GML-DOT 1.0
A Graph Translation Tool

Philip Gabardo

Contents

Chapter 1

Introduction

1.1 Summary

The proposed project is a compiler that translates graphs represented in Graph Modelling Language (GML) to a visual PDF representation produced through DOT code.

GML [5]is a file format standard that is used for representing graphs. GML's key features are portability, simplicity, extensibility and flexibility. The syntax consists of hierarchical key-value lists. Here is an example of a very simple GML file:

```
graph [
comment "This is a sample graph"
directed 1
id n42
label "Hello, I am a graph"
node [
id 1
label "node 1"
]
node [
id 2
label "node 2"
]
node [
id 3
label "node 3"
]
edge [
source 1
target 2
label "Edge from node 1 to node 2"
]
edge [
source 2
target 3
label "Edge from node 2 to node 3"
]
edge [
source 3
target 1
label "Edge from node 3 to node 1"
```

```
]
]
```

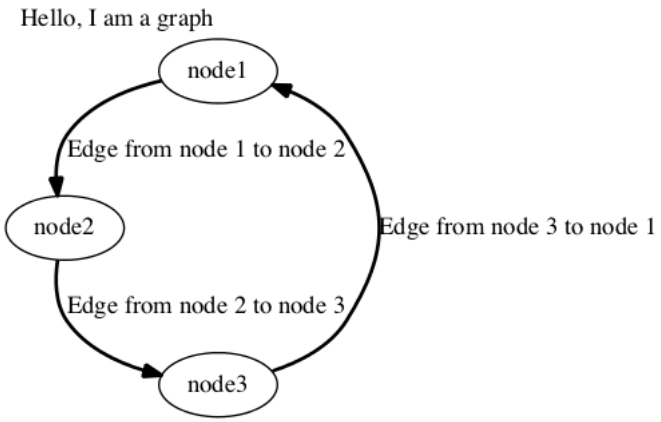
There are several other languages that closely resemble GML, such as GDF [3], GraphML [4], and DOT [2].

GML is widely used, and is the standard file format in the Graphlet graph editor system.

A major drawback of GML is its lack of direct graph translation to png or pdf images, which is essential to users would who like to visualize their graphs. Consequently, many users choose languages such as DOT, which can produce png and pdf images to represent their graphs, despite GML's attempt to become a universal file format for graphs.

The purpose of this compiler is to give GML users the ability to visualize their graphs through translation to DOT code, which can be used to generate a nicely formatted png file. DOT syntax is very simple, and contains single line statements to declare nodes and edges. The compiler will provide a simple, structure-preserving translation. Here is an example of the DOT code and the visual representation (generated using the DOT code) of the graph described in the sample GML provided earlier:

```
graph G {
label = "My graph";
comment = "This is a test";
1 [label = "Node 1"];
2 [label = "Node 2"];
3 [label = "Node 3"];
4 [label = "test"];
1 -- 2 [label = "Edge from node 1 to node 2"];
2 -- 3 [label = "Edge from node 2 to node 3"];
3 -- 1 [label = "Edge from node 3 to node 1"];
}
```



1.2 Tools

The following tools were used for this project:

- **ANTLR4** [6] was be used to generate DOT code from given GML code. ANTLR4 was chosen for two main reasons. First, ANTLR4 is proven to produce correct grammars. The generated parser accepts exactly the language specified in the grammar. If a recursive descent parser was used instead, there would be no immediate proof of correctness, and no easy way to test it. Secondly, ANTLR4 was chosen for its speedy development time. The parser and lexer rules can be constructed extremely fast.

- **DOT** [2] was used to draw the graphs. DOT was chosen because it is can automatically place nodes and edges such that there is minimal overlap (developing this kind of algorithm is outside the scope of this course). Furthermore, DOT has a command-line tool for producing png and pdf files, which many other graph languages that have automatic placement lack. For example, yWorks [12] requires the user to import their graphs to an editor in order to view them.
- **nuweb** [1] was be used to document the project. nuweb was chosen because it supports any programming language (there are no literate programming tools specificly targetted for ANTLR), it is documented well, and it has plenty of user support through forums. nuweb was chosen over noweb because noweb is heavily Unix-dependent, and requires many third party programs in order to be built.

Chapter 2

Implementation

The high-level flow of steps for the compiler is as follows:

1. Parse the GML file to elicit an object based representation of the graphs.
2. Convert the object based representation of the graphs to DOT.

The ANTLR compiler is organized as follows:

"GMLDot.g4" 1≡

⟨ *Grammar Name* 12 ⟩
⟨ *Header* 2 ⟩
⟨ *Members* 3 ⟩
⟨ *Top Level Rule Name* 13 ⟩
⟨ *Graph Rule* 14 ⟩
⟨ *Node Rule* 15 ⟩
⟨ *Edge Rule* 16 ⟩
⟨ *Graph Attribute Declaration Rule* 19 ⟩
⟨ *Node Attribute Declarations Rule* 17 ⟩
⟨ *Edge Attribute Declarations Rule* 18 ⟩
⟨ *Node Attribute Declaration Rule* 20 ⟩
⟨ *Edge Attribute Declaration Rule* 21 ⟩
⟨ *Graph Attribute Rule* 22 ⟩
⟨ *Node Attribute Rule* 23 ⟩
⟨ *Edge Attribute Rule* 24 ⟩
⟨ *Unsupported Attribute Rule* 25 ⟩
⟨ *Section Rule* 26 ⟩
⟨ *Value Rule* 27 ⟩
⟨ *Number Rule* 28 ⟩
⟨ *Integer Rule* 29 ⟩
⟨ *Word Rule* 30 ⟩
⟨ *Character Rule* 31 ⟩
⟨ *String Rule* 32 ⟩
⟨ *Whitespace Rule* 33 ⟩
◇

The following sections will describe, in a detailed, top-down fashion, each of these components. The first section will cover the class definitions that are used to simplify the conversion process and the second section will cover the parsing process.

