mli
	ocamlc -c $<

%.cmx : %.ml
	ocamlfind ocamlopt -c -package llvm $<

.PHONY : clean
clean :
	ocamlbuild -clean
	rm -f utils.bc
	rm -f *.cmx *.cmi *.cmo *.cmx *.o
	rm -f circline parser.ml parser.mli scanner.ml *.cmo *.cmi

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and parser.ml
ast.cmo :
ast.cmx :
cast.cmo : ast.cmo
cast.cmx : ast.cmx
codegen.cmo : cast.cmo
codegen.cmx : cast.cmx
circline.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
circline.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo
parserize.cmx : ast.cmo
```

9.13. Cast.h

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

int32_t VoidtoInt(void* add);
double VoidtoFloat(void* add);
bool VoidtoBool(void* add);
char* VoidtoString(void* add);
struct Node* VoidtoNode(void* add);
struct Graph* VoidtoGraph(void* add);

void* InttoVoid(int32_t val);
void* FloattoVoid(double val);
void* BooltoVoid(bool val);
void* StringtoVoid(char* val);
```

```
void* NodetoVoid(struct Node* val);
void* GraphtoVoid(struct Graph* val);

bool isInt(int32_t d);
bool isFloat(int32_t d);
bool isBool(int32_t d);
bool isString(int32_t d);
bool isNode(int32_t d);
bool isGraph(int32_t d);
```

9.14. cast.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include <stdarg.h>
#include "cast.h"
#include "utils.h"

int32_t VoidtoInt(void* add){
       return *((int32_t*) add);
}

double VoidtoFloat(void* add){
       return *((double*) add);
}

bool VoidtoBool(void* add){
       return *((bool*) add);
}

char* VoidtoString(void* add){
       return (char*) add;
}

struct Node* VoidtoNode(void* add){
       return (struct Node*) add;
}

struct Graph* VoidtoGraph(void* add){
       return (struct Graph*) add;
}

void* InttoVoid(int32_t val){
       int* tmp = (int*)malloc(sizeof(int));
       *tmp = val;
       return (void*) tmp;
}

void* FloattoVoid(double val){
       double* tmp = (double*)malloc(sizeof(double));
       *tmp = val;
       return (void*) tmp;
}
```

```
void* BooltoVoid(bool val){
	bool* tmp = (bool*)malloc(sizeof(bool));
	*tmp = val;
	return (void*) tmp;
}

void* StringtoVoid(char* val){
	return (void*) val;
}

void* NodetoVoid(struct Node* val){
	return (void*) val;
}

void* GraphtoVoid(struct Graph* val){
	return (void*) val;
}

bool isInt(int32_t d){
	return (d==INT);
};
// bool isFloat(int32_t d){return (d==FLOAT);};
bool isFloat(int32_t d){return (d==INT);};
bool isBool(int32_t d){return (d==BOOL);};
bool isString(int32_t d){return (d==STRING);};
bool isNode(int32_t d){return (d==NODE);};
bool isGraph(int32_t d){return (d==GRAPH);};
```

9.15. Hashmap.h

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

#ifndef __HASHMAP_H__
#define __HASHMAP_H__

#define MAP_MISSING -3  /* No such element */
#define MAP_FULL -2   /* Hashmap is full */
#define MAP_OMEM -1   /* Out of Memory */
#define MAP_OK 0      /* OK */

struct hashmap_element{
	char* key;
	int in_use;
	void* data[2];
};

struct hashmap_map{
	int table_size;
	int size;
	int32_t keytype;
	int32_t valuetype;
	struct hashmap_element *data;
```

```
};


typedef int (*Func)(void*, void*, void*);


/*
 * Return an empty hashmap. Returns NULL if empty.
*/
extern struct hashmap_map* hashmap_new(int32_t keytyp,int32_t valuetyp);

/*
 * Iteratively call f with argument (item, data) for
 * each element data in the hashmap. The function must
 * return a map status code. If it returns anything other
 * than MAP_OK the traversal is terminated. f must
 * not reenter any hashmap functions, or deadlock may arise.
 */
extern int hashmap_iterate(struct hashmap_map* m, Func f);

extern int hashmap_print(struct hashmap_map* m);

extern bool hashmap_haskey(struct hashmap_map* m,...);

extern struct List* hashmap_keys(struct hashmap_map* m);
/*
 * Add an element to the hashmap. Return MAP_OK or MAP_OMEM.
 */
extern struct hashmap_map* hashmap_put(struct hashmap_map* m,...);

/*
 * Get an element from the hashmap. Return MAP_OK or MAP_MISSING.
 */
extern void* hashmap_get(struct hashmap_map* m,...);

/*
 * Remove an element from the hashmap. Return MAP_OK or MAP_MISSING.
 */
extern struct hashmap_map* hashmap_remove(struct hashmap_map* m,...);

/*
 * Free the hashmap
 */
extern void hashmap_free(struct hashmap_map* m);

/*
 * Get the current size of a hashmap
 */
extern int hashmap_length(struct hashmap_map* m);
extern int32_t hashmap_keytype(struct hashmap_map* m);
extern int32_t hashmap_valuetype(struct hashmap_map* m);

#endif //__HASHMAP_H__
```

9.16. Hashmap.c

```
/*
```

```
 * Generic map implementation.
 */
#include "hashmap.h"

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include <stdarg.h>
#include "utils.h"
#include "cast.h"
#include "list.h"

#define INITIAL_SIZE (256)
#define MAX_CHAIN_LENGTH (8)


struct hashmap_map* hashmap_new(int32_t keytyp,int32_t valuetyp) {
       struct hashmap_map* m = (struct hashmap_map*) malloc(sizeof(struct hashmap_map));
       m->data = (struct hashmap_element*) calloc(INITIAL_SIZE, sizeof(struct
hashmap_element));
       m->keytype = keytyp;
       m->valuetype = valuetyp;
       m->table_size = INITIAL_SIZE;
       m->size = 0;
       return m;
}


static unsigned long crc32_tab[] = {
      0x00000000L, 0x77073096L, 0xee0e612cL, 0x990951baL, 0x076dc419L,
      0x706af48fL, 0xe963a535L, 0x9e6495a3L, 0x0edb8832L, 0x79dcb8a4L,
      0xe0d5e91eL, 0x97d2d988L, 0x09b64c2bL, 0x7eb17cbdL, 0xe7b82d07L,
      0x90bf1d91L, 0x1db71064L, 0x6ab020f2L, 0xf3b97148L, 0x84be41deL,
      0x1adad47dL, 0x6ddde4ebL, 0xf4d4b551L, 0x83d385c7L, 0x136c9856L,
      0x646ba8c0L, 0xfd62f97aL, 0x8a65c9ecL, 0x14015c4fL, 0x63066cd9L,
      0xfa0f3d63L, 0x8d080df5L, 0x3b6e20c8L, 0x4c69105eL, 0xd56041e4L,
      0xa2677172L, 0x3c03e4d1L, 0x4b04d447L, 0xd20d85fdL, 0xa50ab56bL,
      0x35b5a8faL, 0x42b2986cL, 0xdbbbc9d6L, 0xacbcf940L, 0x32d86ce3L,
      0x45df5c75L, 0xdcd60dcfL, 0xabd13d59L, 0x26d930acL, 0x51de003aL,
      0xc8d75180L, 0xbfd06116L, 0x21b4f4b5L, 0x56b3c423L, 0xcfba9599L,
      0xb8bda50fL, 0x2802b89eL, 0x5f058808L, 0xc60cd9b2L, 0xb10be924L,
      0x2f6f7c87L, 0x58684c11L, 0xc1611dabL, 0xb6662d3dL, 0x76dc4190L,
      0x01db7106L, 0x98d220bcL, 0xefd5102aL, 0x71b18589L, 0x06b6b51fL,
      0x9fbfe4a5L, 0xe8b8d433L, 0x7807c9a2L, 0x0f00f934L, 0x9609a88eL,
      0xe10e9818L, 0x7f6a0dbbL, 0x086d3d2dL, 0x91646c97L, 0xe6635c01L,
      0x6b6b51f4L, 0x1c6c6162L, 0x856530d8L, 0xf262004eL, 0x6c0695edL,
      0x1b01a57bL, 0x8208f4c1L, 0xf50fc457L, 0x65b0d9c6L, 0x12b7e950L,
      0x8bbeb8eaL, 0xfcb9887cL, 0x62dd1ddfL, 0x15da2d49L, 0x8cd37cf3L,
      0xfbd44c65L, 0x4db26158L, 0x3ab551ceL, 0xa3bc0074L, 0xd4bb30e2L,
      0x4adfa541L, 0x3dd895d7L, 0xa4d1c46dL, 0xd3d6f4fbL, 0x4369e96aL,
      0x346ed9fcL, 0xad678846L, 0xda60b8d0L, 0x44042d73L, 0x33031de5L,
      0xaa0a4c5fL, 0xdd0d7cc9L, 0x5005713cL, 0x270241aaL, 0xbe0b1010L,
      0xc90c2086L, 0x5768b525L, 0x206f85b3L, 0xb966d409L, 0xce61e49fL,
      0x5edef90eL, 0x29d9c998L, 0xb0d09822L, 0xc7d7a8b4L, 0x59b33d17L,
      0x2eb40d81L, 0xb7bd5c3bL, 0xc0ba6cadL, 0xedb88320L, 0x9abfb3b6L,
      0x03b6e20cL, 0x74b1d29aL, 0xead54739L, 0x9dd277afL, 0x04db2615L,
      0x73dc1683L, 0xe3630b12L, 0x94643b84L, 0x0d6d6a3eL, 0x7a6a5aa8L,
      0xe40ecf0bL, 0x9309ff9dL, 0x0a00ae27L, 0x7d079eb1L, 0xf00f9344L,
      0x8708a3d2L, 0x1e01f268L, 0x6906c2feL, 0xf762575dL, 0x806567cbL,
```

```
      0x196c3671L, 0x6e6b06e7L, 0xfed41b76L, 0x89d32be0L, 0x10da7a5aL,
      0x67dd4accL, 0xf9b9df6fL, 0x8ebeeff9L, 0x17b7be43L, 0x60b08ed5L,
      0xd6d6a3e8L, 0xa1d1937eL, 0x38d8c2c4L, 0x4fdff252L, 0xd1bb67f1L,
      0xa6bc5767L, 0x3fb506ddL, 0x48b2364bL, 0xd80d2bdaL, 0xaf0a1b4cL,
      0x36034af6L, 0x41047a60L, 0xdf60efc3L, 0xa867df55L, 0x316e8eefL,
      0x4669be79L, 0xcb61b38cL, 0xbc66831aL, 0x256fd2a0L, 0x5268e236L,
      0xcc0c7795L, 0xbb0b4703L, 0x220216b9L, 0x5505262fL, 0xc5ba3bbeL,
      0xb2bd0b28L, 0x2bb45a92L, 0x5cb36a04L, 0xc2d7ffa7L, 0xb5d0cf31L,
      0x2cd99e8bL, 0x5bdeae1dL, 0x9b64c2b0L, 0xec63f226L, 0x756aa39cL,
      0x026d930aL, 0x9c0906a9L, 0xeb0e363fL, 0x72076785L, 0x05005713L,
      0x95bf4a82L, 0xe2b87a14L, 0x7bb12baeL, 0x0cb61b38L, 0x92d28e9bL,
      0xe5d5be0dL, 0x7cdcefb7L, 0x0bdbdf21L, 0x86d3d2d4L, 0xf1d4e242L,
      0x68ddb3f8L, 0x1fda836eL, 0x81be16cdL, 0xf6b9265bL, 0x6fb077e1L,
      0x18b74777L, 0x88085ae6L, 0xff0f6a70L, 0x66063bcaL, 0x11010b5cL,
      0x8f659effL, 0xf862ae69L, 0x616bffd3L, 0x166ccf45L, 0xa00ae278L,
      0xd70dd2eeL, 0x4e048354L, 0x3903b3c2L, 0xa7672661L, 0xd06016f7L,
      0x4969474dL, 0x3e6e77dbL, 0xaed16a4aL, 0xd9d65adcL, 0x40df0b66L,
      0x37d83bf0L, 0xa9bcae53L, 0xdebb9ec5L, 0x47b2cf7fL, 0x30b5ffe9L,
      0xbdbdf21cL, 0xcabac28aL, 0x53b39330L, 0x24b4a3a6L, 0xbad03605L,
      0xcdd70693L, 0x54de5729L, 0x23d967bfL, 0xb3667a2eL, 0xc4614ab8L,
      0x5d681b02L, 0x2a6f2b94L, 0xb40bbe37L, 0xc30c8ea1L, 0x5a05df1bL,
      0x2d02ef8dL
   };

/* Return a 32-bit CRC of the contents of the buffer. */

unsigned long crc32(const unsigned char *s, unsigned int len)
{
  unsigned int i;
  unsigned long crc32val;

  crc32val = 0;
  for (i = 0;  i < len;  i ++)
    {
      crc32val =
        crc32_tab[(crc32val ^ s[i]) & 0xff] ^
          (crc32val >> 8);
    }
  return crc32val;
}

/*
 * Hashing function for a string
 */
unsigned int hashmap_hash_int(struct hashmap_map * m, char* keystring){

    unsigned long key = crc32((unsigned char*)(keystring), strlen(keystring));

        /* Robert Jenkins' 32 bit Mix Function */
        key += (key << 12);
        key ^= (key >> 22);
        key += (key << 4);
        key ^= (key >> 9);
        key += (key << 10);
        key ^= (key >> 2);
        key += (key << 7);
        key ^= (key >> 12);

        /* Knuth's Multiplicative Method */
        key = (key >> 3) * 2654435761;
```

```
        return key % m->table_size;
}

/*
 * Return the integer of the location in data
 * to store the point to the item, or MAP_FULL.
 */
int hashmap_hash(struct hashmap_map* m, char* key){
        int curr;
        int i;

        /* If full, return immediately */
        if(m->size >= (m->table_size/2)) return MAP_FULL;

        /* Find the best index */
        curr = hashmap_hash_int(m, key);

        /* Linear probing */
        for(i = 0; i< MAX_CHAIN_LENGTH; i++){
                if(m->data[curr].in_use == 0)
                        return curr;

                if(m->data[curr].in_use == 1 && (strcmp(m->data[curr].key,key)==0))
                        return curr;

                curr = (curr + 1) % m->table_size;
        }

        return MAP_FULL;
}

/*
 * Doubles the size of the hashmap, and rehashes all the elements
 */
int hashmap_rehash(struct hashmap_map *m){
        int i;
        int old_size;
        struct hashmap_element* curr;

        /* Setup the new elements */
        struct hashmap_element* temp = (struct hashmap_element *)
                calloc(2 * m->table_size, sizeof(struct hashmap_element));
        if(!temp) return MAP_OMEM;

        /* Update the array */
        curr = m->data;
        m->data = temp;

        /* Update the size */
        old_size = m->table_size;
        m->table_size = 2 * m->table_size;
        m->size = 0;

        /* Rehash the elements */
        for(i = 0; i < old_size; i++){
         int status;

         if (curr[i].in_use == 0)
             continue;

                hashmap_put(m, curr[i].key, curr[i].data);
```

```
	}

	free(curr);

	return MAP_OK;
}

/*
 * Add a pointer to the hashmap with some key
 */
struct hashmap_map* hashmap_put(struct hashmap_map* m,...){
	int index;
	void* data1;
	void* data2;
	char* key;
	va_list ap;
	va_start(ap, 2);

	switch (m->keytype) {
		case INT:
			data1 = InttoVoid(va_arg(ap, int));
			key = malloc(16);
			snprintf(key, 16, "%d", VoidtoInt(data1));
			break;

		case STRING:
			data1 = StringtoVoid(va_arg(ap, char*));
			key = VoidtoString(data1);
			//printf("%s\n",key);
			break;

		case NODE:
			data1 = NodetoVoid(va_arg(ap, struct Node*));
			key = malloc(16);
			snprintf(key, 16, "%d", VoidtoNode(data1)->id);
			//printf("%s\n",key);
			break;

		default:
			break;
	}

	switch (m->valuetype) {
		case INT:
			data2 = InttoVoid(va_arg(ap, int));
			break;

		case FLOAT:
			data2 = FloattoVoid(va_arg(ap, double));
			break;

		case BOOL:
			data2 = BooltoVoid(va_arg(ap, double));
			break;

		case STRING:
			data2 = StringtoVoid(va_arg(ap, char*));
			break;

		case NODE:
			data2 = NodetoVoid(va_arg(ap, struct Node*));
```

```
                        break;

                case GRAPH:
                        data2 = GraphtoVoid(va_arg(ap, struct Graph*));
                        break;

                default:
                        break;
        }

        va_end(ap);

        /* Find a place to put our value */
        index = hashmap_hash(m, key);
        while(index == MAP_FULL){
                if (hashmap_rehash(m) == MAP_OMEM) {
                        printf("Error! Hashmap out of Memory\n");
                        exit(1);
                }
                index = hashmap_hash(m, key);
        }

        /* Set the data */
        m->data[index].data[0] = data1;
        m->data[index].data[1] = data2;
        m->data[index].key = key;
        m->data[index].in_use = 1;
        m->size++;

        return m;
}

/*
 * Get your pointer out of the hashmap with a key
 */
// int hashmap_get(map_t in, char* key, any_t *arg){
void* hashmap_get(struct hashmap_map* m,...){
        int curr;
        int i;
        char* key;
        va_list ap;
        va_start(ap, 1);

        switch (m->keytype) {
                case INT:
                        key = malloc(16);
                        snprintf(key, 16, "%d", va_arg(ap, int));
                        break;

                case STRING:
                        key = va_arg(ap, char*);
                        break;

                case NODE:
                        key = malloc(16);
                        snprintf(key, 16, "%d", va_arg(ap, struct Node*)->id);
                        break;

                default:
                        break;
        }
```

```
        va_end(ap);

        /* Find data location */
        curr = hashmap_hash_int(m, key);

        /* Linear probing, if necessary */
        for(i = 0; i<MAX_CHAIN_LENGTH; i++){

         int in_use = m->data[curr].in_use;
         if (in_use == 1){
             if (strcmp(m->data[curr].key,key)==0){
                 // *arg = (m->data[curr].data);
                 // return MAP_OK;
                 return m->data[curr].data[1];
             }
               }

               curr = (curr + 1) % m->table_size;
        }
        printf("Error! Hashmap_Get:Key not Exist!\n");
        exit(1);
}

/*
 * Iterate the function parameter over each element in the hashmap.  The
 * additional any_t argument is passed to the function as its first
 * argument and the hashmap element is the second.
 */
int hashmap_iterate(struct hashmap_map* m, Func f) {

        /* On empty hashmap, return immediately */
        if (hashmap_length(m) <= 0)
                return MAP_MISSING;

        /* Linear probing */
        for(int i = 0; i< m->table_size; i++)
                if(m->data[i].in_use != 0) {
                        int status = f(m->data[i].key, m->data[i].data[0],
m->data[i].data[1]);
                        if (status != MAP_OK) {
                                return status;
                        }
                }

    return MAP_OK;
}

int hashmap_printhelper(char* key, int32_t type, void* value){
        switch (type) {
                case INT:
                        printf("%s: %d",key, VoidtoInt(value));
                        break;

                case FLOAT:
                        printf("%s: %f",key, VoidtoFloat(value));
                        break;

                case BOOL:
                        printf("%s: %d",key, VoidtoBool(value));
                        break;
```

```
		case STRING:
			printf("%s: %s",key, VoidtoString(value));
			break;

		case NODE:
			printf("%s: ",key);
			printNode(VoidtoNode(value));
			break;

		case GRAPH:
			printf("%s: ",key);
			printGraph(VoidtoGraph(value));
			break;

		default:
			break;
	}
	return 0;
}

int hashmap_print(struct hashmap_map* m){
	if (m == NULL) {
		printf("(null)\n");
		return 0;
	}
	printf("{");
	for(int i = 0, c=0; i< m->table_size; i++)
		if(m->data[i].in_use != 0) {
			hashmap_printhelper(m->data[i].key, m->valuetype, m->data[i].data[1]);
			c++;
			if (c < m->size) printf(", ");
		}
	printf("}\n");
	return 0;
}

bool hashmap_haskey(struct hashmap_map* m,...){
	int curr;
	int i;
	char* key;
	va_list ap;
	va_start(ap, 1);

	switch (m->keytype) {
		case INT:
			key = malloc(16);
			snprintf(key, 16, "%d", va_arg(ap, int));
			break;

		case STRING:
			key = va_arg(ap, char*);
			break;

		case NODE:
			key = malloc(16);
			snprintf(key, 16, "%d", va_arg(ap, struct Node*)->id);
			break;

		default:
			break;
	}
```

```
        va_end(ap);
        /* Find data location */
        curr = hashmap_hash_int(m, key);
        /* Linear probing, if necessary */
        for(i = 0; i<MAX_CHAIN_LENGTH; i++){
         int in_use = m->data[curr].in_use;
         if (in_use == 1){
             if (strcmp(m->data[curr].key,key)==0){
                 // *arg = (m->data[curr].data);
                 // return MAP_OK;
                 return 1;
             }
                }
                curr = (curr + 1) % m->table_size;
        }
        return 0;
}

struct List* hashmap_keys(struct hashmap_map* m){
        if (hashmap_length(m) <= 0){
                printf("Error! hashmap_getkey: No keys!\n");
                exit(1);
        }
        struct List* dataset = createList(m->keytype);
        for(int i = 0; i< m->table_size; i++){
                if(m->data[i].in_use != 0) {
                        switch (m->keytype) {
                                case INT:
                                        addList(dataset, VoidtoInt(m->data[i].data[0]));
                                        break;

                                case STRING:
                                        addList(dataset, VoidtoString(m->data[i].data[0]));
                                        break;

                                case NODE:
                                        addList(dataset, VoidtoNode(m->data[i].data[0]));
                                        break;

                                default:
                                        break;
        }
                }
        }
        return dataset;
}

// /*
//  * Remove an element with that key from the map
//  */
struct hashmap_map* hashmap_remove(struct hashmap_map* m,...){
        int i;
        int curr;
        char* key;
        va_list ap;
        va_start(ap, 1);

        switch (m->keytype) {
                case INT:
                        key = malloc(16);
                        snprintf(key, 16, "%d", va_arg(ap, int));
```

```
                    break;

            case STRING:
                    key = va_arg(ap, char*);
                    break;

            case NODE:
                    key = malloc(16);
                    snprintf(key, 16, "%d", va_arg(ap, struct Node*)->id);
                    break;

            default:
                    break;
        }
        va_end(ap);

        /* Find key */
        curr = hashmap_hash_int(m, key);

        /* Linear probing, if necessary */
        for(i = 0; i<MAX_CHAIN_LENGTH; i++){

         int in_use = m->data[curr].in_use;
         if (in_use == 1){
             if (strcmp(m->data[curr].key,key)==0){
                 /* Blank out the fields */
                 m->data[curr].in_use = 0;
                 m->data[curr].data[0] = NULL;
                 m->data[curr].data[1] = NULL;
                 m->data[curr].key = NULL;

                 /* Reduce the size */
                 m->size--;
                 return m;
             }
               }
               curr = (curr + 1) % m->table_size;
        }
        printf("Error! hashmap_remove: Missing data!\n");
        exit(1);
}

// /* Deallocate the hashmap */
// void hashmap_free(map_t in){
//      hashmap_map* m = (hashmap_map*) in;
//      free(m->data);
//      free(m);
// }

// /* Return the length of the hashmap */
int hashmap_length(struct hashmap_map* m){
        if(m != NULL) return m->size;
        else return 0;
}

int32_t hashmap_keytype(struct hashmap_map* m){
        return m->keytype;
}

int32_t hashmap_valuetype(struct hashmap_map* m){
        return m->valuetype;
```

```
}
// int hashmap_print(void* a, void* data1, void* data2){
//     printf("data1: %d\n", *((int*) data1));
//     printf("data2: %s\n", data2);
//     return 0;
// }

// int main(){
//     struct hashmap_map* mymap = hashmap_new(INT, STRING);
//     hashmap_put(mymap, 10, "Hello World");
//     hashmap_put(mymap, 20, "Hello World1");
//     hashmap_put(mymap, 30, "Hello World2");
//     hashmap_print(mymap);
//     printList(hashmap_keys(mymap));
//     //hashmap_iterate(mymap, hashmap_print, 0);
//     //hashmap_remove(mymap, 10);
//     //printf("%s\n", VoidtoString((hashmap_get(mymap, 10))));
//     return 0;
// }
```

