RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN
INSTITUT FÜR INFORMATIK III
Enterprise Information Systems

RDF Doctor: Holistic RDF Syntax Validation and Error Correction

A thesis submitted in fulfillment of the requirements for the degree of
Master of Science in Computer Science

BY

Ahmad Hemid
Matriculation number: 2915575
E-mail: ahemaid@gmx.com

First Evaluator: Prof. Dr. Jens Lehmann
Second Evaluator: Dr. Steffen Lohmann
Supervisor: Lavdim Halilaj

September 2018

Declaration of Authorship

I, Ahmed Hemid, declare that this thesis, titled "RDF Doctor: Holistic RDF Syntax Validation and Error Correction", and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Ahmed Hemid

Signature: _____

Date: _____

I would like to thank

- My parents
- My wife
- My children

Contents

List of Figures

List of Tables

Abstract

Chapter 1

Introduction

1.1 Introduction

More and more the usage of RDF is increasing in many fields in computer science. RDF data representation helps in supporting the machine to perform the normally manual computation work in an automatic fashion. Moreover, the machines will be smarter to understand the data which is represented in RDF format.

The quality of RDF data needs to be ensure before proceeding of any further processing. Most of current parsers which focus on detection of the syntax error fail to detect more than one error, especially, of RDF data represented in Turtle or NTriples format.

This study was encouraged by the tremendous data representation of either Turtle and NTriples. Hence, the intention of the study to afford a user-friendly syntax checker or parser. Such parser or syntax checker should give all errors can be detect inside such data.

1.1.1 Motivation

This study was motivated by several scenarios which require syntax checking of RDF data as an input and ensuring of its quality. To mention one of such scenarios, let's discuss an example shown in *Figure 1.1. The example describes a collaboration system for processing an input data, say for example to perform machine learning analysis. Of course,in a such case, a valid input data to the system is must. The*

input data is verified for further data processing. Most of the current existing systems ensuring syntax-error-free RDF data, are stopped parsing at the first syntax error occurrence, as will be followed in Section **??**.

To stop parsing when the first syntax error is found will introduce much complications. Assuming, the input RDF data contains, for example, 10 syntax errors. Normally, what is happing when an error is found, the system will proceed with no further processing, instead it will report error's existence in the inserted data. The reported error then should be corrected by the user, then after correction data will be send back for re-checking of data's syntax. To make it more complex, imagine that the user will do such correction process for 10 times (remember that data includes 10 errors). Then, what if the data contains hundreds or thousands of errors.

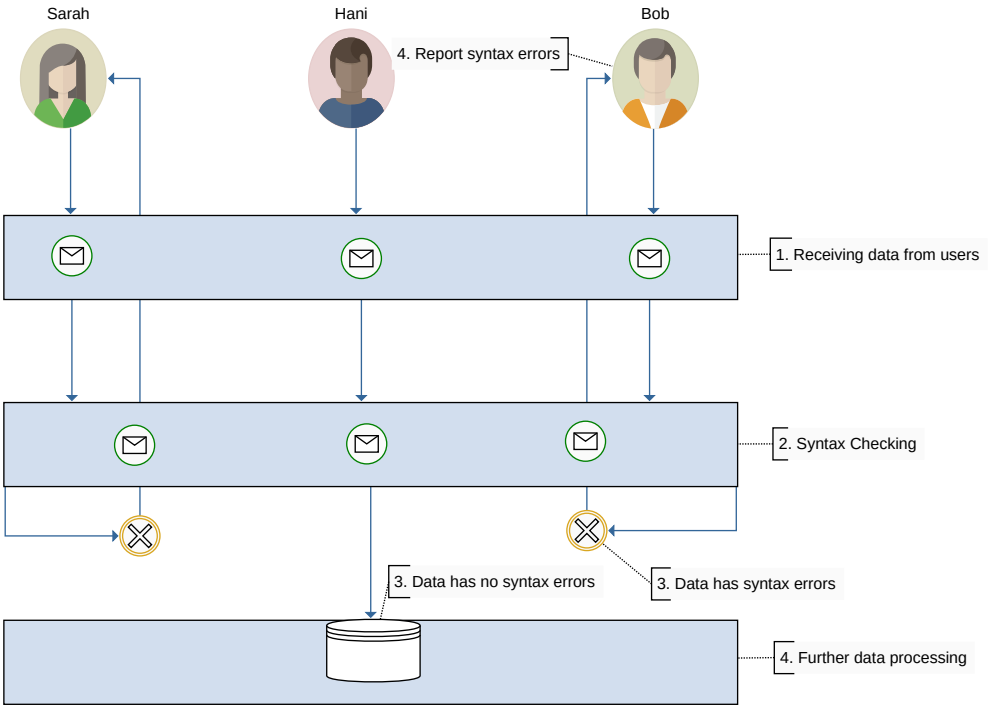

FIGURE 1.1: A motivation example of syntax checking of data before further processing

