
Bu-Ali Sina University

Report

# Implementing DOT Language Using ANTLR

*Mehran Shahidi*

supervised by
Pr. Bathaeian

June 12, 2020

## Summary

In this Project we are going to implement DOT language using ANTLR. For more information about codes visit this repo:

**https://github.com/m3hransh/DOT_Lang**

# Contents

# 1 Introduction

Domain Specific Languages are languages created to support a particular set of tasks, as they are performed in a specific domain.Some DSLs are intended to be used by programmers, and therefore are more technical, while others are intended to be used by someone who is not a programmer and therefore they use less geeky concepts and syntax.

Why using a specific, limited language instead of a generic, powerful one? The short answer is that Domain Specific Languages are limited in the things they can do, but because of their specialization they can do much more in their own limited domain.

Domain Specific Languages are great because:

1. They let you communicate with domain experts. Do you write medical applications? Doctors do not understand when you talk about arrays or for-loops, but if you use a DSL that is about patients, temperature measures and blood pressure they could understand it better than you do
2. They let you focus on the important concepts. They hide the implementation or the technical details and expose just the information that really matters.

In this Report we are going to implement DOT language. DOT is a language that can describe graphs, either directed or non directed.

## 1.1 What is ANTLR?

ANTLR is a parser generator, a tool that helps you to create parsers What you need to do to get an AST:

1. define a lexer and parser grammar
2. invoke ANTLR: it will generate a lexer and a parser in your target language (e.g., Java, Python, C#, Javascript)
3. use the generated lexer and parser: you invoke them passing the code to recognize and they return to you an AST

ANTLR is actually made up of two main parts: the tool, used to generate the lexer and parser, and the runtime, needed to run them.

## 1.2 Lexers and Parsers

A lexer takes the individual characters and transforms them in tokens, the atoms that the parser uses to create the logical structure. These are notes that are important to define lexers and parsers in ANTLR:

- lexer rules are all uppercase, while parser rules are all lowercase
- Rules are typically written in this order: first the parser rules and then the lexer ones
- lexer rules are analyzed in the order that they appear(identifier define first)
- The basic syntax of a rule is easy: there is a name, a colon, the definition of the rule and a terminating semicolon

In the Listing 1 an example of definition of lexer and parser in ANTLR syntax is shown. This is not a complete grammar, but we can already see that lexer rules are all uppercase, while parser rules are all lowercase.

```
/*
 * Parser Rules
 */
operation  : NUMBER '+' NUMBER ;
/*
 * Lexer Rules
 */
NUMBER     : [0-9]+ ;
WHITESPACE : ' ' -> skip ;
```

Listing 1: Example of defining parser and lexer in ANTLR

## 2 DOT Language

The following (Listing 2) is the parser grammar defining the DOT language. First line indicates that these are parser rules. Line 5 causes to use the DOTLexer file that is shown in the Listing 3. Following Rules are typical indications that are in regular expression syntax. if you are not familiar with these rules you can click **here.** Take note that, Terminals are shown in upper-case font and nonterminals in lower-case. literal characters are given in single quotes. Parentheses **(** and **)** indicate grouping when needed. Question marks **?** represent optional items. Vertical bars **|** separate alternatives.

```
1  parser grammar DOTParser;
2  /*
3  * Parser Rules
4   */
5   options { tokenVocab=DOTLexer; }
6
7
8  graph        : STRICT? (GRAPH | DIGRAPH) name? '{' stmt_list'}' ;
9
10 stmt_list    : (stmt ';'? stmt_list)? ;
11
12 stmt         : (node_stmt | edge_stmt | attr_stmt | name '=' name | subgraph) ;
13
14 subgraph     : (SUBGRAPH name?)? '{' stmt_list'}' ;
15
16 attr_stmt    :(GRAPH | NODE | EDGE) attr_list ;
17
18 attr_list    : '[' a_list? ']' attr_list? ;
19
20 a_list       : name '=' name (';'|',')? a_list? ;
21
22 edge_stmt    : (node_name | subgraph) edgeRHS attr_list? ;
23
24 edgeRHS      : EDGEOP (node_name | subgraph) edgeRHS? ;
25
26 node_stmt    : node_name attr_list? ;
27
28 node_name      : name port? ;
29
30 port         : ':' name ( ':' name )? ;
31
32 name           : (ID| STRING| NUMBER) ;
```