9.17. List.h

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

#ifndef _LIST_H_
#define _LIST_H_

// one element of a list.
struct List {
       int32_t type;
       int32_t size;
       void* *arr;
       int32_t curPos;
};

int32_t rangeHelper(int size, int index);
struct List* createList( int32_t type);
struct List* addListHelper( struct List * list, void* addData);
struct List* concatList(struct List* list1, struct List* list2);
struct List* pushList(struct List* list, ...);
struct List* addList(struct List* list, ...);
void* getList(struct List* list, int index);
void* popList(struct List* list);
int32_t setList(struct List* list, int index, ...);
int getListSize(struct List* list);
int32_t removeList(struct List* list, int index);
int32_t pirntList(struct List * list);

#endif
```

9.18. List.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include <stdarg.h>
#include "utils.h"
#include "cast.h"
#include "list.h"

struct List* createList(
	int32_t type
) {
	struct List* new = (struct List*) malloc(sizeof(struct List));
	// default initialize size is 1
	new->size = 1;
	new->type = type;
	// means that the next element would be added at curPos
	new->curPos = 0;
	new->arr = (void**)malloc(new->size * sizeof(void*));
	return new;
}

int rangeHelper(int size, int index){
	if(size <= -index || size <= index || size == 0){
		printf("Error! Index out of Range!\n");
		exit(1);
	}
	if (index < 0){
		index += size;
	}
	return index;
}

struct List* addListHelper(
	struct List * list,
	void* addData
){
	if (list->curPos >= list->size){
		list->size = list->size * 2;
		// double size
		list->arr = (void**) realloc(list->arr, list->size * sizeof (void*));
	}
	*(list->arr + list->curPos) = addData;
	list->curPos++;
	return list;
}

struct List* concatList(struct List* list1, struct List* list2){
	int curPos = list2->curPos;
	for(int i =0; i < curPos; i++){
		list1 = addListHelper(list1, *(list2->arr+i));
	}
	return list1;
}

struct List* addList(struct List* list, ...) {
	if (list == NULL) {
```

```
		printf("[Error] addList() - List doesn't exist. \n");
		exit(1);
	}
	va_list ap;
	va_start(ap, 1);
	void * data;
	int* tmp0;
	double* tmp1;
	bool* tmp2;
	switch (list->type) {
		case INT:
			data = InttoVoid(va_arg(ap, int));
			break;

		case FLOAT:
			data = FloattoVoid(va_arg(ap, double));
			break;

		case BOOL:
			data = BooltoVoid(va_arg(ap, bool));
			break;

		case STRING:
			data = StringtoVoid(va_arg(ap, char*));
			break;

		case NODE:
			data = NodetoVoid(va_arg(ap, struct Node*));
			break;

		case GRAPH:
			data = GraphtoVoid(va_arg(ap, struct Graph*));
			break;

		default:
			break;
	}
  va_end(ap);
  return addListHelper(list, data);
}

void* getList(struct List* list, int index){
	if (list == NULL) {
		printf("[Error] getList() - List doesn't exist. \n");
		exit(1);
	}
	index = rangeHelper(list->curPos, index);
	return *(list->arr + index);
}

void* popList(struct List* list){
	if (list == NULL) {
		printf("[Error] popList() - List doesn't exist. \n");
		exit(1);
	}
	if(list->curPos-1 < 0){
		printf("Error! Nothing Can be poped T.T\n");
		exit(1);
	}
	void* add = *(list->arr + list->curPos-1);
	list->curPos--;
```

```
        return add;
}

int32_t setList(struct List* list, int index, ...){
        if (list == NULL) {
                printf("[Error] setList() - List doesn't exist. \n");
                exit(1);
        }
        index = rangeHelper(list->curPos, index);
        va_list ap;
        va_start(ap, 1);
        void * data;
        int* tmp0;
        double* tmp1;
        bool* tmp2;
        switch (list->type) {
                case INT:
                        data = InttoVoid(va_arg(ap, int));
                        break;

                case FLOAT:
                        data = FloattoVoid(va_arg(ap, double));
                        break;

                case BOOL:
                        data = BooltoVoid(va_arg(ap, bool));
                        break;

                case STRING:
                        data = StringtoVoid(va_arg(ap, char*));
                        break;

                case NODE:
                        data = NodetoVoid(va_arg(ap, struct Node*));
                        break;

                case GRAPH:
                        data = GraphtoVoid(va_arg(ap, struct Graph*));
                        break;

                default:
                        break;
        }
        *(list->arr + index) = data;
        return 0;
}

int getListSize(struct List* list){
        if (list == NULL) {
                printf("[Error] getListSize() - List doesn't exist. \n");
                exit(1);
        }
        return list->curPos;
}

int32_t removeList(struct List* list, int index){
        if (list == NULL) {
                printf("[Error] removelist() - List doesn't exist. \n");
                exit(1);
        }
        index =rangeHelper(list->curPos, index);
```

```
        for(int i=index; i < list->curPos; i++){
                *(list->arr + i) = *(list->arr + i+1);
        }
        list->curPos--;
        return 0;
}

int32_t printList(struct List * list){
        if (list == NULL) {
                printf("(null)\n");
                return 0;
        }
        int curPos = list->curPos - 1;
        if (curPos < 0) {
                printf("list:[]\n");
                return 0;
        }
        int p = 0;
        printf("list:[");
        switch (list->type) {
                case INT:
                        while(p < curPos){
                                printf("%d, ", *((int*)(*(list->arr + p))));
                                p++;
                        }
                        printf("%d", *((int*)(*(list->arr + p))));
                        break;

                case FLOAT:
                        while(p < curPos){
                                printf("%f, ", *((double*)(*(list->arr + p))));
                                p++;
                        }
                        printf("%f", *((double*)(*(list->arr + p))));
                        break;

                case BOOL:
                        while(p < curPos){
                                printf("%s, ", *((bool*)(*(list->arr + p))) ? "true" :
"false");
                                p++;
                        }
                        printf("%s", *((bool*)(*(list->arr + p))) ? "true" : "false");
                        break;

                case STRING:
                        while(p < curPos){
                                printf("%s, ", ((char*)(*(list->arr + p))));
                                p++;
                        }
                        printf("%s", ((char*)(*(list->arr + p))));
                        break;

                case NODE:
                        while(p < curPos){
                                printNode((struct Node*)(*(list->arr + p)));
                                p++;
                        }
                        printNode((struct Node*)(*(list->arr + p)));
                        break;
```

```
		case GRAPH:
			while(p < curPos){
				printGraph((struct Graph*)(*(list->arr + p)));
				p++;
			}
			printGraph((struct Graph*)(*(list->arr + p)));
			break;

		default:
			printf("Unsupported List Type!\n");
			return 1;
	}
	printf("]\n");
	return 0;
	// int p = 0;
	// printf("list:[");
	// while(p < curPos){
	//     printf(fmt, *(list->arr + p));
	//     printf(", ");
	//     p++;
	// }
	// printf(fmt, *(list->arr + curPos));
	// printf("]\n");
	// return 1;
}



// int main() {
//     struct List* a = createList(INT);
//     addList(a, 10);
//     addList(a, 5);
//     addList(a, 7);
//     addList(a, 9);
//     setList(a, 0, 3);
//     a = concatList(a, a);
//     removeList(a, 0);

//     printList(a);
//     //printNode(VoidtoNode(getList(a,2)));
// }
```

9.19. Utils.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include "list.h"

#ifndef _UTILS_H_
#define _UTILS_H_

int32_t printBool(bool a);

/************************************
```

```
	Type Declaration
***********************************/

#define INT 0
#define FLOAT 1
#define BOOL 2
#define STRING 3
#define NODE 4
#define GRAPH 5
#define LIST 6
#define DICT 7
#define EDGE 8

#define RIGHT_LINK 0
#define LEFT_LINK 1
#define DOUBLE_LINK 2


struct Node {
	int32_t id;
	int32_t type;
	int32_t a;
	double b;
	bool c;
	char* d;
};

struct Edge {
	struct Node* sour;
	struct Node* dest;
	int32_t type;
	int32_t a;
	double b;
	bool c;
	char* d;
};

struct Graph {
	int32_t vn;
	int32_t en;
	int32_t vn_len;
	int32_t en_len;
	struct Node* root;
	struct Node** nodes;
	struct Edge* edges;
};



/***********************************
	Node Methods
***********************************/

struct Node* createNode(int32_t id, int32_t type, ...);

void* nodeGetValue(struct Node* node, int32_t type);
int32_t printNode(struct Node * node);

/***********************************
	Edge Methods
***********************************/
```

```
struct Edge createEdge(
	struct Node* sour,
	struct Node* dest,
	int32_t type,
	int32_t a,
	double b,
	bool c,
	char* d
);

int32_t printEdge(struct Edge * edge);
int32_t printEdgeValue(struct Edge * edge);
void* edgeGetValue(struct Edge* edge, int32_t type);

/***********************************
	Graph Methods
***********************************/

struct Graph* createGraph();
struct Graph* copyGraph(struct Graph* a);
struct Graph* mergeGraph(struct Graph* a, struct Graph* b);
struct List* subGraph(struct Graph* a, struct Graph* b);
struct Node* graphGetRoot(struct Graph* g);
int32_t graphSetRoot(struct Graph* g, struct Node * root);
int32_t graphAddList(struct Graph* g, int direction, struct List * l, struct List * edges);
int32_t graphAddNode(struct Graph* g, struct Node * node);
struct List* graphGetAllNodes(struct Graph* g);
struct List* graphRemoveNode(struct Graph* g, struct Node * node);
int32_t graphAddEdgeP( struct Graph* g, struct Node* sour, struct Node* dest, int32_t
type, ...);
int32_t graphAddEdge(
	struct Graph* g,
	struct Node* sour,
	struct Node* dest,
	int32_t type,
	int32_t a,
	double b,
	bool c,
	char* d
);
bool graphEdgeExist(struct Graph* g, struct Node* sour, struct Node* dest);
struct Edge* graphGetEdge(struct Graph* g, struct Node* sour, struct Node* dest);
int32_t graphNumOfNodes(struct Graph* g);
int32_t graphNumOfEdges(struct Graph* g);
struct List* graphGetChildNodes(struct Graph* g, struct Node* rt);
int32_t printGraph(struct Graph* g);

#endif /* #ifndef _UTILS_H_ */
```

9.20. Utils.c

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