let's dig deep to explain what is there in Figure 1.1, It is showing a flow of data from clients or users seeking further data processing. 3 persons are shown in this Figure, their names are Bob, Hani, and Sarah. All of them start with the first phase

by sending the data to be syntacticly checked. The parser starts checking if there syntax errors of the input data, then if such data passed with no syntax errors, it can be forwarded for further phases, i.e. for data processing, otherwise, the input data will send back to the user to correct the errors. Figure 1.1 clearly shows that Sarah and Bob have syntax errors in their input data,then they got an error report, including found errors. In the meanwhile, Hani has received his data processed without getting such an error report, since his input data has no syntax errors.

This study has been encouraged by the illustrated example to find a suitable solution for such cases. The proposed solution will focus on producing a software program that can detect all or almost syntax errors that can be detected in the input data.

1.2 Proposed Problem

The tackled problem in this study is to list all the detected syntax errors found in RDF files; in order to help ontology engineers and users to fix them in one shut. Such a list of errors provided in the output of the syntax checking phase will significantly assist to get rid of a loop of first error notification if multiple errors are found in the input data.

1.3 Contributions

The contributions of this study are:

1. **Reporting list of detected syntax errors found in RDF data:**
2. **Showing expressive error messages:**
3. **Correcting some errors:**

1.4 Thesis Structure

The remainder of this document is structured as follows. In the Related Work chapter

The thesis is organized as follows. The next section presents the problem description. Section **??** *reviews the up-to-date research works. Section* **??** *describes the data set and Section* **??** *presents the results. Finally, Section* **??** *concludes*

Chapter 2

Preliminaries

2.1 RDF Model

The "Semantic Web" [?] term has appeared during the transformation process of Web Development from "Web of documents" to "Web of data", similar to those data are found in databases. W3C defines it as "the Web of linked data" . Figure 2.1 describes the Semantic Web Stack, proposed by W3C. It can be seen, that it contains several technologies to enable users of creating their own data stores on web, building vocabularies, and enforcing processing rules on such data.

In order to make data more and machine-readable, and easier for interchange, RDF Model [?] has been proposed. RDF Model plays the role of data interchange in the layered Semantic Web Stack and leverages the high-level with low-level semantic web tools as it is shown in Figure 2.1.

Figure 2.2(a) exhibits RDF Model's representation of data in triples. A triple is defined by 3 main players: subject, predicate, and object. It has a common similarity of a basic structure of simple sentence which consists of a subject, an object, and a verb. As the verb shows a relation or an event between the other two entities, similar, the predicate connects both subject and object in a certain relation.

A similar Resource-Property-Value structure to Subject-Predicate-Object of RDF Model's representation is drawn in Figure 2.2(b). It is same representation but Subject, Predicate, and Object are replaced with Resource, Property, and Value in sequence.

