Georg-August-University Göttingen

M.Inf.1201

Exploring the Apache Jena Framework

Supervisor:
Prof. Dr. Wolfgang May

Author:
Stefan Siemer

June 18, 2019

Contents

1 Introduction **1**

2 Installation & Setup **1**

2.1 Apache Jena 1

2.2 Tomcat 2

2.3 SemWebJena 2

2.4 JavaCC 3

3 JavaCC Tutorial **3**

3.1 CarGrammar Specification 3

3.2 CarGrammar Implementation 4

4 Jena SPARQL & RDF processing **8**

5 Master's thesis outlook **10**

1 Introduction

In order to work on new possibilities and approaches in the SPARQL language, two things are elementary. On the one hand the fundamental knowledge of the theoretical foundations needed for this formal language. On the other hand a practical framework as a playground is needed. A complete playground gives the opportunity to test small ideas very fast in practice. The following document is a summary of practical work and tutorials needed for this kind of academic work e.g. a master's thesis. Therefore a brief description of the Apache Jena Framework and its underlying architecture, tools and programming languages is given. In the end there is a small outlook on what shall be covered in the upcoming master thesis.

2 Installation & Setup

First of all the various tools and frameworks need to be downloaded, installed and set up. The goal is to have as few duplicate future work as possible.

2.1 Apache Jena

> "A free and open source Java framework for building Semantic Web and Linked Data applications." [1]

While this quote gives the general purpose of the whole framework, the essence of Jena lies in ARQ. ARQ is a query engine that evaluates the SPARQL query language against RDF data. [2] The newest versions of the Apache Jena releases can be found on https://jena.apache.org/download/index.cgi. The frameworks binaries are available in different compressed formats. For developers the source code is fully accessable via the mirrored github at https://gitbox.apache.org/repos/asf?p=jena.git. Older versions can also be found on http://archive.apache.org/dist/jena. All the newer version are build with maven. Maven can be found either on http://maven.apache.org/ or be installed via a package manager e.g. apt on ubuntu with `sudo apt-get install maven`. Afterwards its location needs to be exported and added to the default classpath PATH. This can be done by adding the following two lines to the ~`/.bashrc`:

```
export M2_HOME=/opt/maven
export PATH=${M2_HOME}/bin:${PATH}
```

Now having everything at hand the binary can be build by navigating to the `Jena` folder and executing maven by typing `mvn clean install`. If just the rebuild of a single subfolder is needed the same command can be used in the respective folder. The maven build produces a binary zip-file in the subfolder `apache-jena/target/apache-jena-x.x.x.zip`. This can be unpacked at any desired location and must afterwards be exported in the ~`/.bashrc` to make it accessable via the wrapping command line scripts included in Jena:

```
export JENAROOT='<path to Jena>/jena'
export JENA_HOME='<path to Jena>/jena'
```

```
export PATH=${JENAROOT}/bin:${PATH}
export PATH=${JENA_HOME}/bin:${PATH}
```

Now with Jena set up, the command `sparql -h` gives you the possible arguments and modifications for the command line tool of the sparql query engine. [1]

2.2 Tomcat

"Tomcat is a web servlet container with a simple web server."[3] In order to run tomcat the binaries can be downloaded from https://tomcat.apache.org/. [4] After that the variable `CATALINA_HOME` is to be set in `~/.bashrc`:

```
export CATALINA_HOME=<Path to Tomcat>
```

After putting a webapp foobla.war into the folder `webapps` tomcat can be started via the `startup.sh` script in the `bin` subfolder. The webapp can then be found under `http://localhost:8080/foobla`. Since the development process might require several restarts it is very convenient to add some aliases to the `~/.bashrc`:

```
alias tomstart='<Path to Tomcat>/bin/startup.sh'
alias tomshut='<Path to Tomcat>/bin/shutdown.sh'
alias tomrestart='tomstart && tomshut'
```

