POLITECNICO DI MILANO

Master of Science in Computer Engineering

Dipartimento di Elettronica e Informazione

Extracting Verbal Frames from Natural Language Sentences

Relatore: Prof. Licia Sbattella

Correlatore: Ing. Roberto Tedesco

Thesis by:

Kadir GUNEL , 762552

2013 - 2014

To Seçkin...

Abstract

This study describes an approach for the Italian language by extracting verbal frames to determine their complementary parts within sentences. The model uses a theoretical approach which is designed specifically for Italian. Two sequentially dependent technologies are used for putting this plan into practice : A dependency parser and a rule based system. The parser returns the sentence structure as a set of dependency among words; the rule based system transforms such representation, permitting to recognize the subject, the predicate and the related complements. This model is evaluated by simply giving Italian text files. The results show that the model covers as much information as possible and gives results accordingly. However, the lack of information between verbs and its possible complements might cause inaccurate outcomes when given a sentence. As a future work, this consequence implies the need of an auxiliary system which holds semantical information of Italian verbs and their usages in sentences such like VerbNet and FrameNet systems for English language.

Sommario

La tesi propone un'approccio, mirato alla lingua italiana, per estrarre la struttura dei complementi delle frasi. Il modello si basa su una caratterizzazione della struttura delle frasi, specifica per l'italiano. Due metodologia sono utilizzate: un dependency parser e un sistema a regole di produzione. Il parser ritorna la struttura della frase, sotto forma di relazioni di dipendenza tra le parole; il sistema a regole di produzione elabora tale rappresentazione, riconoscendo il soggetto, il predicato e i relativi complementi. Il modello è stato valutato utilizzando un insieme di testi italiani; i risultati mostrano che il modello ritorna i risultati attesi. Il modello non utilizza informazioni a priori circa i complementi che ciascun verbo può reggere. A causa di ciò, in alcuni casi, i risultati possono essere errati. Come sviluppo futuro, si prevede di incorporare un database semantico dei frame verbali, sul modello proposto da VerbNet o FrameNet per la lingua inglese.

Acknowledgements

I can not separate you from others. You all helped me to find my path for going "*home*". I am thankful to *all of you,* one by one, by my heart.

Kadir GUNEL

Contents

1 Introduction **5**
- 1.1 Motivation . . . 5
- 1.2 Purpose of Thesis . . . 6
- 1.3 Thesis Organization . . . 6

2 Used Technologies **7**
- 2.1 Linguistic Framework . . . 7
  - 2.1.1 Parser Types . . . 7
  - 2.1.2 LinguA Parser . . . 9
  - 2.1.3 TextPro Tool Parser . . . 10
  - 2.1.4 TULE Parser & TUT TreeBank . . . 11
  - 2.1.5 Motivation of Selected Technologies . . . 12
- 2.2 Rule Based Systems . . . 13
  - 2.2.1 Architecture of a Rule Based System . . . 14
  - 2.2.2 Working Strategies of Rule Based Systems . . . 15
  - 2.2.3 Existed Technologies . . . 16
    - 2.2.3.1 Prolog . . . 16
    - 2.2.3.2 Drools JBOSS . . . 16
    - 2.2.3.3 CLIPS . . . 16
    - 2.2.3.4 JESS . . . 16
  - 2.2.4 Motivation for Selected Technology . . . 17
- 2.3 Java . . . 18
- 2.4 JCommander . . . 19

3 Design **20**
- 3.1 Sentence Structure in Italian . . . 20
  - 3.1.1 Complement Types . . . 20
  - 3.1.2 Verbs in Italian . . . 21
- 3.2 Logical Analysis . . . 21
- 3.3 Lexical Approach . . . 22

3.4 System Design . . . 22
3.5 Design of Java Classes . . . 23
3.6 Design of Rule Files . . . 24
3.6.1 First Rule File . . . 25
3.6.2 Second Rule File . . . 26
3.6.3 Third Rule File . . . 27
3.6.4 Summary of Rule Files . . . 29
3.6.5 Scripting . . . 31
3.6.6 Problem of Disordered Words and its Solution . . . 32
3.6.7 Queries . . . 32

4 Implementation 34
4.1 System Architecture . . . 34
4.2 Program vs. Tool Set . . . 35
4.3 Splitting Tools . . . 35
4.4 Details of CallParser . . . 36
4.4.1 Decompilation . . . 39
4.4.2 Reflection . . . 39
4.5 Details of CallRuler . . . 40

5 Conclusion and Future Work 45

Bibliography 45

A User's Manual 48
A.1 CallParser User Manual . . . 48
A.2 CallRuler User Manual . . . 50
A.3 A Crucial Note About the Terminal Commands . . . 50
A.4 Setting Tule Server . . . 51
A.4.1 Steps for Installing TULE Server . . . 51

List of Figures

2.1.1 The LinguA Dependency Tree output for the sentence: "*Io e Marco siamo andati al mare.*" 10
2.1.2 The TextPro Dependency Tree output for the sentence : "*Io e Marco siamo andati al mare.*" 10
2.1.3 The TULE Parser's Dependency Tree for the sentence : "*Io e Marco siamo andati al mare.*" 11
2.1.4 TUT file example 12
2.2.1 A general architecture of a rule based system 14
2.2.2 More detailed architecture of the rule based system 14