Listing 2: Defining Parser for DOT Language in ANTLR

Listing 3 defines the lexer rules. Fragmets are reusable building blocks for lexer rules that are support case-insensatives keywords. The keywords **node, edge, graph, digraph, subgraph,** and **strict** are case-independent. Note also that the allowed compass point values are not keywords, so these strings can be used elsewhere as ordinary identifiers and, conversely, the parser will actually accept any identifier. a name is Any string of alphabetic ([a-zA-Z\200-\377]) characters, underscores ('_') or digits ([0-9]), not beginning with a digit or any double-quoted string ("...") possibly containing escaped quotes (\") .An edgeop is -> in directed graphs and – in undirected graphs. The language supports C++-style comments: /* */ and //. In addition, a line beginning with a '#' character is considered a line output from a C preprocessor (e.g., # 34 to indicate line 34 ) and discarded. Semicolons and commas aid readability but are not required. Also, any amount of whitespace may be inserted between terminals.

```
1  lexer grammar DOTLexer;
2
3  /*
4  * Lexer Rules
5   */
6  fragment A          : ('A'|'a') ;
7  fragment B          : ('B'|'b') ;
8  fragment S          : ('S'|'s') ;
9  fragment Y          : ('Y'|'y') ;
10 fragment H          : ('H'|'h') ;
11 fragment O          : ('O'|'o') ;
12 fragment U          : ('U'|'u') ;
13 fragment T          : ('T'|'t') ;
```

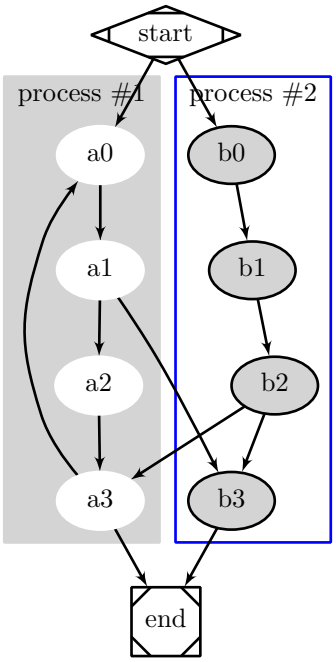
```
fragment R          : ('R'|'r') ;
fragment C          : ('C'|'c') ;
fragment I          : ('I'|'i') ;
fragment G          : ('G'|'g') ;
fragment P          : ('P'| 'p');
fragment D          : ('D'| 'd');
fragment N          : ('N'| 'n');
fragment E          : ('E'| 'e');
fragment DIGIT      : [0-9];

CUR_L: '{';
CUR_R:'}';
SEM:';';
EQ:'=';
BR_L:'[';
BR_R:']';
COMMA:',';
COLON:':';

GRAPH        : G R A P H ;
STRICT       : S T R I C T ;
DIGRAPH      : D I G R A P H ;
NODE         : N O D E ;
EDGE         : E D G E ;
SUBGRAPH     : S U B G R A P H ;
EDGEOP       : ('--'|'->') ;
COMMENT      : '/*' .*? '*/' -> skip ;
LINE_COMMENT: '//' .*? '\r'? '\n' -> skip ;
STRING       : '"' ( '\\"' | . )*? '"' ;
ID           : [_a-zA-Z\u0080-\u00FF][a-zA-Z\u0080-\u00FF_0-9]* ;
NUMBER       : '-'? ( '.' DIGIT+ | DIGIT+ ( '.' DIGIT* )? );
PREPROC      : '#' ~[\r\n]* -> skip ;
WS           : [ \t\n\r]+ -> skip ;
```