2.3 SemWebJena

The SemWebJena tool is a wrapper for an older version (2.10.0) of the Jena framework developed by the DBIS group of Georg-August-University Göttingen. [5] This wrapper is able to combine the Jena framework with various reasoners and is used for research and teaching within the group. It can be found and downloaded in the DBIS svn-repositories via the `svn export` command:

```
svn export https://svn.informatik.uni-goettingen.de/dbis/data-store/mondial-lod
```

Afterwards the complete tool can be build and deployed by calling the `build.xml` via `ant`. Next to the freshly build `mondial-lod.jar` the `ant` call will generate and deploy a `mondial-lod.war` web archive. In order to execute the jar-file conveniently the creation of an alias in the `~/.bashrc` is advantageous:

```
alias query='java -jar <path to folder>/dist/mondial-lod.jar'
```

With the command query -h all possible arguments and modifications are proposed. The web archive is to be put into the `webapps` folder of tomcat. Afterwards tomcat needs to be started with:

```
<path to tomcat>/bin/startup.sh
```

If the server started without any errors the webapp can be accessed via http://localhost:8080/mondial-lod. The default examples from the SemWeb lecture [6] must be stored in `/home/teaching/SemWeb/RDF`. This is due to the teaching examples of the DBIS group on their webserver are in this folder precisely.

2.4 JavaCC

> "Java Compiler Compiler is the most popular parser generator for use with java" [7]

This tool reads in grammar specifications and builds a Java program that parses matching expressions according to the grammar by generating Java source code. Java classes and functions can be used and triggered during the parsing process in order to e.g. fill own containers and objects with respective information. [7]

JavaCC binaries can be downloaded on https://javacc.org/download. Afterwards they need to be unpacked e.g. with `unzip javacc-x.x.zip`. Finally add the `bin` folder to `PATH` e.g. in ~/`.bashrc`:

```
export JAVACC_HOME='<Path to JavaCC>/javacc-x.x'
export PATH=${JAVACC_HOME}/bin:${PATH}
```

A closer look on the usage of JavaCC will be given in the next section 3.

3 JavaCC Tutorial

In this section a small example on JavaCC parsing is given. In the fictive scenario the data of a car shall be read from a file, stored into a suitable Java object and be reprinted as a CSV-tuple. For this purpose an easy grammar for a step by step example is created. All knowledge of the JavaCC syntax in this section originates from the tutorials and code snippets on the JavaCC Website. [7] While there are multiple tutorials for various use cases, this self made example shows the elemental knowledge needed to read and understand the Jena grammar later on.

3.1 CarGrammar Specification

The formal specification of our grammar *CarGrammar* is given by the 4-tuple of non-terminals N, terminals T, productions P and the start symbol S:

$$CarGrammar \coloneqq \{N, T, P, S\}$$

with:

$N \coloneqq \{Name, Color, Company, Price, Input\}$,
$T \coloneqq \{\langle \text{NAME} \rangle, \langle \text{COLOR} \rangle, \langle \text{COMPANY} \rangle, \langle \text{PRICE} \rangle, \langle \text{EOF} \rangle, \langle \text{LINEBREAK} \rangle, \langle \text{NUMERICVALUE} \rangle, \langle \text{STRINGVALUE} \rangle\}$,

Comment. *All ⟨FOOBLA⟩ terminal symbols are regular expressions. The exact regular expressions can be seen in the implementation section in listing 2. Since regular expression can also be expressed by regular sub-grammars within our grammar, they can be seen as a all in one processed terminal symbols.*

$S \coloneqq Input,$

$$
\begin{aligned}
P \coloneqq \{ & \\
& Input && \rightarrow Name \\
& Name && \rightarrow \langle \text{NAME} \rangle \langle \text{STRINGVALUE} \rangle \langle \text{LINEBREAK} \rangle Color \mid Color \\
& Color && \rightarrow \langle \text{COLOR} \rangle \langle \text{STRINGVALUE} \rangle \langle \text{LINEBREAK} \rangle Company \mid Company \\
& Company && \rightarrow \langle \text{COMPANY} \rangle \langle \text{STRINGVALUE} \rangle \langle \text{LINEBREAK} \rangle Price \mid Price \\
& Price && \rightarrow \langle \text{PRICE} \rangle \langle \text{STRINGVALUE} \rangle \langle \text{LINEBREAK} \rangle \langle \text{EOF} \rangle \mid \langle \text{EOF} \rangle \\
\} &
\end{aligned}
$$