3.1.1 Complement categories and their elements in Italian 21
3.4.1 System Overview 23
3.5.1 Word Object Relations 24
3.6.1 Rule Engines and their proper Objects 25
3.6.2 A simple rule for separating a sentence with two predicates 26
3.6.3 Example rule for finding auxiliary verb 27
3.6.4 Dependency tree for "Il gatto di mia soralla" 28
3.6.5 Example rule for finding complemento di specificazione for Subject . . 29
3.6.6 Objects in Rule Engine 1 30
3.6.7 Objects in Rule Engine 2 31
3.6.8 Objects in Rule Engine 3 31
3.6.9 Query example 33

4.1.1 General System Architecture 35
4.3.1 System architecture with an Extra Component 36
4.4.1 CallParser Packages 36
4.4.2 Command UML 37
4.4.3 File Package UML 38
4.4.4 Parser Package UML 38
4.5.1 CallRuler Packages 40

4.5.2 Commander Package 41
4.5.3 Files Package 41
4.5.4 Turin Package 42
4.5.5 Transformer Package 43
4.5.6 Rules Package 44

Chapter 1

Introduction

1.1 Motivation

In recent years, research and development of Artificial Intelligence (A.I.) and its subfield Natural Language Processing (N.L.P.) has increased so well that nowadays companies and academicians work on projects that were science fiction stories only a decade ago.

Today, NLP programs that are available and their development kits in the global market are focused on human to computer interaction as in a computer to understand human language. These products or in a better term programs can be based on speech recognition, text summarization, text translation or natural language[1] which is the best one among out of all. Main contributors to both programs and development kits are large companies like Microsoft, Apple and Google but there are also supporters from universities, like MIT NLP, Stanford NLP and Berkley NLP groups. As it is obvious, all these contributors are based in North America so the programs are developed for English speakers. There are only a few products and APIs that are developed in other languages. Hence this lack encouraged me to build a subsystem for Italian language, which finds the complementary parts of human sentences autonomously. This subsystem will be capable of adapting itself to a well developed NLP system that has the ability to process a natural language other than English. Thus creates the principle motivation for this thesis.

My second motivation for this thesis is too important to be ignored: Education of the youth. Today, both public and private schools are still teaching grammar by using traditional approaches. Since, schools don't offer one-to-one education the possibility of lack of communication between the teacher and the student is very high. Considering that computers are ubiquitous today and that they are spreading unavoidable, the usage of NLP programs could be a complementarity part in both

learning and teaching a natural language.

1.2 Purpose of Thesis

A natural language parser program focuses on generating a general grammatical structure. The program does not give conclusive information only indicates is a group of words forms this complement type. It provides information for all words in a sentence about their lemma, part of speech, position in the sentence and dependency to another words. Also, by using its internal system, it gives basic information about the direct complements of a sentence as in; subject, object and verb. Consequently, groups of words, which form indirect complements by using prepositions, cannot be supplied directly by the parser. As mentioned in the motivation, if there is a need for building a system, such that does sort things like text summarization or text translation. The very first centralization should be covering the lack of producing indirect complements by building a natural language tool. Hence, the focus of this thesis is about closing this lack of information first by finding a proper linguistic method for logical analysis of a given sentence. Secondly, this approach should be implemented in a rule-based system such as the overall system should be both capable and flexible of interacting with other subsystems of larger systems.

1.3 Thesis Organization

The rest of this document is organized into following chapters:

Chapter 2: Used Technologies. This chapter gives information about what kind of technologies are necessary for development of this thesis, which of them are available for Italian. It also explains required coding philosophy by giving reasons in detail.

Chapter 3: Design. This chapter explains the problem and shows a suitable approach for solution.

Chapter 4: Implementation. Explanation of system architecture and implementation details are described by giving code snippets and UML diagrams.

Chapter 5: Conclusion and Future Work. Conclusion and the required improvements for future development are given by complying results of the program.

Chapter 2

Used Technologies

In order to create a system which understands a natural language, there is need of having subsystems which can somehow "*understands*" the grammatical structure. Therefor the meaning of input streams will be passed to the next phases by gradually increasing the information on that input. To achieve this, first a sub-system is needed which is called ***Parser***. Thus forming *the linguistic framework* of the system is done but this is not enough. There is still need of using this information for generating new ones between "*groups of words*". And to produce these "*groups of words*" which form complements of a given sentence another suitable technology is required. To do so a *rule based rule system* is needed.

This chapter will introduce you technologies that are used for building overall system. Please note that to have a successfully running program, deciding the required technologies for that program is crucial. To show this importance, this chapter describes essential technologies, their implementation variants by different groups of people. Finally, reasons why particular ones are selected among others are discussed in detail by giving personal thinking.

2.1 Linguistic Framework

This section starts with the description of different kind of parsers. Then it gives information about different Italian Parsers that are available in the industry. Then how these parsers process same input sentence is shown. At the end of this section motivation for the selected parser is described.

