

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**

[Abstract 2](#_Toc470026910)

[1. Introduction 5](#_Toc470026911)

[1.1. Background 5](#_Toc470026912)

[1.2. Aims and Objectives 5](#_Toc470026913)

[2. Tutorial 6](#_Toc470026914)

[2.1. Hello World 6](#_Toc470026915)

[2.2. Graph Creation 6](#_Toc470026916)

[3. Language Reference Manual 8](#_Toc470026917)

[3.1. Types and Literals 8](#_Toc470026918)

[3.1.1. Primitive Types 8](#_Toc470026919)

[3.1.2. Node 9](#_Toc470026920)

[3.1.3. Graph 9](#_Toc470026921)

[3.1.4. List 12](#_Toc470026922)

[3.1.5. Dict 13](#_Toc470026923)

[3.2. Operators and Expressions 14](#_Toc470026924)

[3.2.1. Comments 14](#_Toc470026925)

[3.2.2. Identifiers 14](#_Toc470026926)

[3.2.3. Arithmetic Operators 14](#_Toc470026927)

[3.2.4. Logical and Relational Operators 15](#_Toc470026928)

[3.2.5. List Operators 15](#_Toc470026929)

[3.2.6. Dict Operators 17](#_Toc470026930)

[3.2.7. Graph 18](#_Toc470026931)

[3.3. Control Flow 23](#_Toc470026932)

[3.3.1. Loops 23](#_Toc470026933)

[3.3.2. Conditionals 23](#_Toc470026934)

[3.4. Program Structure 24](#_Toc470026935)

[3.4.1. Functions 24](#_Toc470026936)

[3.4.2. Scoping & Nested functions 26](#_Toc470026937)

[4. Project Plan 26](#_Toc470026938)

[4.1. Ocaml Style 27](#_Toc470026939)

[4.2. Circline Style 27](#_Toc470026940)

[4.3. Project Timeline 28](#_Toc470026941)

[4.4. Roles and Responsibilities 29](#_Toc470026942)

[4.5. Commits 30](#_Toc470026943)

[4.6. Code Frequency 31](#_Toc470026944)

[5. Language Architecture 32](#_Toc470026945)

[5.1. Scanner (scanner.mll) 33](#_Toc470026946)

[5.2. Parser (parser.mly) 33](#_Toc470026947)

[5.3. Optimizer (organizer.ml) 34](#_Toc470026948)

[5.4. Semantic Check (semant.ml) 35](#_Toc470026949)

[5.5. Code Generator (codegen.ml) 35](#_Toc470026950)

[5.6. C library 35](#_Toc470026951)

[6. Testing 37](#_Toc470026952)

[6.1.1. Examples 37](#_Toc470026953)

[6.1.2. Automated Test Suite 42](#_Toc470026954)

[7. Conclusion 45](#_Toc470026955)

[8. Lessons Learned 45](#_Toc470026956)

[9. Appendix 46](#_Toc470026957)

[9.1. Scanner.mll 46](#_Toc470026958)

[9.2. Parser.mli 47](#_Toc470026959)

[9.3. Parser.mly 48](#_Toc470026960)

[9.4. Ast.ml 52](#_Toc470026961)

[9.5. Cast.ml 54](#_Toc470026962)

[9.6. Organizer.ml 56](#_Toc470026963)

[9.7. Semant.ml 59](#_Toc470026964)

[9.8. Codegen.ml 71](#_Toc470026965)

[9.9. Circline.ml 89](#_Toc470026966)

[9.10. Circline.sh 89](#_Toc470026967)

[9.11. Parserize\_cast.ml 90](#_Toc470026968)

[9.12. compiler.Makefile 92](#_Toc470026969)

[9.13. Cast.h 93](#_Toc470026970)

[9.14. cast.c 94](#_Toc470026971)

[9.15. Hashmap.h 95](#_Toc470026972)

[9.16. Hashmap.c 96](#_Toc470026973)

[9.17. List.h 106](#_Toc470026974)

[9.18. List.c 107](#_Toc470026975)

