COMS 4115 Programming Translator and Translator

Language Reference Manual

Manager:	**Jia Zhang**
System Architect:	**Haikuo Liu**
Language Guru:	**Zehao Song**
Tester:	**Qing Lan**

Table of Contents

1 Introduction 3

2 Types and Literals 3
- 2.1 Primitive Types 3
- 2.2 Node 4
- 2.3 Graph 4
  - 2.3.1 Undirected Graph 4
  - 2.3.2 Directed Graph 5
- 2.4 List 7
- 2.5 Dict 7

3 Operators and Expressions 7
- 3.1 Comments 7
- 3.2 Identifiers 8
- 3.3 Arithmetic Operators 8
- 3.4 Logical and Relational Operators 8
- 3.5 List Operators 9
  - 3.5.1 Size 9
  - 3.5.2 Select 9
  - 3.5.3 Append 9
  - 3.5.4 Concat 10
  - 3.5.5 Delete 10
- 3.6 Dict Operators 10
  - 3.6.1 Add or Update key-value 10
  - 3.6.2 Remove key-value 11
  - 3.6.3 Get key-value 11
- 3.7 Node & Graph Operators 11
  - 3.7.2 Iteration 11
  - 3.7.3 Link and Merge 13

4 Control Flow 17
- 4.1 Loops 17
- 4.2 Conditionals 17

5 Program Structure 18
- 5.1 Import 18
- 5.2 Functions 18
- 5.3 Scoping 19

1 Introduction

Graph is an important data structure in computer science and have a variety of application in the real world. But current computer language is not convenient to define and present. In most case, you need to define a set of vertexes and edges in order to define a graph, which is not straightforward and sometimes annoying.

To facilitate the use of graph, we create an language called Circline. Circline has easier syntax for describing a graph. For 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.

Circline support directed and undirected graph with edge value, graph merging and other graph related operations. Using Circline, you can easily create graph and do some manipulation with graphs like building a binary tree and performing traversal.

The last but not the least, Circline support plotting graph. Whenever you create a graph, you could plot it using the simple "plot" function. Using the plotting function, you could visualize your complicated graph data structure in a more simple way.

2 Types and Literals

2.1 Primitive Types

Name	**Prefix**	**Description**
Boolean	bool	true | false Example: true false
Integer	int	Possible value: 32-bit signed Integer (-2147483638 ~ 2147483647) Example: -123 43 0

Floating point	float	<u>Possible value</u>: A IEEE 754 double-precision (64-bit) numbers <u>Example</u>: 0.356 3.4e-16 1.
String	string	<u>Possible value</u>: A sequence of ASCII enclosed by double quotes <u>Example</u>: “I’m Haikuo! Talent Guy!” “” “Hello world!\n”
Null		A type represent ‘nothing’ <u>Example</u>: null

2.2 Node

Node is used to define a point in the graph, it could be linked to other nodes or graph, and could be assigned any kinds of type value. However, each node could only store a single value. If multiple information are required, dict type could be used.

node(1)
node(true)
node(4.5)
node(“Hello world!”)
node([1, 2, 3])
node({ name: “circline”, value: 123 })
node(a) /* Keep a reference to other nodes */

2.3 Graph

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

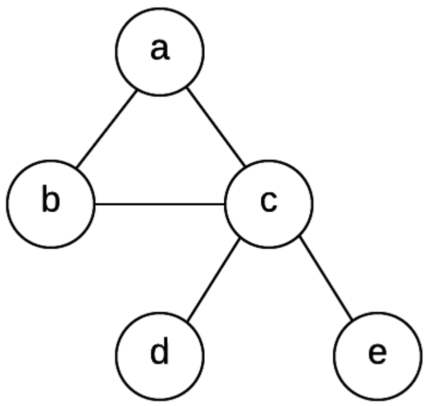
1. All nodes shown in the graph.
2. The Edge (direction, value), by which the nodes are connected.