2.1 Class Definitions

Several non-default Java data structures are used in class definitions.

⟨ *Header* 2 ⟩ ≡

```
@header {
        import java.util.HashMap;
        import java.util.ArrayList;
        import java.util.Set;
}
```

◇

Fragment referenced in 1.

Three classes are defined to simplify the conversion process.

⟨ *Members* 3 ⟩ ≡

```
@members {
        ⟨ Node Class Definition 4 ⟩
        ⟨ Edge Class Definition 5 ⟩
        ⟨ Graph Class Definition 6 ⟩
}
```

◇

Fragment referenced in 1.

The Node class is used to represent nodes declared in the GML. Each Node object contains a hash map that is used to store the attributes of the node. Each attribute name is stored as a key in the map, while each attribute value is stored as a value in the map.

⟨ *Node Class Definition* 4 ⟩ ≡

```
class Node {
        private HashMap<String, String> attributes;

        public Node(HashMap<String, String> attributes){
                this.attributes = attributes;
        }
}
```

◇

Fragment referenced in 3.

The Edge class is used to represent edges declared in the GML. Objects of the Edge class contain a hash map that is used to store the attributes of the edge. This hash map is used in the same way as in the Node class.

⟨ *Edge Class Definition* 5 ⟩ ≡

```
class Edge {
        private HashMap<String, String> attributes;

        public Edge(HashMap<String, String> attributes){
                this.attributes = attributes;
        }
}
```

◇

Fragment referenced in 3.

The Graph class is used to represent the entire graph declared in the GML. A Graph object contains a list of Node objects, a list of Edge objects, and a hash map to store attributes of the graph. This hash map is used in the same way as in the Node and Edge classes. A Graph object also has toDot() method.

⟨ *Graph Class Definition* 6 ⟩ ≡

```
class Graph {
        private ArrayList<Node> nodes;
        private ArrayList<Edge> edges;
        private HashMap<String, String> attributes;

        public Graph(ArrayList<Node> nodes, ArrayList<Edge> edges,
                 HashMap<String, String> attributes){
                this.nodes = nodes;
                this.edges = edges;
                this.attributes = new HashMap<String, String>();
                this.attributes.put("directed", "0");
                this.attributes.putAll(attributes);
        }

        ⟨ toDot Definition 7 ⟩
}
```

◇

Fragment referenced in 3.

The toDot() method converts the object representation of the GML to DOT.

⟨ *toDot Definition* 7 ⟩ ≡

```
private void toDot(){
        ⟨ Print Header 8 ⟩
        ⟨ Print Graph Attributes 9 ⟩
        ⟨ Print Nodes 10 ⟩
        ⟨ Print Edges 11 ⟩
        System.out.println("}");
}
```

◇

Fragment referenced in 6.

First, the 'directed' attribute is checked to determine the type of the DOT graph (digraph or a graph). The name of the DOT graph is set to G by default. The header (type and name) of the graph is printed to the screen. The 'directed' attribute is then deleted, so that it will not be printed with other graph attributes.

⟨ *Print Header* 8 ⟩ ≡

```
boolean directed = this.attributes.get("directed").equals("1");
if (directed){
        System.out.println("digraph G {");
}
else{
        System.out.println("graph G {");
}

this.attributes.remove("directed");
```

◇

Fragment referenced in 7.

Next, the parsed attributes of the graph are printed to the screen in DOT format (<ATTRIBUTE> = <VALUE>). This is easily done by iterating over the keys of the hash map of graph attributes.

⟨ *Print Graph Attributes* 9 ⟩ ≡

```
Set<String> keys = attributes.keySet();
for(String key: keys){
        System.out.println("\t" + key + " = " + attributes.get(key) + ";");
}
```

◇

Fragment referenced in 7.

Then, the parsed nodes are printed in DOT format (<NODE_ID> [<ATTRIBUTE1> = <VALUE1>, <ATTRIBUTE2> = <VALUE2>, ..., <ATTRIBUTEN> = <VALUEN>]). The 'id' attribute is printed first and then deleted from the hash map of attributes, so that it will not be printed with other node attributes. Then, the node attributes are printed. This is easily done by iterating over hash map of node attributes.

⟨ *Print Nodes* 10 ⟩ ≡

```
for (Node node: nodes){
        String nodeString = "\t" + node.attributes.get("id") + " [";
        node.attributes.remove("id");
        keys = node.attributes.keySet();  //get all keys
        for(String key: keys)
        {
                nodeString +=  key + " = " + node.attributes.get(key) + ", ";
        }
        if (keys.size() > 0)
                nodeString = nodeString.substring(0, nodeString.length()-2);
        nodeString += "];\n";
        System.out.print(nodeString);
}
```

◇

Fragment referenced in 7.

Finally, the parsed edges are printed in DOT format (<SOURCE_NODE_ID> (– if undirected — -¿ if directed) <TARGET_NODE_ID> [<ATTRIBUTE1> = <VALUE1>, <ATTRIBUTE2> = <VALUE2>, ..., <ATTRIBUTEN> = <VALUEN>]. The 'source' and 'target' attributes are printed first and then

deleted from the hash map of attributes, so that they will not be printed with other edge attributes. Then, the edge attributes are printed. This is done easily by iterating over the hash map of attributes for the edge.

