



COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

COMS 4115 Programming Translator and Translator

Circline

An Easy Graph Language

Manager:	Jia Zhang	jz2784
System Architect:	Haikuo Liu	hl3023
Language Guru:	Zehao Song	zs2324
Tester:	Qing Lan	ql2282

Table of Contents

1. Introduction	3
2. Language Features.....	4
Simple Graph Syntax.....	4
Universal Graph Language	4
Flexible Data Structure	4
Easy Graph Rendering.....	4
3. Language Design and Syntax.....	4
Storage of Node & Graph.....	4
Comments	5
Keywords	5
Operator.....	6
Built-in functions	7
Basic Usage.....	10
Define a new Node.....	10
Define a new Graph.....	11
4. Code Examples	13
Linked List	13

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. Language Features

Simple Graph Syntax

Using Circline, most of simple graph (directed or undirected graph, edge with or without values) could be created by a single statement. Combining several simple graphs could generate a complicated graph.

Universal Graph Language

Many graphs are composed of nodes and lines, such as tree and linked list, and Circline provides a more concise way to present graphs by taking advantage of this. Circline is able to define all kinds of graphs with its data structure and draw them with lines and nodes. The language will adjust the graphs' location and size with its built-in algorithms.

Flexible Data Structure

The Core Node function accept multiple data types (int/float, string, boolean ...). Once defining/adding nodes in graph, the data type of different node could also be different. This feature will provide user to design more complicated functional graph. Such as Decision Tree or State Machine.

Easy Graph Rendering

Circline is capable of plotting graph. By using the "plot" function, user could draw their graphs in an easy manner. Circline will handle the graph detail like the distance between nodes and lines automatically and draw the graph using OpenGL.

3. Language Design and Syntax

Storage of Node & Graph

Nodes are stored in a global node pool in memory.

Graphs just keep the reference to each node and stored the information of edges. Thus, if two graphs share the same node, modify the value of the node in one graph would result in the change in another graph.

There is no edge type, since everything regarding the edges could be retrieved by graph-node pair.

Any operation on graphs would generate a new graph.

Comments

Syntax	Comment Style
<code>/* Multiline comments */</code>	Multiline comment
<code>//</code>	Single line comment

Keywords

Key word	Definition
int	Defines an integer, which has ...bit.
float	float represents a float number, which has ...bit.
bool	To define a boolean expression is correct (true) or wrong (false), for true or false only.
string	string keyword is used to represent a sequence of characters, it's included in "".
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.
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.
If else	Used as <code>if(boolean expression) {/*if statement*/} else {/*else statement*/}</code> : Same as if else statement in java.
for	Used as <code>for (initialization; termination;increment) {/*for statement*/}</code> , same as for statement in java.
continue	Continue to the next loop in for loop.
break	break from a for loop when the control flow encounters the break.
map	Defines a list of key-value pairs, the key should be unique. The value of key or value could be any type, but types of same map should be same. <code>map aMap = {name: "circle", value: 1};</code>

list	Defines a list of value, which could be any type. list aList = [1, 2, "my type", {name: 'circline'}];
null	null could be applied to node and graph , it means a non-existing node or an empty graph. Example Use: graph gh = node(1) -> null; // Create a graph with single node.
void	Primary type, means the value havn't been assigned.
print	Print the result value
plot	Plot the Graph

Operator

Name	Operator	Node, Node	Node, Graph	Graph, Graph
MERGE	+	/	Update the node in graph	Merge
ASSIGN	=	Define a New Node	/	Define a new Graph
LINK	--	Node -- Node => Graph	Node -- Root => Graph	/
LINKR	->	Node -> Node => Graph	Node -> Root => Graph	/
LINKL	<-	Node Left Left Link to Node Right	Node <- Root => Graph	/

Name	Operator	Int/Float	String	List	Map
PLUS	+	Adding value from value	Contact two string	Connect the former tail to the latter head	/
MINUS	-	Subtract value from value	/	/	/
MULTIPLY	*	Multiply two	/	/	/

		value			
DIVIDE	/	Value divided value	/-	/	/
ASSIGN	=	Substitute the right value to the left	Assign the right value to the left	Assign an empty list or with some value	Assign a new map variable
AND	&	bool	bool	bool	bool
OR		bool	bool	bool	bool
NOT	!	bool	bool	bool	bool
EQUAL	==	Compare the value	Compare the string	/	/
GT	>	Compare the value	/	/	/
LT	<	Compare the value	/	/	/
GE	>=	Compare the value	/	/	/
LE	<=	Compare the value	/	/	/

Built-in functions

T refers to any type.

