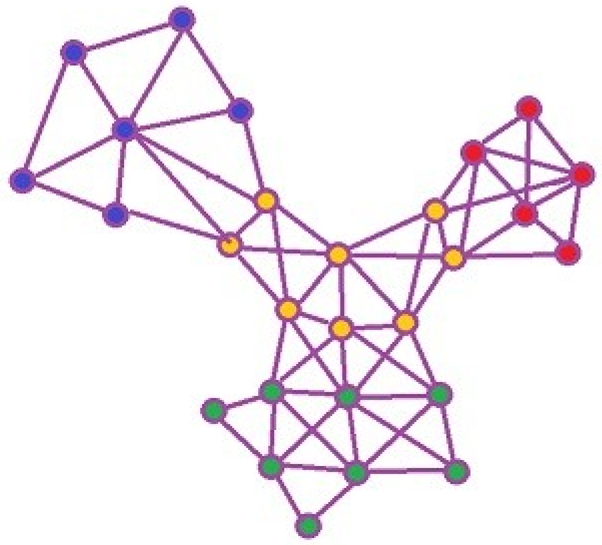
COMS W4115 Programming Languages & Translators

GIRAPHE



Language Reference Manual

Name	UNI
Dianya Jiang	dj2459
Vince Pallone	vgp2105
Minh Truong	mt3077
Tongyun Wu	tw2568
Yoki Yuan	yy2738

1 Lexical Elements

1.1 Identifiers

Identifiers are strings used for naming different elements, such as variables, functions, classes. These identifiers are case sensitive, and can involve letters, digits, and underscore ‘_’, but should always start with a letter. These rules are described by the definitions involving regular expressions below:

identifier	:=	(letter) (letter | digit | underscore)*
digit	:=	‘0’ - ‘9’
letter	:=	uppercase_letter | lowercase_letter
uppercase_letter	:=	‘A’ - ‘Z’
lowercase_letter	:=	‘a’ - ‘z’

1.2 Keywords

int Edge Graph Node def add
else return while bool remove if
new double
for main string char List Map INF
null void

1.3 Literals

1.3.1 String Literals

- string literals are ASCII characters inside double quotation marks. C escape character rules are applied as same.

Escape Sequence	Description
\”	Insert a double quote at this point.
\\	Insert a backslash at this point
\n	Insert a newline at this point
\t	Insert a tab at this point

\r	Insert a carriage return at this point
\b	Insert a backspace at this point
\f	Insert a formfeed at this point

1.3.2 Integer Literals

- Giraffe only uses decimal integer literals, i.e., there is no Hex or Oct numbers. A valid integer literal is either a decimal value in the valid integer value range, +INF or -INF. Also note that +INF and -INF is a single lexeme, i.e., there is no space between +/- and INF.

1.3.3 Boolean Literals

- A boolean literal represents a truth value and can have the value ***true*** or ***false*** (case sensitive).

1.4 Delimiters

1.4.1 Parentheses and Braces

Parentheses are used to force evaluation of parts of a program in a specific order. They are also used to enclose arguments for a function.

1.4.2 Commas

Commas are used to separate function arguments.

1.4.3 Brackets

Brackets are used for array initialization, assignment, and access.

1.4.4 Semicolon

A semicolon is used to terminate a sequence of code.

1.4.5 Curly Braces

Curly braces are used to enclose function definitions, blocks of code, function definitions. In general, blocks enclosed within curly braces do not need to be terminated with semicolons.

1.4.6 Periods

Periods are used for accessing fields of object.

1.4.7 Whitespace

Whitespace (unless used in a string literal) is used to separate tokens, but has no special meaning otherwise. List of whitespace characters: spaces, tabs, newlines, vertical tabs, and formfeed characters.

2 Data Types

Giraffe is statically typed. The types of all variables are known at compile time and cannot be changed.

2.1 Primitive Data Types

int (5)	32-bit signed integers that can range from −2,147,483,648 to 2,147,483,647
double (8.7)	Represented by IEEE 754 double-precision 64-bit number
char (‘Z’)	An ASCII character
str (“Hello!”)	A set of chars
bool (true)	true or false value
null	null
void	Empty return type

2.2 Non-Primitive Data Types

List	A sequence of the same type of data in square braces separated by commas
Map	A hashed set of keys and values of any type
Node	A vertex represents in (x,y)
Edge	A line between two nodes, possibly with weight and/or direction
Graph	A set of nodes and edges

2.2.1 Declaring lists

- Declare a list by specifying a type, followed by brackets enclosing the number of elements in the list, followed by an identifier for the list
- All lists dynamically sized
- The following will declare a list of 10 integers called "myList"

```
int[10] myList;
```

- Or you could do the following for a list of Nodes of length 0

```
Node[] myList;
```

2.2.2 Accessing and setting list elements

- List elements can be accessed by providing the desired index of the element you wish to access enclosed in brackets next to the identifier
- The following will set the element at index 1 of "myList" to 0

```
myList[1] = 0;
```

2.2.3 List length

- The following will return the length of list "myList"

```
myList.len();
```

2.2.4 List add

- To add an element to "myList"

```
myList.add(5);
```

2.2.5 List remove

- To remove the element at index 0 of “myList”

```
myList.remove(0);
```

2.3 Nodes

- To initialize a node with no edges, use the new keyword

```
Node n = new Node();
```

- You can optionally feed the constructor with a graph and list of nodes to create edges with

```
Node n = new Node(g,myList);
```

2.4 Edges

- To connect two nodes with an edge, use the pipe operator

```
n | m;
```

- But be careful, the nodes must exist in a graph together to be linked

2.5 Graphs

- To initialize a graph with no nodes or edges, use the new keyword

```
Graph g = new Graph();
```