[9.19. Utils.h 111](#_Toc470026976)

[9.20. Utils.c 113](#_Toc470026977)

[9.21. Main.c 128](#_Toc470026978)

[9.22. Parser Test Cases 129](#_Toc470026979)

[9.23. Parser Test Makefile 135](#_Toc470026980)

[9.24. Parserize.ml 135](#_Toc470026981)

[9.25. Scanner Test Cases 138](#_Toc470026982)

[9.26. Scanner Test Makefile 141](#_Toc470026983)

[9.27. tokenize.ml 142](#_Toc470026984)

[9.28. Semantic Check Test Cases 143](#_Toc470026985)

[9.29. Semantic Check Makefile 164](#_Toc470026986)

[9.30. semantic\_check.ml 165](#_Toc470026987)

[9.31. Code Generator Test Cases 165](#_Toc470026988)

[9.32. Codegen Test Makefile 209](#_Toc470026989)

[9.33. test\_scanner.sh 209](#_Toc470026990)

[9.34. test\_parser.sh 210](#_Toc470026991)

[9.35. test\_semantic.sh 210](#_Toc470026992)

[9.36. test\_code\_gen.sh 211](#_Toc470026993)

[9.37. ../circline.sh 212](#_Toc470026994)

[9.38. Makefile 212](#_Toc470026995)

# Introduction

## 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.

## 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.

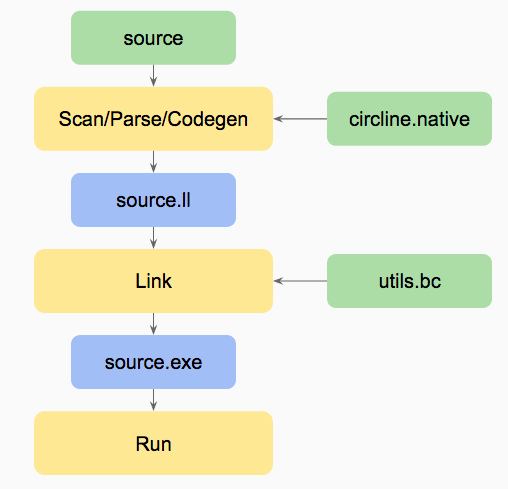
As an example, to create a graph with three nodes (a, b, c), where a is root node and link with node b and c. We could easily define it like:

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.

# Tutorial



## 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! |

## 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.

# Language Reference Manual

## Types and Literals

### 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 |

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

1. cast

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

The cast functions could retrieve the value in the node.

### 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.

#### 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 |

#### 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  -------------------------------------- |

### 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] );  print( [1.2, -3.4] );  print( ["a", "ab", "abc"] );  print( [true, false, 1>2] ); | => list:[1, 2, 3, 4]  => list:[1.200000, -3.400000]  => list:[a, ab, abc]  => 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<float> lf = [1.2, -3.4];  print(lf);  list<string> ls = ["a", "ab"];  print(ls);  list<bool> lb = [1<2, 1>2];  print(lb); | => list:[1, 2, 3, 4]  => list:[1.200000, -3.400000]  => list:[a, ab]  => 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<graph> lgh = [a, b->c ];  print( lgh.get(0).size() );  print( lgh.get(1).size() ); | => list:[1.000000, 2.000000, 3.000000]  => 1  => 2 |

### 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 \*/ |

## Operators and Expressions

### 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.

### 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 |

### 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 + []. |

### 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.

### List Operators

#### .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 |

#### .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 |

#### .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”] |

#### .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 |

#### .pop()

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

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

#### 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 |

#### .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);  aList.remove(-1); | => [“fat”, “cat”, “cat”]  => [“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] |

### Dict Operators

#### .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” |

#### .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 |

#### .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” }\*/

#### .size()

Return the size of Dictionary

#### .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);  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)); | => {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 |

### Graph

#### Definition

There are three link operators:

1. “--” Double Link
2. “->” Right Link
3. “<-” Left Link

##### <node> <op> <Edge Value> & <node> => <graph>

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

##### <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)

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

##### <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] ]

#### Methods

##### .root()

Return the root of a graph.