⟨ *Print Edges* 11 ⟩ ≡

```
for (Edge edge: edges){
        String edgeString;
        if (directed) {
                edgeString = "\t" + edge.attributes.get("source")
                        + " -> " + edge.attributes.get("target") + " [";
        }
        else {
                edgeString = "\t" + edge.attributes.get("source")
                        + " -- " + edge.attributes.get("target") + " [";
        }
        edge.attributes.remove("source");
        edge.attributes.remove("target");
        keys = edge.attributes.keySet();  //get all keys
        for(String key: keys)
        {
                edgeString +=  key + " = " + edge.attributes.get(key) + ", ";
        }
        if (keys.size() > 0)
                edgeString = edgeString.substring(0, edgeString.length()-2);
        edgeString += "];\n";
        System.out.print(edgeString);
}
```

◇

Fragment referenced in 7.

2.2 Parsing

The grammar is named GMLDot:

⟨ *Grammar Name* 12 ⟩ ≡

```
grammar GMLDot;
```

◇

Fragment referenced in 1.

The top level rule name is “graph”.

⟨ *Top Level Rule Name* 13 ⟩ ≡

```
r: graph;
```

◇

Fragment referenced in 1.

The graph rule is initialized with two empty lists for its nodes and edges and a hashmap for its attributes. The rule recognizes the keyword “graph”, followed by an opening square bracket, followed by a series of nodes, edges, or graph attribute declarations in no specific order (according to https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf, “It (GML) should be flexible enough that a specific order of declarations is not needed”), terminated by a closing square

bracket. Whenever a node is parsed, it is added to the list of nodes. Similarly, whenever an edge is parsed, it is added to the list of edges. Whenever a graph attribute is parsed, the attribute name and value is added as a key value pair to the graph attribute hash map. When the terminating square bracket is parsed, a Graph object is created using the objects initialized at the beginning of the rule. The toDot() method is called to print the DOT representation of the graph.

⟨ *Graph Rule* 14 ⟩ ≡

```
graph
@init {ArrayList<Node> nodes = new ArrayList<Node>();
        ArrayList<Edge> edges = new ArrayList<Edge>();
        HashMap<String, String> declarations = new HashMap<String, String>();}
: 'graph' '['
(
        node
        {
                nodes.add($node._node);
        }
        |
        edge
        {
                edges.add($edge._edge);
        }
        |
        graphAttrDeclaration
        {
                declarations.putAll($graphAttrDeclaration.declaration);
        }
                )*
']'
{
        Graph graph = new Graph(nodes, edges, declarations);
        graph.toDot();
};
```

◇

Fragment referenced in 1.

The node rule returns a Node object. The rule recognizes the keyword "node" followed by an opening square bracket, followed by a series of node attribute declarations (nodeAttrDeclarations), followed by a terminating square bracket. A Node object is created at the end of the rule, using the declarations that were parsed as the hash map for the constructor.

⟨ *Node Rule* 15 ⟩ ≡

```
node returns [Node _node]: 'node' '[' nodeAttrDeclarations ']'
{
        $_node = new Node($nodeAttrDeclarations.declarations);
};
```

◇

Fragment referenced in 1.

The edge rule returns an Edge object. The rule works in a very similar fashion to how the node rule works.

⟨ *Edge Rule* 16 ⟩ ≡

```
edge returns [Edge _edge]: 'edge' '[' edgeAttrDeclarations ']'
{
        $_edge = new Edge($edgeAttrDeclarations.declarations);
};
◇
```

Fragment referenced in 1.

The nodeAttrDeclarations rule parses a series of node attribute declarations, and returns a hash map where the keys are attributes and the values are corresponding attribute values. The rule is initialized with an empty hash map. Every time a node attribute declaration (nodeAttrDeclaration) is parsed, its corresponding hash map is merged with the rule's hash map.

⟨ *Node Attribute Declarations Rule* 17 ⟩ ≡

```
nodeAttrDeclarations returns [HashMap<String, String> declarations]
@init { $declarations = new HashMap<String, String>(); }
: (nodeAttrDeclaration {$declarations.putAll($nodeAttrDeclaration.declaration);})*;
◇
```

Fragment referenced in 1.

The edgeAttrDeclarations rule parses a series of edge attribute declarations, and returns a hash map where the keys are attributes and the values are corresponding attribute values. This rule works in a very similar fashion to how the nodeAttrDeclarations rule works.

⟨ *Edge Attribute Declarations Rule* 18 ⟩ ≡

```
edgeAttrDeclarations returns [HashMap<String, String> declarations]
@init { $declarations = new HashMap<String, String>(); }
: (edgeAttrDeclaration {$declarations.putAll($edgeAttrDeclaration.declaration);})*;
◇
```

Fragment referenced in 1.

The graphAttrDeclaration rule parses a graph attribute and returns a hash map. If the attribute is translatable to DOT, then the hash map includes a single key-value pair where the key is the attribute and the value is the corresponding attribute value. If the attribute is not translatable to DOT, a comment is printed that warns the user that the graph attribute was parsed but will not be translated, and the hash map that is returned is empty.

⟨ *Graph Attribute Declaration Rule* 19 ⟩ ≡

```
graphAttrDeclaration returns [HashMap<String, String> declaration]
@init {$declaration = new HashMap<String, String>();}
:(graphAttribute VALUE)
{
        $declaration.put($graphAttribute.text, $VALUE.text);
}
| (unsupportedAttribute (VALUE | section))
{
        System.out.printf("//Warning: the graph attribute \"%s\" was parsed but is not "
                + "supported by DOT,so it will not be translated.\n", $unsupportedAttribute.text);
};
◇
```

Fragment referenced in 1.

The nodeAttrDeclaration rule parses a node attribute and returns a hash map. This rule works in a similar fashion to how the graphAttrDeclaration rule works.

⟨ *Node Attribute Declaration Rule* 20 ⟩ ≡

```
nodeAttrDeclaration returns [HashMap<String, String> declaration]
@init { $declaration = new HashMap<String, String>(); }
: (nodeAttribute VALUE)
{
        $declaration.put($nodeAttribute.text, $VALUE.text);
}
| (unsupportedAttribute (VALUE | section))
{
        System.out.printf("//Warning: the node attribute \"%s\" was parsed but is not"
                + " supported by DOT,so it will not be translated.\n", $unsupportedAttribute.text);
};
```

◇

Fragment referenced in 1.