2.1.1 Parser Types

There are different types of NLP parsers and each finds different usages in both research and industry. The main ones can be listed as :

- Shallow Parser
- Semantic Parser
- Shallow Semantic Parser
- Probabilistic Parser
- Full Parser

A *Shallow Parser* analyzes a given sentence in a way that it outputs only the syntactical information by dividing the sentence into chunks[2]. It can be associated to the Lexical Analysis of a programming language. And due to this it is called *light parsing* too . Example :"*Jack and Jill went up the hill to fetch a pail of water.*" An output from Illionis Shallow Parser for this example[3] :

```
[NP Jack and Jill ] ; NP– Noun Phrase
[VP went ] ; VP – Verb Phrase
[ADVP up ] ; ADVP – Adverb Phrase
[NP the hill ] ; NP– Noun Phrase
[VP to fetch ]; VP – Verb Phrase
[NP a pail ] ; NP– Noun Phrase
[PP of ] ; PP – Prepositional Phrase
[NP water ]; NP– Noun Phrase .
```

A *Semantic Parser* analyzes the given text so that it tries to match the text with a formal meaning representation. This type of NLP parser requires extra work because in order to build a semantic parser the most used approach is to use human experts[4]. The reason for that is a sentence can have the same meaning but with a different syntax such example is :

"*Utah is next to Idaho.*" and "*Utah shares a border with Idaho.*"

Researchers try to free this manual assignment by using ML techniques such as *Supervised* and *Unsupervised.* A working example for semantic parser is the Stanford's SEMPRE Parser.

A *Shallow Semantic Parser* finds the predicate of the sentence and then it tries to match the predicate with different questions which give different complements. These questions are *Who, Where, Which, Where, Why, etc.* An example is :

"*Shaw Publishing offered Mr. Smith a reimbursement last March.*"

And by using the predicate all other components of the sentence are determined : Shaw Publishing is *Agent,* Mr. Smith is *Recepient* , a reimbursement is *Theme,* last March is *Time.*

A *Probabilistic Parser* is one of the most promising and widely used parser in the NLP research area. What it does is to simply use its a priori knowledge, which is also called trained data, and it tries to estimate the outcome of a given sentence as accurate as it has in its knowledge. The most popular one is the Stanford NLP group's probabilistic parser called Stanford Parser.

A *Full Parser* is a kind of parser which besides the morphological description of words in the sentence, it gives the dependency relation between different token components of a sentence. The concept of Full parser can be compared with a Shallow Parser more easily than the other kinds of parser in this list. An example for the Full Parser is :

"Me and Marc went to the sea side."

And the dependency relation between words can be given as (from Stanford Parser [5]):

```
nsubj(went-4, Me-1)
conj_and(Me-1, Marc-3)
nsubj(went-4, Marc-3)
root(ROOT-0, went-4)
det(side-8, the-6)
nn(side-8, sea-7)
prep_to(went-4, side-8)
```

2.1.2 LinguA Parser

LinguA is developed by Italian Natural Language Processing laboratory. LinguA combines machine learning techniques and rule-based systems and offers following services :

- Sentence splitting
- Tokenization
- Part-of-speech tagging
- lemmatization Dependency parsing

By giving these services it also supports a visualization tool which is available online. And it allows users to download analyzed sentences in *CoNLL format* which keeps lemma, coarse and fine grained Part-of-Speech, morphological features, and syntactic Dependency information about given sentences.

In addition to Italian parser, LinguA offers also English parser sevice.

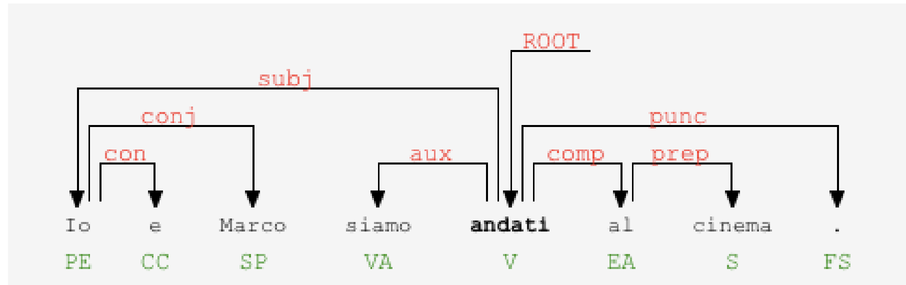

Figure 2.1.1: The LinguA Dependency Tree output for the sentence: "*Io e Marco siamo andati al mare.*"

2.1.3 TextPro Tool Parser

TextPro tool is developed by the Bruno Kessler Foundation. It is a suite of tools which are performing a number of NLP task. It provides the following services :

- Tokenization
- Morphological Analyzer
- Lemmatizer
- Chunker which is used for syntactical analysis.
- Tagger
- Dependency Parser
- A statistical based tool which recognizes temporal expressions in the text

It has an online demo version like LinguA and for researchers, they provide a free version of the TextPro tool.

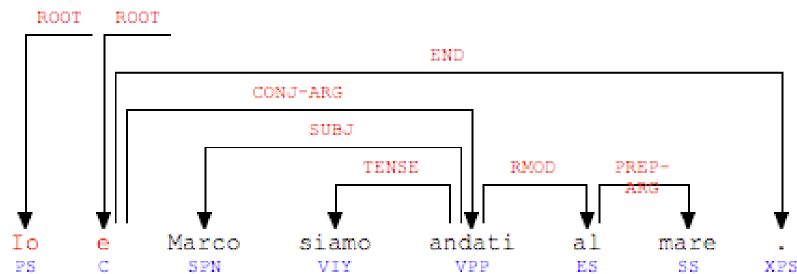

Figure 2.1.2: The TextPro Dependency Tree output for the sentence : "*Io e Marco siamo andati al mare.*"