Listing 3: Defining Lexer for DOT Language in ANTLR

Figure 1 shows an example of executing the language using **dot2tex** console program.

```
digraph G {
subgraph cluster0 {
node [style=filled,color=white];
style=filled;
color=lightgrey;
a0 -> a1 -> a2 -> a3;
label = "process #1";
}
subgraph cluster1 {
node [style=filled];
b0 -> b1 -> b2 -> b3;
label = "process #2";
color=blue
}
start -> a0;
start -> b0;
a1 -> b3;
b2 -> a3;
a3 -> a0;
a3 -> end;
b3 -> end;
start [shape=Mdiamond];
end [shape=Msquare];
}
```

(a)




(b)

Figure 1: Example of executing DOT Language uisng dot2tex console program

## 2.1 Subgraphs and Clusters

Subgraphs play two roles. First, a subgraph can be used to represent graph structure, indicating that certain nodes and edges should be grouped together. This is the usual role for subgraphs and typically specifies semantic information about the graph components. It can also provide a convenient shorthand for edges. An edge statement allows a subgraph on both the left and right sides of the edge operator. When this occurs, an edge is created from every node on the left to every node on the right. For example, the specification

```
A -> {B C}
```

is equivalent to:

```
A -> B
A -> C
```

In the second role, a subgraph can provide a context for setting attributes. For example, a subgraph could specify that blue is the default color for all nodes defined in it. In the context of graph drawing, a more interesting example is:

```
subgraph {
rank = same; A; B; C;
}
```

This (anonymous) subgraph specifies that the nodes A, B and C should all be placed on the same rank.

## 2.2 Lexical and Semantic Notes

A graph must be specified as either a digraph or a graph. Semantically, this indicates whether or not there is a natural direction from one of the edge's nodes to the other. Lexically, a digraph must specify an edge using the edge operator -> while a undirected graph must use –. Operationally, the distinction is used to define different default rendering attributes. For example, edges in a digraph will be drawn, by default, with an arrowhead pointing to the head node. For ordinary graphs, edges are drawn without any arrowheads by default.

A graph may also be described as strict. This forbids the creation of multi-edges, i.e., there can be at most one edge with a given tail node and head node in the directed case. For undirected graphs, there can be at most one edge connected to the same two nodes. Subsequent edge statements using the same two nodes will identify the edge with the previously defined one and apply any attributes given in the edge statement. For example, the graph

```
strict graph {
  a -- b
  a -- b
  b -- a [color=blue]
}
```

will have a single edge connecting nodes a and b, whose color is blue.

If a default attribute is defined using a node, edge, or graph statement, or by an attribute assignment not attached to a node or edge, any object of the appropriate type defined afterwards will inherit this attribute value. This holds until the default attribute is set to a new value, from which point the new value is used. Objects defined before a default attribute is set will have an empty string value attached to the attribute once the default attribute definition is made.

Note, in particular, that a subgraph receives the attribute settings of its parent graph at the time of its definition. This can be useful; for example, one can assign a font to the root graph and all subgraphs will also use the font. For some attributes, however, this property is undesirable. If one attaches a label to the root graph, it is probably not the desired effect to have the label used by all subgraphs. Rather than listing the graph attribute at the top of the graph, and the resetting the attribute as needed in the subgraphs, one can simply defer the attribute definition in the graph until the appropriate subgraphs have been defined.

If an edge belongs to a cluster, its endpoints belong to that cluster. Thus, where you put an edge can effect a layout, as clusters are sometimes laid out recursively.

There are certain restrictions on subgraphs and clusters. First, at present, the names of a graph and it subgraphs share the same namespace. Thus, each subgraph must have a unique name. Second, although nodes can belong to any number of subgraphs, it is assumed clusters form a strict hierarchy when viewed as subsets of nodes and edges.