The edgeAttrDeclaration rule parses an edge attribute and returns a hash map. This rules works in a similar fashion to how the graphAttrDeclaration and nodeAttrDeclaration rules work.

⟨ *Edge Attribute Declaration Rule* 21 ⟩ ≡

```
edgeAttrDeclaration returns [HashMap<String, String> declaration]
@init { $declaration = new HashMap<String, String>(); }
: (edgeAttribute VALUE)
{
        $declaration.put($edgeAttribute.text, $VALUE.text);
}
| (unsupportedAttribute (VALUE | section))
{
        System.out.printf("//Warning: the edge attribute \"%s\" was parsed but is not"
                + " supported by DOT,so it will not be translated.\n", $unsupportedAttribute.text);
};
```

◇

Fragment referenced in 1.

There are four GML graph attribute rules that are directly translatable to DOT. Therefore, the graphAttribute rule is defined as follows:

⟨ *Graph Attribute Rule* 22 ⟩ ≡

```
graphAttribute: 'label' | 'directed' | 'comment' | 'URL';
```

◇

Fragment referenced in 1.

Similarly, there are four GML node attribute rules that are directly translatable to DOT. Therefore, the nodeAttribute rule is defined as follows:

⟨ *Node Attribute Rule* 23 ⟩ ≡

```
nodeAttribute: 'id' | 'name' | 'label' | 'comment';
```

◇

Fragment referenced in 1.

Likewise, there are four GML edge attribute rules that are directly translatable to DOT. Therefore, the edgeAttribute rule is defined as follows:

⟨ *Edge Attribute Rule* 24 ⟩ ≡

```
edgeAttribute: 'source' | 'target' | 'label' | 'comment';
```

◇

Fragment referenced in 1.

An unsupported GML attribute is defined as any word, in an attempt to make the compiler as flexible as possible. This works because the graphAttributeDeclaration, nodeAttrDeclaration and edgeAttrDeclaration rules are structured to parse translatable attributes before attempting to parse unsupported attributes.

⟨ *Unsupported Attribute Rule* 25 ⟩ ≡

```
unsupportedAttribute:  WORD;
```

◇

Fragment referenced in 1.

Some unsupported attributes have 'sections' as values. A section is defined recursively as follows:

⟨ *Section Rule* 26 ⟩ ≡

```
section: '[' (WORD VALUE)*  (WORD section)* (WORD VALUE)* ']';
```

◇

Fragment referenced in 1.

A GML value is defined to be either a number or a string:

⟨ *Value Rule* 27 ⟩ ≡

```
VALUE: NUMBER | STRING;
```

◇

Fragment referenced in 1.

A number can either be a floating point number or an integer.

⟨ *Number Rule* 28 ⟩ ≡

```
NUMBER: INT ('.' INT)?;
```

◇

Fragment referenced in 1.

An integer is one or more digits.

⟨ *Integer Rule* 29 ⟩ ≡

```
INT: ('0' .. '9')+;
```

◇

Fragment referenced in 1.

A word is one or more characters.

⟨ *Word Rule* 30 ⟩ ≡

```
WORD: CHAR+;
```

◇

Fragment referenced in 1.

A character is defined to be any non-whitespace character.

⟨ *Character Rule* 31 ⟩ ≡

```
CHAR: ~[ \t\r\n];
```

◇

Fragment referenced in 1.

A String is defined to be a series of non-newline characters enclosed by quotation marks.

⟨ *String Rule* 32 ⟩ ≡

```
STRING: '"' (~[\r\n"] | '""')* '"';
```

◇

Fragment referenced in 1.

Whitespace is ignored while parsing.

⟨ *Whitespace Rule* 33 ⟩ ≡

```
WS : [ \t\r\n]+ -> skip;
```

◇

Fragment referenced in 1.

2.3 Testing

Two test benches were used for testing the compiler.

The first test bench was generated manually, and consists of a set of GML files and corresponding expected DOT files. The GML files were constructed to test edge cases of the compilation process. For each test case, the GML file was compiled and the outputted DOT file was diffed with the expected DOT file. If there were no differences, the test passed. Otherwise, the test failed.

The second test bench was derived from an online source [11], and consists only of GML files. Each GML file is extremely large. For each test case, the GML file was compiled and a png was generated fromt the outputted DOT file. The png was inspected for correctness.

All test cases passed.

2.3.1 Test Bench 1

Test 1

"test1.gml" 34≡

```
graph [

]
```
◇

"expected1.dot" 35≡

```
graph G {
}
```
◇

Test 2

"test2.gml" 36≡

```
graph [
URL "www.test.com"
node [
id 1
label "Node 1"
]
node [
id 2
label "Node 2"
]
node [
id 3
label "Node 3"
]
node [
id 4
label "test"
]
edge [
source 1
target 2
label "Edge from node 1 to node 2"
]
edge [
source 2
target 3
label "Edge from node 2 to node 3"
]
edge [
source 3
target 1 label
"Edge from node 3 to node 1"
]
comment "This is a test"
label "My graph"
defaultnodesize 4
]
◇
```

"expected2.dot" 37≡

```
//Warning: the graph attribute "defaultnodesize" was parsed but is not supported by DOT,so it will not be
graph G {
        label = "My graph";
        URL = "www.test.com";
        comment = "This is a test";
        1 [label = "Node 1"];
        2 [label = "Node 2"];
        3 [label = "Node 3"];
        4 [label = "test"];
        1 -- 2 [label = "Edge from node 1 to node 2"];
        2 -- 3 [label = "Edge from node 2 to node 3"];
        3 -- 1 [label = "Edge from node 3 to node 1"];
}
```