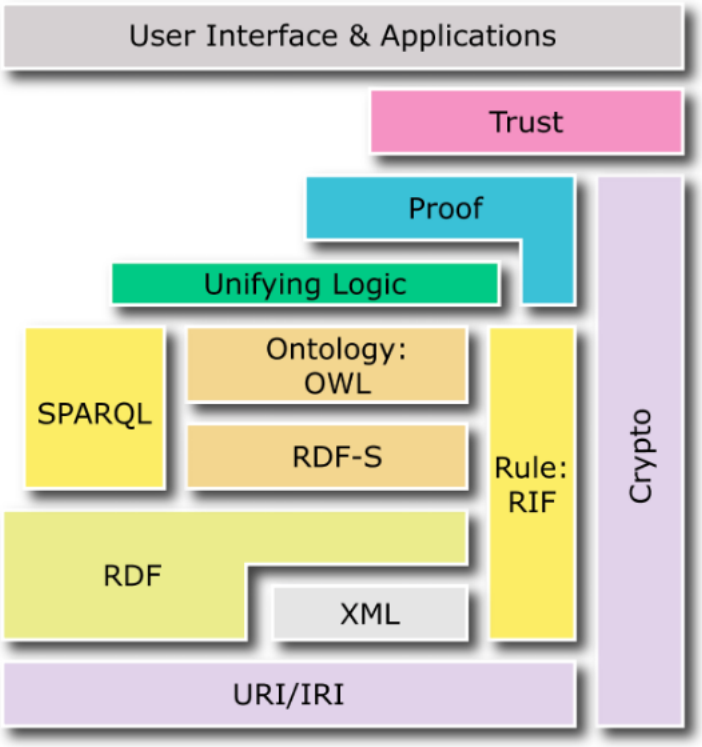

FIGURE 2.1: The Semantic Web Stack [**?**]

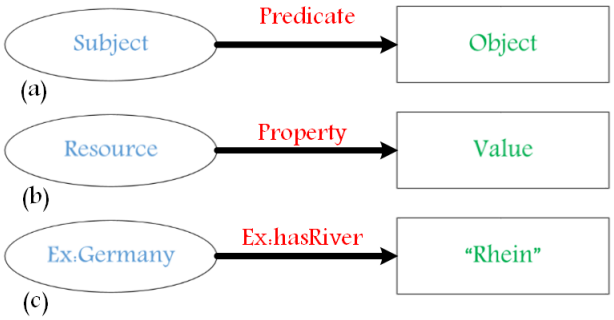

FIGURE 2.2: RDF Model's Structure

2.1.1 Simple RDF Model Example

*To make it more clear, a simple example of RDF Modeling is giving in Figure 2.2(c). To represent a fact that "German has The Rhein river", Ex:Germany is Subject/Resource, Ex:hasRiver is Predicate/Property, and "Rhein" is Object/Value. The first two value are Uniform Resource Identifiers (URIs) where "Ex:" is a defined path in RDF files and the following part is an extension of the path. The last value "Rhein" is a literal or a string. More examples can be viewed at [**?**].*

2.2 Parsing and Error Recovery

2.2.1 Parsing and Grammar Definitions

*It is of great benefit to shed a little light on parsing topic to understand later the approach of this study. In [**?**], Gabriele Tomassetti has defined* ***Parsing*** *by:*

"The analysis of an input to organize the data according to the rule of a grammar"*. It can be intelligibly recognized that parsing deals with some input data and tries to analyses it based on a given grammer. let's have a look on the definition of grammer based on [**?**].* ***Grammer*** *is* ***"A formal grammar is a set of rules that describes syntactically a language"****. In the definition, rules are the significant part of the grammar to define the language's syntax.*

2.2.2 Parser's Role

The parser is an essential part in a compiler. Figure Figure 2.3 draws the role of the parser in a compiler. The first phase "Lexical Analyzer" receives the input data, then produces tokens for the next phase "Parser". Parser constructs a parse tree based on its grammar rules and gets the next token till it consumes all tokens. The parse tree as an parser's output is delivered to the next phases of a compiler. If any errors are found, lexical errors can be generated by Lexical Analyzer and Parser is the producer of both Syntax and semantic errors.A symbol table is data structure entries to store some declared values in the input, such as in programming languages object and variable names. It is used by both Lexical Analyzer and Parser as it is shown in Figure Figure 2.3.

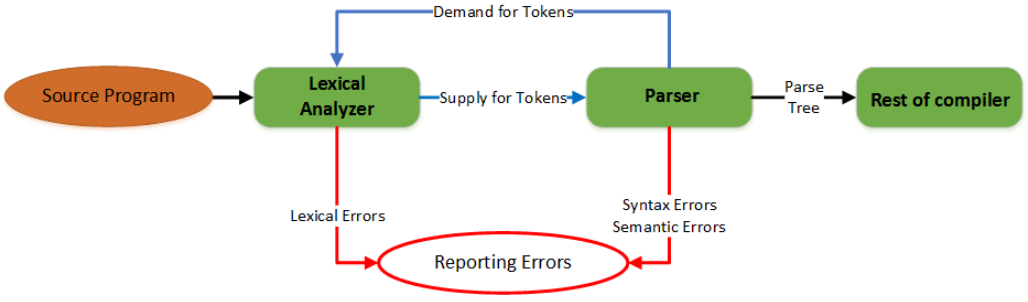

FIGURE 2.3: The Role of Parser

2.2.3 Parser's Constructions

In order to build a parser, there are two options to get such objective, either by building your own hand-written parser or getting an off-the-shelf parser generator tool. follows both approaches are discussed:

1) Hand-writing Parsers: *someone writes his own parser, including the lexical analyzer and the parser. On the one hand this way gives more flexibility in handling several problems triggered during development but on the other hand it consumes much time and mind.*

2) Parser Generators: *someone uses a tool to generate a parser to do specific parsing task, by writing a compilable grammar to such tool. The most advantage of using those tools is the less consumed time to build a parser.*

2.2.4 Error Recovery Methodologies in Parsers

*Error Recovery Strategies are the behavior of a system once an error is detected. Compiler's researchers Aho et al [**?**] have classified such methodologies as follows:*

- ***Panic-Mode Recovery****: Once an error has been discovered, the parser ignores and skips next input symbols one by one till a recognized set of tokens is detected. In spite of discarding sometimes huge amount of input symbols with checking them for further error detection, it is considered a simple parsing mode and do not fall in an infinite loop while other modes may do.*
- ***Phrase-Level Recovery****: To continue the parsing process, the parser performs local correction. A typical local correction in RDF is to insert a missing dot or delete extraneous dots.*
- ***Error Productions****: By expecting the common errors, production rules can be inserted into the grammar. Then, the generated parser is well-informed about such errors. Error diagnostics can be easily done in this case since the error in our hand.*
- ***Global Correction****: The parser tries to reduce as much as possible number of change operations (Insertion, deletion or modification) when dealing with an incorrect input token to reduce globally total cost of error correction. To make it more clear, let's assume an incorrect input statement X is giving in grammar G, the parser constructs a closest error-free parse tree of statement Y to replace statment X, such that changes are small as possible.*

2.3 ANTLR Parser Generator

ANTLR is an handy tool and easy way to generate a specific domain language parser. It is an parser generator to do automatic generation of a source code of such parser without much coding and with less time. The basic principle used by ANTLR is define language's rules which draws the syntax and the semantic of the language the parser build for.

As has been previously discussed, the compiler has two main subsystems: lexer and parser. Both lexer and parser are needed to have their rules defined in ANTLR grammar file. Figure Figure 2.4 demonstrates an auto-generated parser program, generated based on ANTLR tool. follows is a sequence steps **?** *, describes such parsing process:*

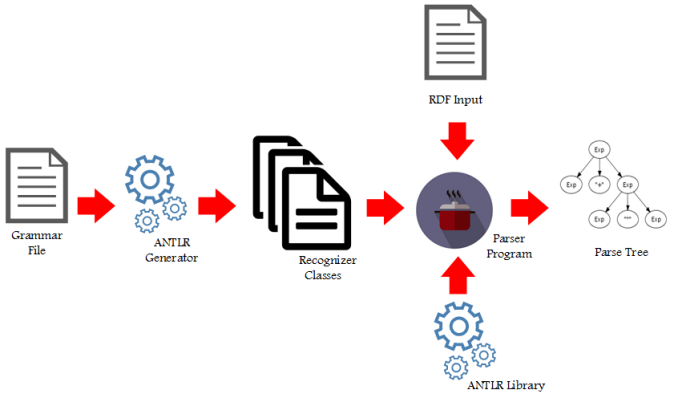

FIGURE 2.4: Parsing process based on ANTLR parser generator

1. **Writing a grammar file:** *ordinarily, a grammar called a parsing expression grammar, or PEG is required to build a parser. ANTLR needs a context-free grammar, crafted with Extended Backus-Naur Form (EBNF). EBNF consists of a sequence of rules. These rules describe the syntax and the semantic of an input which needs to be parsed. Either Terminal or Non-terminal entities are the components of these rules. On the one hand, Terminals are the leaf elements where they have no grammatical structure, e.g., words or numbers. On the other hand, Terminal entities are those with a definite grammatical structure and name, e.g., triple component.*
2. **Generation of recognizer target_based classes by ANTLR generator:** *a significant feature of ANTLR is its capability of generated the auto-generated parse code for a variety of the programming languages* **?** *: Java, C#, Python (2 and 3), JavaScript, Go, C++, and Swift.*