### 2.3 Implementaion of Lexer and Parser

Now that we have written the lexer and parser rules using ANTLR format, we can run our files against ANTLR and get our parser in the desired target language that we want. Before that, to use testing tools, we can compile our grammar with the java and see that it gives us the desired AST by using the graphical representation of the tree against some valid inputs in the DOT language. Before that, the installation of things that is needed is shown on this website **here**. After that, you can get the parse tree for the specificed input with these commands:

```
$ antlr4 *.g4 -o test
$ javac DOT*.java
$ grun DOT graph
> digraph G {
> subgraph cluster0 {
> style=filled;
> a0 -> a1;
> label = "1";
> }
> subgraph cluster1 {
> node [style=filled];
> b0 -> b1 -> b2 -> b3;
> label = "2";
> }
> start -> a0;
}
```

Lines proceded by $ are commands and lines proceded by > are input. The output parse tree is shown in Figure 2.

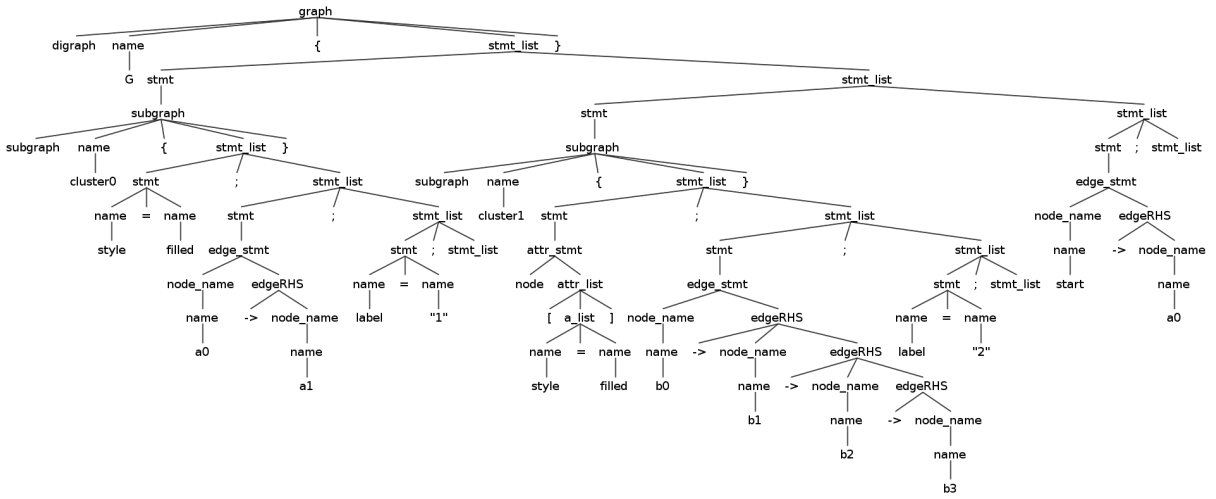



Figure 2: Parse tree picture of the Above commands.

## 3 Implementing in Python

In this section, first, the lexer and parser in python as a target language will be generated. Then we Implement a simple and limited Listener that support some important part of the language. In the next section we'll elaborate on this and create a browser base application for that.