API of list (list aL = [])		
Function Name	Function Signature	Description
add	aL.add(T p1)=>list	Add p1 to the list
get	aL.get(int index)=>T	Get existing value from index
remove	aL.remove(int index)	Remove the value from index
size	aL.size() => int	Return the length
concat	aL.concat(list p1)=>list	Combine two lists, the same as (list + list)
set	aL.set(T p1, int index)	Update the value at given index
isEmpty	aL.isEmpty() => bool	The same as aL.size() == 0

API of map (map aM = {})		
Function Name	Function Signature	Description
add	aM.add(T k, T v)=>map	Add (k, v) key-value pair into the map.
get	aM.get(T key)=>T value	Get existing value from key
remove	aM.remove(T key)	Remove the key-value pair
contains	aM.contains(T key) =>boolean	Detect if the key pair exist in the map
size	aM.size() => integer	Get the size of the map (Number of valid key-value pair)
isEmpty	aM.isEmpty() => bool	The same as aM.size() == 0

API of node (list aN = node(T))		
Function Name	Function Signature	Description
get	aN.get()=>T	Return the data stored in node
set	aN.set(T) => node	Update the data stored in node
link	aN.link(node n1, string edge_type, T value) => graph	<p>Link two nodes together by <edge_type> with a edge value of <value>.</p> <p>The root of the graph is always aN.</p> <pre> aN.link(n1, "--", value); => aN -- n1(value); aN.link(n2, "->", value); => aN -> n1(value); aN.link(n2, "<-", value); => aN <- n1(value); </pre>

API of graph(list aG = node(1) -> node(2))		
Function Name	Function Signature	Description
merge	aG.merge(graph g)=>graph	<p>Merge two graph together and return a new graph, the root of the new graph is aG's root. If there are conflicts between two graphs, replace the old graph's nodes or edges with the new graph.</p> <p>aG.merge(g); => aG + g;</p>
link	aG.link(node n1, string edge_type, T edge_value[, node n2]) => graph	<p>Link the node n1 with the graph aG by connecting n1 & n2 by <edge_type> with value.</p> <p>The root of the graph remains unchanged.</p> <p>If n2 is defined yet not existed in aG, throw an error. If n2 is not defined, use Graph.root as default.</p> <p>aG.link(n1, "->", 2); => aG(2) + n1 -> aG.getRoot()(2);</p> <p>aG.link(n1, "<-", 2, n2); => aG + n1 <- n2(2);</p>
getRoot	aG.getRoot() => node	Get the root node.
setRoot	aG.setRoot(node n) => graph	Set the root as the node n in the graph. If n is not existed in the graph, throw an error.
contains	aG.contains(node n) => boolean	Check whether node n is in the graph.
	aG.contains(node n1, node n2[, string edge_type]) => boolean	Check whether edge n1 <edge_type> n2 exists in the graph aG. If edge_type is not given, all of "--", "->" and "<-" are valid.
edge	aG.edge(node n1, node n2) => map	<p>Return the edge between n1 & n2.</p> <p>Example:</p> <p>1. Single Directed Edge aG = n1 -> n2("haha"); aG.edge(n1, n2); => { "->": "haha" }</p> <p>2. Double Directed Edge</p>