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

##### .size()

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

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

#### Operator

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

##### 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 |
|  |  |

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

##### 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.

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

## Control Flow

### Loops

|  |  |
| --- | --- |
| **while loops** | |
| dict<int> di = {0:0};  int i = 1;  while (i<3) {  di.put(i, i);  i = i + 1;  }  print(di); | => {2: 2, 1: 1, 0: 0} |
| **for loops** | |
| list<int> li = [0];  int i;  for (i=1; i<5; i=i+1) {  li.add(i);  }  print(li); | => list:[0, 1, 2, 3, 4] |

### 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 |

## Program Structure

### Functions

#### 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 |

#### 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;  } |

### 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 \*/ |

# 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.

## 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.

## 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.

## Project Timeline

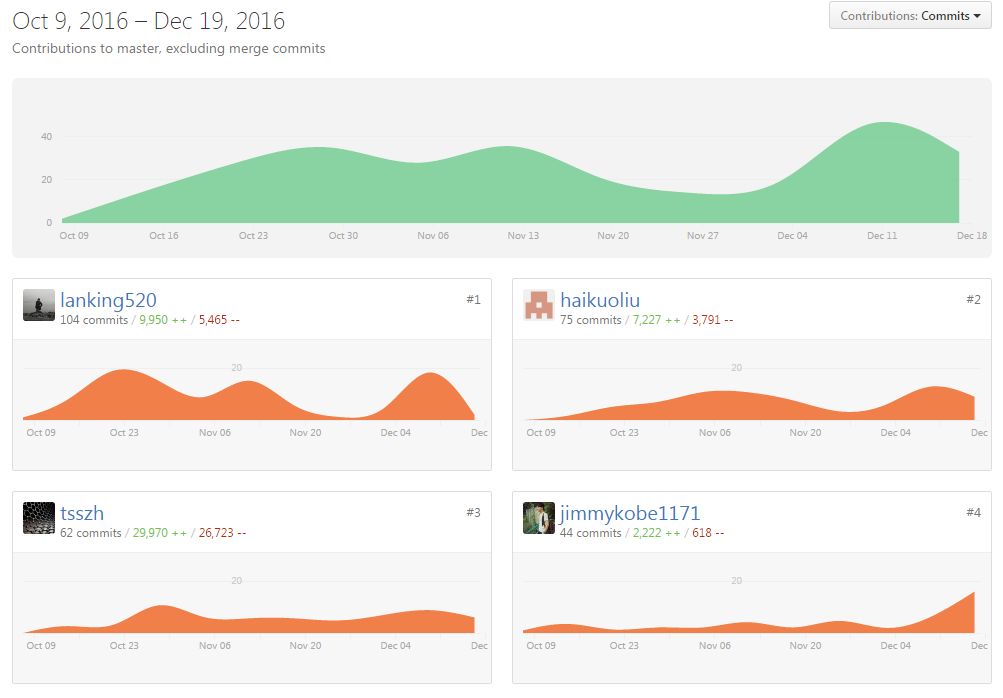
|  |  |
| --- | --- |
| Time | Achievement |
| 10.22 | Finish Scanner and Ast |
| 10.23 | Finish Tokenize and test files |
| 10.25 | Established Travis CI online testing |
| 10.31 | Build Parser and parserize file for testing |
| 11.1 | Makefile Linking the whole project |
| 11.6 | Parser second step 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>