3.2 CarGrammar Implementation

In order to have solid control over the car data within Java, a data wrapping class is needed. For the previously defined scenario a simple class will do the job. First of all some strings to store the name, color and company of the car as well as an integer for the price is needed. Since the object is to be build "on the fly" (while parsing the data), getter and setter methods are far more important than any complex constructor. In the end the implementation of the `serialize` method gives the possibility to handle the data as a 4-tuple of $(name, color, company, price)$. The source code of this class can be seen in listing 1

```
1  class CarJava{
2      private String name;
3      private String color;
4      private String company;
5      private int price;
6
7      // Default Constructor
8      public CarJava(){}
9
10     //Setter
11     public void setName(String name){ this.name = name;}
12     public void setColor(String color){ this.color = color;}
13     public void setCompany(String company){ this.company = company;}
14     public void setPrice(String price){ this.price = Integer.valueOf(price);}
15
16     //Getter
17     public String getName(){ return name;}
18     public String getColor(){ return color;}
```

```
19      public String getCompany(){ return company;}
20      public int getPrice(){ return price;}
21
22      //Serialize the object as CSV
23      public String serialize_as_CSV(){
24          return ""
25              + getName() + ","
26                  + getColor() + ","
27              + getCompany() + ","
28              + getPrice() + "\n";
29      }
30  }
```

Listing 1: Java class to store car data

In the listing 2 the definitions of TOKENS is given. TOKENS serve as terminal symbols. Parsing these is not only needed for determining the next rule but can also be used to return the parsed content e.g. as a string. As an example the string “bla bla” can be matched by ⟨STRINGVALUE⟩ and also its stringvalue can be returned to e.g. set an object property.

```
1   TOKEN:
2   {
3       < STRINGVALUE: ["a"-"z","A"-"Z"] ( ["a"-"z","A"-"Z","_"," ","0"-"9"] )* >
4       |
5       < NUMERICVALUE: ["0"-"9"] ( ["0"-"9"] )* >
6       |
7       < LINEBREAK: "\n" >
8       |
9       < NAME: "name: " >
10      |
11      < COLOR: "color: " >
12      |
13      < COMPANY: "company: " >
14      |
15      < PRICE: "price: " >
16  }
```

Listing 2: Regular Expression representing the terminals

The production rules of JavaCC look very similar to regular Java functions. Once chosen, a production rule will also behave like a Java function e.g. defining objects and returning values. As an example the production rule `Stringvalue()` assigns the parsed part to a token and returns its string value. JavaCC is designed to commit to terminal choices without any backtracking algorithm. In oder to commit to a terminal the `Lookahead` value is specified. This value determines how many TOKENS in the future are to

be considered for the next choice. By default this value is one. Further the processing of the grammar can be extended by e.g. logical “or” “|” or a logical “optional” “[Foobla()]”. Listing 3 shows all productions needed for the `CarGrammar` example.

```
1  void Input():{}
2  { [Name()] [Color()] [Company()] [Price()] <EOF> }
3
4  void Name():{String val;}
5  { <NAME> val = Stringvalue() {car.setName(val);} <LINEBREAK> }
6
7  void Color():{String val;}
8  { <COLOR> val = Stringvalue() {car.setColor(val);} <LINEBREAK> }
9
10 void Company():{String val;}
11 { <COMPANY> val = Stringvalue() {car.setCompany(val);} <LINEBREAK> }
12
13 void Price():{String val;}
14 { <PRICE> val = Numericvalue() {car.setPrice(val);} <LINEBREAK> }
15
16 String Stringvalue(): {Token t;}
17 { t = <STRINGVALUE> {return t.image;} }
18
19 String Numericvalue(): {Token t;}
20 { t = <NUMERICVALUE> {return t.image;} }
```