```
#include <stdarg.h>
#include "utils.h"
#include "hashmap.c"
#include "list.c"
#include "cast.c"

int32_t printBool(bool a) {
	printf("%s\n", a ? "true" : "false");
	return 0;
}

/************************************
	Node Methods
************************************/

struct Node* createNode(int32_t id, int32_t type, ...) {
	struct Node* new = (struct Node*) malloc(sizeof(struct Node));
	new->id = id;
	new->type = type;

	va_list ap;
	va_start(ap, 1);
	switch (type) {
		case INT:
			new->a = va_arg(ap, int);    break;
		case FLOAT:
			new->b = va_arg(ap, double); break;
		case BOOL:
			new->c = va_arg(ap, bool);   break;
		case STRING:
			new->d = va_arg(ap, char*);  break;
		default:
			break;
	}
  va_end(ap);
  return new;
}

void* nodeGetValue(struct Node* node, int32_t type) {
		if (node == NULL) {
			printf("[Error] Node doesn't exist!\n");
			exit(1);
		}
		void* res;
		switch (type) {
			case INT:
				if (node->type == INT)
					res = InttoVoid(node->a);
				else if (node->type == FLOAT)
					res = InttoVoid((int)node->b);
				else if (node->type == BOOL)
					res = InttoVoid( node->c ? 1 : 0 );
				else {
					res = InttoVoid(0);
				}
				break;
			case FLOAT:
				if (node->type == INT)
					res = FloattoVoid((double)node->a);
				else if (node->type == FLOAT)
					res = FloattoVoid(node->b);
```

```
                        else if (node->type == BOOL)
                                res = FloattoVoid( node->c ? 1 : 0 );
                        else {
                                res = FloattoVoid(0);
                        }
                        break;
                case BOOL:
                        if (node->type == INT)
                                res = BooltoVoid(node->a != 0);
                        else if (node->type == FLOAT)
                                res = BooltoVoid(node->b != 0);
                        else if (node->type == BOOL)
                                res = BooltoVoid(node->c);
                        else {
                                res = BooltoVoid(false);
                        }
                        break;
                case STRING:
                        if (node->type == STRING)
                                res = StringtoVoid(node->d);
                        else{
                                res = StringtoVoid("");
                        }
                        break;
                default:
                        printf("[Error] Edge Value Type Error!\n");
                        exit(1);
                        break;
        }
        return res;
}

int32_t printNode(struct Node * node) {
        if (node == NULL) {
                printf("(null)\n");
                return 0;
        }
        switch (node->type) {
                case 0:
                        printf("node%3d: %d\n", node->id, node->a);
                        break;
                case 1:
                        printf("node%3d: %f\n", node->id, node->b);
                        break;
                case 2:
                        printf("node%3d: %s\n", node->id, node->c ? "true" : "false");
                        break;
                case 3:
                        printf("node%3d: %s\n", node->id, node->d);
                        break;
                default:
                        printf("node%3d\n", node->id);
                        break;
        }
        return 0;
}

/***********************************
        Edge Methods
***********************************/
```

```
struct Edge createEdge(
	struct Node* sour,
	struct Node* dest,
	int32_t type,
	int32_t a,
	double b,
	bool c,
	char* d
) {
	struct Edge e = {sour, dest, type, a, b, c, d};
	return e;
}

void* edgeGetValue(struct Edge* edge, int32_t type) {
	if (edge == NULL) {
		printf("[Error] Edge doesn't exist!\n");
		exit(1);
	}
	void* res;
	switch (type) {
		case INT:
			if (edge->type == INT)
				res = InttoVoid(edge->a);
			else if (edge->type == FLOAT)
				res = InttoVoid((int)edge->b);
			else if (edge->type == BOOL)
				res = InttoVoid( edge->c ? 1 : 0 );
			else {
				res = InttoVoid(0);
			}
			break;
		case FLOAT:
			if (edge->type == INT)
				res = FloattoVoid((double)edge->a);
			else if (edge->type == FLOAT)
				res = FloattoVoid(edge->b);
			else if (edge->type == BOOL)
				res = FloattoVoid( edge->c ? 1 : 0 );
			else {
				res = FloattoVoid(0);
			}
			break;
		case BOOL:
			if (edge->type == INT)
				res = BooltoVoid(edge->a != 0);
			else if (edge->type == FLOAT)
				res = BooltoVoid(edge->b != 0);
			else if (edge->type == BOOL)
				res = BooltoVoid(edge->c);
			else {
				res = BooltoVoid(false);
			}
			break;
		case STRING:
			if (edge->type == STRING)
				res = StringtoVoid(edge->d);
			else {
				res = StringtoVoid("");
			}
			break;
		default:
```

```
                    printf("[Error] Edge Value Type Error!\n");
                    exit(1);
                    break;
        }
        return res;
}

int32_t printEdge(struct Edge * edge) {
        if (edge == NULL) {
                printf("(null)\n");
                return 0;
        }
        switch (edge->type) {
                case 0:
                        printf("edge%3d->%3d: %d\n", edge->sour->id, edge->dest->id, edge->a);
                        break;
                case 1:
                        printf("edge%3d->%3d: %f\n", edge->sour->id, edge->dest->id, edge->b);
                        break;
                case 2:
                        printf("node%3d->%3d: %s\n", edge->sour->id, edge->dest->id, edge->c ?
"true" : "false");
                        break;
                case 3:
                        printf("edge%3d->%3d: %s\n", edge->sour->id, edge->dest->id, edge->d);
                        break;
                default:
                        printf("edge%3d->%3d\n", edge->sour->id, edge->dest->id);
                        break;
        }
        return 0;
}

int32_t printEdgeValue(struct Edge * edge) {
        if (edge == NULL) {
                printf("(null)\n");
                return 0;
        }
        switch (edge->type) {
                case 0:
                        printf("%d\n", edge->a);
                        break;
                case 1:
                        printf("%f\n", edge->b);
                        break;
                case 2:
                        printf("%s\n", edge->c ? "true" : "false");
                        break;
                case 3:
                        printf("%s\n", edge->d);
                        break;
                default:
                        printf("[Error] Unknown Edge Value Type!\n");
                        exit(1);
                        break;
        }
        return 0;
}

/************************************
        Graph Methods
```

```
************************************/

int32_t graphAddEdgeHelper(struct Graph* g, struct Edge e) {
	if (g == NULL) exit(1);
	int i;
	for (i=0; i < g->en; i++) {
		if (g->edges[i].sour == e.sour && g->edges[i].dest == e.dest) {
			g->edges[i] = e;
			return 0;
		}
	}
	g->edges[i] = e;
	g->en ++;
	return 0;
}

int32_t graphAddEdgeP( struct Graph* g, struct Node* sour, struct Node* dest, int32_t
type, ...) {
	if (g == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	if (sour == dest) return 0;
	if (g->en + 1 >= g->en_len) {
		printf("[Error] # Graph Edges reach the limit!\n");
		exit(1);
	}
	if (graphAddNode(g, sour) > 0) exit(1);
	if (graphAddNode(g, dest) > 0) exit(1);

	// Assign the Edge Value
	struct Edge e = createEdge(sour, dest, type, 0, 0, 0, NULL);
	va_list ap;
	va_start(ap, 1);
	void* tmp = va_arg(ap, void*);
	switch (type) {
		case INT:
			e.a = *((int*)tmp);   break;
		case FLOAT:
			e.b = *((double*)tmp);       break;
		case BOOL:
			e.c = *((bool*)tmp);  break;
		case STRING:
			e.d = ((char*)tmp);   break;
		default:
			break;
	}
  va_end(ap);

	int i;
	// Edges already exist in the graph
	for (i=0; i<g->en; i++) {
		if (g->edges[i].sour == sour && g->edges[i].dest == dest) {
			g->edges[i] = e;
			return 0;
		}
	}
	g->edges[i] = e;
	g->en++;
	return 0;
}
```

```
int32_t graphAddEdge(
	struct Graph* g,
	struct Node* sour,
	struct Node* dest,
	int32_t type,
	int32_t a,
	double b,
	bool c,
	char* d
) {
	if (g == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	if (sour == dest) return 0;
	if (g->en + 1 >= g->en_len) {
		printf("[Error] # Graph Edges reach the limit!\n");
		exit(1);
	}
	if (graphAddNode(g, sour) > 0) exit(1);
	if (graphAddNode(g, dest) > 0) exit(1);
	int i;
	// Edges already exist in the graph
	for (i=0; i<g->en; i++) {
		if (g->edges[i].sour == sour && g->edges[i].dest == dest) {
			g->edges[i].type = type;
			g->edges[i].a = a;
			g->edges[i].b = b;
			g->edges[i].c = c;
			g->edges[i].d = d;
			return 0;
		}
	}
	struct Edge e = createEdge(sour, dest, type, a, b, c, d);
	g->edges[i] = e;
	g->en++;
	return 0;
}

bool graphEdgeExist(struct Graph* g, struct Node* sour, struct Node* dest) {
	if (g == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	int i;
	for (i=0; i<g->en; i++) {
		if (g->edges[i].sour == sour && g->edges[i].dest == dest) {
			return true;
		}
	}
	return false;
}
struct Edge* graphGetEdge(struct Graph* g, struct Node* sour, struct Node* dest) {
	if (g == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	int i;
	for (i=0; i<g->en; i++) {
		if (g->edges[i].sour == sour && g->edges[i].dest == dest) {
```

```
                      return &g->edges[i];
               }
       }
       return NULL;
}

/*
       Split the graph into a list of graphs, in which all graphs are connected.
 */
struct List* splitGraph(struct Graph * gh) {
       struct List* l = createList(GRAPH);
       if (gh == NULL) return l;

       gh = copyGraph(gh);
       struct Node* root = gh->root;
       struct Graph* gh_tmp = NULL;
       int vn = gh->vn, en = gh->en, max_vn = gh->vn, max_en = gh->en;
       int i, j, k;
       struct List* queue = createList(NODE);
       struct Node* node = NULL, *node_tmp = NULL;

       while (vn > 0) {
               gh_tmp = createGraph();
               for (i=0; i<max_vn; i++) {
                      if (gh->nodes[i] != NULL) break;
               }
               addList(queue, gh->nodes[i]);
               while (getListSize(queue) > 0) {
                      node = (struct Node*) getList(queue, 0);
                      removeList(queue, 0);
                      graphAddNode(gh_tmp, node);
                      for (k=0; k<max_vn; k++) {
                             if (gh->nodes[k] == node) {
                                    gh->nodes[k] = NULL;
                                    vn--;
                                    break;
                             }
                      }
                      if (k == max_vn) continue;
                      for (j=0; j<max_en; j++) {
                             if (gh->edges[j].type != -9 && gh->edges[j].sour == node) {
                                    node_tmp = gh->edges[j].dest;
                             } else if (gh->edges[j].type != -9 && gh->edges[j].dest ==
node) {
                                    node_tmp = gh->edges[j].sour;
                             } else {
                                    node_tmp = NULL;
                             }
                             if (node_tmp == NULL ) continue;
                             addList(queue, node_tmp);
                             graphAddEdgeHelper(gh_tmp, gh->edges[j]);
                             gh->edges[j].type = -9;
                      }
               }
               // Adjust the root to the original one
               bool hasRoot = false;
               for (i=0; i<gh_tmp->vn; i++) {
                      if (gh_tmp->nodes[i] == root) {
                             gh_tmp -> root = root;
                             hasRoot = true;
                             break;
```

```
                    }
            }
            // Make sure the subgrpah with original root is the first in the list
            if (hasRoot && getListSize(l) > 0) {
                    addList(l, (struct Graph*)getList(l, 0));
                    setList(l, 0, gh_tmp);
            } else {
                    addList(l, gh_tmp);
            }
    }
    free(gh);
    return l;
}

struct Graph* createGraph() {
    struct Graph* g = (struct Graph*) malloc( sizeof(struct Graph) );
    g->vn = 0;
    g->en = 0;
    g->vn_len = 32;
    g->en_len = 128;
    g->root = NULL;
    g->nodes = (struct Node**) malloc( sizeof(struct Node*) * 16 );
    g->edges = (struct Edge*) malloc( sizeof(struct Edge) * 64 );
    return g;
}
struct Graph* copyGraph(struct Graph* a) {
    if (a == NULL) return NULL;
    struct Graph* g = (struct Graph*) malloc( sizeof(struct Graph) );
    memcpy(g, a, sizeof(struct Graph));
    g->nodes = (struct Node**) malloc( sizeof(struct Node*) * a->vn_len );
    g->edges = (struct Edge*) malloc( sizeof(struct Edge) * a->en_len );
    int i;
    for (i=0; i<a->vn; i++) {
            g->nodes[i] = a->nodes[i];
    }
    struct Edge* tmp;
    for (i=0; i<a->en; i++) {
            tmp = (struct Edge*) malloc( sizeof(struct Edge) );
            memcpy(tmp, &a->edges[i], sizeof(struct Edge));
            g->edges[i] = *tmp;
    }
    return g;
}

struct Graph* mergeGraph(struct Graph* a, struct Graph* b) {
    if (b == NULL) return copyGraph(a);
    if (a == NULL) return copyGraph(b);

    struct Graph* gh = copyGraph(a);
    // Check whether two graph have shared nodes
    int i; int j;
    bool hasShared = false;
    for (i=0; i < a->vn; i++) {
            for (j=0; j < b->vn; j++) {
                    if (a->nodes[i] == b->nodes[j]) {
                            hasShared = true;
                            break;
                    }
            }
            if (hasShared) break;
    }
```

```
	if (!hasShared) return gh; // Return the copy of graph a

	for (i=0; i< b->vn; i++) {
		graphAddNode(gh, b->nodes[i]);
	}
	for (i=0; i< b->en; i++) {
		graphAddEdgeHelper(gh, b->edges[i]);
	}
	return gh;
}
struct Node* graphGetRoot(struct Graph* g) {
	if (g == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	return g->root;
}
int32_t graphSetRoot(struct Graph* g, struct Node * root) {
	if (g == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	if (root == NULL) {
		printf("[Error] Root node doesn't exist!\n");
		exit(1);
	}
	int i;
	for (i=0; i<g->vn; i++) {
		if (g->nodes[i] == root) {
			g->root = root;
			return 0;
		}
	}
	printf("[Error] Root doesn't exist in the graph!\n");
	exit(1);
}

struct List* subGraph(struct Graph* a, struct Graph* b) {
	if (a == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	struct Graph* gh = copyGraph(a);
	if (b == NULL || b->en <= 0) {
		struct List* l = createList(GRAPH);
		addList(l, gh);
		return l;
	}
	int i, j, k;
	for (i = 0; i < b->en; i++) {
		struct Edge e = b->edges[i];
		for (j = 0; j < gh->en; j++) {
			if (gh->edges[j].sour == e.sour && gh->edges[j].dest == e.dest) {
				gh->edges[j] = gh->edges[gh->en-1];
				gh->en --;
				break;
			}
		}
	}
	return splitGraph(gh);
}
```

```
int32_t graphAddList(struct Graph* g, int direction, struct List * l, struct List * edges) {
	if (g == NULL || g->root == NULL || l == NULL) {
		printf("[Error] Graph or List doesn't exist!\n");
		exit(1);
	}
	int i, j, lsize = l->curPos, rsize = edges == NULL ? 0 : edges->curPos;
	if (lsize != rsize && rsize > 1) {
		printf("[Error] Edge List Not Compatible with Node/Graph List!\n");
		exit(1);
	}
	for (i=0; i<lsize; i++) {
		struct Node * node = NULL;
		if (l->type == GRAPH) {
			// Merge the graph
			struct Graph * gh_tmp = (struct Graph*)l->arr[i];
			node = gh_tmp->root;
			for (j=0; j< gh_tmp->vn; j++) {
				graphAddNode(g, gh_tmp->nodes[j]);
			}
			for (j=0; j< gh_tmp->en; j++) {
				graphAddEdgeHelper(g, gh_tmp->edges[j]);
			}
		} else if (l->type == NODE) {
			node = (struct Node*)l->arr[i];
		} else {
			printf("[Error] GraphAddList List Type doesn't supported!!\n");
			exit(1);
		}

		if (node == NULL) continue;
		if (edges != NULL && edges->curPos > 0) {
			int edgePos = edges->curPos == 1 ? 0 : i;
			switch (direction) {
				case RIGHT_LINK:
					graphAddEdgeP(g, g->root, node, edges->type,
edges->arr[edgePos]); break;
				case LEFT_LINK:
					graphAddEdgeP(g, node, g->root, edges->type,
edges->arr[edgePos]); break;
				case DOUBLE_LINK:
					graphAddEdgeP(g, g->root, node, edges->type,
edges->arr[edgePos]);
					graphAddEdgeP(g, node, g->root, edges->type,
edges->arr[edgePos]);
					break;
				default:
					break;
			}
		} else {
			switch (direction) {
				case RIGHT_LINK:
					graphAddEdgeP(g, g->root, node, -1, NULL); break;
				case LEFT_LINK:
					graphAddEdgeP(g, node, g->root, -1, NULL); break;
				case DOUBLE_LINK:
					graphAddEdgeP(g, g->root, node, -1, NULL);
					graphAddEdgeP(g, node, g->root, -1, NULL);
					break;
				default:
					break;
```