- You can optionally feed the constructor with a list of nodes and a list of lists of edges for each node

```
Graph g = new Graph(nodeList, listsOfEdges);
```

- Another option involves using node addition to create a graph

```
n + m;
```

2.6 Map

2.6.1 Declaring a map

- To initialize an empty map

```
Map myMap = new HashMap();
```

2.6.2 Adding key and value

- To add the key and value in the map

```
myMap.put(key, value);
```

2.6.3 Clear map

- To clear all the key and related value in the map

```
myMap.clear();
```

2.6.4 Map size

- To get the size of the map

```
myMap.size();
```

2.6.5 Map remove

- To remove a value associate with a key

```
myMap.remove(key);
```

2.6.6 Get value from map

- To get a value associate with a key

```
myMap.get(key);
```

2.6.7 Check key

- Return true if the map contains the key, otherwise return false

```
myMap.containsKey(key);
```

2.6.8 Check value

- Return true if the map contains the value, otherwise return false

```
myMap.containsValue(value);
```

3 Expressions and Operators

3.1 Expressions

Expressions consist of one or more operands with zero or more operators. Innermost expressions are evaluated first, as determined by grouping into parentheses, and operator precedence helps determine order of evaluation. Expressions are otherwise evaluated left to right.

3.2 Operators

3.2.1 Basic

The table below presents the language operators (including assignment operators, mathematical operators, logical operators, and comparison operators), descriptions, and associativity rules. Operator precedence is highest at the top and lowest at the bottom of the table.

Operator	Description	Associativity
.	Element Access	Left-to-Right
[]	List Access	Right-to-left
!	Logical Not	
* / %	Multiplication, division, remainder	Left-to-right
+-	Addition, subtraction	
< <= > >=	Inequality Operators	
== !=	Comparison Operators	
&&	Logical And	
||	Logical Or	
=	Assignment	Right-to-left

Node/Graph:

+	Node + Node = Graph	Return graph of two nodes
	Graph + Node = Graph	Return graph with node added
-	Graph - Node = Graph	Return graph with node removed
|	Node | Node = Edge	Return edge of newly connected nodes if they are in the same graph

;	end of line
//	Start of a one line comment
/*	Start of a comment block
*/	End of a comment block

4 Control Flow

if (expression) {
   …
}
	Execute the statements between curly braces if expression is true

if (expression) {
   …
} else {
   …
}
	Can have an optional else statement. One can also nest the if-else statement.

while (loop condition) {
         …
}
	The while statement is used to execute a block of code continuously in a loop until the specified condition is no longer met. If the condition is not met upon initially reaching the while loop, the code is never executed

for (initialization; loop condition; increment;) {
       …
}
	Iterate until loop condition is met, incrementing each time

for (*member*   in   *collection*) {
      …
}
	Execute the following block on each element of the collection

continue	Continue to next iteration of the loop
break	Break out of the loop
return	Return signals that what follows it should be the return values
in	Syntax sugar

5 Functions

5.1 Function Definitions

- Function definitions consist of an initial keyword “def,” a return type, a function identifier, a set of parameters and their types, and then a block of code to execute when that function is called with the specified parameters. An example of an addition function definition is as follows:

```
def int sum(int a, int b)  { return  a+b; }
```

5.2 Calling Functions

- A function can be called its identifier followed by its params in parentheses. for example:

```
sum(0,100);
```

6 Program Structure and Scope

6.1 Importing Libraries

Use the #import syntax at the top of a .gf file to import GF libraries. A GF library is a file with extension .gf that consists of a list of GF functions. To get access to a GF library’s functions, simply copy the library file to the lib/ folder if it’s not there already, then add the line “#import [libraryName]” to the top of any GF program that uses that library.

```
#import stdlib
#import typeConversionLib
```

6.2 Scope

Any declarations made within the program that are not within one the block of an if statement, a while statement, and a function definition are available for reference any point later in the program. Declarations made within blocks of an if statement, a while statement, or a function definition are only available for reference within that block. Declarations are never visible to any code that comes before it in the program.

7 Built-in Functions

7.1 The print function

The print function can be used to print out strings, integers, and booleans to the command line. The general structure for calling the print function is as follows:

```
print ( "GIRAPHE" );
print ( 62 );
```

API of Graph(G)		
Name	Function Expression	Description
add	G.add(n)	Add node to graph
merge	G.merge(g)	Merge Graph g with Graph G and return a new graph.
contains	G.contains(n)	Check whether node n is in the graph
size	G.size()	Return the number of nodes in G
path	G.path(n1,n2)	Return shortest path between nodes
is empty	G.isEmpty()	Return whether the graph has node or not
remove	G.remove(n)	Remove node from Graph G
get articulation points	G.articulations()	Get articulation points
get a node	G.getNode(int id)	Returns the node with specified id
get all nodes	G.getAllNodes()	returns the list of nodes in the graph
get all edges	G.getAllEdges()	returns the list of edges in the graph
get edge count	G.getEdgeCount()	returns the number of edges in the graph
get node count	G.getNodeCount()	returns the number of nodes in the

		graph

API of Node(N)		
Name	Function Expression	Description
value	N.value()	Return the value of node
neighbor	N.nbr()	Return a list of all connected nodes in an undirected graph
child	N.child()	Return a list of nodes which are children of the current node in a directed graph
parent	N.parent()	Return the parents of a node in a directed graph
getID	N.getID()	Returns the unique integer ID of the specified node

API of Edge(E)		
Name	Function Expression	Description
start	E.start()	Return the starting point of one edge
end	E.end()	Return the end of one edge
weight	E.weight()	Set the weight of the edge
nodes	E.nodes()	Return a list of two nodes connected by the edge E
label	E.label()	Return the boolean value, true if the edge is been labeled
remove	E.remove()	Remove edge E