Listing 3: Productions with function names beein non-terminals

The JavaCC section from `PARSERBEGIN` to `PARSEREND` gives the opportunity to implement everything needed for the parsing process e.g. objects to be filled. It is also very common to inherit some functions and background processing of the parser from a PARSERBASE class by just specifying this inheritance. In the car example (listing 4) this code section is used to create a fresh `CarJava` object to be filled during the parsing process. Also a main function is included to enable playing with the grammar right away. Line 11 shows the starting non-terminal symbol `Input()` beeing called to initialize the parsing process.

```
1  PARSER_BEGIN(CarGrammar)
2  public class CarGrammar {
3
4      static CarJava car;
5
6      /** Main entry point. */
7      public static void main(String args[]) throws ParseException {
8          CarGrammar parser = new CarGrammar(System.in);
9              car = new CarJava();
```

```
10          /** Calling the start symbol*/
11          parser.Input();
12          System.out.print(car.serialize_as_CSV());
13      }
14
15  }
16  PARSER_END(CarGrammar)
```

Listing 4: Core part of the CarData Parser

Having the `CarGrammar.jj` and `CarJava.java` at hand the java compiler compiler `javacc` compiles all the needed source codes. Afterwards all Java files need to be compiled with a the regular java compiler.

```
1  $ javacc CarGrammar.jj
2  Java Compiler Compiler Version 6.0_1 (Parser Generator)
3  (type "javacc" with no arguments for help)
4  Reading from file CarGrammar.jj . . .
5  File "TokenMgrError.java" is being rebuilt.
6  File "ParseException.java" is being rebuilt.
7  File "Token.java" is being rebuilt.
8  File "SimpleCharStream.java" is being rebuilt.
9  Parser generated successfully.
10 $ javac *.java
11 $
```

Listing 5: Building the final parser program

After creating the Java program `CarGrammar`, a small testing example with random values is given in listing 6.

```
1  name: Golf 6
2  color: blue
3  company: Volkswagen
4  price: 20000
```

Listing 6: Data of a Car to be parsed

The above `CarData.txt` can then be used as input for the program. The output is a nicely formatted CSV-formatted string.

```
1  $ java CarGrammar < CarData.txt
2  Golf 6,blue,Volkswagen,20000
```

Listing 7: Output with complete & correct Datafile

As a second test example the same data from `CarData.txt` 6 is used with the deletion of line 2. The results show the same CSV-format with a `null` value for the missing line 2.

```
1 $ java CarGrammar < CarDataIncomplete.txt
2 Golf 6,null ,Volkswagen ,20000
```

Listing 8: Output with incomplete & correct Datafile

The last testcase shows the parsers behaviour with syntactically wrong input by adding the line "doors: 4" to the `CarData.txt` file. The parser prints out an exception with the malicious string and what was expected instead.

```
1 $ java CarGrammar < CarDataIncorrect.txt
2 Exception in thread "main" ParseException: Encountered " <STRINGVALUE> "doors
    "" at line 5, column 1.
3 Was expecting:
4     <EOF>
5         at CarGrammar.generateParseException(CarGrammar.java:275)
6         at CarGrammar.jj_consume_token(CarGrammar.java:213)
7         at CarGrammar.Input(CarGrammar.java:53)
8         at CarGrammar.main(CarGrammar.java:12)
```

Listing 9: Data of a car to be parsed

All in all this small example gives a brief insight in how to use `JavaCC`. Also the toy-example itself is build very similar to the grammar used in the Apache Jena framework.