```
                        }
                }
        }
        return 0;
}

int32_t graphAddNode(struct Graph* g, struct Node * node) {
        if (g == NULL) {
                printf("[Error] Graph doesn't exist!\n");
                exit(1);
        }
        if (g->vn + 1 >= g->vn_len) {
                printf("[Warning] # Graph Nodes reach the limit!\n");
                exit(1);
        }
        int i;
        // Nodes already exist in the graph
        for (i=0; i<g->vn; i++) {
                if (g->nodes[i] == node) return 0;
        }
        // Update the root if the graph is empty
        if (g->root == NULL) {
                g->root = node;
        }
        g->nodes[i] = node;
        g->vn++;
        return 0;
}

struct List* graphRemoveNode(struct Graph* gh, struct Node * node) {
        if (gh == NULL) {
                printf("[Error] Graph doesn't exist!\n");
                exit(1);
        }
        gh = copyGraph(gh);
        int i, j;
        // Remove Node
        for (i=0; i<gh->vn; i++) {
                if (gh->nodes[i] == node) {
                        for (j=i; j<gh->vn-1; j++) {
                                gh->nodes[j] = gh->nodes[j+1];
                        }
                        gh->nodes[j] = NULL;
                        gh->vn--;
                }
        }
        if (gh->root == node) {
                gh->root = gh->vn == 0 ? NULL : gh->nodes[0];
        }
        // Remove Edges
        for (i=0, j=gh->en-1; i<=j;) {
                if (gh->edges[i].sour == node || gh->edges[i].dest == node) {
                        gh->edges[i] = gh->edges[j];
                        gh->en--;
                        j--;
                } else {
                        i++;
                }
        }
        return splitGraph(gh);
}
```

```
struct List* graphGetAllNodes(struct Graph* g) {
	if (g == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	struct List* ret = createList(NODE);
	int i;
	for (i=0; i < g->vn; i++) {
		addList(ret, g->nodes[i]);
	}
	return ret;
}

int32_t graphNumOfNodes(struct Graph* g) {
	if (g == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	return g->vn;
}

int32_t graphNumOfEdges(struct Graph* g) {
	if (g == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	return g->en;
}

struct List* graphGetChildNodes(struct Graph* g, struct Node* rt) {
	if (g == NULL) {
		printf("[Error] Graph doesn't exist!\n");
		exit(1);
	}
	struct List* children = createList(NODE);
	if (rt == NULL) return children;
	int i;
	for (i=0; i< g->en; i++) {
		if (g->edges[i].sour == rt) {
			addList(children, g->edges[i].dest);
		}
	}
	return children;
}

int32_t printGraph(struct Graph* g) {
	if (g == NULL) {
		printf("(null)\n");
		return 0;
	}
	printf("------------------------------------\n");
	printf("#Nodes: %d  ", g->vn);
	if (g->root != NULL) {
		printf("Root Node: %d\n", g->root->id);
	} else {
		printf("\n");
	}
	int i;
	for (i=0; i<g->vn; i++) {
		printNode(g->nodes[i]);
```

```
        }
        printf("#Edges: %d\n", g->en);
        for (i=0; i<g->en; i++) {
               printEdge(&g->edges[i]);
        }
        printf("--------------------------------------\n");
        return 0;
}



//test list
// int main() {
        // test list
        // struct List* list = createList(1);
        // printf("list type:%d\n", list->type);
        // struct List* newList = addList(addList(addList(addList(list, 52), 53), 54), 55);
        // printList(list);

        // struct Node* a = createNode(1, 0, 12, 0, 0, NULL);
        // struct Node* b = createNode(2, 1, 0, 1.2, 0, NULL);
        // struct Node* c = createNode(3, 2, 0, 0, 0, NULL);
        // struct Node* d = createNode(4, 3, 0, 0, 1, "Hello World!");
        // struct Graph* g = createGraph();
        // graphAddNode(g, a);
        // graphAddNode(g, b);
        // graphAddNode(g, c);
        // graphAddNode(g, d);
        // graphAddEdge(g, a, b, 3,0,0,0,"Edge1");
        // graphAddEdge(g, b, c, 2,0,0,1,NULL);

        // struct Graph* g2 = copyGraph(g);
        // g->edges[0].d = "ffff";
        // d->d = "????";
        // graphAddEdge(g2, c, d, 1,0,3.3,0,NULL);

        // printGraph(g);
        // printf("***************************\n");
        // printGraph(g2);

        // void * ptr = "xxx";
        // printf("%s\n", get_str_from_void_ptr(ptr));
        // exit(1);
// }



// below is the test for dict

// #include <stdlib.h>
// #include <stdio.h>
// #include <assert.h>

// #include "hashmap.h"

// #define KEY_MAX_LENGTH (256)
// #define KEY_PREFIX ("somekey")
// #define KEY_COUNT (1024*1024)

// typedef struct data_struct_s
// {
```

```
//     char key_string[KEY_MAX_LENGTH];
//     int number;
// } data_struct_t;


// int main()
// {
//     int index;
//     int error;
//     map_t mymap;
//     char key_string[KEY_MAX_LENGTH];
//     data_struct_t* value;

//     mymap = hashmap_new();

//     /* First, populate the hash map with ascending values */
//     /* Store the key string along side the numerical value so we can free it later */
//     value = malloc(sizeof(data_struct_t));
//     value->number = 1;
//     strcpy(value->key_string, "Warrior");
//     printf("%s\n", value->key_string);
//     hashmap_put(mymap, value->key_string, value);
//     data_struct_t* tmp = malloc(sizeof(data_struct_t));
//     int a = hashmap_get(mymap, value->key_string, (void**)(&tmp));
//     printf("%s:%d", tmp->key_string, tmp->number);
//     // error = hashmap_remove(mymap, key_string);
//     /* Now, destroy the map */
//     hashmap_free(mymap);

//     exit(1);
// }


//     struct List* list = createList(1);
//     printf("list type:%d\n", list->type);
//     struct List* newList = addList(addList(addList(addList(list, 52), 53), 54), 55);
//     printList(list);
// }

// test graph
// int main() {
//     struct Node* a = createNode(1, 3, "a");
//     struct Node* b = createNode(2, 3, "b");
//     struct Node* c = createNode(3, 3, "c");
//     struct Node* d = createNode(4, 3, "d");
//
//     struct Graph* g = createGraph();
//     graphAddNode(g, a);
//     graphAddNode(g, b);
//     graphAddNode(g, c);
//     graphAddNode(g, d);
//     graphAddEdgeP(g, a, b, STRING, "a->b");
//     graphAddEdgeP(g, a, c, STRING, "a->c");
//     graphAddEdgeP(g, a, d, STRING, "a->d");
//     graphAddEdgeP(g, c, d, STRING, "c->d");
//
//
//     struct Graph* g1 = createGraph();
//     graphAddNode(g1, a);
//     graphAddNode(g1, b);
//     graphAddNode(g1, c);
```

```
//      graphAddNode(g1, d);
//      // graphAddEdgeP(g1, a, b, STRING, "a->b");
//      graphAddEdgeP(g1, a, c, STRING, "a->c");
//      // graphAddEdgeP(g1, a, d, STRING, "a->d");
//      graphAddEdgeP(g1, c, d, STRING, "c->d");
//
//      struct List* l = subGraph(g, g1);
//
//
//      printf("The list size is: %d\n", getListSize(l));
//      int i;
//      for (i=getListSize(l)-1; i>=0; i-- ) {
//              printf("==============================\n");
//              printGraph( getList(l, i) );
//              printf("==============================\n");
//      }
// }


// int main() {
//      printf("%f", (float)1 );
// }
```

9.21. Main.c

```
/*
 * A unit test and example of how to use the simple C hashmap
 */

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

#include "hashmap.h"
#include "hashmap.c"

#define KEY_MAX_LENGTH (256)
#define KEY_PREFIX ("somekey")
#define KEY_COUNT (1024*1024)

typedef struct data_struct_s
{
    char key_string[KEY_MAX_LENGTH];
    int number;
} data_struct_t;

int main()
{
    int index;
    int error;
    map_t mymap;
    char key_string[KEY_MAX_LENGTH];
    data_struct_t* value;

    mymap = hashmap_new();

    /* First, populate the hash map with ascending values */
```

```
    /* Store the key string along side the numerical value so we can free it later */
    // value = malloc(sizeof(data_struct_t));

    // value->number = 1;
    // strcpy(value->key_string, "Warrior");
    // printf("%s\n", value->key_string);
    void * str = "xxx";
    hashmap_put(mymap, "a", str);
    // data_struct_t* tmp = malloc(sizeof(data_struct_t));
    char* a = hashmap_get(mymap, "a");
    printf("%s", a);
    // error = hashmap_remove(mymap, key_string);
    /* Now, destroy the map */
    hashmap_free(mymap);

    return 1;
}
```

9.22. Parser Test Cases

_arithmetic.in

1 + 5.4;
1 - 5.4;
1 * 5.4;
1 / 5.4;
1 % 5.4;
-42;
1 + -43;
1 * 2 + 3 * 4;
1 / 2 % 3 % 4;
1 + 2 - 3 / 4;
1 * 2 + 3;

_arithmetic.out

Expr(Binop(Num_Lit(1), Add, Num_Lit(5.4)));
Expr(Binop(Num_Lit(1), Sub, Num_Lit(5.4)));
Expr(Binop(Num_Lit(1), Mult, Num_Lit(5.4)));
Expr(Binop(Num_Lit(1), Div, Num_Lit(5.4)));
Expr(Binop(Num_Lit(1), Mod, Num_Lit(5.4)));
Expr(Unop(Sub, Num_Lit(42)));
Expr(Binop(Num_Lit(1), Add, Unop(Sub, Num_Lit(43))));
Expr(Binop(Binop(Num_Lit(1), Mult, Num_Lit(2)), Add, Binop(Num_Lit(3), Mult, Num_Lit(4))));
Expr(Binop(Binop(Binop(Num_Lit(1), Div, Num_Lit(2)), Mod, Num_Lit(3)), Mod, Num_Lit(4)));
Expr(Binop(Binop(Num_Lit(1), Add, Num_Lit(2)), Sub, Binop(Num_Lit(3), Div, Num_Lit(4))));

```
Expr(Binop(Binop(Num_Lit(1), Mult, Num_Lit(2)), Add, Num_Lit(3)));
```

_conditional.in

```
aList = ["str1","str2","str3"];
int i;
for(i = 0; i<= 5; i=i+1){
    if (str == "str2"){
        3+2;
    }
    else{
        /* do something */
    }
}
```

_conditional.out

```
Expr(Assign(aList, List(String_Lit(str1), String_Lit(str2), String_Lit(str3))));
Var_dec(Local(int, i, Noexpression));
For(Assign(i, Num_Lit(0));Binop(Id(i), Leq, Num_Lit(5));Assign(i, Binop(Id(i), Add,
Num_Lit(1)))){If(Binop(Id(str), Equal, String_Lit(str2))){Expr(Binop(Num_Lit(3), Add,
Num_Lit(2)));} Else{}}
```

_dict.in

```
{};
{"a":"b"};
{"a":"b","c":"d"};
{"a":1,"c":true};
```

_dict.out

```
Expr(Dict());
Expr(Dict(key:String_Lit(a),value:String_Lit(b)));
Expr(Dict(key:String_Lit(a),value:String_Lit(b), key:String_Lit(c),value:String_Lit(d)));
Expr(Dict(key:String_Lit(a),value:Num_Lit(1), key:String_Lit(c),value:Bool_lit(true)));
```

_function.in

```
int func(int a, int b) {
    int c = 0;
    return a + b + c;
}
func(1, 2);
func(a, b);
```

_function.out

```
Func(int func (Formal(int, a), Formal(int, b)) {Var_dec(Local(int, c, Num_Lit(0)));
Return(Binop(Binop(Id(a), Add, Id(b)), Add, Id(c)));})
Expr(Call(func, [Num_Lit(1), Num_Lit(2)]));
Expr(Call(func, [Id(a), Id(b)]));
```

_graph.in

```
a--b;
a--2&b--4&c;
a--2&[b--c,f];
a--[2&b--3&c, 1&d--1&[e,f--g]];
```

_graph.out

```
Expr(Graph_Link(Id(a), DLink, Id(b), WithEdge, Null));
Expr(Graph_Link(Id(a), DLink, Graph_Link(Id(b), DLink, Id(c), WithEdge, Num_Lit(4)),
WithEdge, Num_Lit(2)));
Expr(Graph_Link(Id(a), DLink, List(Graph_Link(Id(b), DLink, Id(c), WithEdge, Null), Id(f)),
WithEdge, Num_Lit(2)));
Expr(Graph_Link(Id(a), DLink, List(Graph_Link(Id(b), DLink, Id(c), WithEdge, Num_Lit(3)),
Graph_Link(Id(d), DLink, List(Id(e), Graph_Link(Id(f), DLink, Id(g), WithEdge, Null)), WithEdge,
Num_Lit(1))), WithEdge, List(Num_Lit(2), Num_Lit(1))));
```

_list.in

```
[];
[1,2,3];
[1,"2",2.0,true,false,1+1];
```

_list.out

```
Expr(List());
Expr(List(Num_Lit(1), Num_Lit(2), Num_Lit(3)));
Expr(List(Num_Lit(1), String_Lit(2), Num_Lit(2.), Bool_lit(true), Bool_lit(false), Binop(Num_Lit(1),
Add, Num_Lit(1))));
```

_literals.in

```
20;
20.0;
"str";
true;
false;
```

_literals.out

```
Expr(Num_Lit(20));
Expr(Num_Lit(20.));
Expr(String_Lit(str));
Expr(Bool_lit(true));
Expr(Bool_lit(false));
```

_node.in

```
node(1);
node("a");
node(false);
node([1,"2"]);
```

_node.out

```
Expr(Node(Num_Lit(1)));
Expr(Node(String_Lit(a)));
Expr(Node(Bool_lit(false)));
Expr(Node(List(Num_Lit(1), String_Lit(2))));
```

_relational.in

```
1==1;
1!=1;
1<1;
1<=1;
1>1;
1>=1;
true and true;
true or false;
```

_relational.out

```
Expr(Binop(Num_Lit(1), Equal, Num_Lit(1)));
Expr(Binop(Num_Lit(1), Neq, Num_Lit(1)));
Expr(Binop(Num_Lit(1), Less, Num_Lit(1)));
Expr(Binop(Num_Lit(1), Leq, Num_Lit(1)));
Expr(Binop(Num_Lit(1), Greater, Num_Lit(1)));
Expr(Binop(Num_Lit(1), Geq, Num_Lit(1)));
Expr(Binop(Bool_lit(true), And, Bool_lit(true)));
Expr(Binop(Bool_lit(true), Or, Bool_lit(false)));
```

_type_dec.in

```
int a;
float a;
bool a;
node a;
graph a;
```

```
list<int>a;
list<float>a;
list<string>a;
list<node>a;
list<graph>a;
dict<int>a;
dict<float>a;
dict<string>a;
dict<node>a;
dict<graph>a;
int a = 1;
float a = 1.0;
bool a = true;
node a = node(1);
graph a = node(1)--node(2);
list<int> a = [];
dict<int>a = {};
```

_type_dec.out

```
Var_dec(Local(int, a, Noexpression));
Var_dec(Local(float, a, Noexpression));
Var_dec(Local(bool, a, Noexpression));
Var_dec(Local(node, a, Noexpression));
Var_dec(Local(graph, a, Noexpression));
Var_dec(Local(list<int>, a, Noexpression));
Var_dec(Local(list<float>, a, Noexpression));
Var_dec(Local(list<string>, a, Noexpression));
Var_dec(Local(list<node>, a, Noexpression));
Var_dec(Local(list<graph>, a, Noexpression));
Var_dec(Local(dict<int>, a, Noexpression));
Var_dec(Local(dict<float>, a, Noexpression));
Var_dec(Local(dict<string>, a, Noexpression));
Var_dec(Local(dict<node>, a, Noexpression));
Var_dec(Local(dict<graph>, a, Noexpression));
Var_dec(Local(int, a, Num_Lit(1)));
Var_dec(Local(float, a, Num_Lit(1.)));
Var_dec(Local(bool, a, Bool_lit(true)));
Var_dec(Local(node, a, Node(Num_Lit(1))));
Var_dec(Local(graph, a, Graph_Link(Node(Num_Lit(1)), DLink, Node(Num_Lit(2)), WithEdge,
Null)));
Var_dec(Local(list<int>, a, List()));
```

Var_dec(Local(dict<int>, a, Dict()));

9.23. Parser Test Makefile

```
# test/parser Makefile
#  - builds the parserize executable for printing parsed strings from stdin

OCAMLC = ocamlc
OBJS = ../../compiler/_build/parser.cmo ../../compiler/_build/scanner.cmo parserize.cmo
INCLUDES = -I ../../compiler/_build

default: parserize

all:
	cd ..; make all

parserize: $(OBJS)
	$(OCAMLC) $(INCLUDES) -o parserize $(OBJS)

%.cmo: %.ml
	$(OCAMLC) $(INCLUDES) -c $<

%.cmi: %.mli
	$(OCAMLC) $(INCLUDES) -c $<

.PHONY: clean
clean:
	rm -f parserize *.cmo *.cmi

# # Generated by ocamldep *.ml
# parserize.cmo: ../../compiler/_build/parser.cmi ../../compiler/_build/ast.cmi
# parserize.cmx: ../../compiler/_build/parser.cmi ../../compiler/_build/ast.cmi
```

9.24. Parserize.ml

```
open Ast
open Printf

(* Unary operators *)
let txt_of_unop = function
  | Not -> "Not"
  | Neg -> "Sub"

(* Binary operators *)
let txt_of_binop = function
  (* Arithmetic *)
  | Add -> "Add"
  | Sub -> "Sub"
  | Mult -> "Mult"
  | Div -> "Div"
  | Mod -> "Mod"
  (* Boolean *)
  | Or -> "Or"
  | And -> "And"
```