3. **Feeding an input file for parsing:** *as an input, ANTLR can parse text documents without additional libraries. Users send their input files for parsing where the parsing process will be achieved based on the crafted grammar in the first step.*
4. **Parsing procedure:** *figure 2.4 shows that it is the cooking time in this step where all materials and Ingredients are ready. The materials are: 1) the auto-generated parser, 2) the ANTLR library. The ingredients are one or more text files to be parsed.*
5. **Delivering of a report and parse tree as an output:** *now the cake is ready for eating, an output of this process is given. A parse tree and a report of collected errors can be such output.*

Chapter 3

Related Work

In order to validate RDF code, either by pasting URL where it exists or by uploading a file, almost the available tools and applications that we could find, will only give the first occurred error. Moreover, semantic developers and engineers will struggle in debugging their codes and they need alternative tools that could be more helpful. To the best of our knowledge, there is no comparable prior work regarding fault-tolerant tool to validate syntactically RDF serialization formats expect one that works only for RDF/XML format, the following text sheds light on this tool. The new proposed tool should feature prominently in listing of all errors included in the code.

*This section reviews the related research works have been done and presents the current state of the art of RDF syntax validating. Despite the long record of RDF syntax validation research with many of theoretical models or practical tools, we can hardly find a research that describes the challenge of detection multiple syntax errors inside RDF code. During our journey of checking the existing tools that provide such validation service, The W3C RDF validation tool [**?**]W3C:Validation:Online was firstly checked, it is available online for parsing and validating RDF/XML codes. It uses the ARP parser of Jena [**?**] as a backend. However, it fails in detection of multiple syntax errors, the first error in the order will be only released. K. Tolle developed a Validating RDF Parser (VRP) [**?**] in his thesis, VRP is a Java-based build tool, and it validates RDF/XML code semantically and syntactically. Nevertheless, the validation service provided by VRP is limited to RDF/XML and does not support other RDF serialization formats, especially those formats which are structured in triples such as N3, NTriple, and Turtle. In this work, extension of syntax validation to other formats is planned.*

*The journey to check the existing tools that validate RDF serialization formats other than RDF/XML is continued. As previously stated, Jena RDF toolkit [**?**] offers validation service based on ARP parser. It can be used as a command-line program (standalone) or as an API within another application. Despite its ability to validate numerous RDF serialization formats, including RDF/XML, again, the first error is only reported. Some of the tools validating RDF formats use the following core techniques as a significant part of their implementations:*

- ***ARP-parser-dependable approach :*** *both W3C RDF validation tool [**?**] and RDF Validator and Converter [**?**] use ARP parser of Jena [**?**]. Moreover, the latter focuses more on triple-based serialization formats, validating them and converting from one format to another, where the former validates only RDF/XML format.*
- ***N3-parser-dependable approach :*** *N3 parser can also be used for syntax validation. In the online IDLab Turtle Validator [**?**], N3 parser powered by N3 NodeJS library is used. As well, same approach was used to build a turtle editor with syntax validation in [**?**].*
- ***Shape expressions approach :*** *in [**?**] a turtle parser was developed based on shape expressions. Shape expressions validates RDF through declaring of constraints on the RDF model, if the declared constraints are violated, then RDF is invalid. Furthermore, Shape expressions describes the RDF graph on regular expressions base.*

When it comes to the application side, N3-parser-dependable approach can be fitted perfectly in the new tool, since it is built using Nodejs library. This can improve the performance of the tool, especially when it is validating a large RDF code. Moreover, the first two approaches are more expressive in explaining the syntax error and its location, where as, the tool used the third approach is less expressive.

In this research, our intention goes toward inventing a fancy tool that lists all syntax errors with an improved performance. The proposed tool can have a solution for the explained issue in either two ways:

- ***Patching the output errors of parsers :*** *while reviewing the source codes of others' tools, an error event by an error handler will be emitted to show the first occurred error. An idea of looping inside the RDF code and fixing eachtime the first error can be suggested. Fixing the error can be by either deleting the triple made the error, removing or adding a punctuation, inserting*

a dummy IRI for an incorrect or missing one, etc, then reprase the RDF code again and again till the end of the code .

- ***Parser Optimization :*** *this needs to review deeply the whole code of the parser and improve its method. The improvement should list all syntax errors that the parser can detect. Both parsers built with N3-parser-dependable approach or Shape expressions approach can be optimized to reach our goals while the optimization of the latter inherits more complexity.*

To end this section, after describing the actual issue, reviewing the state of art of research works related to it, and finally presenting the possible solutions, we can say that both two solutions can solve the issue, but it seems to us that the second solution more efficient than the first, since this is the normal way how acually most of editors of programming languages work, to alert on-the-fly syntax errors to the programmer, even before compilation .