Examples: (a, b, c are all references to nodes)

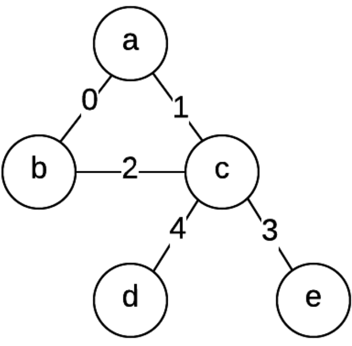
2.3.1 Undirected Graph

a -- b -- c -- [a,d,e]

/* Define undirected graph without edge value. */



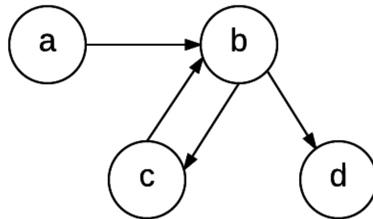
a -- b&0 -- c&2 -- [a&1, d&3, e&4]
/* Define undirected graph with edge value */



2.3.2 Directed Graph

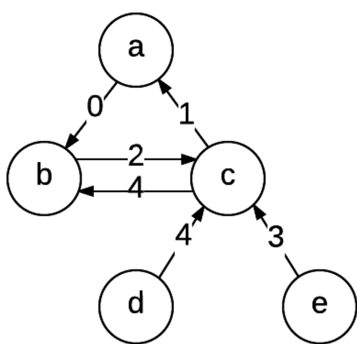
a -> b -> [c -> b, d]

/* Define a simple directed graph, which could be defined by a single statement */

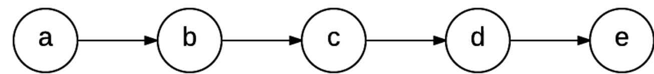


a -> b&0 -> c&2 -> a&1 + d -> c&3 -> b&4 + e -> c&4

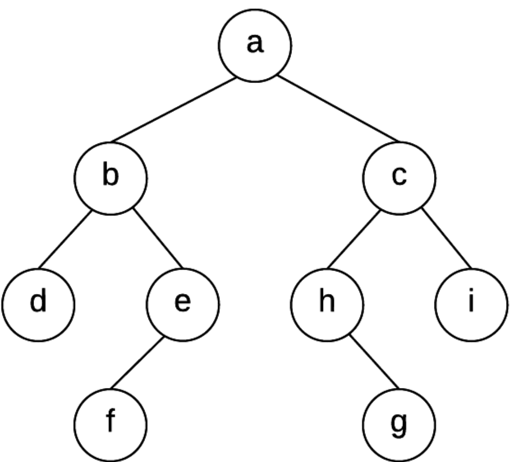
/* Define a complicated directed graph with edge values, which cannot be defined by a single statement, through graph merging. */



a -> b -> c -> d -> e
/* Define a linked list */



bool l = true;
bool r = false;
a -> [b&l -> [d&l, e&r -> f&l], c&r -> [h&l -> g&r, i&r]]
/* Define a binary tree Since the edge of BST has direction, assign a direction value for each edge. */



Attention: <node> will be automatically converted to graph with single node when necessary. Examples:

graph gh = node(0);

graph[] ghList = [node(0), node(1) -- node(2)]

2.4 List

List literals are a sequence of literals enclosed in square braces. The items are separated by commas or semicolons. The list in our language support the pop, push, removeFirst operation, which is very convenient for users to use it as a stack. For example, these are valid lists:

[1, 2, 3, 4]
[“a”, “ab”, “abc”]
[“True”, “False”]

This is not allowed:

[1, “apple”, 3]

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.

2.5 Dict

Dict defines a list of key-value pairs. The key-value pairs in it is are enclosed by curly braces, and are separated by commas. For a given key-value pair, the key should be subject to the convention of the identifiers, and it should be unique. For a given dict, the key could be only one type, which can be chosen from all types in Circlin, and we should declare it. But the value could be any type in all circumstances. A key-value pair should contains both key and value, no single key or single value is allowed.
Here are some examples:
dict<string> = {“name1”: “circle”, “name2”: 1}
dict<int> = {1: “circle”, 2: 1}
dict<int> = {1: aDict1, 2: [1,2,34]}
dict<string> = {“name1”:1, 1:”name2”} /* error */

3 Operators and Expressions

3.1 Comments

Only one type of the comment is accepted, enclosed by “/*” and “*/”, such as
/* write an Ocaml a line */
or
/* Write an Ocaml a line,

Keep you happy a day */
The elements in the middle are automatically ignored.