```
  | Equal -> "Equal"
  | Neq -> "Neq"
  | Less -> "Less"
  | Leq -> "Leq"
  | Greater -> "Greater"
  | Geq -> "Geq"
  (* Graph *)
  | ListNodesAt -> "Child_Nodes_At"
  | ListEdgesAt -> "Child_Nodes&Edges_At"
  | RootAs -> "Root_As"

let txt_of_graph_op = function
  | Right_Link -> "RLink"
  | Left_Link -> "LLink"
  | Double_Link -> "DLink"

let txt_of_var_type = function
  | Void_t -> "void"
  | Null_t -> "null"
  | Int_t -> "int"
  | Float_t -> "float"
  | String_t -> "string"
  | Bool_t -> "bool"
  | Node_t -> "node"
  | Graph_t -> "graph"
  | Dict_Int_t -> "dict<int>"
  | Dict_Float_t -> "dict<float>"
  | Dict_String_t -> "dict<string>"
  | Dict_Node_t -> "dict<node>"
  | Dict_Graph_t -> "dict<graph>"
  | List_Int_t -> "list<int>"
  | List_Float_t -> "list<float>"
  | List_String_t -> "list<string>"
  | List_Node_t -> "list<node>"
  | List_Graph_t -> "list<graph>"

let txt_of_formal = function
| Formal(vtype, name) -> sprintf "Formal(%s, %s)" (txt_of_var_type vtype) name


let txt_of_formal_list formals =
  let rec aux acc = function
    | [] -> sprintf "%s" (String.concat ", " (List.rev acc))
    | fml :: tl -> aux (txt_of_formal fml :: acc) tl
  in aux [] formals

let txt_of_num = function
  | Num_Int(x) -> string_of_int x
  | Num_Float(x) -> string_of_float x

(* Expressions *)
let rec txt_of_expr = function
  | Num_Lit(x) -> sprintf "Num_Lit(%s)" (txt_of_num x)
  | Bool_lit(x) -> sprintf "Bool_lit(%s)" (string_of_bool x)
  | String_Lit(x) -> sprintf "String_Lit(%s)" x
  | Null -> sprintf "Null"
  | Node(x) -> sprintf "Node(%s)" (txt_of_expr x)
  | Unop(op, e) -> sprintf "Unop(%s, %s)" (txt_of_unop op) (txt_of_expr e)
  | Binop(e1, op, e2) -> sprintf "Binop(%s, %s, %s)"
      (txt_of_expr e1) (txt_of_binop op) (txt_of_expr e2)
  | Graph_Link(e1, op1, e2, e3) -> sprintf "Graph_Link(%s, %s, %s, WithEdge, %s)"
```

```
      (txt_of_expr e1) (txt_of_graph_op op1) (txt_of_expr e2) (txt_of_expr e3)
  | Id(x) -> sprintf "Id(%s)" x
  | Assign(e1, e2) -> sprintf "Assign(%s, %s)" e1 (txt_of_expr e2)
  | Noexpr -> sprintf "Noexpression"
  | ListP(l) -> sprintf "List(%s)" (txt_of_list l)
  | DictP(d) -> sprintf "Dict(%s)" (txt_of_dict d)
  | Call(f, args) -> sprintf "Call(%s, [%s])" (f) (txt_of_list args)
  | CallDefault(e, f, args) -> sprintf "CallDefault(%s, %s, [%s])" (txt_of_expr e) f
(txt_of_list args)

(*Variable Declaration*)
and txt_of_var_decl = function
  | Local(var, name, e1) -> sprintf "Local(%s, %s, %s)"
    (txt_of_var_type var) name (txt_of_expr e1)

(* Lists *)
and txt_of_list = function
  | [] -> ""
  | [x] -> txt_of_expr x
  | _ as l -> String.concat ", " (List.map txt_of_expr l)

(* Dict *)
and txt_of_dict_key_value = function
  | (key, value) -> sprintf "key:%s,value:%s" (txt_of_expr key) (txt_of_expr value)

and txt_of_dict = function
  | [] -> ""
  | [x] -> txt_of_dict_key_value x
  | _ as d -> String.concat ", " (List.map txt_of_dict_key_value d)

(* Functions Declaration *)
and txt_of_func_decl f =
  sprintf "%s %s (%s) {%s}"
    (txt_of_var_type f.returnType) f.name (txt_of_formal_list f.args) (txt_of_stmts f.body)

(* Statements *)
and txt_of_stmt = function
  | Expr(expr) -> sprintf "Expr(%s);" (txt_of_expr expr)
  | Func(f) -> sprintf "Func(%s)" (txt_of_func_decl f)
  | Return(expr) -> sprintf "Return(%s);" (txt_of_expr expr)
  | For(e1,e2,e3,s) -> sprintf "For(%s;%s;%s){%s}"
    (txt_of_expr e1) (txt_of_expr e2) (txt_of_expr e3) (txt_of_stmts s)
  | If(e1,s1,s2) -> sprintf "If(%s){%s} Else{%s}"
    (txt_of_expr e1) (txt_of_stmts s1) (txt_of_stmts s2)
  | While(e1, s) -> sprintf "While(%s){%s}"
    (txt_of_expr e1) (txt_of_stmts s)
  | Var_dec(var) -> sprintf "Var_dec(%s);" (txt_of_var_decl var)
and txt_of_stmts stmts =
  let rec aux acc = function
      | [] -> sprintf "%s" (String.concat "\n" (List.rev acc))
      | stmt :: tl -> aux (txt_of_stmt stmt :: acc) tl
  in aux [] stmts

(* Program entry point *)
let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  let result = txt_of_stmts program in
  print_endline result
```

9.25. Scanner Test Cases

_arithmetic.in

+ - * / %

_arithmetic.out

PLUS
MINUS
TIMES
DIVIDE
MOD

_boolean_operation.in

true false

_boolean_operation.out

BOOL_LITERAL
BOOL_LITERAL

_bracket.in

[ ] { } ( )

_bracket.out

LEFTBRACKET
RIGHTBRACKET
LEFTCURLYBRACKET
RIGHTCURLYBRACKET
LEFTROUNDBRACKET
RIGHTROUNDBRACKET

_comment.in

```
/*
This is comment.
int a = 3;
if else { }
*/
```

_comment.out

_comparator.in

```
> >= < <=
```

_comparator.out

```
GREATER
GREATEREQUAL
SMALLER
SMALLEREQUAL
```

_graph_operator.in

```
-- -> <-
```

_graph_operator.out

```
LINK
RIGHTLINK
LEFTLINK
```

_integer_float.in

10 10.0 11.1

_integer_float.out

INT_LITERAL
FLOAT_LITERAL
FLOAT_LITERAL

_logic_opearation.in

and or not if else for break continue in return

_logic_opearation.out

AND
OR
NOT
IF
ELSE
FOR
BREAK
CONTINUE
IN
RETURN

_primary_type.in

int float string bool node graph list dict null

_primary_type.out

INT
FLOAT
STRING
BOOL
NODE

GRAPH
LIST
DICT
NULL

_quote.in

"

_quote.out

QUOTE

_separator.in

; , = : .

_separator.out

SEMICOLUMN
SEQUENCE
ASSIGN
COLUMN
DOT

9.26. Scanner Test Makefile

```
# test/scanner Makefile
#  - builds the tokenize executable for printing scanned tokens from stdin

OCAMLC = ocamlc
OBJS = ../../compiler/_build/scanner.cmo tokenize.cmo
INCLUDES = -I ../../compiler/_build

default: tokenize

all:
	cd ..; make all
```

```
tokenize: $(OBJS)
	$(OCAMLC) $(INCLUDES) -o tokenize $(OBJS)

%.cmo: %.ml
	$(OCAMLC) $(INCLUDES) -c $<

%.cmi: %.mli
	$(OCAMLC) $(INCLUDES) -c $<

.PHONY: clean
clean:
	rm -f tokenize *.cmo *.cmi

# Generated by ocamldep *.ml
tokenize.cmo:
tokenize.cmx:
```

9.27. tokenize.ml

```
open Parser

let stringify = function
  (* calculation *)
  | PLUS -> "PLUS"   | MINUS -> "MINUS"
  | TIMES -> "TIMES" | DIVIDE -> "DIVIDE"
  | MOD -> "MOD"
  (* separator *)
  | SEMICOLUMN -> "SEMICOLUMN" | SEQUENCE -> "SEQUENCE"
  | ASSIGN -> "ASSIGN"         | COLUMN -> "COLUMN"
  | DOT -> "DOT"
  (* logical operation *)
  | AND -> "AND"      | OR -> "OR"
  | NOT -> "NOT"      | IF -> "IF"
  | ELSE -> "ELSE"    | FOR -> "FOR"
  | WHILE -> "WHILE"  | BREAK -> "BREAK"
  | CONTINUE -> "CONTINUE" | IN -> "IN"
  (* comparator *)
  | EQUAL -> "EQUAL"          | NOTEQUAL -> "NOTEQUAL"
  | GREATER -> "GREATER"      | GREATEREQUAL -> "GREATEREQUAL"
  | SMALLER -> "SMALLER"      | SMALLEREQUAL -> "SMALLEREQUAL"
  (* graph operator *)
  | LINK -> "LINK"            | RIGHTLINK -> "RIGHTLINK"
  | LEFTLINK -> "LEFTLINK"    | AT -> "AT"
  | AMPERSAND -> "AMPERSAND"  | SIMILARITY -> "SIMILARITY"
  (* identifier *)
  | ID(string) -> "ID"
  (* primary type *)
  | INT -> "INT"          | FLOAT -> "FLOAT"
  | STRING -> "STRING"    | BOOL -> "BOOL"
  | NODE -> "NODE"        | GRAPH -> "GRAPH"
  | LIST -> "LIST"        | DICT -> "DICT"
  | NULL -> "NULL"        | VOID -> "VOID"
  (* quote *)
  | QUOTE -> "QUOTE"
  (* boolean operation *)
  (* bracket *)
  | LEFTBRACKET -> "LEFTBRACKET"          | RIGHTBRACKET -> "RIGHTBRACKET"
```

```
  | LEFTCURLYBRACKET -> "LEFTCURLYBRACKET" | RIGHTCURLYBRACKET -> "RIGHTCURLYBRACKET"
  | LEFTROUNDBRACKET -> "LEFTROUNDBRACKET" | RIGHTROUNDBRACKET -> "RIGHTROUNDBRACKET"
  (* End-of-File *)
  | EOF -> "EOF"
  (* Literals *)
  | INT_LITERAL(int) -> "INT_LITERAL"
  | FLOAT_LITERAL(float) -> "FLOAT_LITERAL"
  | STRING_LITERAL(string) -> "STRING_LITERAL"
  | BOOL_LITERAL(bool) -> "BOOL_LITERAL"
  | RETURN -> "RETURN"

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let rec print_tokens = function
    | EOF -> " "
    | token ->
      print_endline (stringify token);
      print_tokens (Scanner.token lexbuf) in
  print_tokens (Scanner.token lexbuf)
```

9.28. Semantic Check Test Cases

_access_outer_func_variable.in

```
int foo() {
  int a = 0;
  int bar() {
    return a + 1;
  }
  bar();
}

foo();
```

_access_outer_func_variable.out

_illegal_assignment.in

```
int a = 3.1;
```

_illegal_assignment.out

```
illegal assignment int = float in a = 3.1
```

_illegal_binary_operation1.in

1+"hh";

_illegal_binary_operation1.out

illegal binary operator int + string in 1 + hh

_illegal_binary_operation2.in

1 == 1.1;

_illegal_binary_operation2.out

illegal binary operator int == float in 1 == 1.1

_illegal_binary_operation3.in

1 < "hh";

_illegal_binary_operation3.out

illegal binary operator int < string in 1 < hh

_illegal_binary_operation4.in

true and 1;

_illegal_binary_operation4.out

illegal binary operator bool and int in true and 1 _illegal_binary_operation5.in

```
1.1%1;
```

_illegal_binary_operation5.out

```
illegal binary operator float % int in 1.1 % 1
```

_illegal_unary_operation1.in

```
-"hh";
```

_illegal_unary_operation1.out

```
illegal unary operator - string in - hh
```

_illegal_unary_operation2.in

```
not 1;
```

_illegal_unary_operation2.out

```
illegal unary operator not int in not 1
```

_incompatible_func_arg_type.in

```
int foo(int a, int b) {
   return a + b;
}

foo("1",3);
```

_incompatible_func_arg_type.out

incompatible argument type string, but int is expected

_inconsistent_dict_element_type.in

```
dict<int> a = {"a": 1, "b": "c"};
```

_inconsistent_dict_element_type.out

dict can not contain objects of different types: int and string

_inconsistent_list_element_type.in

```
list<int> a = [1, "a"];
```

_inconsistent_list_element_type.out

list can not contain objects of different types: int and string

_invalid_dict_get1.in

```
dict<int> a = {"a": 1, "b":2};
a.get();
```

_invalid_dict_get1.out

dict get method should only take one argument of type int, string or node: a

_invalid_dict_get2.in

```
dict<int> a = {"a": 1, "b":2};
```

a.get(1.1);

_invalid_dict_get2.out

dict get method should only take one argument of type int, string or node: a

_invalid_dict_keys1.in

```
dict<int> a = {"a": 1, "b":2};
a.keys(1);
```

_invalid_dict_keys1.out

dict keys method do not take arguments: a

_invalid_dict_keys2.in

```
dict<int> a = {"a": 1, "b":2};
list<float> b = a.keys();
```

_invalid_dict_keys2.out

illegal assignment list<float> = list<null> in b = function call a.keys

_invalid_dict_put1.in

```
dict<int> a = {"a": 1, "b":2};
a.put();
```

_invalid_dict_put1.out

dict put method should only take two argument of type (int, string or node) and int: a _invalid_dict_put2.in

```
dict<int> a = {"a": 1, "b":2};
a.put(1.1, 2);
```

_invalid_dict_put2.out

dict put method should only take two argument of type (int, string or node) and int: a

_invalid_dict_put3.in

```
dict<int> a = {"a": 1, "b":2};
a.put(1, "2");
```

_invalid_dict_put3.out

dict put method should only take two argument of type (int, string or node) and int: a

_invalid_dict_remove1.in

```
dict<int> a = {"a": 1, "b":2};
a.remove();
```

_invalid_dict_remove1.out

dict remove method should only take one argument of type int, string or node: a

_invalid_dict_remove2.in

```
dict<int> a = {"a": 1, "b":2};
a.remove(1.1);
```

_invalid_dict_remove2.out

dict remove method should only take one argument of type int, string or node: a

_invalid_dict_size.in

dict<int> a = {"a": 1, "b":2};
a.size(1);

_invalid_dict_size.out

dict size method do not take arguments: a

_invalid_dict_type1.in

list<int> a = [1,2];
{"a": a};

_invalid_dict_type1.out

invalid dict type: list<int>

_invalid_empty_dict.in

dict<int> a = {};

_invalid_empty_dict.out

invalid empty dict declaration: dict

_invalid_empty_list.in

list<int> a = [];

_invalid_empty_list.out

invalid empty list declaration: list

_invalid_expr_after_return.in

```
int foo() {
    return 1;
    int a = 1;
}
```

_invalid_expr_after_return.out

nothing may follow a return

_invalid_graph_edge_at.in

```
node a = node("1");
node b = node("1");
graph g = a -- b;
g@(a,1);
```

_invalid_graph_edge_at.out

invalid graph edge at: g

_invalid_graph_edges.in

```
node a = node("1");
node b = node("1");
node c = node("1");
graph g = a--b--c;
g.edges(1);
```

_invalid_graph_edges.out

```
graph edges method do not take arguments: g
```

_invalid_graph_link.in

```
int a = 1;
node b = node("1");
a -- b;
```

_invalid_graph_link.out

```
left side of graph link should be node type: a
```

_invalid_graph_list_node_at.in

```
node a = node("1");
node b = node("1");
node c = node("1");
graph g = a--b--c;
int d = 1;
g@d;
```

_invalid_graph_list_node_at.out

```
invalid graph list node at: g @ d
```

_invalid_graph_nodes.in

```
node a = node("1");
node b = node("1");
node c = node("1");
graph g = a--b--c;
g.nodes(1);
```

_invalid_graph_nodes.out

```
graph nodes method do not take arguments: g
```

_invalid_graph_root.in

```
node a = node("1");
node b = node("1");
node c = node("1");
graph g = a--b--c;
g.root(1);
```

_invalid_graph_root.out

```
graph root method do not take arguments: g
```

_invalid_graph_root_as.in

```
node a = node("1");
node b = node("1");
node c = node("1");
graph g = a--b--c;
int d = 1;
g~d;
```

_invalid_graph_root_as.out

```
invalid graph root as: g ~ d
```

_invalid_graph_size.in

```
node a = node("1");
node b = node("1");
node c = node("1");
graph g = a -- b -- c;
g.size(1);
```

_invalid_graph_size.out

graph size method do not take arguments: g

_invalid_list_add1.in

```
list<int> a = [1,2,3];
a.add(1, 2);
```

_invalid_list_add1.out

list add method should only take one argument of type int: a

_invalid_list_add2.in

```
list<int> a = [1,2,3];
a.add("1");
```

_invalid_list_add2.out

list add method should only take one argument of type int: a

_invalid_list_get1.in

```
list<int> a = [1,2,3];
a.get(1, 2);
```

_invalid_list_get1.out

list get method should only take one argument of type int: a _invalid_list_get2.in

```
list<int> a = [1,2,3];
a.get("1");
```

_invalid_list_get2.out

list get method should only take one argument of type int: a

_invalid_list_pop.in

```
list<int> a = [1,2,3];
a.pop(1);
```

_invalid_list_pop.out

list pop method do not take arguments: a

_invalid_list_push1.in

```
list<int> a = [1,2,3];
a.push(1, 2);
```

_invalid_list_push1.out

list push method should only take one argument of type int: a

_invalid_list_push2.in

```
list<int> a = [1,2,3];
a.push("1");
```

_invalid_list_push2.out

list push method should only take one argument of type int: a

_invalid_list_remove1.in