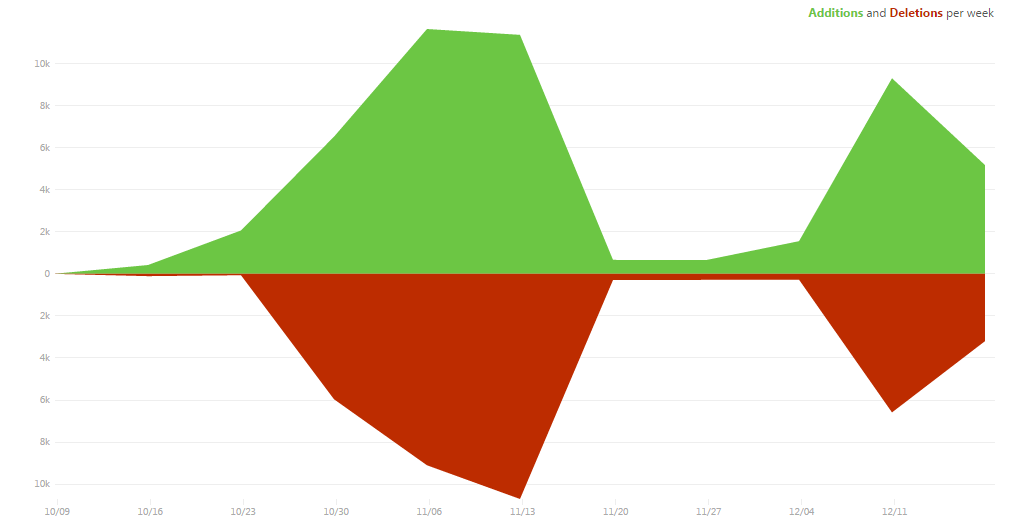
## Roles and Responsibilities

|  |  |
| --- | --- |
| **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 |