		<pre>aG = n1->n2("aa") + n1<-n2("bb"); aG.edge(n1, n2); => {"->": "aa", "<": "bb"}</pre> <p>3. Undirected Edge <pre>aG = n1 -- n2("haha"); => {"--": "haha"}</pre></p>
remove	<code>aG.remove(node n) => graph</code>	Remove node n and it's connecting bonds
size	<code>aG.size()=> integer</code>	Get the number of the nodes in graph
succ	<code>aG.succ(node n)=> list</code>	<p>Return the successors of node n. The return value is a list of map, which contains two key-value pairs:</p> <pre>{ node: <a reference to node>, edge: <data stored at the edge> }</pre> <p>Example: <pre>aG = a -> [b(0), c(1)] + a<-[d(2)]; aG.succ(a); => [{ node: b, edge: 0 }, { node: c, edge: 1 }]</pre></p>
dfs	<code>aG.dfs(node n) => list</code>	Return a list of node reference, following the order of depth first search started at node n.
bfs	<code>aG.bfs(node n) => list</code>	Return a list of node reference, following the order of breadth first search started at node n.
tpsort	<code>aG.tpsort(node n) => list</code>	Topological Sort of graph started at node n. Return a list of node reference.

Basic Usage

Define a new Node

```
node a = node( <node value> )
```

To retrieve the node value, use **a.get()**

<node value> : int, float, bool, string, list or map

Example	Retrieved Value
node a = node(1);	a.get() => 1
node a = node(2.9);	a.get() => 2.9
node a = node("Jobs");	a.get() => "Jobs"
node a = node(true);	a.get() => true
node a = node({ name: "Circline", description: "Powerful Language"});	a.get() => { name: "Circline", description: "Powerful Language"}

Define a new Graph

Define graphs with edge values:

graph gh = node <link type> [<node or graph>]

Define graphs without edge values:

graph gh = node <link type> [<node or graph> (<edge value>), ...]

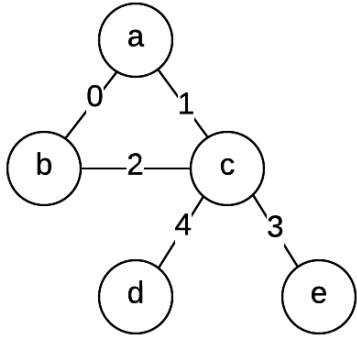
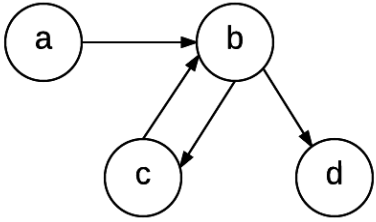
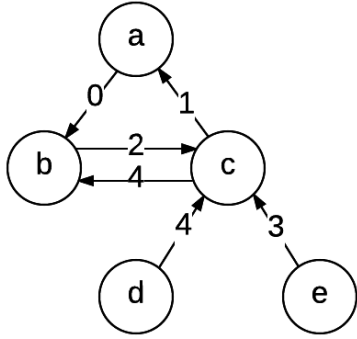
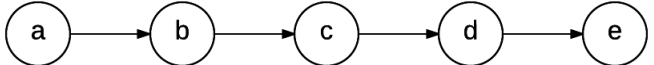
Create graphs via Graph Merging

graph gh = <graph> + <graph>

<link type> : --, -> or <-

<edge value> : int, float, bool, string, list or map

Example	Graph
<pre>graph gh = a -- b -- c -- [a,d,e]; /* Define undirected graph without edge value. */</pre>	<pre> graph TD a((a)) --- b((b)) a --- c((c)) b --- c c --- d((d)) c --- e((e)) </pre>

<pre>graph g = a -- b(0) -- c(2) -- [a(1), d(3), e(4)]; /* Define undirected graph with edge value */</pre>	
<pre>graph gh = a -> b -> [c -> b, d]; /* Define a simple directed graph, which could be defined by a single statement. */</pre>	
<pre>graph gh = 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. */</pre>	
<pre>graph linkedlist = a -> b -> c -> d -> e; /* Define a linked list */</pre>	