## 3.1 Generate lexer and parser in Python

To make the process straightforward, a Gradle-build file is created to do the commands. This is the Gradle file script:

```
/*
 * This file was generated by the Gradle 'init' task.
 *
 * This is a general purpose Gradle build.
 * Learn how to create Gradle builds at https://guides.gradle.org/
    creating-new-gradle-builds
 */
apply plugin: 'java'
repositories {
    jcenter()
}
dependencies {
    runtime 'org.antlr:antlr4:4.8'
}

task generateLexer(type:JavaExec) {
  def lexerName = "DOTLexer"
  inputs.file("$ANTLR_SRC/${lexerName}.g4")
  outputs.file("$GEN_SRC/${lexerName}.py")
  outputs.file("$GEN_SRC/${lexerName}.interp")
  outputs.file("$GEN_SRC/${lexerName}.tokens")
  main = 'org.antlr.v4.Tool'
  classpath = sourceSets.main.runtimeClasspath
  args = ['-Dlanguage=Python3', "${lexerName}.g4", '-o', '../main-generated']
  workingDir = ANTLR_SRC
}
task generateParser(type:JavaExec) {
  dependsOn generateLexer
  def lexerName = "DOTLexer"
  def parserName = "DOTParser"
  inputs.file("$ANTLR_SRC/${parserName}.g4")
  inputs.file("$GEN_SRC/${lexerName}.tokens")
  outputs.file("$GEN_SRC/${parserName}.py")
  outputs.file("$GEN_SRC/${parserName}.interp")
  outputs.file("$GEN_SRC/${parserName}.tokens")
  main = 'org.antlr.v4.Tool'
  classpath = sourceSets.main.runtimeClasspath
  args = ['-Dlanguage=Python3', "${parserName}.g4", '-o', '../main-generated']
  workingDir = ANTLR_SRC
}
```

Listing 4: Gradle script

Two tasks are defined, *generateLexer*, and *generateParser* that take proper inputs and generate files in a proper path that is specified. **ANTLR_SRC** and **GEN_SRC** are specified in the *gradle.properties* file.

## 3.2 Setting up a Simple DOT Intepreter

In the previous section, files *DOTLexer.py*, *DOTParser.py* and *DOTParserListener.py* were generated. So now we'll override *DOTParserListener.py* with our own Listener in another file is called *GraphDOTListener.py*. Before that, we go through the steps of how to use our listener in Python. And for that, we need to first install the **antlr4-python3-runtime**. The proper way to do that is by using *pip*.

```
import sys
import antlr4
from DOTLexer import DOTLexer
from DOTParser import DOTParser
from GraphDOTListener import GraphDOTListener



def main(file_name):
    chars = antlr4.FileStream(file_name)
```

```
11      lexer = DOTLexer(chars)
12      stream = antlr4.CommonTokenStream(lexer)
13      parser = DOTParser(stream)
14      tree = parser.graph()
15
16      GraphDOT = GraphDOTListener()
17      walker = antlr4.ParseTreeWalker()
18      walker.walk(GraphDOT, tree)
19
20      GraphDOT.show_graph()
21
22
23  if __name__ == "__main__":
24      if len(sys.argv) > 1:
25          main(sys.argv[1])
```

Listing 5: a simple code of using our listener with passing networkx graph to it

Without further ado, let's look at our code that is shown in the Listings 5. From line 1 to line 7, needed modules are imported. sys module to read terminal arguments for specifying input files. antlr4 is our ANTLR runtime and has functions for reading inputs from files and walking through our Listener function. Our generated lexer and parser and then our Listener that we'll implement in the next section.

The most important things happen in lines 11 through 19. In the line 11, characters are read from the file. Then these characters plug into the lexer to take tokens, and then using antlr4, we get the stream of the tokens and pass it to the parser to create the parse tree. Then the graph is passed to the Listener, and with the antlr4 helper functions, we walk through the tree to manipulate the graph object.

### 3.2.1 Setting up the Listener

In our first iteration of setting up the Listener. We redefine our Language Rules in more easier and straightforward terms.

1. Our graph is directed or undirected and this is defined in the graph rules so we override our enterGraph:

```
def __init__(self):
        self.g = None

def enterGraph(self, ctx:DOTParser.GraphContext):
    if ctx.GRAPH() is not None:
        self.g  = nx.Graph()
    elif ctx.DIGRAPH() is not None:
        self.g = nx.DiGraph()
    # label property specify the name of the graph
    self.g.graph['label'] = ctx.name().getText()
```