◇

Test 3

"test3.gml" 38≡

```
graph [
comment "This is a test"
node [
id 1
label "Node 1"
]
node [
id 2
label "Node 2"
]
edge [
source 1
target 2
label "Edge from node 1 to node 2"
]
node [
id 3
label "Node 3"
]
node [
id 4
label "test"
]
edge [
source 2
target 3
label "Edge from node 2 to node 3"
]
edge [
source 3
target 1 label
"Edge from node 3 to node 1"
]
label "My graph"
]
```

◇

"expected3.dot" 39≡

```
graph G {
        label = "My graph";
        comment = "This is a test";
        1 [label = "Node 1"];
        2 [label = "Node 2"];
        3 [label = "Node 3"];
        4 [label = "test"];
        1 -- 2 [label = "Edge from node 1 to node 2"];
        2 -- 3 [label = "Edge from node 2 to node 3"];
        3 -- 1 [label = "Edge from node 3 to node 1"];
}
```

◇

Test 4

"test4.gml" 40≡

```
graph [
node [
id 1
label ""
]
node [
id 2
label ""
]
node [
id 3
label ""
]
edge [
source 1
target 2
label ""
]
edge [
source 2
target 3
label ""
]
edge [
source 3
target 1 label
""
]
comment ""
directed 1
isPlanar 1
]
◇
```

"expected4.dot" 41≡

```
//Warning: the graph attribute "isPlanar" was parsed but is not supported by DOT,so it will not be transla
digraph G {
        comment = "";
        1 [label = ""];
        2 [label = ""];
        3 [label = ""];
        1 -> 2 [label = ""];
        2 -> 3 [label = ""];
        3 -> 1 [label = ""];
}
◇
```

Test 5

"test5.gml" 42≡

```
graph
[ hierarchic  1
  directed  1
  node
  [ id  0
    graphics
    [ x 200.0
      y 0.0
    ]
    LabelGraphics
    [ text  "January" ]
  ]
  node
  [ id  1
    graphics
    [ x 425.0
      y 75.0
    ]
    LabelGraphics
    [ text  "December"  ]
  ]
  edge
  [ source  1
    target  0
    LabelGraphics
    [ text  "Happy New Year!"
      model "six_pos"
      position  "head"
    ]
  ]
]
◇
```

"expected5.dot" 43≡

```
//Warning: the graph attribute "hierarchic" was parsed but is not supported by DOT,so it will not be trans
//Warning: the node attribute "graphics" was parsed but is not supported by DOT,so it will not be translat
//Warning: the node attribute "LabelGraphics" was parsed but is not supported by DOT,so it will not be tra
//Warning: the node attribute "graphics" was parsed but is not supported by DOT,so it will not be translat
//Warning: the node attribute "LabelGraphics" was parsed but is not supported by DOT,so it will not be tra
//Warning: the edge attribute "LabelGraphics" was parsed but is not supported by DOT,so it will not be tra
digraph G {
        0 [];
        1 [];
        1 -> 0 [];
}
◇
```

Test 6

"test6.gml" 44≡

```
graph
[ hierarchic  1
  directed  1
  node
  [ id  0
  ]
  node
  [ id  1
  ]
  edge
  [ source  1
    target  0
  ]
]
◇
```

"expected6.dot" 45≡

```
//Warning: the graph attribute "hierarchic" was parsed but is not supported by DOT,so it will not be trans
digraph G {
        0 [];
        1 [];
        1 -> 0 [];
}
◇
```

Test 7

"test7.gml" 46≡

```
graph [
 node [
 test "estseesest"
 id 7
 label "5"
 edgeAnchor "corners"
 labelAnchor "n"
 graphics [
 center [ x 82.0000 y 42.0000 ]
 w 16.0000
 h 16.0000
 type "rectangle"
 fill "#000000"
 ]
 ]
 node [
 id 15
 label "13"
 edgeAnchor "corners"
 labelAnchor "c"
 graphics [
 center [ x 73.0000 y 160.000 ]
 w 16.0000
 h 16.0000
 type "rectangle"
 fill "#FF0000"
 ]
 ]
 edge [
 label "24"
 labelAnchor "first"
 source 7
 target 15
 graphics [
 type "line"
 arrow "last"
 Line [
 point [ x 82.0000 y 42.0000 ]
 point [ x 10.0000 y 10.0000 ]
 point [ x 100.000 y 100.000 ]
 point [ x 80.0000 y 30.0000 ]
 point [ x 120.000 y 230.000 ]
 point [ x 73.0000 y 160.000 ]
 ]
 ]
 ]
]
```