```
list<int> a = [1,2,3];
a.remove(1, 2);
```

_invalid_list_remove1.out

list remove method should only take one argument of type int: a

_invalid_list_remove2.in

```
list<int> a = [1,2,3];
a.remove("1");
```

_invalid_list_remove2.out

list remove method should only take one argument of type int: a

_invalid_list_set1.in

```
list<int> a = [1,2,3];
a.set(1);
```

_invalid_list_set1.out

list set method should only take two argument of type int and int: a

_invalid_list_set2.in

```
list<int> a = [1,2,3];
a.set("1", 2);
```

_invalid_list_set2.out

list set method should only take two argument of type int and int: a

_invalid_list_set3.in

```
list<int> a = [1,2,3];
a.set(1, "2");
```

_invalid_list_set3.out

list set method should only take two argument of type int and int: a

_invalid_list_size.in

```
list<int> a = [1,2,3];
a.size(1);
```

_invalid_list_size.out

list size method do not take arguments: a

_invalid_list_type1.in

```
list<int> a = [1,2];
[a,a];
```

_invalid_list_type1.out

invalid list type: list<int>

_invalid_return_type.in

```
int foo() {
    return "1";
}
```

_invalid_return_type.out

```
wrong function return type: string, expect int
```

_legal_binary_operation.in

```
1+1;
1.1+1.1;
1+1.1;
1.1+1;
1-1;
1.1-1.1;
1-1.1;
1.1-1;
1*1;
1.1*1.1;
1*1.1;
1.1*1;
1/1;
1.1/1.1;
1/1.1;
1.1/1;
1==1;
1.1==1.1;
1!=1;
1.1!=1.1;
1<2;
1<2.1;
1.1<2;
1.1<2.1;
1<=2;
1<=2.1;
1.1<=2;
1.1<=2.1;
1>2;
```

```
1>2.1;
1.1>2;
1.1>2.1;
1>=2;
1>=2.1;
1.1>=2;
1.1>=2.1;
true and false;
false or true;
2 % 1;
```

_legal_binary_operation.out

_legal_unary_operation.in

```
-1;
-1.1;
not true;
not false;
```

_legal_unary_operation.out

_redefine_print_func.in

```
int print() {
   return 1;
}

print();
```

_redefine_print_func.out

```
function print may not be defined
```

_support_default_funcs.in

```
print("a");
printf("a");
printb(true);
int("a");
string(1);
float(1);
bool(1);
```

_support_default_funcs.out

_undeclared_variable.in

```
int a = 1;
a + b;
```

_undeclared_variable.out

```
undeclared identifier b
```

_unmatched_func_arg_len.in

```
int foo(int a, int b) {
    return a + b;
}

foo(1,2,3);
```

_unmatched_func_arg_len.out

```
args length not match in function call: main.foo
```

_unsupport_graph_list_edge_at.in

```
node a = node("1");
node b = node("1");
node c = node("1");
graph g = a--b--c;
g@@c;
```

_unsupport_graph_list_edge_at.out

```
unsupport graph list edge at: g @@ c
```

_valid_assignment.in

```
float v1 = 1;
string v2 = null;
node v3 = null;
graph v4 = null;

list<int> v5 = null;
list<float> v6 = null;
list<string> v7 = null;
list<node> v8 = null;
list<graph> v9 = null;
list<bool> v10 = null;

dict<int> v11 = null;
dict<float> v12 = null;
dict<string> v13 = null;
dict<node> v14 = null;
dict<graph> v15 = null;

dict<int> a = {"a":1};
list<int> v16 = a.keys();
list<string> v17 = a.keys();
list<node> v18 = a.keys();
```

_valid_assignment.out

_valid_dict_operation.in

```
dict<int> a = {"a": 1, "b":2};
a.put("c", 3);
a.put(1, 3);
node n = node("a");
a.put(n, 2);
a.get("a");
a.get(1);
a.get(n);
a.remove("a");
a.remove(1);
a.remove(n);
a.size();
a.keys();
list<int> c = a.keys();
list<string> d = a.keys();
list<node> f = a.keys();
```

_valid_dict_operation.out

_valid_graph_operation.in

```
node a = node("1");
node b = node("1");
node c = node("1");
graph g = a--b--c;
g.root();
g.size();
g.nodes();
g.edges();
graph g2 = a--b--c;
graph g3 = g + g2;
list<graph> g4 = g - b;
list<graph> g5 = g - g2;
```

_valid_graph_operation.out

_valid_list_operation.in

```
list<int> a = [1,2,3];
a.add(2);
a.remove(0);
a.push(2);
a.pop();
a.get(0);
a.set(0, 5);
list<float> b = [1.1, 2.2];
list<string> c = ["a", "b"];
node n1 = node("1");
list<node> d = [n1];
graph g = n1 -- null;
list<graph> e = [g];
list<bool> f = [true, false];
```

_valid_list_operation.out

_valid_return_type.in

```
float f1() {
   return 1;
}

string f2() {
   return null;
}

node f3() {
   return null;
}

graph f4() {
   return null;
```

```
}

list<int> f5() {
    return null;
}

list<string> f6() {
    return null;
}

list<float> f7() {
    return null;
}

list<node> f8() {
    return null;
}

list<graph> f9() {
    return null;
}

list<bool> f10() {
    return null;
}

dict<int> f11() {
    return null;
}

dict<float> f12() {
    return null;
}

dict<string> f13() {
    return null;
}

dict<node> f14() {
    return null;
}

dict<graph> f15() {
    return null;
```

```
}

list<int> f16() {
    dict<int> g = {"a": 1};
    return g.keys();
}

list<string> f17() {
    dict<int> g = {"a": 1};
    return g.keys();
}

list<node> f18() {
    dict<int> g = {"a": 1};
    return g.keys();
}
```

_valid_return_type.out

9.29. Semantic Check Makefile

```
# test/semantic_check Makefile
#  - builds the semantic_check executable for semantic checking strings from stdin

OCAMLC = ocamlc
OBJS
= ../../compiler/_build/parser.cmo ../../compiler/_build/scanner.cmo ../../compiler/_build/a
st.cmo ../../compiler/_build/cast.cmo ../../compiler/_build/organizer.cmo ../../compiler/_bu
ild/semant.cmo semantic_check.cmo
INCLUDES = -I ../../compiler/_build

default: semantic_check

all:
	cd ..; make all

semantic_check: $(OBJS)
	$(OCAMLC) $(INCLUDES) -o semantic_check $(OBJS)

%.cmo: %.ml
	$(OCAMLC) $(INCLUDES) -c $<

%.cmi: %.mli
	$(OCAMLC) $(INCLUDES) -c $<

.PHONY: clean
clean:
```

```
        rm -f semantic_check *.cmo *.cmi
```

9.30. semantic_check.ml

```
(* Program entry point *)
open Printf

let _ =
    let lexbuf = Lexing.from_channel stdin in
    let ast = Parser.program Scanner.token lexbuf in
    let cast = Organizer.convert ast in
    try
        Semant.check cast
    with
        Semant.SemanticError(m) -> print_endline m
        (* Semant.SemanticError(m) -> raise (Failure(m)) *)
```

9.31. Code Generator Test Cases

_cast.in

```
node a = node(1);
node b = node(1.2);
node c = node(true);
node d = node("Hello");
graph g = null;

/* int() */

print( int(23) );
print( int(a) );
print( int(b) );
g = a -- 2& b;
print( int( g@(a,b) ) );
print( int( (a -- 4.4& b)@(a,b) ) );

/* bool() */

print( bool(1>3) );
print( bool(1<3) );
print( bool(a) );
print( bool(b) );
print( bool(c) );
g = a -- (2<3)& b;
print( bool( g@(a,b) ) );
```

```
print( bool( (a -- 4.4& b)@(a,b) ) );

/* float() */

print( float(23) );
print( float(3.4) );
print( float(a) );
print( float(b) );
g = a -- 3.2& b;
print( float( g@(a,b) ) );
print( float( (a -- 6& b)@(a,b) ) );

/* string() */

print( string("Hello") );
print( string(d) );
print( string( (a -- "Edge"& b)@(a,b) ) );
```

_cast.out

```
23
1
1
2
4
false
true
true
true
true
true
true
23.000000
3.400000
1.000000
1.200000
3.200000
6.000000
Hello
Hello
Edge
```

_dict.in

```
dict<int> d_int = {1: 11, 2: 22, 3: 33};
print(d_int);
print(d_int.get(1));
print(d_int.put(4, 44));
print(d_int.remove(2));
print(d_int.size());
list<int> l_int = d_int.keys();
print(l_int);
print(d_int.has(2));
print(d_int.has(3));

node n1 = node(1);
node n2 = node(2);
node n3 = node(3);

dict<graph> d_graph = {n1: n1->n2, n2: n2->n3, n3: n3->n1};
print(d_graph);
print(d_graph.get(n1));
print(d_graph.put(n3, n3->n2));
print(d_graph.remove(n2));
print(d_graph.size());
list<node> l_node = d_graph.keys();
print(d_graph);
print(d_graph.has(n2));
```

_dict.out

```
{2: 22, 1: 11, 3: 33}
11
{2: 22, 1: 11, 3: 33, 4: 44}
{1: 11, 3: 33, 4: 44}
3
list:[1, 3, 4]
false
true
{2: --------------------------------------
#Nodes: 2  Root Node: 2
node  2: 1
node  1: 2
```

```
#Edges: 1
edge  2->  1
--------------------------------------
, 1: --------------------------------------
#Nodes: 2  Root Node: 1
node  1: 2
node  0: 3
#Edges: 1
edge  1->  0
--------------------------------------
, 0: --------------------------------------
#Nodes: 2  Root Node: 0
node  0: 3
node  2: 1
#Edges: 1
edge  0->  2
--------------------------------------
}
--------------------------------------
#Nodes: 2  Root Node: 2
node  2: 1
node  1: 2
#Edges: 1
edge  2->  1
--------------------------------------
{2: --------------------------------------
#Nodes: 2  Root Node: 2
node  2: 1
node  1: 2
#Edges: 1
edge  2->  1
--------------------------------------
, 1: --------------------------------------
#Nodes: 2  Root Node: 1
node  1: 2
node  0: 3
#Edges: 1
edge  1->  0
--------------------------------------
, 0: --------------------------------------
#Nodes: 2  Root Node: 0
node  0: 3
node  1: 2
#Edges: 1
```

```
edge  0->  1
--------------------------------------
, }
{2: --------------------------------------
#Nodes: 2  Root Node: 2
node  2: 1
node  1: 2
#Edges: 1
edge  2->  1
--------------------------------------
, 0: --------------------------------------
#Nodes: 2  Root Node: 0
node  0: 3
node  1: 2
#Edges: 1
edge  0->  1
--------------------------------------
, }
3
{2: --------------------------------------
#Nodes: 2  Root Node: 2
node  2: 1
node  1: 2
#Edges: 1
edge  2->  1
--------------------------------------
, 0: --------------------------------------
#Nodes: 2  Root Node: 0
node  0: 3
node  1: 2
#Edges: 1
edge  0->  1
--------------------------------------
, }
false
```

_dict_node.in

```
node a = node("a");
node b = node("b");

dict<node> d = { a: a };
```

```
print("dict<node> d = { a: a}");

printf("d.size() => %d\n", d.size());

printf("d.has(a) => ");
print(d.has(a));

printf("d.get(a) => %s\n", string( d.get(a) ));

printf("d.size() => %d\n", d.size());

printf("d.has(b) => ");
print(d.has(b));

print("d.put(b, b)");
d.put(b, b);

printf("d.size() => %d\n", d.size());

printf("d.has(b) => ");
print(d.has(b));


int i;
list<node> l = d.keys();
printf("d.keys() => [ ");
for (i=0; i<d.size()-1; i=i+1) {
  printf("%s, ", string(l.get(i)));
}
if (d.size() > 0) {
  printf("%s ]\n", string(l.get(i)));
} else {
  print("]");
}
```

_dict_node.out

```
dict<node> d = { a: a}
d.size() => 1
d.has(a) => true
```

```
d.get(a) => a
d.size() => 1
d.has(b) => false
d.put(b, b)
d.size() => 2
d.has(b) => true
d.keys() => [ a, b ]
```

_graph_direct_def.in

```
node a = node("a");
node b = node("b");
node c = node("c");
node d = node("d");
node e = node("e");

void printGraph(graph g) {
  string getNode(int i) {
    return string( nodes.get(i) );
  }
  printf("Root: %s\n", string(g.root()));

  printf("Nodes: ");
  list<node> nodes = g.nodes();
  int size = g.size();
  int i;
  int j;
  for (i=0; i < size - 1; i=i+1) {
    printf( "%s, ", getNode(i) );
  }
  if (size > 0) {
    print( getNode(i) );
  }

  printf("Edges:\n");
  node a;
  node b;
  for (i=0; i < size; i=i+1) {
    for (j=0; j<size; j=j+1) {
      a = nodes.get(i);
      b = nodes.get(j);
      if ( g@(a,b) != null ) {
```

```
        printf("%s -> %s\n", string(a), string(b));
      }
    }
  }
}

print("a->null");
printGraph(a->null);

print("----------------------------------");

print("a<-b--c->d");
printGraph(a<-b--c->d);

print("----------------------------------");

print("a<-a--b");
printGraph(a<-a--b);

print("----------------------------------");

print("a->[b->c, c->d]");
printGraph(a->[b->c, c->d]);

print("----------------------------------");

print("a->[b, c, d]");
printGraph(a->[b, c, d]);

print("----------------------------------");

print("a->[b, c<-d]");
printGraph(a->[b, c<-d]);
```

_graph_direct_def.out

```
a->null
Root: a
Nodes: a
Edges:
----------------------------------
a<-b--c->d
```

```
Root: a
Nodes: c, d, b, a
Edges:
c -> d
c -> b
b -> c
b -> a
----------------------------------
a<-a--b
Root: a
Nodes: a, b
Edges:
a -> b
b -> a
----------------------------------
a->[b->c, c->d]
Root: a
Nodes: a, b, c, d
Edges:
a -> b
a -> c
b -> c
c -> d
----------------------------------
a->[b, c, d]
Root: a
Nodes: a, b, c, d
Edges:
a -> b
a -> c
a -> d
----------------------------------
a->[b, c<-d]
Root: a
Nodes: a, b, c, d
Edges:
a -> b
a -> c
d -> c
```

_graph_edge.in

```
node a = node("a");
node b = node("b");
node c = node("c");
node d = node("d");
node e = node("e");



print("<node> -> <edge> & <node/graph>");
graph gh = a -> 2&b -> 1.2&c -> (1>3)&d -> (1<2)&e -> "Hello"&a;
print( gh@(a,b) );
print( gh@(b,c) );
print( gh@(c,d) );
print( gh@(d,e) );
print( gh@(e,a) );
gh = a -> 2&a;
print( gh@(a,a) );



print("<node> -> <edge> & [ <node> ]");
gh = a -> false&[b, c];
print( gh@(a,b) );
print( gh@(a,c) );
print( gh@(b,c) );

print("<node> -> [ <edge> & <node> ]");
gh = a -> [1&b, 2.0&c];
print( gh@(a,b) );
print( gh@(a,c) );
print( gh@(b,c) );


print("<node> -> <edge> & [ <graph> ]");
graph g1 = a -> "a->b"&b;
graph g2 = c -> "c->d"&d;
gh = e -- "EEE"&[g1, g2];
print( gh@(a,b) );
print( gh@(c,d) );
print( gh@(e,a) );
print( gh@(a,e) );
print( gh@(e,c) );
print( gh@(c,e) );
print( gh@(a,c) );
```

```
print("<node> -> [ <edge> & <graph> ]");
gh = e -- ["e--a"&g1, "e--c"&g2];
print( gh@(a,b) );
print( gh@(c,d) );
print( gh@(e,a) );
print( gh@(a,e) );
print( gh@(e,c) );
print( gh@(c,e) );
print( gh@(a,c) );



print("<node> -> <edge> & [ <node/graph> ]");
gh = a -> 2&[b, c, d->3&e];
print( gh@(a,b) );
print( gh@(a,c) );
print( gh@(a,d) );
print( gh@(d,e) );
print( gh@(a,e) );

print("<node> -> [ <edge> & <node/graph> ]");
gh = a -> ["a->b"&b, "a->c"&c, "a->d"&d<-"e->d"&e];
print( gh@(a,b) );
print( gh@(a,c) );
print( gh@(a,d) );
print( gh@(d,e) );
print( gh@(e,d) );
print( gh@(a,e) );
```

_graph_edge.out

```
<node> -> <edge> & <node/graph>
2
1.200000
false
true
Hello
(null)
<node> -> <edge> & [ <node> ]
false
false
```

```
(null)
<node> -> [ <edge> & <node> ]
1.000000
2.000000
(null)
<node> -> <edge> & [ <graph> ]
a->b
c->d
EEE
EEE
EEE
EEE
(null)
<node> -> [ <edge> & <graph> ]
a->b
c->d
e--a
e--a
e--c
e--c
(null)
<node> -> <edge> & [ <node/graph> ]
2
2
2
3
(null)
<node> -> [ <edge> & <node/graph> ]
a->b
a->c
a->d
(null)
e->d
(null)
```

_graph_merge.in

```
node a = node("a");
node b = node("b");
node c = node("c");
node d = node("d");
node e = node("e");
```

