
Bu-Ali Sina University

Report

# Implementing DOT Language Using ANTLR

*Mehran Shahidi*

supervised by
Pr. Bathaeian

June 10, 2020

## Summary

Summary of your project.

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

# 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) ;
```

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') ;
```

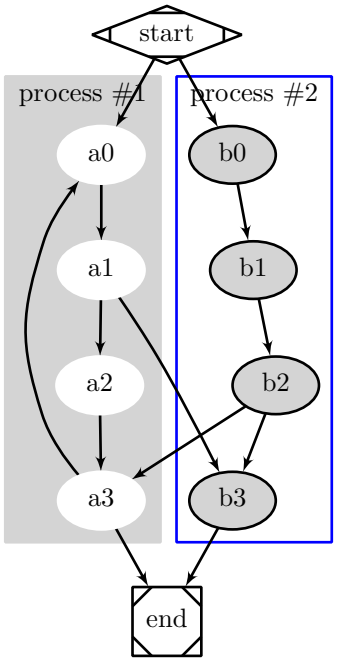
```
14 fragment R           : ('R'|'r') ;
15 fragment C           : ('C'|'c') ;
16 fragment I           : ('I'|'i') ;
17 fragment G           : ('G'|'g') ;
18 fragment P           : ('P'| 'p');
19 fragment D           : ('D'| 'd');
20 fragment N           : ('N'| 'n');
21 fragment E           : ('E'| 'e');
22
23 CUR_L: '{';
24 CUR_R:'}';
25 SEM:';';
26 EQ:'=';
27 BR_L:'[';
28 BR_R:']';
29 COMMA:',';
30 COLON:':';
31
32 GRAPH        : G R A P H ;
33 STRICT       : S T R I C T ;
34 DIGRAPH      : D I G R A P H ;
35 NODE         : N O D E ;
36 EDGE         : E D G E ;
37 SUBGRAPH     : S U B G R A P H ;
38 EDGEOP       : ('--'|'->') ;
39 COMMENT      : '/*' .*? '*/' -> skip ;
40 LINE_COMMENT: '//' .*? '\r'? '\n' -> skip ;
41 STRING       : '"' ( '\\"' | . )*? '"' ;
42 ID           : [_a-zA-Z\u0080-\u00FF][a-zA-Z\u0080-\u00FF_0-9]* ;
43 PREPROC      : '#' ~[\r\n]* -> skip ;
44 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 specified 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.

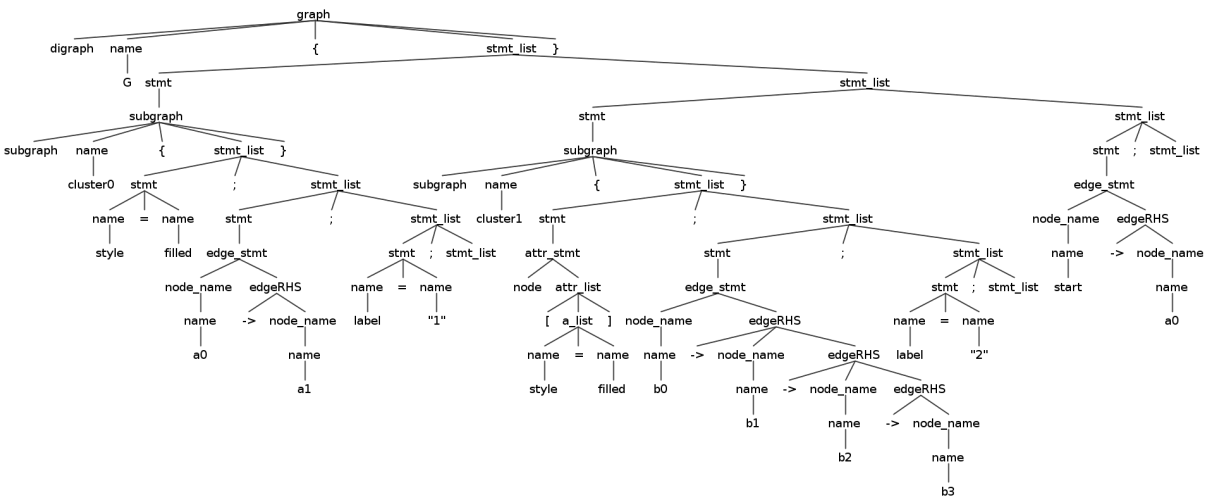



Figure 2: Parse tree picture of the Above commands.