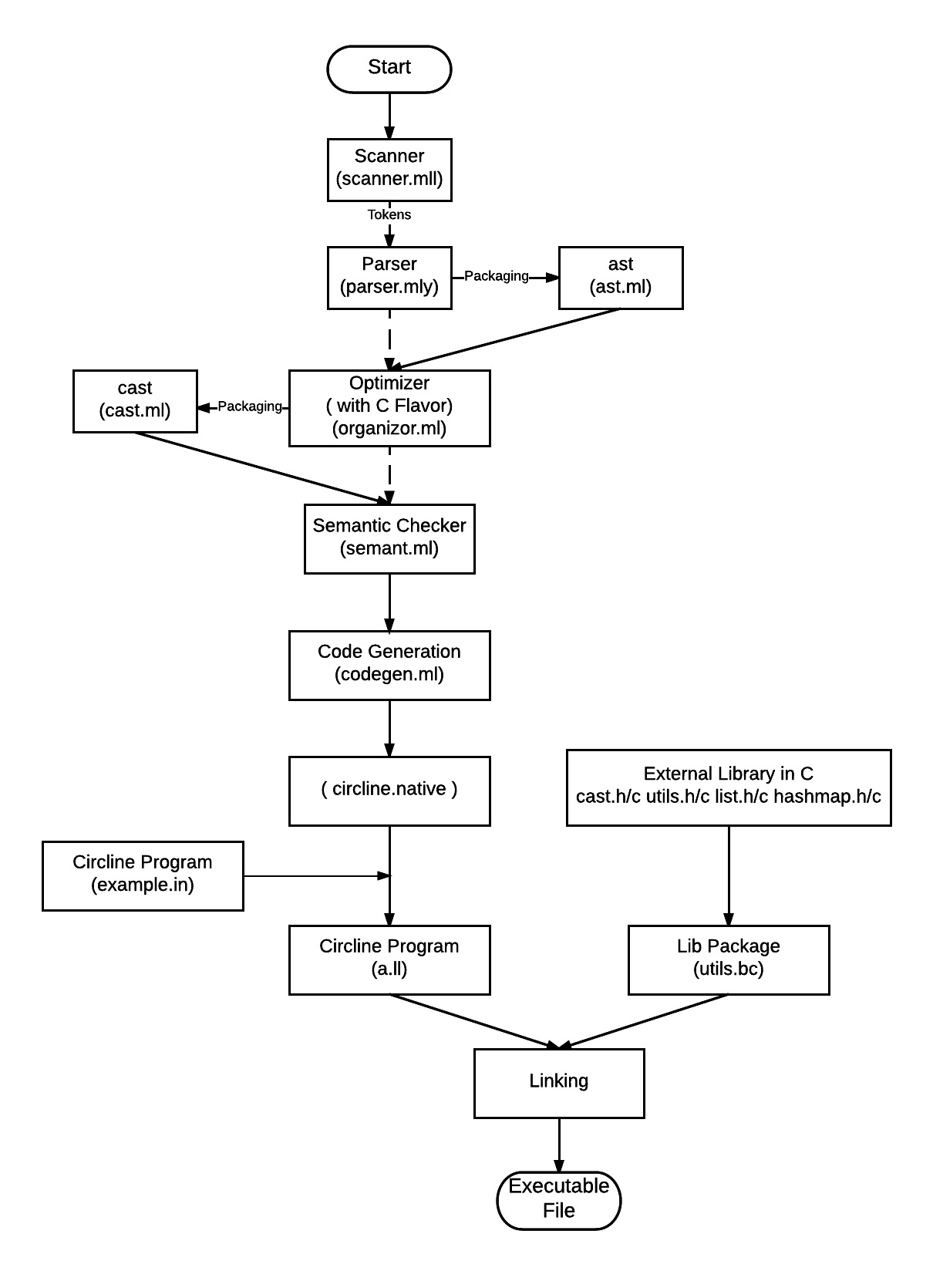
## Commits



## Code Frequency



# Language Architecture



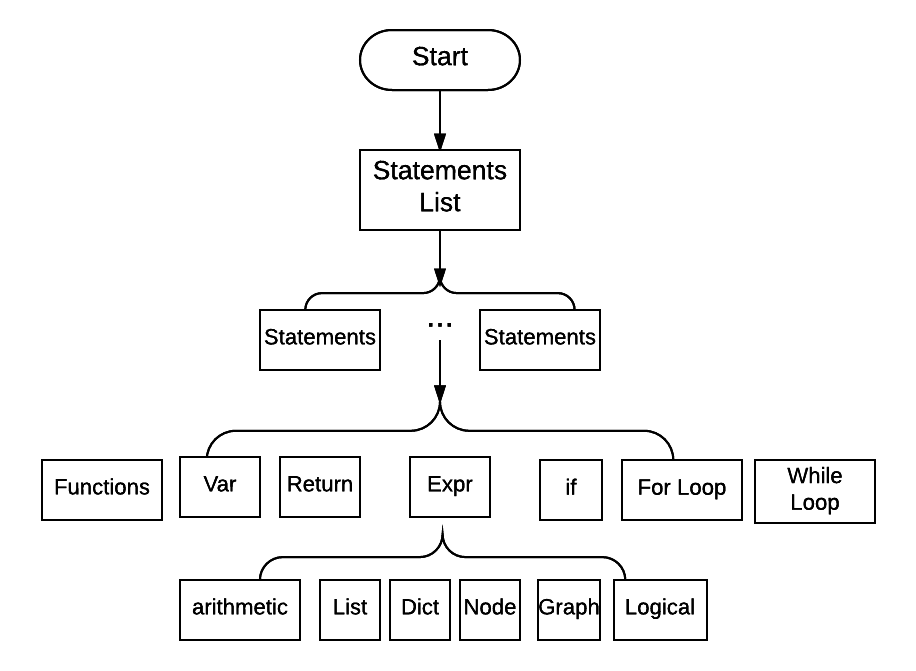
## Scanner (scanner.mll)

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 \*/).