2. We can define edges with – and -> between nodes. They are not different. if the graph is defined as directed, they are one-way edge from the first node to the second, otherwise they are simple edge.

```
def exitEdge_stmt(self, ctx:DOTParser.Edge_stmtContext):
    if ctx.node_name() is not None:
        first_node = ctx.node_name().getText()
        for second_node in ctx.edgeRHS().nodes:
            # add attributes if exist to the edges
            attrs = ctx.attr_list().attrs if ctx.attr_list() is not None else {}
            self.g.add_edge(first_node,second_node,**attrs)
            first_node = second_node

def exitEdgeRHS(self, ctx:DOTParser.EdgeRHSContext):
    ctx.nodes=[]

    if ctx.node_name() is not None:
        ctx.nodes.append(ctx.node_name().getText())
    if ctx.edgeRHS() is not None:
        ctx.nodes += ctx.edgeRHS().nodes
```

The *exitEdgeRHS* put nodes after the first node in a list is called nodes in a way that we can call that variable from its contex in the top level rule *edge_stmt* through the method *exitEdge_stmt*. Then the edges will be added pairwise to the graph.

3. We can set attributes in two ways:

   (a) With *attr_list* in *node_stmt* or *edge_stmt* that only influence the preceding nodes or edges in that statement.

```
# Like the edge_stmt check if has a following attributs
# If it exits it will add to the preceded node
def exitNode_stmt(self, ctx:DOTParser.Node_stmtContext):
    if ctx.attr_list() is not None:
        self.g.add_node(ctx.node_name().getText(),**ctx.attr_list().attrs)

def exitAttr_list(self, ctx:DOTParser.Attr_listContext):
    ctx.attrs = {}
    if ctx.a_list() is not None:
        ctx.attrs.update(ctx.a_list().attrs)
        if ctx.attr_list() is not None:
            ctx.attrs.update(ctx.attr_list().attrs)

def exitA_list(self, ctx:DOTParser.A_listContext):
    ctx.attrs = {}
    ctx.attrs[ctx.name()[0].getText()] = ctx.name()[1].getText()
    if ctx.a_list() is not None:
        ctx.attrs += ctx.a_list().attrs
```

   (b) Using *attr_stmt* that has three types *GRAPH* ,*NODE* and *EDGE* that NODE and EDGE only effect following nodes and edges that define after that. To support this feature two attibutes will define for current Node and edges default attributes.

```

def __init__(self):
   self.g = None
   # add this to the attributes to the class
   self.node_att={}
   self.edge_att={}

def exitAttr_stmt(self, ctx:DOTParser.Attr_stmtContext):
    if ctx.NODE() is not None:
        for k,v in ctx.attr_list().attrs:
            self.node_att[k] = v
    elif ctx.EDGE() is not None:
        for k,v in ctx.attr_list().attrs:
            self.edge_att[k] = v
    else:
        for k,v in ctx.attr_list().attrs:
            self.g.graph[k] = v
```

   Now we need to change *node_stmt* and *edge_stmt* to add these default attributes too.