◇

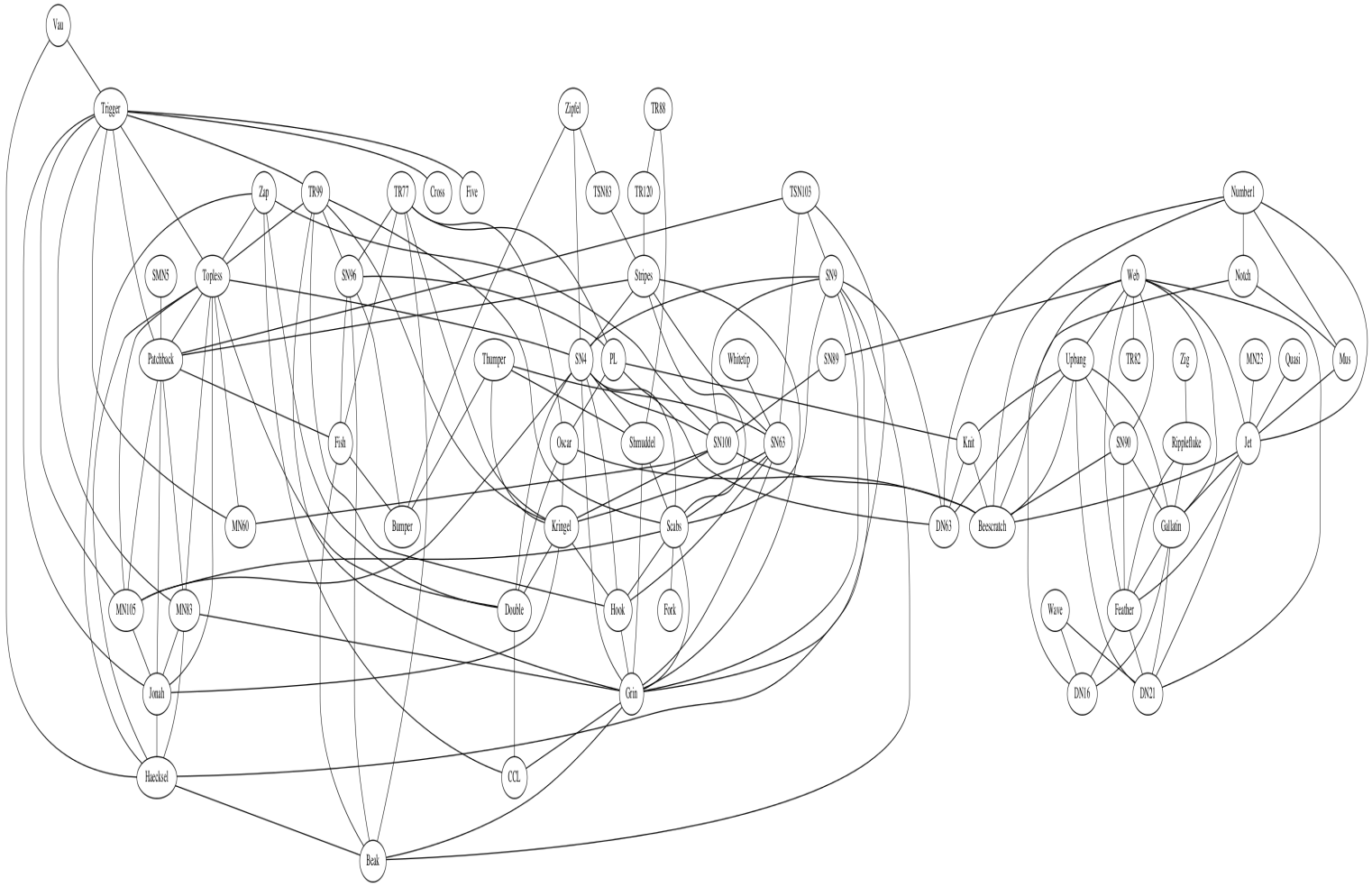
"expected7.dot" 47≡

```
//Warning: the node attribute "test" was parsed but is not supported by DOT,so it will not be translated.
//Warning: the node attribute "edgeAnchor" was parsed but is not supported by DOT,so it will not be transl
//Warning: the node attribute "labelAnchor" was parsed but is not supported by DOT,so it will not be trans
//Warning: the node attribute "graphics" was parsed but is not supported by DOT,so it will not be translat
//Warning: the node attribute "edgeAnchor" was parsed but is not supported by DOT,so it will not be transl
//Warning: the node attribute "labelAnchor" was parsed but is not supported by DOT,so it will not be trans
//Warning: the node attribute "graphics" was parsed but is not supported by DOT,so it will not be translat
//Warning: the edge attribute "labelAnchor" was parsed but is not supported by DOT,so it will not be trans
//Warning: the edge attribute "graphics" was parsed but is not supported by DOT,so it will not be translat
graph G {
        7 [label = "5"];
        15 [label = "13"];
        7 -- 15 [label = "24"];
}
◇
```