## 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:



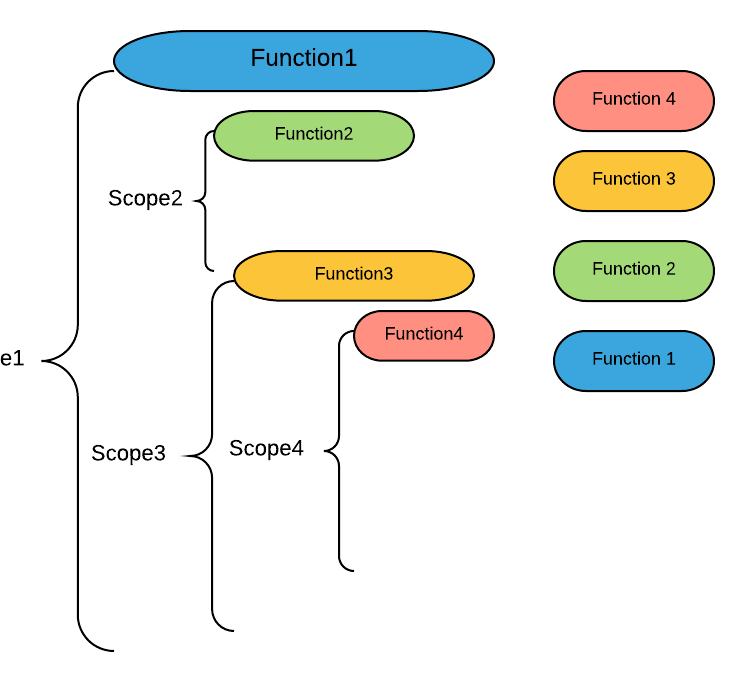
## Optimizer (organizer.ml)

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.

## 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.

## 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
   1. Declare all variables and stored into name-variable map
   2. Translate each statement
      1. 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.

## C library

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

1. List Operation
   1. create list
   2. concat list
   3. list add/push elements
   4. list get elements
   5. list setelements
   6. list remove/pop elements
   7. get list size
   8. print out list
2. Dict Operation
   1. create dict
   2. dict put key-value pairs
   3. dict remove key-value pairs
   4. dict get value
   5. check key existence in dict
   6. get dict size
   7. get list of keys
   8. print out dict
3. Node Operation
   1. create node
   2. get node value
4. Graph Operation
   1. create graph
   2. add nodes / edges
   3. remove nodes
   4. graph merge
   5. graph subtraction
   6. graph get / set root
   7. graph get all nodes
   8. get neighbors of node in the graph
   9. graph get edges
   10. print out graph

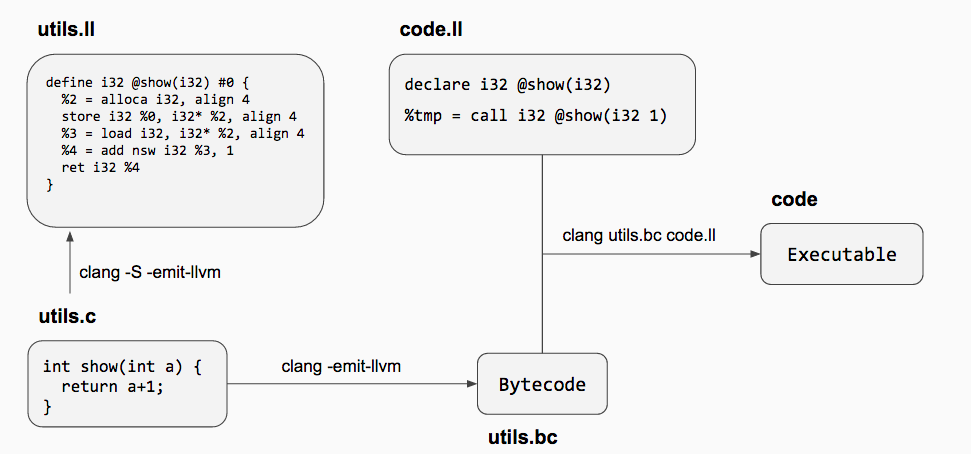
How to use the C library in ocaml code generator? Here is a brief illustration.

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**.



# Testing

### Examples

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

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

#### > Dijkstra’s Algorithm for Shortest Path

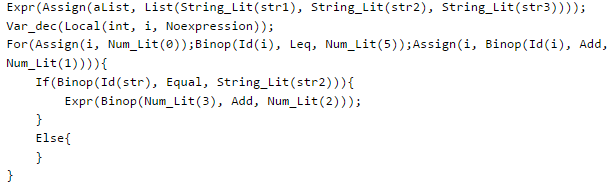
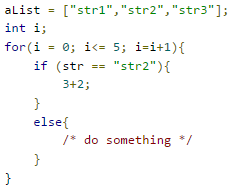
|  |
| --- |
| void dijkstra(graph gh, node sour) {  dict<int> distance = { sour: 0 };  list<node> queue = gh.nodes();  dict<node> parent = {sour: sour};  int i;  for (i=0; i<queue.size(); i=i+1) {  distance.put(queue.get(i), 2147483647);  parent.put(queue.get(i), null);  }  distance.put(sour, 0);  while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