```
print("a -> 0&b +  c -> [1&a, 2&b, 4&d, 3&e]");

graph g = a -> 0&b +  c -> [1&a, 2&b, 4&d, 3&e];

printGraph(g);

print("-------------------------------------");
print("No shared nodes! Return the first graph.");
print("a->0&b + c->1&d");
printGraph( a->0&b + c->1&d );

print("-------------------------------------");
print("Shared edges. Update the edge value with the second one.");
print("a -- 0&b -- 2&c -- 1&a + b -- 3&c");
printGraph( a -- 0&b -- 2&c -- 1&a + b -- 3&c );

void printGraph(graph g) {
  string getNode(int i) {
    return string( nodes.get(i) );
  }
  printf("Root: %s\n", string(g.root()));

  printf("Nodes: ");
  list<node> nodes = g.nodes();
  int size = g.size();
  int i;
  int j;
  for (i=0; i < size - 1; i=i+1) {
    printf( "%s, ", getNode(i) );
  }
  if (size > 0) {
    print( getNode(i) );
  }

  printf("Edges:\n");
  node a;
  node b;
  for (i=0; i < size; i=i+1) {
    for (j=0; j<size; j=j+1) {
      a = nodes.get(i);
      b = nodes.get(j);
      if ( g@(a,b) != null ) {
        printf("%s -> %s : %d\n", string(a), string(b), int(g@(a,b)));
```

```
      }
    }
  }
}
```

_graph_merge.out

```
a -> 0&b +  c -> [1&a, 2&b, 4&d, 3&e]
Root: a
Nodes: a, b, c, d, e
Edges:
a -> b : 0
c -> a : 1
c -> b : 2
c -> d : 4
c -> e : 3
--------------------------------------
No shared nodes! Return the first graph.
a->0&b + c->1&d
Root: a
Nodes: a, b
Edges:
a -> b : 0
--------------------------------------
Shared edges. Update the edge value with the second one.
a -- 0&b -- 2&c -- 1&a + b -- 3&c
Root: a
Nodes: c, a, b
Edges:
c -> a : 1
c -> b : 3
a -> c : 1
a -> b : 0
b -> c : 3
b -> a : 0
```

_graph_method.in

```
node a = node("a");
node b = node("b");
```

```
node c = node("c");
node d = node("d");
node e = node("e");

graph gh = a->b->c;

print("graph gh = a->b->c");
printf("gh.root() => %s\n", string(gh.root()) );
printf("gh.size() => %d\n", gh.size() );
print("g2 = gh~b => Return a new graph with different root");
graph g2 = gh~b;
printf("gh.root() => %s\n", string(gh.root()) );
printf("gh.nodes() => ");
showNodeList( gh.nodes() );
printf("g2.root() => %s\n", string(g2.root()) );
printf("g2.nodes() => ");
showNodeList( gh.nodes() );

printf("(d<-e).root() => %s\n", string((d<-e).root()) );

printf("(a--[b,c]).root() => %s\n", string((a--[b,c]).root()) );

printf("((a--[b,c])~c).root() => %s\n", string(((a--[b,c])~c).root()) );

printf("(a->[b->c, d<-e]).size() => %d\n", (a->[b->c, d<-e]).size() );

printf("(a->[b->c, d<-e]).nodes() =>");
showNodeList( (a->[b->c, d<-e]).nodes() );

void showNodeList(list<node> l) {
  if (l == null) { return; }
  int i; int size = l.size();
  printf("[");
  for (i=0; i < size-1; i=i+1) {
    printf("%s, ", string( l.get(i) ) );
  }
  if (size > 0) {
    printf("%s]\n", string(l.get(i)));
  } else {
    print("]");
  }
}
```

_graph_method.out

```
graph gh = a->b->c
gh.root() => a
gh.size() => 3
g2 = gh~b => Return a new graph with different root
gh.root() => a
gh.nodes() => [b, c, a]
g2.root() => b
g2.nodes() => [b, c, a]
(d<-e).root() => d
(a--[b,c]).root() => a
((a--[b,c])~c).root() => c
(a->[b->c, d<-e]).size() => 5
(a->[b->c, d<-e]).nodes() =>[a, b, c, d, e]
```

_graph_sub_graph.in

```
node a = node("a");
node b = node("b");
node c = node("c");
node d = node("d");
node e = node("e");

graph g1 = a -- 0&b -- 2&c -- [1&a, 3&d, 4&e];
graph g2;

print("a -- 0&b -- 2&c -- [1&a, 3&d, 4&e] - a -- c -- b");
g2 = a -- c -- b;
printGraphList(g1 - g2);
print("-----------------------------");

print("The subgrpah with the original root is guaranteed to be the first in the list.");
print("(a -- 0&b -- 2&c -- [1&a, 3&d, 4&e])~e - a -- c -- b");
printGraphList((a -- 0&b -- 2&c -- [1&a, 3&d, 4&e])~e - a -- c -- b);

void printGraphList(list<graph> l) {
  int i;
  for (i = 0; i < l.size(); i=i+1 ) {
    printf("******  Graph %d  ******\n", i);
    printUndirectedGraph(l.get(i));
```

```
  }
}

void printUndirectedGraph(graph g) {
  string getNode(int i) {
    return string( nodes.get(i) );
  }
  printf("Root: %s\n", string(g.root()));

  printf("Nodes: ");
  list<node> nodes = g.nodes();
  int size = g.size();
  int i;
  int j;
  for (i=0; i < size - 1; i=i+1) {
    printf( "%s, ", getNode(i) );
  }
  if (size > 0) {
    print( getNode(i) );
  }

  printf("Edges:\n");
  node a;
  node b;
  for (i=0; i < size; i=i+1) {
    for (j=i+1; j<size; j=j+1) {
      a = nodes.get(i);
      b = nodes.get(j);
      if ( g@(a,b) != null ) {
        printf("%s -- %s : %d\n", string(a), string(b), int(g@(a,b)));
      }
    }
  }
}
```

_graph_sub_graph.out

```
a -- 0&b -- 2&c -- [1&a, 3&d, 4&e] - a -- c -- b
******  Graph 0  ******
Root: a
Nodes: a, b
Edges:
```

```
a -- b : 0
******  Graph 1  ******
Root: c
Nodes: c, d, e
Edges:
c -- d : 3
c -- e : 4
-----------------------------
The subgrpah with the original root is guaranteed to be the first in the list.
(a -- 0&b -- 2&c -- [1&a, 3&d, 4&e])~e - a -- c -- b
******  Graph 0  ******
Root: e
Nodes: c, d, e
Edges:
c -- d : 3
c -- e : 4
******  Graph 1  ******
Root: a
Nodes: a, b
Edges:
a -- b : 0
```

_graph_sub_node.in

```
node a = node("a");
node b = node("b");
node c = node("c");
node d = node("d");
node e = node("e");

graph g = a -- 0&b -- 2&c -- [1&a, 3&d, 4&e];

print("a -- 0&b -- 2&c -- [1&a, 3&d, 4&e]     -    e");
printGraphList(g-e);
print("-----------------------------");

print("a -- 0&b -- 2&c -- [1&a, 3&d, 4&e]     -    c");
printGraphList(g-c);
print("-----------------------------");

print("The subgrpah with the original root is guaranteed to be the first in the list.");
print("(a -- 0&b -- 2&c -- [1&a, 3&d, 4&e])~d   -  c");
```

```
printGraphList(g~d-c);

void printGraphList(list<graph> l) {
  int i;
  for (i = 0; i < l.size(); i=i+1 ) {
    printf("******  Graph %d  ******\n", i);
    printUndirectedGraph(l.get(i));
  }
}

void printUndirectedGraph(graph g) {
  string getNode(int i) {
    return string( nodes.get(i) );
  }
  printf("Root: %s\n", string(g.root()));

  printf("Nodes: ");
  list<node> nodes = g.nodes();
  int size = g.size();
  int i;
  int j;
  for (i=0; i < size - 1; i=i+1) {
    printf( "%s, ", getNode(i) );
  }
  if (size > 0) {
    print( getNode(i) );
  }

  printf("Edges:\n");
  node a;
  node b;
  for (i=0; i < size; i=i+1) {
    for (j=i+1; j<size; j=j+1) {
      a = nodes.get(i);
      b = nodes.get(j);
      if ( g@(a,b) != null ) {
        printf("%s -- %s : %d\n", string(a), string(b), int(g@(a,b)));
      }
    }
  }
}
```

_graph_sub_node.out

```
a -- 0&b -- 2&c -- [1&a, 3&d, 4&e]     -    e
******  Graph 0  ******
Root: a
Nodes: c, a, d, b
Edges:
c -- a : 1
c -- d : 3
c -- b : 2
a -- b : 0
-----------------------------
a -- 0&b -- 2&c -- [1&a, 3&d, 4&e]     -    c
******  Graph 0  ******
Root: a
Nodes: a, b
Edges:
a -- b : 0
******  Graph 1  ******
Root: d
Nodes: d
Edges:
******  Graph 2  ******
Root: e
Nodes: e
Edges:
-----------------------------
The subgrpah with the original root is guaranteed to be the first in the list.
(a -- 0&b -- 2&c -- [1&a, 3&d, 4&e])~d   -  c
******  Graph 0  ******
Root: d
Nodes: d
Edges:
******  Graph 1  ******
Root: a
Nodes: a, b
Edges:
a -- b : 0
******  Graph 2  ******
Root: e
Nodes: e
Edges:
```

_id_defalut_assign.in

```
int a;
printf("int a; => ");
print(a);
printf("int a; a == 0; => ");
print(a == 0);
printf("int fun(){ } => ");
int intNull() {  }
print( intNull() );
print("----------------------------------");

float b;
printf("float b; => ");
print(b);
printf("float b; b == 0; => ");
print(b==0);
printf("float fun(){ } => ");
float floatNull() {  }
print( floatNull() );
print("----------------------------------");

bool c;
printf("bool c; => ");
print(c);
printf("bool c; c == false; => ");
print(c==false);
printf("bool fun(){ } => ");
bool boolNull() {  }
print( boolNull() );
print("----------------------------------");

string d;
printf("string d; => ");
print(d);
printf("string d; d == null; => ");
print(d == null);
printf("string fun(){ } => ");
string stringNull() {  }
print( stringNull() );
printf("string fun(){ return null; } => ");
string stringNull2() { return null; }
print( stringNull2() );
```

```
printf("string fun(){ return \"\"; } => ");
string stringNull3() { return ""; }
print( stringNull3() );
print("---------------------------------");

list<int> e;
printf("list<int> e; => ");
print(e);
printf("list<int> e; e == null; => ");
print(e == null);
printf("list<int> fun(){ } => ");
list<int> listIntNull() {  }
print( listIntNull() );
printf("list<int> fun(){ return null; } => ");
list<int> listIntNull2() { return null; }
print( listIntNull2() );
print("---------------------------------");

dict<float> f;
printf("dict<float> f; => ");
print(f);
printf("dict<float> f; f == null; => ");
print(f==null);
printf("dict<float> fun(){ } => ");
dict<float> dictFloatNull() {  }
print( dictFloatNull() );
printf("dict<float> fun(){ return null; } => ");
dict<float> dictFloatNull2() { return null; }
print( dictFloatNull2() );
print("---------------------------------");

node n;
printf("node n; => ");
print(n);
printf("node n; n==null; => ");
print(n==null);
printf("node fun(){ return null; } => ");
node nodeNull() { return null; }
print( nodeNull() );
printf("node fun(){ } => ");
node nodeNull2() { }
print( nodeNull2() );
print("---------------------------------");
```

```
graph g;
printf("graph g; => ");
print(g);
printf("graph n; g==null; => ");
print(g==null);
printf("graph fun(){ return null; } => ");
graph graphNull() { return null; }
print( graphNull() );
printf("graph fun(){ } => ");
graph graphNull2() { }
print( graphNull2() );
```

_id_defalut_assign.out

```
int a; => 0
int a; a == 0; => true
int fun(){ } => 0
----------------------------------
float b; => 0.000000
float b; b == 0; => true
float fun(){ } => 0.000000
----------------------------------
bool c; => false
bool c; c == false; => true
bool fun(){ } => false
----------------------------------
string d; => (null)
string d; d == null; => true
string fun(){ } => (null)
string fun(){ return null; } => (null)
string fun(){ return ""; } =>
----------------------------------
list<int> e; => (null)
list<int> e; e == null; => true
list<int> fun(){ } => (null)
list<int> fun(){ return null; } => (null)
----------------------------------
dict<float> f; => (null)
dict<float> f; f == null; => true
dict<float> fun(){ } => (null)
dict<float> fun(){ return null; } => (null)
----------------------------------
```

```
node n; => (null)
node n; n==null; => true
node fun(){ return null; } => (null)
node fun(){ } => (null)
---------------------------------
graph g; => (null)
graph n; g==null; => true
graph fun(){ return null; } => (null)
graph fun(){ } => (null)
```

_list.in

```
print("------------------test for list of int type------------------");

list<int> l_int = [1, 2, 3];
print(l_int);
l_int.add(4);
print(l_int);
print(l_int.get(0));
l_int.set(0, 4);
print(l_int);
l_int.remove(0);
print(l_int);
print(l_int.size());
print(l_int.pop());
print(l_int);
print(l_int.push(5));

print("------------------test for list of float type------------------");

list<float> l_float = [1.0, 2.0, 3.0];
print(l_float);
l_float.add(4.0);
print(l_float);
print(l_float.get(0));
l_float.set(0, 4.0);
print(l_float);
l_float.remove(0);
print(l_float);
print(l_float.size());
print(l_float.pop());
print(l_float);
```

```
print(l_float.push(5.0));

print("------------------test for list of float type------------------");

list<string> l_string = ["a", "b", "c"];
print(l_string);
l_string.add("d");
print(l_string.get(0));
l_string.set(0, "e");
print(l_string);
l_string.remove(0);
print(l_string);
print(l_string.size());
print(l_string.pop());
print(l_string);
print(l_string.push("x"));

print("------------------test for list of bool type------------------");


list<bool> l_bool = [true, false, true];
print(l_bool);
l_bool.add(false);
print(l_bool.get(0));
l_bool.set(0, false);
print(l_bool);
l_bool.remove(0);
print(l_bool);
print(l_bool.size());
print(l_bool.pop());
print(l_bool);
print(l_bool.push(true));

print("------------------test for list of node type------------------");

node n1 = node(1);
node n2 = node(2);
node n3 = node(3);
list<node> l_node = [n1, n2, n3];
print(l_node);
l_node.add(node(4));
print(l_node.get(0));
l_node.set(0, node("x"));
print(l_node);
```

```
l_node.remove(0);
print(l_node);
print(l_node.size());
print(l_node.pop());
print(l_node);
print(l_node.push(node("y")));

print("------------------test for list of graph type------------------");

list<graph> l_graph = [n1->n2, n2->n3, n3->n1];

print(l_graph);
l_graph.add(n1<-n2);
print(l_graph.get(0));
l_graph.set(0, n1--n2);
print(l_graph);
l_graph.remove(0);
print(l_graph);
print(l_graph.size());
print(l_graph.pop());
print(l_graph);
print(l_graph.push(node(5)->node(6)));
```

_list.out

```
------------------test for list of int type------------------
list:[1, 2, 3]
list:[1, 2, 3, 4]
1
list:[4, 2, 3, 4]
list:[2, 3, 4]
3
4
list:[2, 3]
list:[2, 3, 5]
------------------test for list of float type------------------
list:[1.000000, 2.000000, 3.000000]
list:[1.000000, 2.000000, 3.000000, 4.000000]
1.000000
list:[4.000000, 2.000000, 3.000000, 4.000000]
list:[2.000000, 3.000000, 4.000000]
3
```

```
4.000000
list:[2.000000, 3.000000]
list:[2.000000, 3.000000, 5.000000]
------------------test for list of float type------------------
list:[a, b, c]
a
list:[e, b, c, d]
list:[b, c, d]
3
d
list:[b, c]
list:[b, c, x]
------------------test for list of bool type------------------
list:[true, false, true]
true
list:[false, false, true, false]
list:[false, true, false]
3
false
list:[false, true]
list:[false, true, true]
------------------test for list of node type------------------
list:[node  7: 1
node  6: 2
node  5: 3
]
node  7: 1
list:[node  3: x
node  6: 2
node  5: 3
node  4: 4
]
list:[node  6: 2
node  5: 3
node  4: 4
]
3
node  4: 4
list:[node  6: 2
node  5: 3
]
list:[node  6: 2
node  5: 3
node  2: y
```

```
]
-------------------test for list of graph type------------------
list:[--------------------------------------
#Nodes: 2  Root Node: 7
node  7: 1
node  6: 2
#Edges: 1
edge  7->  6
--------------------------------------
--------------------------------------
#Nodes: 2  Root Node: 6
node  6: 2
node  5: 3
#Edges: 1
edge  6->  5
--------------------------------------
--------------------------------------
#Nodes: 2  Root Node: 5
node  5: 3
node  7: 1
#Edges: 1
edge  5->  7
--------------------------------------
]
--------------------------------------
#Nodes: 2  Root Node: 7
node  7: 1
node  6: 2
#Edges: 1
edge  7->  6
--------------------------------------
list:[--------------------------------------
#Nodes: 2  Root Node: 7
node  7: 1
node  6: 2
#Edges: 2
edge  7->  6
edge  6->  7
--------------------------------------
--------------------------------------
#Nodes: 2  Root Node: 6
node  6: 2
node  5: 3
#Edges: 1
```

```
edge  6->  5
--------------------------------------
--------------------------------------
#Nodes: 2  Root Node: 5
node  5: 3
node  7: 1
#Edges: 1
edge  5->  7
--------------------------------------
--------------------------------------
#Nodes: 2  Root Node: 7
node  7: 1
node  6: 2
#Edges: 1
edge  6->  7
--------------------------------------
]
list:[--------------------------------------
#Nodes: 2  Root Node: 6
node  6: 2
node  5: 3
#Edges: 1
edge  6->  5
--------------------------------------
--------------------------------------
#Nodes: 2  Root Node: 5
node  5: 3
node  7: 1
#Edges: 1
edge  5->  7
--------------------------------------
--------------------------------------
#Nodes: 2  Root Node: 7
node  7: 1
node  6: 2
#Edges: 1
edge  6->  7
--------------------------------------
]
3
--------------------------------------
#Nodes: 2  Root Node: 7
node  7: 1
node  6: 2
```