3.2 Identifiers

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

Valid names:
Merge_sort, apple, a3b_a21, a12, I_Like_Ocaml

Invalid names:
_bash, 1st, 3a5, I-Like-ocaml

3.3 Arithmetic Operators

The arithmetic operators are +, -, *, /, %.
The + and - operators have same precedence, * and / have same precedence. * and / have a higher precedence than + and -. We support automatic promotion of int and float, which we can have int + int, float + float and int + float, same as other arithmetic operators. But other primary types are not allowed as for the arithmetic operators.
Valid input:
1 + 2
1 - 2
1 * 2
1 / 2
1.0 + 2.0
1.0 * 2, 1 / 2.0
1002 % 2

Invalid Input:
1.0 + true
1 + “”
1 + {}
1 + [].

3.4 Logical and Relational Operators

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

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

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

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

3.5 List Operators

3.5.1 Size

Size operator can return the length of a list, it’s used as aList.size.
int[] aList = [1, 2, 3, 4];
aLisrt.size /* 4 */

3.5.2 Select

A specific element can be selected and used by using the index of the element:
int[] aList = [1, 2, 3, 4];
aList[1]; /* 2 */
To select multiple elements, we can use the colon. Its syntax is like aList[:index], where index is between [0, aList.size], and this will give you a new list whose elements are aList[0], aList[1]...aList[index - 1].
aList[:2]; /* [1,2] */
aList[:4]; /* [1,2,3,4] */
aList[:5]; /* error */
Besides, user can get the last element of a list by setting the index as -1, but this is the only allowed negative index in the list.
aList[-1]; /* 4 */
aList[-2]; /* error */
Furthermore, if user wants to get a segment of the list, he can use aList[i,j], the returned results will include aList[i]...aList[j - 1].
aList[1:3]; /* [1,2] */

3.5.3 Append

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.
Valid inputs:
[1] + 2; /* [1, 2] */
[“str1”] + “str2”; /* [“str1”,”str2”] */
Invalid inputs:
[1] + “str” /* error */

3.5.4 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]; /* [1,2,3,4,5,6] */
[“str1”,”str2”,”str3”] + [“str4”,”str5”,”str6”]; /* [1,2,3,4,5,6] */
[“str1”,”str2”,”str3”] + [1,2,3]; /* error */

3.5.5 Delete

User can delete a specific element of the list using .remove(index), where index should be a number between 0 and the size of the list minus 1, for example:

string[] aList = [“big”, “fat”, “cat”, “cat”];
aList.remove(0); /*[“fat”, “cat”, “cat”] */
aList.remove(-1); /* error */
aList.remove(3); /* error */

Pop
Pop operation can be interpreted as aList.remove(aList.size - 1), and it’ll return the elements which is removed. It’s used as aList.pop.

Push
Push operation is used as aList.push(new element), and it can be interpreted as aList = aList + new element.

RemoveFirst
RemoveFirst operation is used as aList.removeFirst, it can be interpreted as aList = aList[1:aList.size].

3.6 Dict Operators

3.6.1 Add or Update key-value

To add a new value in the dict, we can simply use syntax aDict[keyname] = valuename, for which there is no same keyname in the dict before adding.
To update a key-value pair, we use a similar syntax as add, difference is that the keyname has already existed in the dict and we update the value with the keyname.

dict<string> aDict = {};
aDict[“pig”] = true; /* aDict = {“pig”:true} */
aDict[“dog”] = “bob”; /* aDict = {“dog”:”bob”, “pig”:true} */
aDict[“dog”] = “sam”; /* aDict = {“dog”:”sam”, “pig”:true} */

3.6.2 Remove key-value

aDict.remove(keyname) allows user to remove a specific key-pair value. However, the keyname should exist, otherwise there will be an error.
aDict.remove(“pig”); /* {“dog”:”sam”} */
aDict.remove(“pig”); /* {“error”} */

3.6.3 Get key-value

aDict[keyname] returns the value of a key, and the keyname should exist. Although this is similar to getting a value in a string, we’ve already defined the type of the variable, so it’s not a problem.
aDict[“dog”]; /* “sam” */
aDict[“dod”]; /* error */

3.7 Node & Graph Operators

3.7.1.1 Graph Utilities

3.7.1.2 <graph>.root => <node>

Return the root of a graph.