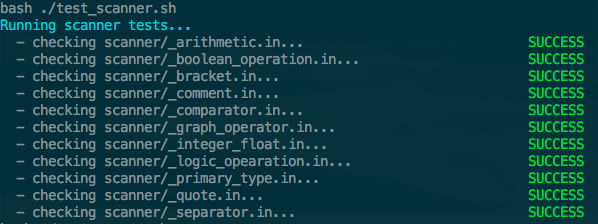
### Automated Test Suite

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.

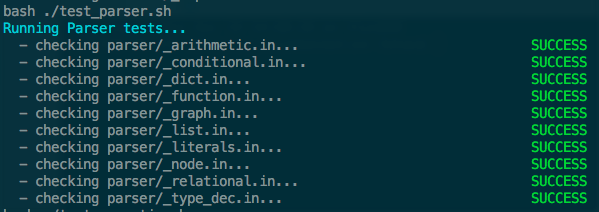


In order to test it automatically, all of the test file were linked with makefile.

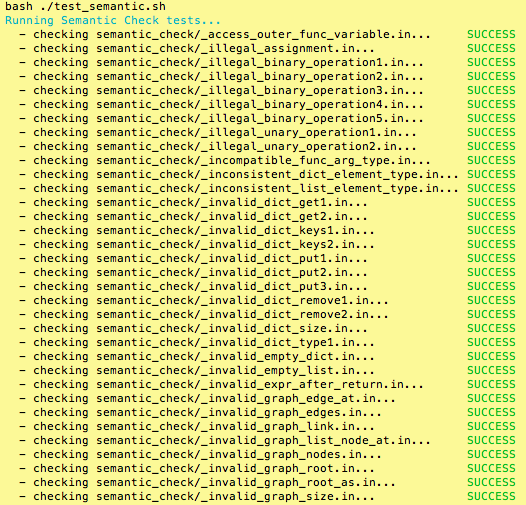
#### Scanner Test Cases

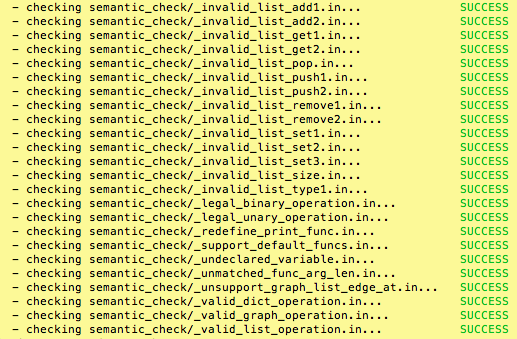


#### Parser Test Cases



#### Semantic Check Test Cases





#### Code Generator Test Cases



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

# 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.

# 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.

# Appendix

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

## 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} |

## 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 |

## 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) } |

## 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 |
|  |

## 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 |
|  |

## 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) |
|  |

## 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 |
|  |

## 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 |
|  |

## 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) |

## 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 |

## 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 |
|  |

## 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 |
|  |

## 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); |
|  |

## 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);}; |
|  |

## 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\_\_ |
|  |

## 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;  // } |
|  |

## 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 |
|  |

## 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)));  // } |
|  |

## 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\_ \*/ |
|  |

## 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 );  // } |
|  |

## 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;  } |
|  |

## 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()));

## 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 |

## 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 |

## 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

## 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: |

## 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) |

## 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

## 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/ast.cmo ../../compiler/\_build/cast.cmo ../../compiler/\_build/organizer.cmo ../../compiler/\_build/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 |

## 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)) \*) |

## 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

## 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 |

## 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 |

## 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 |

## 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 |

## 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 |

## ../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 |

## 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 |