Chapter 4

Approach

4.1 Challenges Before

Before reaching the proposed solution, different methods have been tired to reach the objective. The main strategy was the continuation of parsing after error detection with reporting the detected error. The following 3 tools were individually tested:

1. **Jena RIOT API**
2. **RDF4J RIO API**
3. **N3 Parser**

In the first 2 tools, the focus were to handling the error and continue based on predefined error messages but unfortunately the error message sometime are unexpected and not from the predefined ones. The last tool handles errors of lexer and parser in different methods, that results in inability to collect them altogether in one method.

4.2 Proposed Solution

4.2.1 Algorithmic Representation

Algorithm 1: The algorithmic representation of the proposed solution

1 indent=2em [1] $syntaxRules \leftarrow correctSyntaxRules\&uncorrectSyntaxRules$ token in inputText && $inputText \neq EOF$ tokenToBeMatched += token syntaxRules contains tokenToBeMatched uncorrectSyntaxRules contains tokenToBeMatched Save this syntax error with tokenToBeMatched string to the error report CorrectionIsSelected ThisErrorCanBeRecovered Recover the wrong syntax with a correct one Save This correction to the output report $tokenToBeMatched \leftarrow$ "" $token \leftarrow$ ""

Chapter 5

Implementation

In this chapter, details of the implementation practical part of the study is discussed. The architecture, modules, development tools, installation, and configurations of such part are the main role of this chapter.

5.1 Architecture

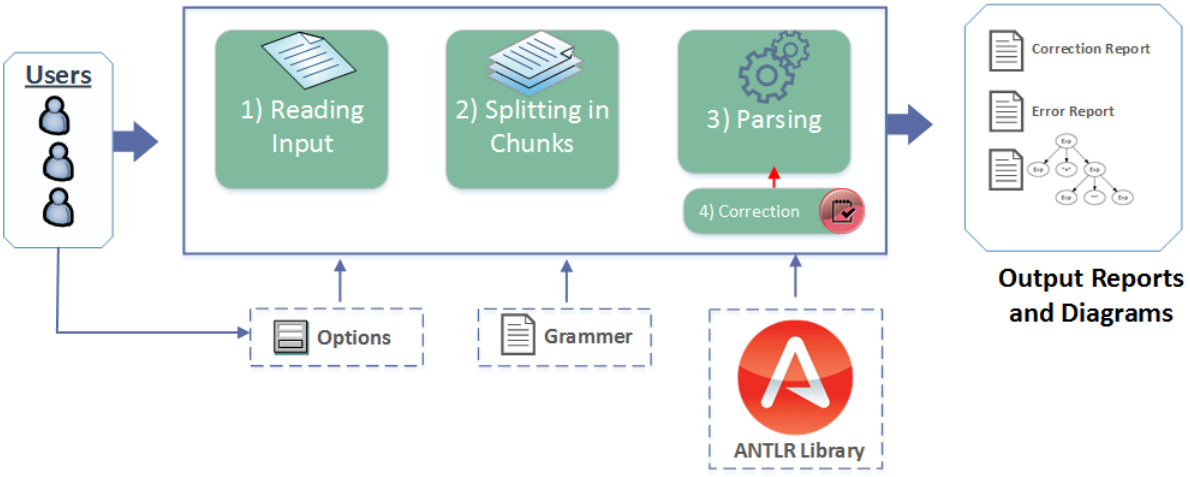

FIGURE 5.1: The proposed solution architecture

5.2 Modules

5.3 Tools

5.3.1 Hardware

5.3.2 Software

5.4 Installation and Configurations

Chapter 6

Evaluation

6.1 Experiments and Results

To evaluate the proposed solution, firstly, it was tested against standard test cases, secondly, it was compared against XTurtle framework, lastly, it was validate with arbitrary large RDF files. This chapter discusses all of such details in the following text.

6.1.1 Testing with RDF SuiteTest

The evaluation phase finds its starting point at **?** *where Test Suite files of Turtle serialization are found. Any new parser (focused on the same RDF serialization) can its proficiency be validated against these files.*

In **?**

6.1.2 Comparing the Proposed solution and Xturtle

6.1.3 Validation against arbitrary large RDF files

6.2 Accuracy

6.3 Scalability

Chapter 7

Conclusions and Future Work