4 Jena SPARQL & RDF processing

This section gives a brief overview of `SPARQL` and `RDF` processing. The specification of the SPARQL grammar is given in the file `master.jj`. By using `javacc`, this grammar is compiled to a parser that will be used by default in the `QueryFactory`. A QueryFactory object describes how a Query object shall be build e.g. what parser shall be used. Having a parser at hand the QueryFactory can create a Query object from a `SPARQL-file`. Those Query objects are the internal Java representation of every information parsed by the Parser e.g. query form, prefixes. Query objects do also contain a list of references to RDF source-files. These sources get parsed seperately by a `DatasetFactory`. The created model is then, together with the query object, given to the `QueryExecutionFactory` to create a `QueryExecution`. The internal conversion of the abstract Query object to a `Algebra-Plan` is a currently not of interest and therefore omitted. The following Figure 1 gives a fundamental overview of the processing as a simple flow chart. [8] [9]

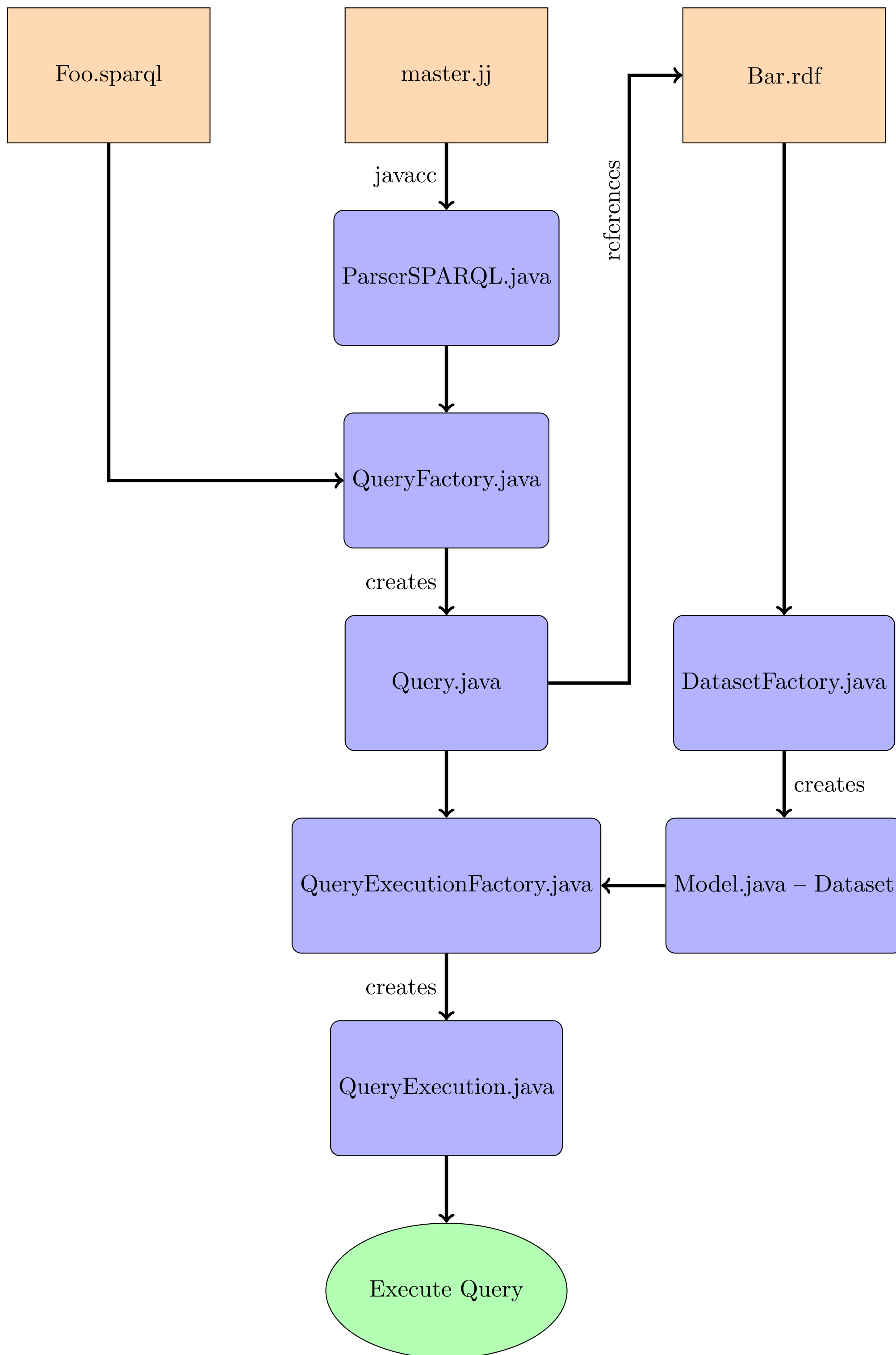

Figure 1: Jena flow from parsing files to an execution object

5 Master's thesis outlook

Typically a SPARQL query has the three main building blocks of `SELECT`, `FROM` and `WHERE`. The SELECT is the most common representative of the possible query forms. In the FROM section typically references to RDF source-files are given. Finally the WHERE describes the graph pattern to be matched for possible solutions. Listing 10 gives a simple example of a typical SPARQL query returning all subjects of the triples in `foobla.n3`.

```
1 prefix : <foo://bla/blubb#>
2 select ?S
3 from <foobla.n3>
4 where { ?S ?P ?O }
```

Listing 10: Regular simple SPARQL query

If there is an other `CONSTRUCT` query instead of a RDF file, this is called a `nested CONSTRUCT` query. Such a nested CONSTRUCT query is shown in Listing 11.

```
1 prefix : <foo://bla/names#>
2 select ?X
3 from
4     { construct { ?G :grandchild ?X . ?X :name ?N}
5       from <file:john.n3>
6       from <file:parents.n3>
7       where {?G :hasChild ?P . ?P :hasChild ?X . ?X :name ?N}
8     }
9 where { ?X ?P ?Y }
```

Listing 11: Nested CONSTRUCT query[6]