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

3.7.1.3 <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.
((a -- b -- c) ~ b).root => b

3.7.1.4 <graph>.size => <int>

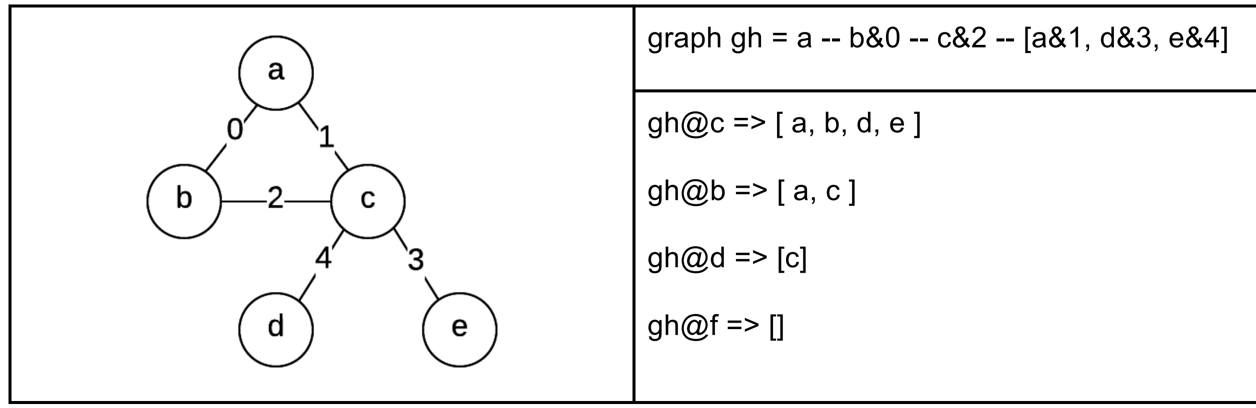
Return the size of the graph.

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

3.7.2 Iteration

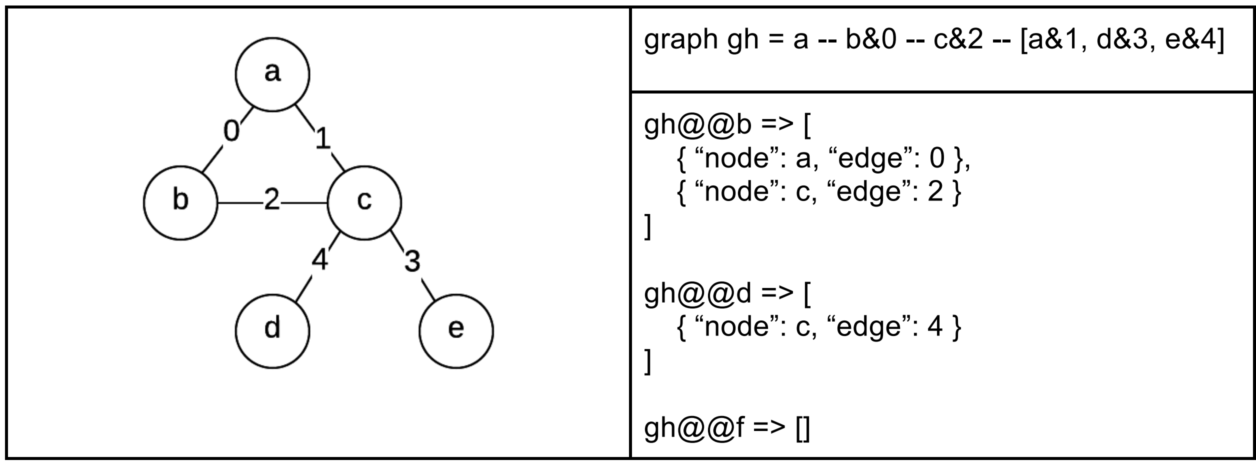
3.7.2.1 <graph> @ <node> => <node>[]

Return a list of nodes, which are children of the specific node in a certain graph.



3.7.2.2 <graph>@@<node> => dict<string>[] /* <dict<string>>.node = <node>, <dict<string>>.edge = <Edge Value> */

Return a list of node-edge pairs (stored as dict), which are children of the specific node in a certain graph.



3.7.2.3 <graph>@(<node>, <node>) => <Edge Value>

Return the edge value between two nodes in a certain graph.
Attention: Order of the node pairs determines the direction of the edge.

a	graph gh = a -> b&0 -> c&2 -> a&1 + d -> c&3 -> b&4 + e -> c&4

	gh@(a,b) => 0 gh@(b,a) => null gh@(b,c) => 2 gh@(c,b) => 4

3.7.2.4 <graph>.nodes => <node>[]

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

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

3.7.2.5 <graph>.edges => <Edge Value>[]

Return a list of the edge values in the graph with random order, if the edge values are not of the same type, throws an error.