```
#Edges: 1
edge  6->  7
--------------------------------------
list:[--------------------------------------
#Nodes: 2  Root Node: 6
node  6: 2
node  5: 3
#Edges: 1
edge  6->  5
--------------------------------------
--------------------------------------
#Nodes: 2  Root Node: 5
node  5: 3
node  7: 1
#Edges: 1
edge  5->  7
--------------------------------------
]
list:[--------------------------------------
#Nodes: 2  Root Node: 6
node  6: 2
node  5: 3
#Edges: 1
edge  6->  5
--------------------------------------
--------------------------------------
#Nodes: 2  Root Node: 5
node  5: 3
node  7: 1
#Edges: 1
edge  5->  7
--------------------------------------
--------------------------------------
#Nodes: 2  Root Node: 1
node  1: 5
node  0: 6
#Edges: 1
edge  1->  0
--------------------------------------
]
```

_list_automatic_conversion.in

```
node a = node("a");
node b = node("b");
node c = node("c");
node d = node("d");
node e = node("e");

list<graph> l1 = [
  a, b, c, d->e
];

int i;
int size = l1.size();
graph gh;
for (i=0; i<size; i=i+1) {
  printf("graph %d: root - %s, nodes - %d\n",
    i, string( l1.get(i).root() ), l1.get(i).size());
}


print([1, 2, 3.]);
```

_list_automatic_conversion.out

```
graph 0: root - a, nodes - 1
graph 1: root - b, nodes - 1
graph 2: root - c, nodes - 1
graph 3: root - d, nodes - 2
list:[1.000000, 2.000000, 3.000000]
```

_node_var_type.in

```
node a = null;

print(a);

a = node(1);

print( int(a) );
```

```
a = node(-3.4);

print( float(a) );

a = node(1>2);

print( bool(a) );

a = node(true);

print( bool(a) );

a = node("Hello World!");

print( string(a) );
```

_node_var_type.out

```
(null)
1
-3.400000
false
true
Hello World!
```

_print_test.in

```
print(23);
print(-1.2);
print(1>2);
print(true);
print("Hello World!");
print(null);
print(node("a"));
print([1, 2, 3]);
print({"a": 1, "b": 2});
print(1, true, "Hello~");

int a = 1;
```

```
float b = 1.2;
string d = "What!";

printf("%d--\n%.2f--\n%s\n", a, b , d);
```

_print_test.out

```
23
-1.200000
false
true
Hello World!
null
node  0: a
list:[1, 2, 3]
{b: 2, a: 1}
1
true
Hello~
1--
1.20--
What!
```

_test.in

```
print("Hello World!");
```

_test.out

```
Hello World!
```

example_bfs.in

```
list<node> bfs(graph gh, node r) {
  if (gh == null or gh.size() == 0) { return null; }

  int i; node curr; node tmp_n; list<node> children;
```

```
  dict<node> set = { r: r };
  list<node> res = null;

  list<node> queue = [ r ];
  while (queue.size() > 0) {
    curr = queue.get(0); queue.remove(0);
    if (res == null) { res = [curr]; } else { res.add(curr); }

    children = gh@curr;
    for (i=0; i<children.size(); i=i+1) {
      tmp_n = children.get(i);
      if (not set.has( tmp_n )) {
        set.put( tmp_n, tmp_n );
        queue.add(tmp_n);
      }
    }
  }

  return res;
}

void printNodeList(list<node> l) {
  int i;
  printf("[ ");
  for (i=0; i<l.size()-1; i = i+1) {
    printf("%s, ", string( l.get(i) ));
  }
  if (l.size() > 0) {
    printf("%s ]\n", string( l.get(i) ));
  } else {
    print("]");
  }
}

node a = node("a");
node b = node("b");
node c = node("c");
node d = node("d");
node e = node("e");
node f = node("f");
node g = node("g");

graph gh;
```

```
print("a--[b, c--[e, f], d]");
gh = a--[b, c--[e, f], d];
printf("bfs(gh, a): ");
printNodeList( bfs(gh, a) );

printf("bfs(gh, b): ");
printNodeList( bfs(gh, b) );

printf("bfs(gh, c): ");
printNodeList( bfs(gh, c) );

printf("bfs(gh, d): ");
printNodeList( bfs(gh, d) );

printf("bfs(gh, e): ");
printNodeList( bfs(gh, e) );

printf("bfs(gh, f): ");
printNodeList( bfs(gh, f) );
```

example_bfs.out

```
a--[b, c--[e, f], d]
bfs(gh, a): [ a, b, c, d, e, f ]
bfs(gh, b): [ b, a, c, d, e, f ]
bfs(gh, c): [ c, e, f, a, b, d ]
bfs(gh, d): [ d, a, b, c, e, f ]
bfs(gh, e): [ e, c, f, a, b, d ]
bfs(gh, f): [ f, c, e, a, b, d ]
```

example_dfs.in

```
list<node> dfs(graph gh, node r) {
  if (gh == null or gh.size() == 0) { return null; }

  int i; node curr; node tmp_n; list<node> children;
  bool found;
  dict<int> set = { r: 0 };
  list<node> res = [r];
```

```
  list<node> stack = [ r ];
  while (stack.size() > 0) {
    curr = stack.get( stack.size() - 1 );
    set.put(curr, 1);

    children = gh@curr;
    found = false;
    for (i=0; (not found) and (i<children.size()); i=i+1) {
      tmp_n = children.get(i);
      if (not set.has( tmp_n )) { set.put( tmp_n, 0 ); }
      if (set.get(tmp_n) == 0) {
        stack.push(tmp_n);
        res.add(tmp_n);
        found = true;
      }
    }
    if (not found) {
      set.put(r, 2);
      stack.pop();
    }
  }

  return res;
}

void printNodeList(list<node> l) {
  int i;
  printf("[ ");
  for (i=0; i<l.size()-1; i = i+1) {
    printf("%s, ", string( l.get(i) ));
  }
  if (l.size() > 0) {
    printf("%s ]\n", string( l.get(i) ));
  } else {
    print("]");
  }
}

node a = node("a");
node b = node("b");
node c = node("c");
node d = node("d");
node e = node("e");
node f = node("f");
```

```
node g = node("g");

graph gh;

print("a--[b, c--[e, f], d]");
gh = a--[b, c--[e, f], d];
printf("dfs(gh, a): ");
printNodeList( dfs(gh, a) );

printf("dfs(gh, b): ");
printNodeList( dfs(gh, b) );

printf("dfs(gh, c): ");
printNodeList( dfs(gh, c) );

printf("dfs(gh, d): ");
printNodeList( dfs(gh, d) );

printf("dfs(gh, e): ");
printNodeList( dfs(gh, e) );

printf("dfs(gh, f): ");
printNodeList( dfs(gh, f) );
```

example_dfs.out

```
a--[b, c--[e, f], d]
dfs(gh, a): [ a, b, c, e, f, d ]
dfs(gh, b): [ b, a, c, e, f, d ]
dfs(gh, c): [ c, e, f, a, b, d ]
dfs(gh, d): [ d, a, b, c, e, f ]
dfs(gh, e): [ e, c, f, a, b, d ]
dfs(gh, f): [ f, c, e, a, b, d ]
```

example_dijkstra.in

```
node a = node("a");
node b = node("b");
node c = node("c");
node d = node("d");
```

```
node e = node("e");
node f = node("f");
node g = node("g");

graph gh = a->[
        1&b->1&e->[4&g->1&b, 2&c],
        5&c->[1&g, 1&f->1&c],
        3&d->[2&c, 3&f]
];

printGraph(gh);

print("\nDijkstra Results:");

dijkstra(gh, a);

void dijkstra(graph gh, node sour) {
        dict<int> distance = { sour: 0 };
        list<node> queue = gh.nodes();
        dict<node> parent = {sour: sour};
        int i;
        for (i=0; i<queue.size(); i=i+1) {
                distance.put(queue.get(i), 2147483647);
                parent.put(queue.get(i), null);
        }
        distance.put(sour, 0);

        while (queue.size() > 0) {
                updateDistance( findMin() );
        }
        queue = gh.nodes();
        for (i=0; i<queue.size(); i=i+1) {
                showRes(queue.get(i));
        }

        node findMin() {
                node minNode = queue.get(0);
                int minDis = distance.get(minNode);
                int minIndex = 0;

                int i; node tmp;
                for (i = 1; i < queue.size(); i=i+1) {
                        tmp = queue.get(i);
                        if ( distance.get(tmp) < minDis ) {
```

```
					minNode = tmp;
					minDis = distance.get(tmp);
					minIndex = i;
				}
			}
			queue.remove(minIndex);
			return minNode;
		}

		void updateDistance(node u) {
			int i; int dv; int dis; node v;
			list<node> neighs = gh@u;
			int du = distance.get(u);
			for (i = 0; i<neighs.size(); i=i+1) {
				v = neighs.get(i);
				dv = distance.get(v);
				dis = int( gh@(u, v) );
				if ((dis + du) < dv) {
					distance.put(v, dis+du);
					parent.put(v, u);
				}
			}
		}

		void showRes(node dest) {
			list<node> res = [dest];
			node tmp = parent.get(dest);
			while (tmp != null) {
				res.add( tmp );
				tmp = parent.get(tmp);
			}
			int i;
			printf("%s -> %s : %d [ ", string(sour), string(dest), distance.get(dest) );
			for (i=res.size()-1; i > 0; i=i-1) {
				printf("%s, ", string( res.get(i) ));
			}
			if (i == 0) {
				printf("%s ]\n", string( res.get(i) ));
			} else {
				print("]");
			}
		}
	}
```

```
void printGraph(graph g) {
  string getNode(int i) {
    return string( nodes.get(i) );
  }
  printf("Root: %s\n", string(g.root()));

  printf("Nodes: ");
  list<node> nodes = g.nodes();
  int size = g.size();
  int i;
  int j;
  for (i=0; i < size - 1; i=i+1) {
    printf( "%s, ", getNode(i) );
  }
  if (size > 0) {
    print( getNode(i) );
  }

  printf("Edges:\n");
  node a;
  node b;
  for (i=0; i < size; i=i+1) {
    for (j=0; j<size; j=j+1) {
      a = nodes.get(i);
      b = nodes.get(j);
      if ( g@(a,b) != null ) {
        printf("%s -> %s : %d\n", string(a), string(b), int(g@(a,b)));
      }
    }
  }
}
```

example_dijkstra.out

```
Root: a
Nodes: a, e, g, b, c, f, d
Edges:
a -> b : 1
a -> c : 5
a -> d : 3
e -> g : 4
e -> c : 2
```

```
g -> b : 1
b -> e : 1
c -> g : 1
c -> f : 1
f -> c : 1
d -> c : 2
d -> f : 3

Dijkstra Results:
a -> a : 0 [ a ]
a -> e : 2 [ a, b, e ]
a -> g : 5 [ a, b, e, c, g ]
a -> b : 1 [ a, b ]
a -> c : 4 [ a, b, e, c ]
a -> f : 5 [ a, b, e, c, f ]
a -> d : 3 [ a, d ]
```

test_arith.in

```
print(1+1);
print(2-1);
print(2*3);
print(9/4);
print(8/4);
print(5%3);
print(1.2+1);
print(1.2-1);
print(1-1.2);
print(2*0.4);
print(9./4);
print(-8);
print(-2.1);
print(-1);
print(-2.1);
```

test_arith.out

```
2
1
6
2
```

```
2
2
2.200000
0.200000
-0.200000
0.800000
2.250000
-8
-2.100000
-1
-2.100000
```

test_if.in

```
int a = 2;
if (a < 3) {
	print(a);
}
if(a>3) {
	print(10);
}
else
{
	print("True");
}
float b = 0;
if (b < 3) {
	print(b);
}
bool c = true;
if (c) {
	print("True");
}
```

test_if.out

```
2
True
0.000000
True
```

test_inner_var_access.in

```
int d = 1;
int b(int c) {
  int d = 2;
  int a() {
    return d + c;
  }
  return a();
}
print(b(3));
print(d);
```

test_inner_var_access.out

```
5
1
```

test_node_basic.out

```
node  3: 1
node  2: 1.200000
node  1: Hello World!
node  0: true
node  4: 22
```

test_while.in

```
int a = 0;
while (a < 3) {
	print(a);
	a = a + 1;
}
float b = 0;
while (b < 3) {
	print(b);
```

```
	b = b + 1;
}
bool c = true;
while (c) {
	print(c);
	c = not c;
}
```

test_while.out

```
0
1
2
0.000000
1.000000
2.000000
true
```

circline.sh

```
# Check whether the file "utils.bc" exist
file="utils.bc"
if [ ! -e "$file" ]
then
	clang -emit-llvm -o utils.bc -c ../compiler/lib/utils.c -Wno-varargs
fi

if [ $# -eq 1 ]
then
   ../compiler/circline.native <$1 >a.ll
else
   ../compiler/circline.native $1 <$2 >a.ll
fi
clang -Wno-override-module utils.bc a.ll -o $1.exe
./$1.exe
rm a.ll
rm ./$1.exe

# /usr/local/opt/llvm38/bin/clang-3.8
```

9.32. Codegen Test Makefile

```
# circling: test Makefile
#  - builds all files needed for testing, then runs tests

default: test

all: clean
	cd ..; make all

test: clean build
	clang -emit-llvm -o utils.bc -c ../compiler/lib/utils.c -Wno-varargs
	bash ./test_scanner.sh
	bash ./test_parser.sh
	bash ./test_semantic.sh
	bash ./test_code_gen.sh

build:
	cd scanner; make
	cd parser; make
	cd semantic_check; make

.PHONY: clean
clean:
	rm -f utils.bc
	cd scanner; make clean
	cd parser; make clean
	cd semantic_check; make clean
```

9.33. test_scanner.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

result=true

INPUT_FILES="scanner/*.in"
printf "${CYAN}Running scanner tests...\n${NC}"

for input_file in $INPUT_FILES; do
    output_file=${input_file/.in/.out}
    scanner/tokenize < $input_file | cmp -s $output_file -
    if [ "$?" -eq 0 ]; then
       printf "%-65s ${GREEN}SUCCESS\n${NC}" "  - checking $input_file..."
    else
       printf "%-65s ${RED}ERROR\n${NC}" "  - checking $input_file..." 1>&2
       result=false
    fi
done
```

```
exit 0

# if $result; then
#       exit 0
# else
#       exit 1
# fi
```

9.34. test_parser.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

result=true

INPUT_FILES="parser/*.in"
printf "${CYAN}Running Parser tests...\n${NC}"

for input_file in $INPUT_FILES; do
    output_file=${input_file/.in/.out}
    parser/parserize < $input_file | cmp -s $output_file -
    if [ "$?" -eq 0 ]; then
       printf "%-65s ${GREEN}SUCCESS\n${NC}" "  - checking $input_file..."
    else
       printf "%-65s ${RED}ERROR\n${NC}" "  - checking $input_file..." 1>&2
       result=false
    fi
done

cd ../compiler;
ocamlyacc -v parser.mly;

exit 0

# if $result; then
#       exit 0
# else
#       exit 1
# fi
```

9.35. test_semantic.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'
```

```
result=true

INPUT_FILES="semantic_check/*.in"
printf "${CYAN}Running Semantic Check tests...\n${NC}"

for input_file in $INPUT_FILES; do
    output_file=${input_file/.in/.out}
    semantic_check/semantic_check < $input_file | cmp -s $output_file -
    if [ "$?" -eq 0 ]; then
       printf "%-65s ${GREEN}SUCCESS\n${NC}" "  - checking $input_file..."
    else
       printf "%-65s ${RED}ERROR\n${NC}" "  - checking $input_file..." 1>&2
        result=false
    fi
done

exit 0

# if $result; then
#     exit 0
# else
#     exit 1
# fi
```

9.36. test_code_gen.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

result=true

INPUT_FILES="code_gen/*.in"
printf "${CYAN}Running code_gen tests...\n${NC}"



for input_file in $INPUT_FILES; do
    output_file=${input_file/.in/.out}
    sh ./circline.sh $input_file | cmp -s $output_file -
    if [ "$?" -eq 0 ]; then
       printf "%-65s ${GREEN}SUCCESS\n${NC}" "  - checking $input_file..."
    else
       printf "%-65s ${RED}ERROR\n${NC}" "  - checking $input_file..." 1>&2
        result=false
    fi
done

exit 0

# if $result; then
#      exit 0
# else
```

```
#       exit 1
# fi
```

9.37. ../circline.sh

```
# Check whether the file "utils.bc" exist
file="utils.bc"
if [ ! -e "$file" ]
then
        clang -emit-llvm -o utils.bc -c ./compiler/lib/utils.c -Wno-varargs
fi

if [ $# -eq 1 ]
then
    ./compiler/circline.native <$1 >a.ll
else
    ./compiler/circline.native $1 <$2 >a.ll
fi
clang -Wno-override-module utils.bc a.ll -o $1.exe
./$1.exe
rm a.ll
rm ./$1.exe

# /usr/local/opt/llvm38/bin/clang-3.8
```

9.38. Makefile

```
# circling: Makefile
#  - main entry point for building compiler and running tests

default: build

all: clean build

build:
	cd compiler; make

test: clean build
	cd tests; make

.PHONY: clean
clean:
	cd compiler; make clean
	cd tests; make clean
```