2.1.4 TULE Parser & TUT TreeBank

The *TULE* parser is built by a group of researchers from Turin University. Main aim of this parser is to produce a dependency tree by using a tree bank, called TUT Bank. TULE parser is implemented in a way that it can handle four different languages: Italian, English, Catalan and French. It also supplies a graphical user interface which produces dependency tree in a tree structure. Each leaf on tree has information such as : *Part of Speech, relation between words, lemma, etc.*

By time of writing, the developers of TULE parser discontinued to support their product but they gave an opportunity to install the parser server to a local computer. So that it is still usable. [1]

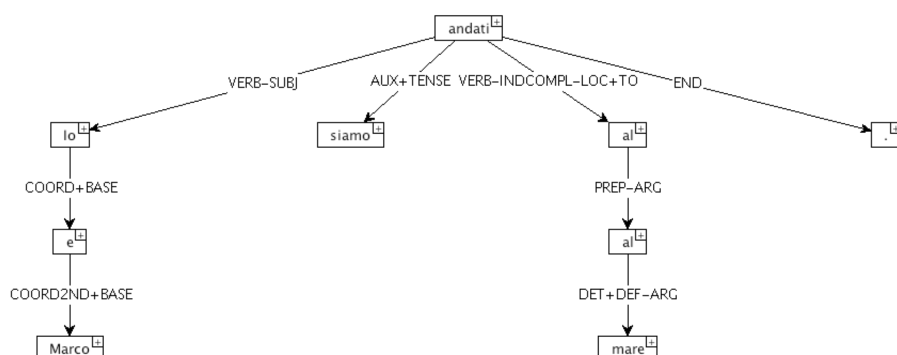

Figure 2.1.3: The TULE Parser's Dependency Tree for the sentence : "*Io e Marco siamo andati al mare.*"

On the other hand, *TUT* (Turin University TreeBank) is a treebank project [6]which contains 2860 italian sentences with 5 different categories. This treebank includes also 200 english sentences. All these sentences are annotated morphologically, semantically and syntactically.

As shown in figure 2.1.3, TULE parser uses TUT Treebank in order to obtain dependency trees. It is seen that every word in the sentence has a dependency relationship with one or the other. These relations between words are provided by using the TUT. TUT uses ARS (Augmented Relational Structure) annotations for relational dependencies of words in a sentence. By using ARS structure, every relation can include derived values from morpho-syntactic, functional-syntactic and syntactic-semantic.

Let's take the same example : *"Io e Marco siamo andati al mare."* The output for this example from TUT by using the TULE parser is the following :

[1]For more information on installing the TULE read users' manual in Appendix A.

```
1 Io (IO PRON PERS ALLVAL SING 1 LSUBJ) [5;VERB-SUBJ]
2 e (E CONJ COORD COORD) [1;COORD+BASE]
3 Marco (MARCO NOUN PROPER M SING ££NAME) [2;COORD2ND+BASE]
4 siamo (ESSERE VERB AUX IND PRES INTRANS 1 PL) [5;AUX+TENSE]
5 andati (ANDARE VERB MAIN PARTICIPLE PAST INTRANS PL M) [0;TOP-VERB]
6 al (A PREP MONO) [5;VERB-INDCOMPL-LOC+TO]
6.1 al (IL ART DEF M SING) [6;PREP-ARG]
7 mare (MARE NOUN COMMON M SING) [6.1;DET+DEF-ARG]
8 . (#\\. PUNCT) [5;END]
```

Figure 2.1.4: TUT file example

For the example given above has 4 different parts for each word in a sentence :

5 andati (ANDARE VERB MAIN PARTICIPLE PAST INTRANS PL M) [0;TOP-VERB]

1. The position of word in the given sentence, colored number with magenta in the example above.
2. The word's usage in sentence, colored word with cyan in the example.
3. The semantical characteristic of word, series of words colored with blue in the example.
4. The dependency number of that word which shows position of another word in the sentence and its relation role in the sentence. Both are colored with green in the given example above.

2.1.5 Motivation of Selected Technologies

Before going further to the next technological system for building a NLP tool it is required to describe main motivational criteria for selecting TULE and TUT.

Since our aim is to analyze a natural language, in this case *Italian*, it is necessary for a linguistic tool which can handle italian phrases accurately as much as it can. First of all, the bad news is that both in industry and in academia there are only a few options for languages other than English. And this really makes it hard to find a good dependency parser for Italian.

From the list of parsers that are given above the dependency output of TULE parser makes a clear difference among others that TULE can handle sentences most accurately and can give as much information about words in sentence as possible. And also considering easiness of use makes the TULE and TUT the selected linguistic framework for this thesis.

2.2 Rule Based Systems

The use of selected programming paradigm affects implementation phase very deeply so it is important to cogitate about an appropriate programming paradigm before writing the code. Therefor at the end of coding phase the selected programming paradigm could save developer time and code complexity by not giving any priority from the cleanness, shortness and functionality of the code. Considering this long but absolute statement, in order to find complementary parts of sentence in any natural language using an imperative language paradigm is *possible* but *not suitable*. Because an imperative (or procedural) language paradigm forces to know every logical detail[2] in order to control the flow of the program. So it can be said that the imperative programming paradigm focuses on *HOW* the computer should do.

On the other hand, from the perspective of a developer of such a tool, the focus should be on *WHAT* the computer is to do, not on *HOW*. By using a declarative language paradigm, this *WHAT* can be “declared” without touching every logical detail[7]. Thus giving the programmer the flexibility of thinking more as a human being rather than as a machine.