Now our listener ends up like this:

```
import sys
import antlr4
from DOTParser import DOTParser
from DOTParserListener import DOTParserListener
import networkx as nx
import matplotlib.pyplot as plt


class GraphDOTListener(DOTParserListener):
    def __init__(self):
       self.g = None
       self.node_att={}
       self.edge_att={}

    def enterGraph(self, ctx:DOTParser.GraphContext):
```

```
16          if ctx.GRAPH() is not None:
17              self.g  = nx.Graph()
18          elif ctx.DIGRAPH() is not None:
19              self.g = nx.DiGraph()
20          # label property specify the name of the graph
21          self.g.graph['label'] = ctx.name().getText()
22
23
24      def exitEdge_stmt(self, ctx:DOTParser.Edge_stmtContext):
25          if ctx.node_name() is not None:
26              first_node = ctx.node_name().getText()
27              for second_node in ctx.edgeRHS().nodes:
28                  if ctx.attr_list():
29                      attrs = self.edge_att.copy()
30                      attrs.update(ctx.attr_list().attrs)
31                  else:
32                      attrs = self.edge_att
33                  self.g.add_edge(first_node,second_node,**attrs)
34                  first_node = second_node
35
36      def exitEdgeRHS(self, ctx:DOTParser.EdgeRHSContext):
37          ctx.nodes=[]
38
39          if ctx.node_name() is not None:
40              ctx.nodes.append(ctx.node_name().getText())
41              if ctx.edgeRHS() is not None:
42                  ctx.nodes += ctx.edgeRHS().nodes
43
44      def exitAttr_stmt(self, ctx:DOTParser.Attr_stmtContext):
45          if ctx.NODE() is not None:
46              for k,v in ctx.attr_list().attrs.items():
47                  self.node_att[k] = v
48          elif ctx.EDGE() is not None:
49              for k,v in ctx.attr_list().attrs.items():
50                  self.edge_att[k] = v
51          else:
52              for k,v in ctx.attr_list().attrs.items():
53                  self.g.graph[k] = v
54
55      def exitNode_stmt(self, ctx:DOTParser.Node_stmtContext):
56          if ctx.attr_list():
57              attrs = self.node_att.copy()
58              attrs.update(ctx.attr_list().attrs)
59          else:
60              attrs = self.node_att
61
62          self.g.add_node(ctx.node_name().getText(),**attrs)
63
64      def exitAttr_list(self, ctx:DOTParser.Attr_listContext):
65          ctx.attrs = {}
66          if ctx.a_list() is not None:
67              ctx.attrs.update(ctx.a_list().attrs)
68              if ctx.attr_list() is not None:
69                  ctx.attrs.update(ctx.attr_list().attrs)
70
71      def exitA_list(self, ctx:DOTParser.A_listContext):
72          ctx.attrs = {}
73          ctx.attrs[ctx.name()[0].getText()] = ctx.name()[1].getText()
74          if ctx.a_list() is not None:
75              ctx.attrs.update(ctx.a_list().attrs)
76
77      def show_graph(self):
78          color_map = []
79
80          for node in self.g:
81              color= self.g.nodes[node].get('color')
82              if color is not None:
83                  color_map.append(color)
84              else:
85                  color_map.append('gray')
86          edge_color_map = []
87          for edge in self.g.edges:
```

```
            color= self.g[edge[0]][edge[1]].get('color')
            if color is not None:
                edge_color_map.append(color)
            else:
                edge_color_map.append('gray')
        weights = nx.get_edge_attributes(self.g, 'weight')
        print(self.g.adj)
        pos = nx.spring_layout(self.g)
        nx.draw(self.g,pos, node_color=color_map, edge_color=edge_color_map,
    with_labels=True)
        nx.draw_networkx_edge_labels(self.g,pos,edge_labels= weights)
        plt.title(label = self.g.graph['label'])
        plt.show()
```

Package `networkx` is used for storing and manipulating the graph and package `matplotlib` is used to show the graph in show_graph helper method that is defined for the *GraphDOTListener*.

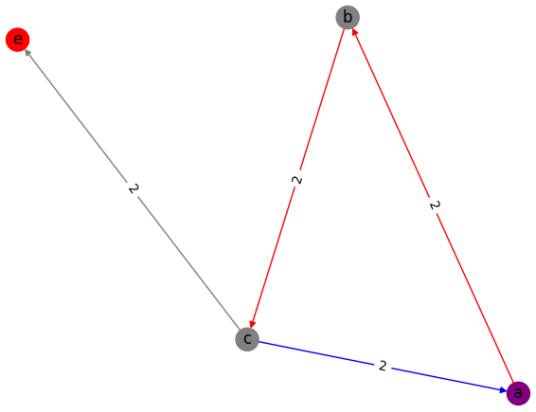
Now if you run the code above with command:

```
python main.py input.txt
```

you get the following result is shown in Figure 3.

```
digraph g1{
    a -> b -> c [color=red, weight=2];
    a [color=purple]
    c->a[color=blue, weight=2]
    node [color = red]
    e;
    edge[weight=2]
    c -- e;
}
```

(a) input.txt




(b) result of code (a)

Figure 3: Running main.py on input.txt file (a) result in figure (b)