These kind of queries are not yet possible in `SPARQL 1.1`, because they are believed to be fully replaceable by nested SELECT queries. This is more or less proven by Polleres et al. [10] by defining a rewriting algorithm. However these approaches of rewriting such a query are exponential in the depth of nested CONSTRUCTS. [10] That is why a nested CONSTRUCT might be a useful feature in the SPARQL language. The upcoming master's thesis will go deeper into the theoretical parts of nested CONSTRUCT queries. Also a experimental implementation for testing these kind of queries will be created. Therefore the Jena framework will be slightly rewritten. The goal is to have the grammar accept nested CONSTRUCTS and afterwards handle these in a suitable way. If these nested CONCSTRUCTS are a useful feature, they will be added to the SemwebJena tool.

References

[1] The Apache Software Foundation. Apache Jena. https://jena.apache.org/index.html, 2019. [Online; accessed 11.06.2019].

[2] The Apache Software Foundation. ARQ. https://jena.apache.org/documentation/query/index.html, 2019. [Online; accessed 11.06.2019].

[3] Wolfgang May. Playground page for the XML Course. http://www.stud.informatik.uni-goettingen.de/xml-lecture/#tomcat, 2019. [Online; accessed 11.06.2019].

[4] The Apache Software Foundation. Apache Tomcat. https://tomcat.apache.org/, 2019. [Online; accessed 11.06.2019].

[5] Wolfgang May. Databases and Information Systems. https://www.dbis.informatik.uni-goettingen.de/, 2019. [Online; accessed 11.06.2019].

[6] Wolfgang May. Semantic Web Lecture. https://www.dbis.informatik.uni-goettingen.de/Teaching/SemWeb-WS1819/, 2018/2019. [Online; accessed 11.06.2019].

[7] JavaCC Team. JavaCC - The Java Parser Generator. https://javacc.org/, 2019. [Online; accessed 11.06.2019].

[8] The Apache Software Foundation. ARQ - Application API. https://jena.apache.org/documentation/query/app_api.html, 2019. [Online; accessed 11.06.2019].

[9] S. Harris and A. Seaborne. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/, 2019. [Online; accessed 11.06.2019].

[10] Axel Polleres, Juan Reutter, and Egor V Kostylev. Nested constructs vs. sub-selects in sparql. *Alberto Mendelzon International Workshop on Foundations of Data Management*, 10, 2016.