Since a declarative programming paradigm is needed, so a system which includes a declarative language is needed. Thus *Rule Based Systems* are introduced. “*A Rule Based System is a system that uses rules to derive conclusions from premises*.” [8] A Rule Based System can be used in all areas where an algorithmic solution does not exists clearly like Classification, Prediction, Diagnosis and Pattern Recognition

The elementary parts of a rule can be shown as follows :

1. **IF part** (LHS - Left Hand Side)
2. **THEN part** (RHS- Right Hand Side)

Example Rule: (*youDriving* and *yourCellPhoneRings*) => (*doNotAnswerPhone*)

In order to run a rule, data is indispensable in the system that is called *The Domain*. So given the rule above and a proper domain system concludes that you should not answer your phone while driving.

The main difference which separates a rule based system from imperative languages paradigm is that rule based systems can automatically infer to fire a rule wherever and whenever it sees a statement which matches with proper rule. But in imperative paradigm if execution of a particular statement is passed then there is no way to execute that particular piece of code during the lifecycle of that execution.

[2]This means tons of nested if-else statements

2.2.1 Architecture of a Rule Based System

The general architecture of a rule based system is composed of 3 different components that are *working memory*, *inference engine* and *knowledge base* (or rule base).

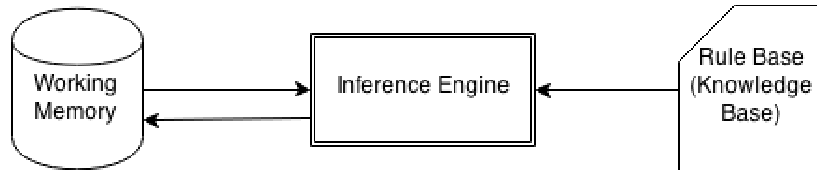

Figure 2.2.1: A general architecture of a rule based system

The working memory holds the data that you need in order to run other two components. Working memory is also called fact base. The knowledge base has all constrains that can trigger entire system. The inference engine itself is the trigger. The link between knowledge base and inference engine is directed towards to the inference engine. On the other hand, the link between the working memory and the inference engine is double linked. This means that inference engine is the one which can change the state of working memory.

From these 3 components the inference engine plays the central role for a Rule Based System. So let's take a closer look.

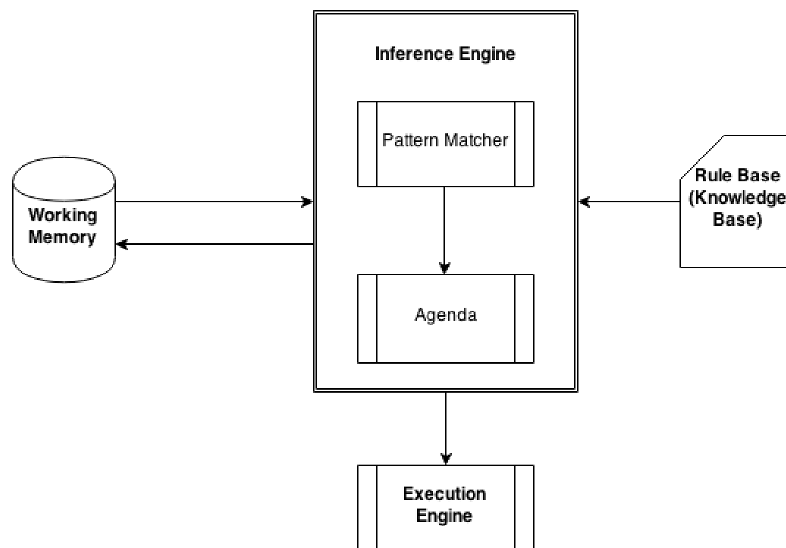

Figure 2.2.2: More detailed architecture of the rule based system

An inference engine composed of three additional parts that are : *Pattern Matcher*, *Agenda* and *Execution Engine*. Aim of inference engine is to match all defined rules in the knowledge base by comparing them to working memory and by using the pattern matcher as its subcomponent.

With more easy terms, inference engine takes list of facts from working memory and forms different possible combinations with them. It does it so fast by using a special algorithm called Rete[9] then it tries to apply them on the rule set. If rule pattern matches with fact combination then inference engine is triggered and possibly a new fact is born in the working memory. This hole process is repeated in order to see if there are more. This process can be thought as the triggering system in relational databases.

Since there are set of rules in a rule based system which are processed in parallel one rule might conflict with another one. To solve this problem a mechanism is necessary to save the overall system from crashes thus a mechanism called *Agenda* is created. An agenda holds the necessary information so that no rule can conflict with each other. In other words, we know that a rule based system is working in a huge while loop and rules are run in parallel. Think if a rule conflicts with another rule; meaning that they are triggering each other all the time. Then the program might not stop until it finds a fact which does not cause any more triggering. Agenda simply solves this problem by deciding a strategy for firing rules in order.

And finally, *the execution engine* is the part which executes THEN part of a rule (or RHS in terms of rule based systems).

2.2.2 Working Strategies of Rule Based Systems

Basically, there are 2 kinds of working strategy of a Rule based system that are :

1. Forward Chaining
2. Backward Chaining

A rule based system can include both like Jess, Drools or either of one like Prolog.