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

3.7.3 Link and Merge

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

Link two nodes together with specified edge value, and return a graph, whose root is the first node.
There are three link operators:

1. "--" Double Link
2. "->" Right Link
3. "<-" Left Link

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

Link the first node and the root of the graph with specified edge value, and return a graph, whose root is the first node.
There are three link operators:

1. "--" Double Link
2. "->" Right Link
3. "<-" Left Link

Attention: The following statements are equal to each other.

1. a -- b&1-- c&2
2. a -- (b--c&2)&1

3.7.3.3 <node> <op> <graph or node>[]&<Edge Value> => <graph>

This is a shorthand of the following formal definition of graph, if the edge value are all of the same.
<node> <op> [<graph or node>&<Edge Value>, ...] => <graph>
There are three link operators:

1. "--" Double Link
2. "->" Right Link
3. "<-" Left Link

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

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

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

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

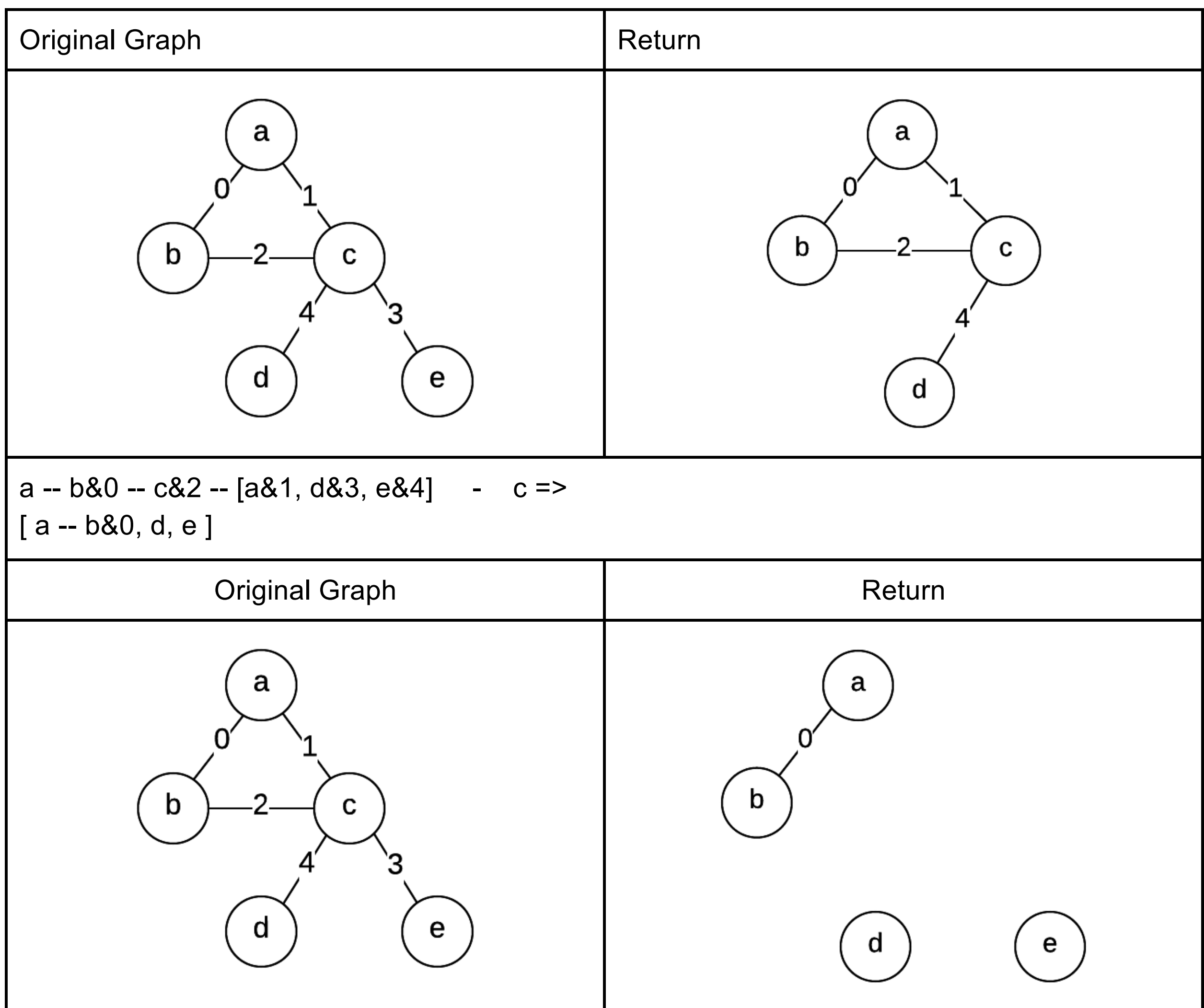
Attention 2: If the edge value are not of the same, must use the full definition.
a -- [b&1, c&2, d&3 -- e&4]

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

3.7.3.4 <graph> <op> <node> => <graph>[]

Remove Nodes:
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.