2.3.2 Test Bench 2

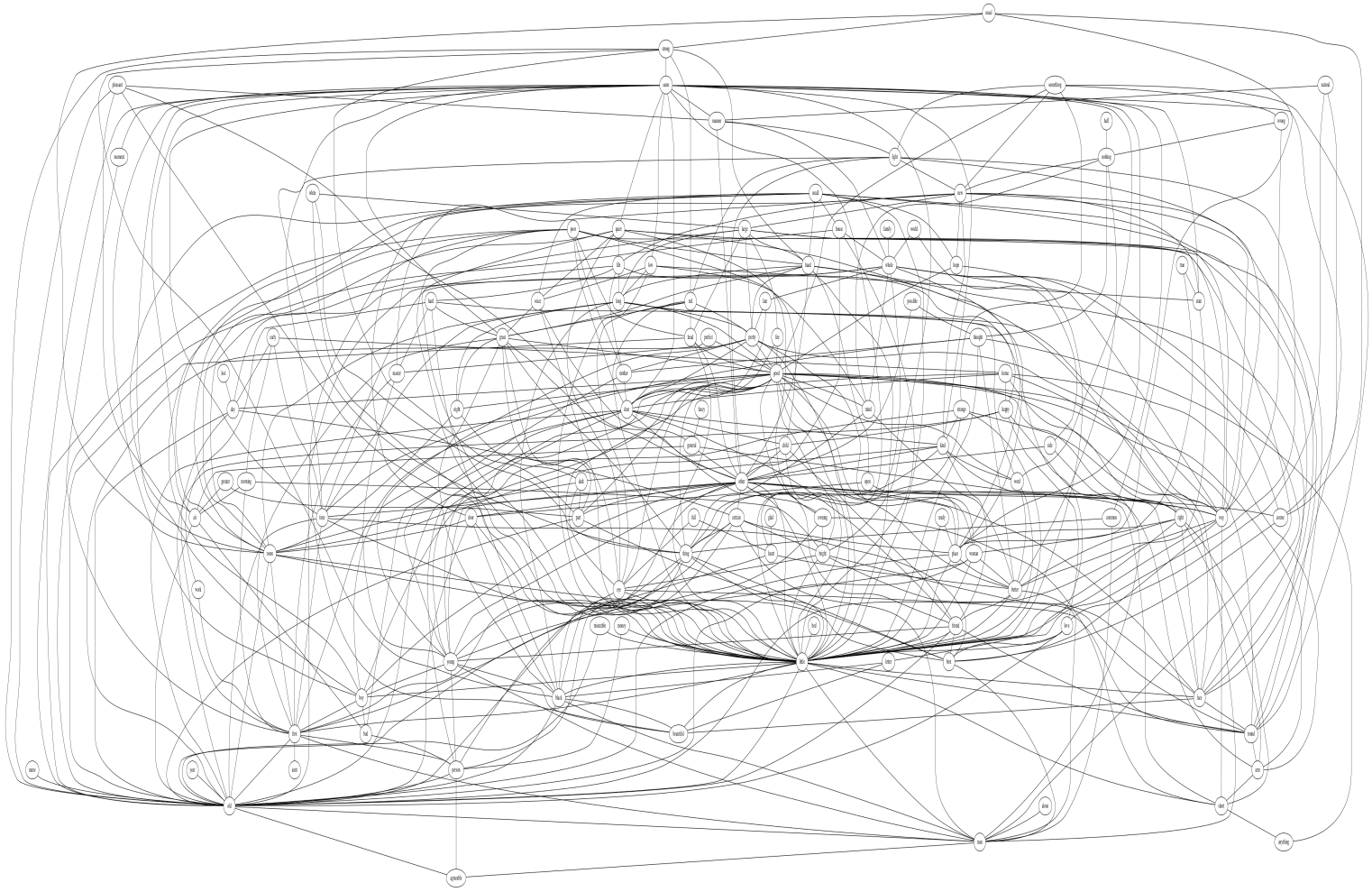
Test 1

dolpins.gml [8]



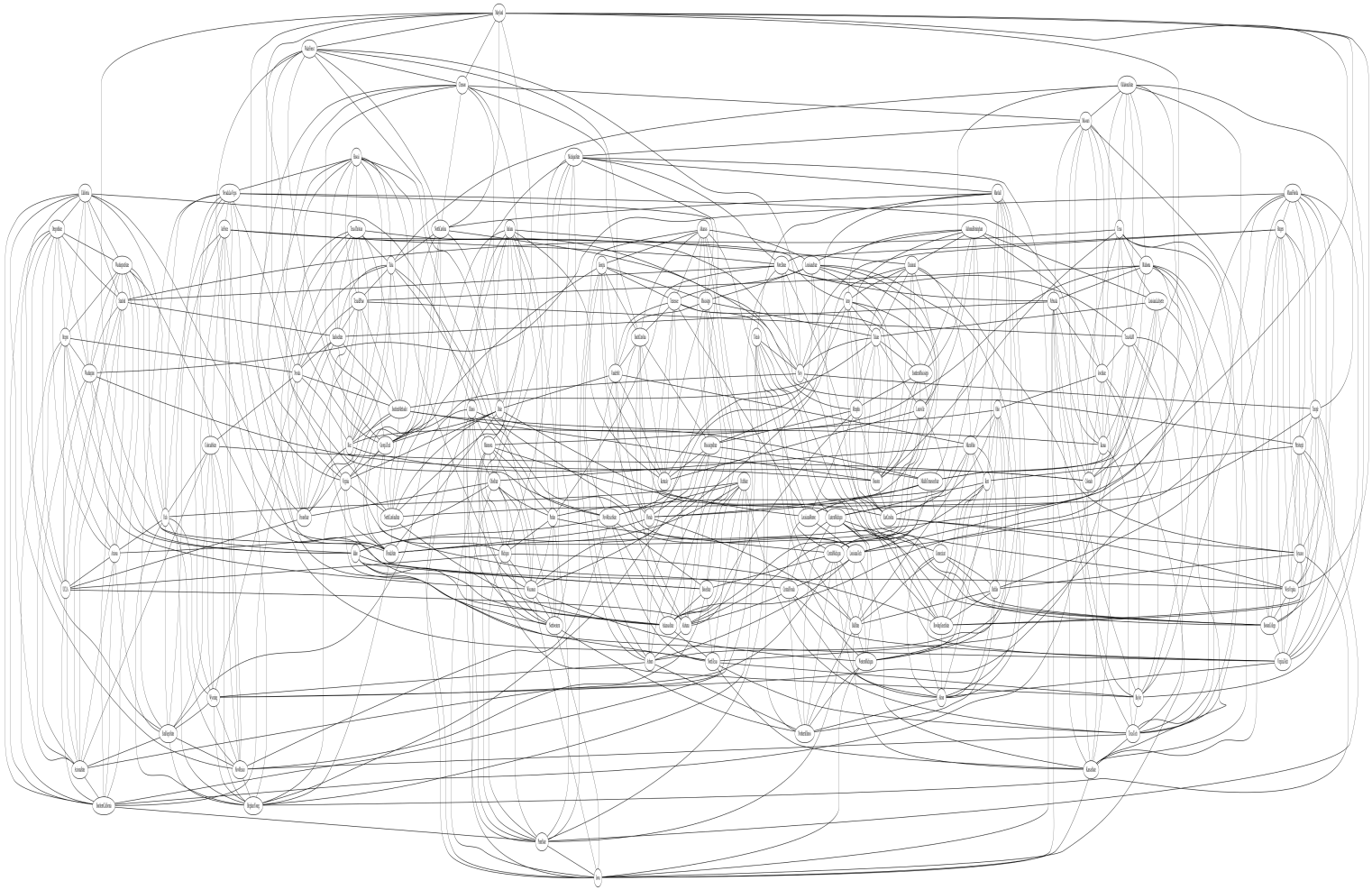
Test 2

adjnoun.gml [8]