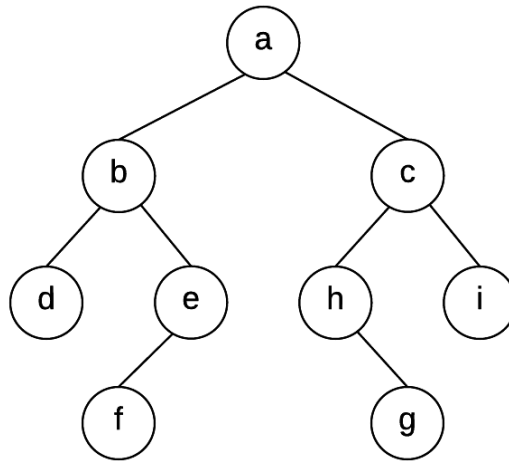
```

bool l = true;
bool r = false;

graph btree =
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.
*/

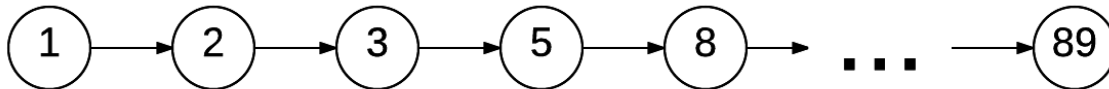
```



4. Code Examples

Linked List

Generate a linked list:



```

// Recursive Version
graph function fibonacci(int n, int prev) {
    if (n > 100) return null;
    return node(n) -> fibonacci(n+prev, n);
}
graph gh = fibonacci(1, 1);

```

```

// Iterative Version
graph gh = node(1) -> null;
node prev = gh.getRoot();
for (int n = 2; n <= 100;) {
    node curr = node(n);

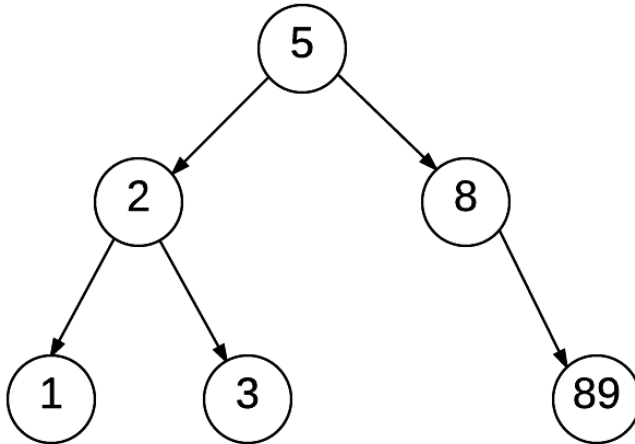
```

```

gh = gh + prev -> curr;
n = n + prev.get();
prev = curr;
}

```

Generate a binary search tree: [5, 2, 1, 3, 8, 89]



```

// Directly Define
bool l = true;
bool r = false;
graph bst = node(5) -> [ node(2)(l) -> [ node(1)(l), node(3)(r) ],
                        node(8)(r) -> node(89)(r) ]

```

```

// Though BST Insertion Algorithm
bool LEFT = true;
bool RIGHT = false;

// Get Left & Right Node helper functions
node getEdge(graph bst, node root, bool direction) {
    list succs = bst.succ(root);
    int i = 0;
    for (; i < succs.size(); i = i + 1) {
        map curr = succs.get(i);
        if (curr.get('edge') == direction) return curr.get('node');
    }
    return null;
}

node getLeft(graph bst, node root) {
    return getEdge(bst, root, LEFT);
}

node getRight(graph bst, node root) {
    return getEdge(bst, root, RIGHT);
}

// BST Insertion Algorithm

```

```

void insert(graph bst, node root, node n) {
    if (root == null) return void;

    node left = getLeft(bst, root);
    node right = getRight(bst, root);

    if (n.get() <= root.get()) {
        if (left == null) return bst.link(n, "<-", LEFT, root);
        return insert(bst, left, n);
    } else {
        if (right == null) return bst.link(n, "<-", RIGHT, root);
        return insert(bst, right, n);
    }
}

// Generate the Tree
list source = [5, 2, 1, 3, 8, 89];
graph bst = node(source.get(0)) -> null;
int i = 1;
for (; i < source.size(); i = i+1) {
    insert(graph, graph.getRoot(), node(source.get(i)));
}

```