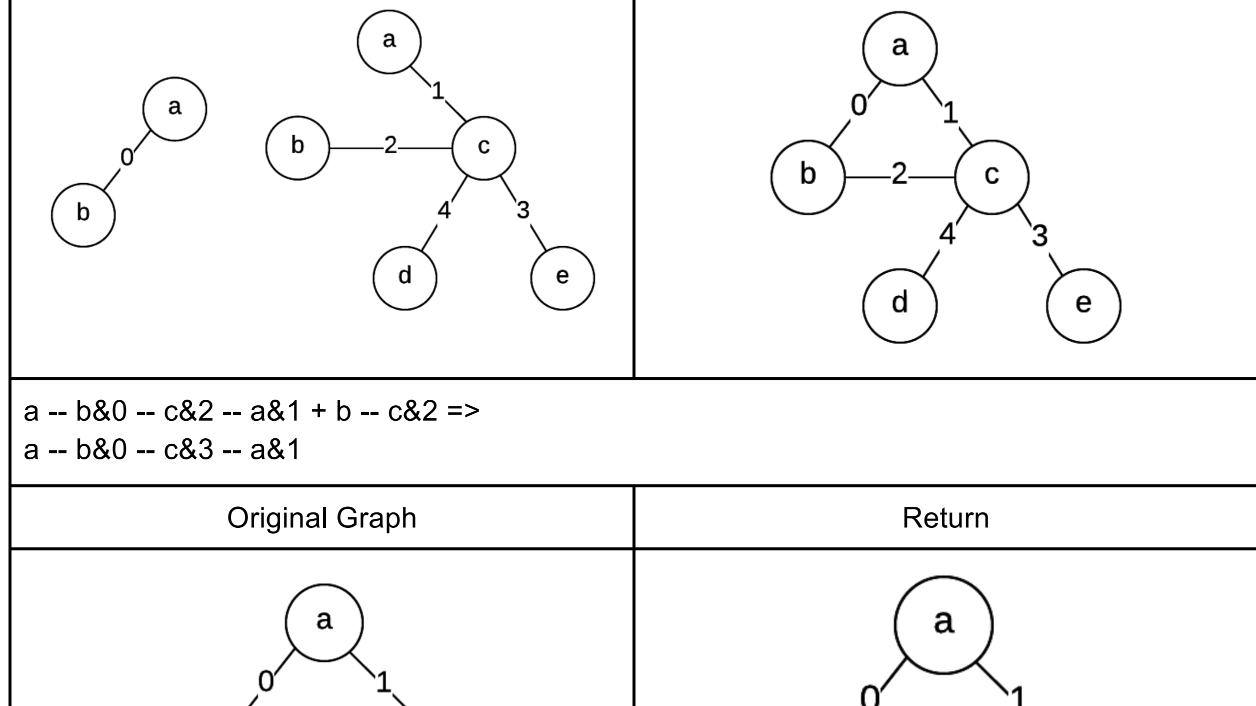
```
a -- b&0 -- c&2 -- [a&1, d&3, e&4]     -    e =>
[ a -- b&0 -- c&2 -- [ a&1, d&4 ] ]
```



3.7.3.5 <graph> + <graph> => <graph>

Merge Graphs / Update Edges:
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.