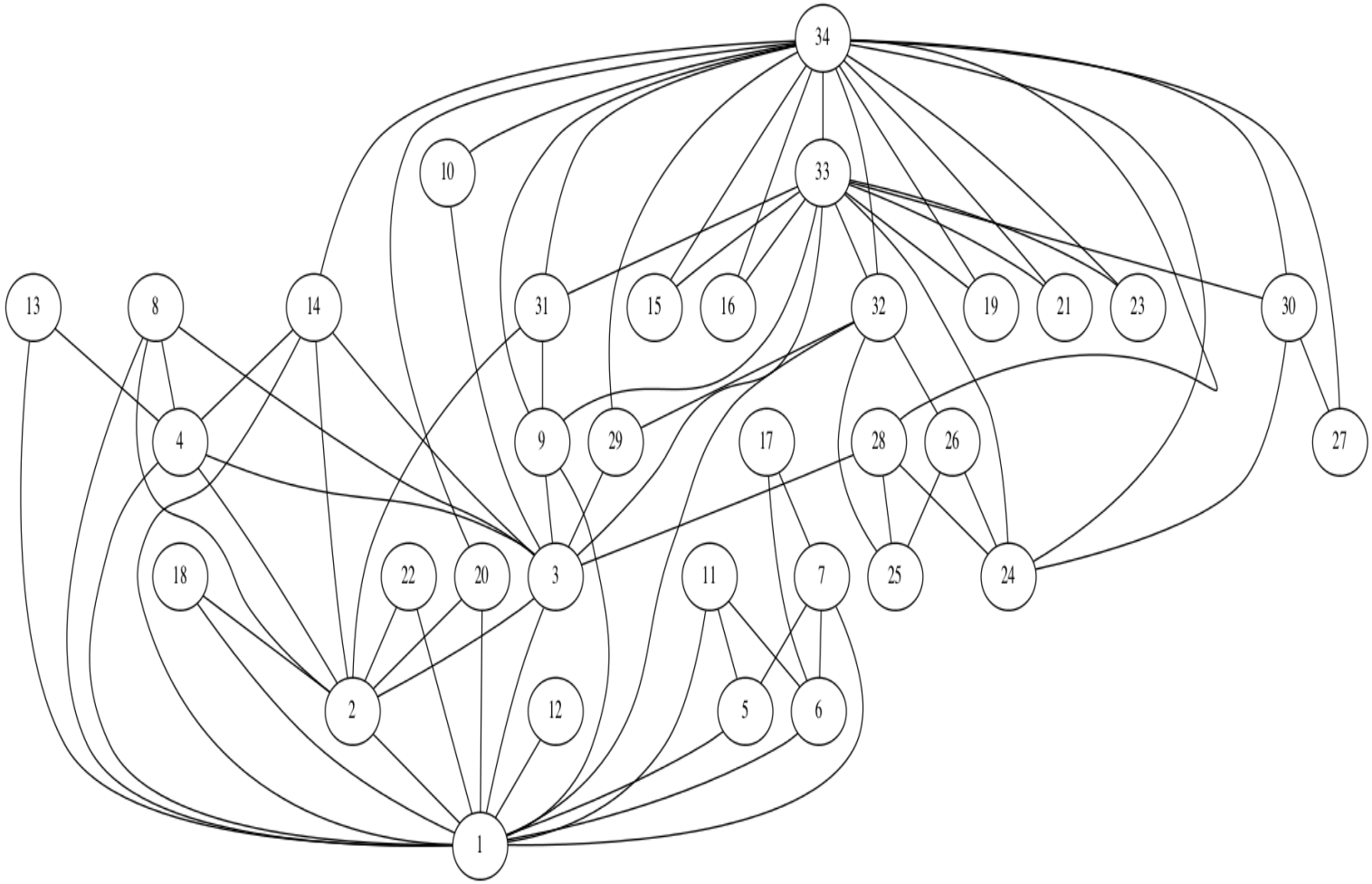
Test 3

football.gml [8]



Test 4

karate.gml [8]



Bibliography

[1] Preston Briggs. Nuweb version 1.57 a simple literate programming too. http://nuweb.sourceforge.net/nuwebdoc.pdf. iv

[2] Emden Gansner. Drawing graphs with dot. http://www.graphviz.org/pdf/dotguide.pdf, 2015. iii, iv

[3] Gephi. Gdf format. https://gephi.org/users/supported-graph-formats/gdf-format/. iii

[4] Gephi. Graphml format. https://gephi.org/users/supported-graph-formats/graphml-format/. iii

[5] Michael Himsolt. Gml: A portable graph file format. https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf. ii

[6] Terence Parr. Antlr 4 documentation. https://github.com/antlr/antlr4/blob/master/doc/index.md, 2015. iii

[7] Irvine University of California. adjnoun.gml. http://networkdata.ics.uci.edu/data/adjnoun/adjnoun.gml.

[8] Irvine University of California. dolphins.gml. http://networkdata.ics.uci.edu/data/dolphins/dolphins.gml. xxv, xxvi, xxvii, xxviii

[9] Irvine University of California. football.gml. http://networkdata.ics.uci.edu/data/football/football.gml.

[10] Irvine University of California. karate.gml. http://networkdata.ics.uci.edu/data/karate/karate.gml.

[11] Irvine University of California. Network data repository. http://networkdata.ics.uci.edu/. xiv

[12] yWorks. yworks. https://gephi.org/users/supported-graph-formats/graphml-format/. iv