In Forward Chaining, the starting point for evaluating a rule is the IF part. It tries to match working memory with rule's IF part. So the strategy of Forward Chaining is to ask the question: "*Do I have it?*" to working memory. If it *has then* fires the consequence otherwise it doesn't. On the other hand, in Backward Chaining the starting point for evaluating a rule is THEN part. So the strategy for matching a rule in Backward Chaining is asking the question to Inference Engine "*To have this consequence, what should I have?*" This approach is also known as *Gaol Seeking*. These strategies are the main differences between rule engines.

2.2.3 Existed Technologies

There are many rule based systems as product. Most used ones are Prolog, JBoss Drools, CLIPS and Jess. Mainly they differentiate on how the engine executes rules as it is described in the previous section. Let's take a look at different rule based systems.

2.2.3.1 Prolog

Prolog is a general purpose logic programming language. It mostly uses declarative programming paradigm. And it is used mainly in natural language processing and other fields of Artificial Intelligence. Rule engine of Prolog uses backward chaining mechanism in order to evaluate its rules. Main importance which distinguishes Prolog from others is that Pure Prolog is Turing-complete[10].

2.2.3.2 Drools JBOSS

JBoss Drools is used mainly by enterprises as an expert system. It uses declarative programming paradigm. It supports JSR-94 Java standards which makes it 100% java compatible. For examining defined rules, it gives option to use both Forward and Backward Chaining approaches. It uses an enhanced version of Rete algorithm in order to make combinations of facts during the evaluation process.

JBoss has an enterprise project which includes several facilities for enterprises. And apart from the enterprise version, it has also a community project which provides free version. This free version has more features compared to the enterprise version but it is less steady. In time these features are transferred to enterprise edition with a steady state.

2.2.3.3 CLIPS

Clips is the most widely used expert system tool. It is written in C. It uses declarative programming paradigm and also it includes a complete object oriented language called COOL. It has an interface for Java but is not supporting JSR-94 Java standards completely. It has a similar syntax to Lisp programming language.

2.2.3.4 JESS

Jess is a rule engine and a variant of CLIPS. It supports JSR-94 Java standards hence it is 100% compatible with Java. It uses declarative programming paradigm with Lisp like syntax. It provides a dynamic programming environment but it can also be used with a different kind of XML which is created specifically for JESS that is called JessML.

Rule engine in Jess can be programmed to be used both for Forward Chaining and Backward Chaining approaches. It uses Rete Algorithm for evaluating defined rules. Apart from traditional rule engines Jess also provides an additional feature which allows user to write queries for retrieving the facts from working memory. So that a user can use it as a relational database.

Jess provides the facility to programmers while they write their code in Java with object oriented paradigm. A developer can call Java objects, data structures like arrayLists, hashMaps etc. inside it self. It also permits to be called from inside Java. Moreover, a user can define new rules or even functions of Jess inside Java. If Jess user explores that additional commands are needed then he/she can extend the main abilities of Jess by inserting new functionalities inside it. Jess can also be used while developing GUI applications which need some rules.

For programming environment Jess provides an Eclipse plug-in which has even a debugger inside. But there is a community of people who built other plugins for text editors like Emacs.

A *dummy rule example* and its file *dummy-rule.clp* is written in Jess as follows:

```
(defrule if-phone-rings-while-driving
        "a description of the rule is written in here"
        (driving (status TRUE)) ;this is a fact in the working memory
        (phone_rings (status TRUE)) ;this is also a fact in the working memory
   => ;if two are matched then printout
         (printout t "Do not answer!" crlf))
```

2.2.4 Motivation for Selected Technology

While considering selection of a proper rule based system five criteria are considered :

1. Compatibility with the programming language used in the thesis,
2. Rule engine strategy: Forward and Backward Chaining
3. Ease of learning phase,
4. Stability of the overall system,
5. The virtual weight on the program of the thesis[3]

Programming language for this thesis is Java [4]. So first criteria requires a rule based system which has libraries that are 100% compatible with Java. CLIPS is not a good

[3] in terms of space in storage.

[4] Its reasons are described in the next section

candidate for this. It is true that it has an interface for Java but it does not have the required specifications determined by Java which might cause problems during the implementation. Thus it makes CLIPS eliminated.

For the second criteria the approach of rule engines is considered. Both strategies are important in specific fields; sometimes it seems like Forward Chaining is advantageous but sometimes it is the other way around. So the selected rule engine should support both. Thus Prolog is eliminated.

The third and the forth criteria can be combined together for JBoss Drools and Jess. It can be said that they have similar syntax structure with minor differences. Both systems can work with Java objects. Stabilities of both systems are good. Even though community edition of Drools is less steady than the enterprise edition, it is still good for use as a rule based system. Jess and JBoss Drools can both work on java objects.

So in order to determine the selected rule engine a new criteria was necessary which creates the fifth criteria. The space occupied by Jess is much smaller than the space occupied by the JBoss. Libraries of Jess needs a space nearly to 1,1MB and Drools needs 98MB of space. So, Jess is selected as the default rule engine for this project.

2.3 Java

The main programming paradigm for thesis is *Object Oriented* besides the declarative paradigm of rule engine. And there are many programming languages in industry which support object oriented paradigm. But for the thesis it is chosen Java programming language due to the below facts:

- Java has a large API.
- The documentation support is really good.
- Performance compared to other similar languages such as Ruby, Python is much better.
- Portability of the code to other platforms and operating systems without recompiling the source code.
- Most of the development tools support mainly Java.
- The integrity with Jess; the rule engine used for thesis.