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



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

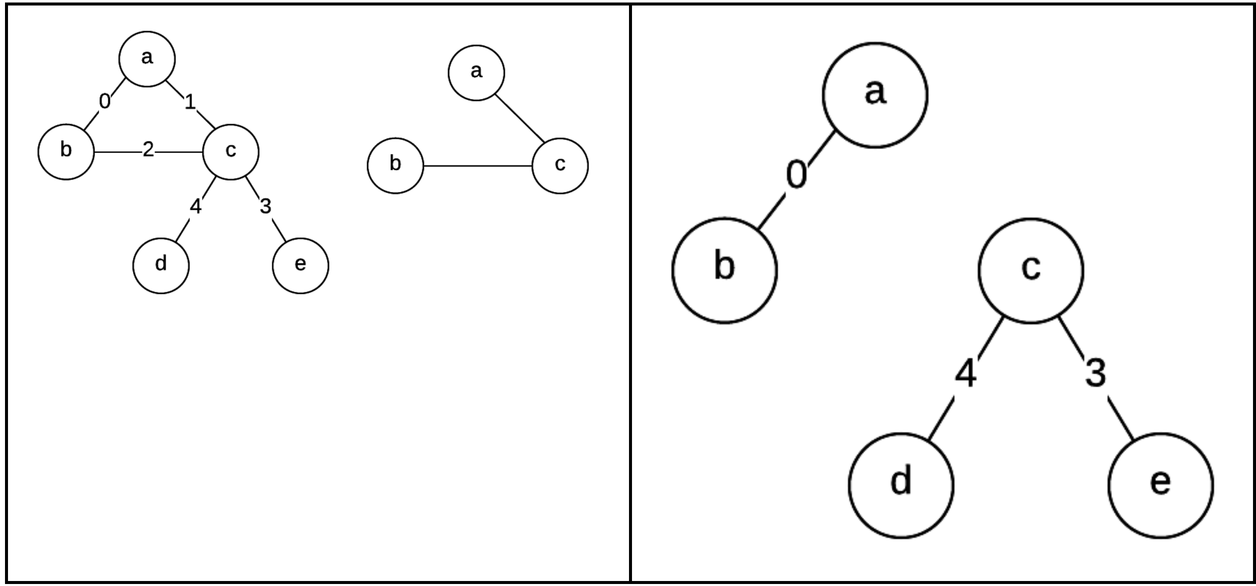
Original Graph	Return

3.7.3.6 <graph> - <graph> => <graph>[]

Remove Edges:
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 -- b&0 -- c&2 -- [a&1, d&4, e&3] - a -- c -- b =>
[a -- b&0, d -- c&4 -- e&3]

Original Graph	Return



4 Control Flow

4.1 Loops

Used as for (type x in list/dict) {/*for statement*/}
We support both the list and dict in for loop.
First, we should declare the type of the element for the list or dict. When we want to iterate through a list, the type should be same as the type of list and should be only one according to the definition of the list; and when we want iterate through the dict, x is each key in the dict and we can get the value of the key using the select future of dict. x could be only used in the body of for statement.

For list, we maintain the order of elements as it is in list. For dict, we don't maintain the order of the keys.

4.2 Conditionals

There are two forms of conditional expressions in our language:
1. if (boolean expression) {statement} else {statement}
2. break: used to break current for loop.
It's basically a java like syntax, but the difference is that the statement must be surrounded in the curly braces.
Here's an example for the control flow:

```
aList = ["str1","str2","str3"];
for(string str in aList){
   if (str == "str2"){
      break;
   }
```

```
    else{
        /* do something */
    }
}
```

5 Program Structure

5.1 Import

Programs begin with imports(if they exist). Import are specified in the following format:
import module_name

When the import is executed, it copies all variables and functions from module. So you could access the variables and function directly by their names instead of adding the module name in front of them. For example, if there is a module name **module1** and it include variables **v1**. Then in another module **module2**, you could use the v1 directly.

```
\* module1  *\
Int v1 = 1;

\* module2  *\
Import module1

Int v2 = v1 + 1 \* v2 should be 2 *\
```

5.2 Functions

Functions are defined using keyword *func*, function could be passed into the argument of another function. Lambda function is supported. To declare lambda function, you could use keyword *lambda*. Below is an example of function declaration and usage.

```
\* function is passed into the argument of function sum *\
func sum( func a, int[] nums) {
        int res = 0
        for (int tmp in nums) {
                res = res + a(tmp);
        }
        return res;
}

int[] nums = [1,2,3];
\* declaration of lambda function and pass it to function sum *\
sum(lambda x : x%2 == 0, nums );
```

```
func b(x) {
	return x%2==0;
}
sum(b, nums);
```

5.3 Scoping

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. However, you could not change the value of global variable inside the function.

```
int a = 1

func foo() {
	print  a \* it will print out 1 *\
	a = 2    \* it will raise exception cause you cannot change the value of global variable *\
}
```