2.4 JCommander

There is no GUI for the program of this thesis. This is done intentionally and its justification is explained in detail in the implementation chapter. For this reason a small and efficient terminal library was in search of and was found out that *JCommander* is a very small library which provides interaction between user and program.

Chapter 3

Design

In order to develop a program which tries to find complementary parts of a natural language, it is essential to represent how sentences are formed in that particular language and how the logical analysis for that particular language is done. Then it is required to represent an appropriate approach which is suitable for that language. In this chapter, these 3 arguments are introduced to show the design phase of this thesis.

3.1 Sentence Structure in Italian

In Italian, phrases have a structure like the following :

SUBJECT - PREDICATE - COMPLEMENTS

And complements of a phrase is divided into 2 subcategories which are *Complementi Diretti* and *Complementi Indiretti.*

3.1.1 Complement Types

This type of complements form the basic structure in Italian sentences. Complementi Diretti are the ones which connect directly to the predicate of a sentence and they do not have any preposition[1]. There are 3 kinds of Complementi Diretti : *Complemento Oggetto, Complemento Predicaticvo del Soggetto* and *Complemento Predicativo dell'Oggetto.*

Complementi Indiretti are the ones which connect to the predicate indirectly and have prepositions.[2].

[1]Actually, this information is partially correct some of direct complements can have prepositions[11]. But simplification of text it is discarded.

[2]Note that Complementi Indiretti are not limited with those in the figure. There are many more of Complementi Indiretti

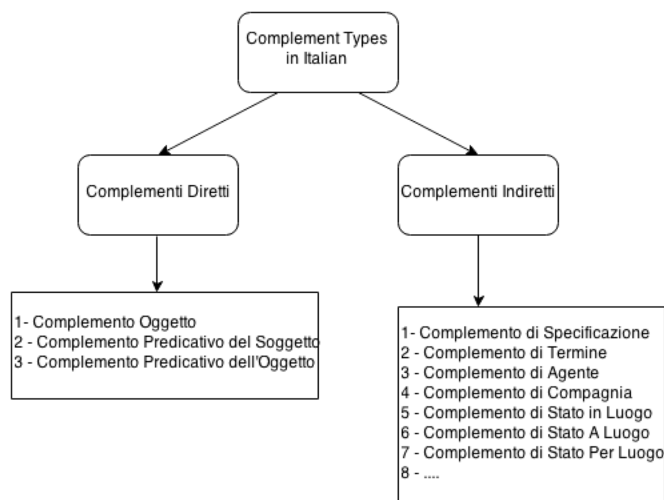

Figure 3.1.1: Complement categories and their elements in Italian

3.1.2 Verbs in Italian

Verbs in Italian are declared according to the subject which does the action, as in any language. But the declarations of verbs show difference from one language to another. In Italian there are 6 different declarations for each verb tense. In order to show this difference more clearly, it is better to compare it with a very known language which is English. When a verb is declared in English it can be declared in 2 different ways :

1. Verb's non-simple form : takes the suffix -s (for present tense verbs) and used by 3rd person singular.
2. Verb's simple form : does not take any suffix and is used by all other persons (I, You, We, They) except 3rd person singular.

On the other hand, in Italian verb structure is more complicated but this complication gives more expressive power in a sentence. Such that it is not required to express explicitly subject of the sentence because the verb itself implicitly refers to it hence subject is known a priori by the verb.

3.2 Logical Analysis

In the broadest sense, the word analysis means *"the process of separating something into its constituent elements."*[3] [12] And in linguistic, it means "*the use of separate,*

[3]In English Logical Analysis has more deep meaning but in this text Logical Analysis is considered as in the linguistics content.

short words and word order rather than inflection or agglutination to express grammatical structure."[12] In other words, in linguistics Logical Analysis[4] focuses on the individual elements in a sentence. And it asks questions to the predicate of the sentence for determining the syntactical function of each element ; like who, with who, where, etc.

3.3 Lexical Approach

The approach that is used in order to find complementary parts of a given sentence is so called *Lexical Approach*. In this approach, predicate of the sentence plays the role of an atom and all other words are arguments of that atom which try to attach it only if atom permits them. More formally, Lexical Approach says that the verb by itself includes very crucial and important information about number and type of arguments which can be bound to itself[13].

This theoretical method is the one which is needed in order to find complements precisely. Therefor there is the need of a subsystem which processes sentences and gives as much information as possible about the predicate and the words that are bound to it. This step can be completed by using a *dependency parser* but still there is a lack on how we could match complementary parts and information of words in a sentence. This gap can be filled with a rule based system.

3.4 System Design

Since our design aims to follow Lexical Approach which focuses on relation between verb and its arguments. We need to find a way to determine this dependency relation of words to predicate. As it is mentioned in technologies part, for the purpose of finding complements a parser which produces dependency trees is the only option so it is required to use it in our system. Now, we have necessary information which comes from dependency tree but this is not enough. We need to find out an autonomous way for determining which preposition forms a particular complement type and this can be done by defining rules hence a rule based system is an obligation.

Our general system design can be explained with a drawing more easily so, please look at figure 3.4.1. At first we have a text which is composed of Italian sentences and we give this text to a *blue toolbox* which is actually the program of this thesis. This toolbox generates the dependency trees of each sentences after that it sends these dependency relations to rule engine system which has predefined rules file for Italian grammar. These rules try to match themselves with information of dependency relations and once they find a match, they return it to toolbox. After all dependency

[4] Analisi Logica in Italian

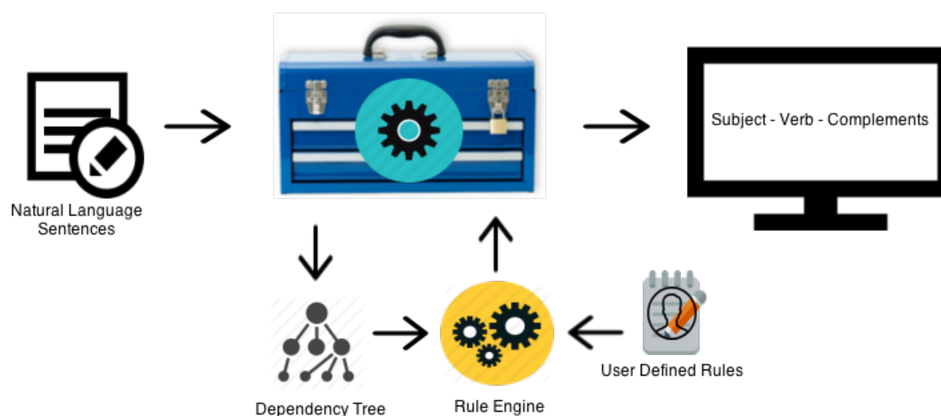

Figure 3.4.1: System Overview

relations are consumed by defined rules, *blue toolbox* writes found complements in a text file for showing them to user.

3.5 Design of Java Classes

The design procedure of Java classes that are interacting with rule engines plays essential role in the overall system. These classes are forced to be POJO objects by Jess. POJO is the acronym for Plain Old Java Object. Main characteristic of a POJO object class is for each property that is defined in that class has a setter and a getter method. It can be said that POJO paradigm is the most simplistic approach in terms of intelligibility for object oriented paradigm. We have 6 different POJO classes that are interacting with rule engine sets. These object classes are : *Word, R1Word, ourSubject, ourVerb, ourObject* and *ourNotKnownYet.* Let's observe the relation between them and the design of each object classes.

First of all, Word object type is the "mother" of R1Word object class and R1Word object is the "mother" of the rest. So it can be said that all object classes are actually Word objects; see figure 3.5.1.

A Word object contains information about a specific word in a sentence such as its position, lemma, part of speech(POS), relation with verb, etc. These information are taken from the dependency tree and converted from streams of characters to meaningful strings by creating Word object. When passing from one rule engine to another this Word object [5] adds itself new information so it needs to transform itself to another object type but of course this necessary information depends to the scope

[5]By Word object we mean all its children classes

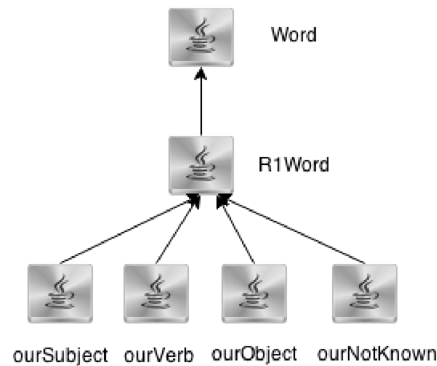

Figure 3.5.1: Word Object Relations

of the rule set. Please, check the figure 3.6.1. This figure shows each step of object transformation. Rule engine 1 and rule set 1 is designed to work with only Word Object and after leaving this engine, before entering rule engine 2 Word object is not anymore Word Object but R1Word object. The same happens in rule engine 3.

3.6 Design of Rule Files

The design of rule files are separated into 3 different rule categories. These rule sets (or categories) have different roles and are linked to each other consecutively. Each rule set possesses its own rule engine and each rule engine can work with only proper Word object types; as described in the previous section. As a brief description the aim of each rule set has its own scope :

1. Rule Set 1: This rule set breaks a taken sentence into number of main predicates by considering its complements it holds.
2. Rule Set 2: After first rule set finishes its mission second rule set starts to find subject, verb and object of sentence and leaves the indirect complements for the next phase.
3. Rule Set 3: This rule set recognizes indirect complements that are defined in its rule file.

Now, let's take a closer look to each rule file and their purposes.

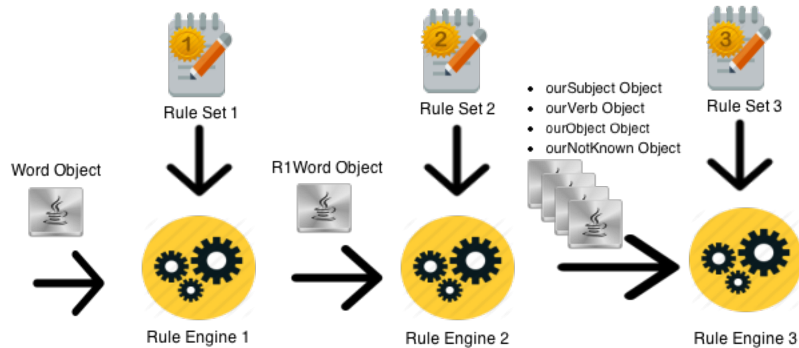

Figure 3.6.1: Rule Engines and their proper Objects

3.6.1 First Rule File

Our main criteria on logical analysis is *Lexical Approach* as described in section 3.3. By using definition of Lexical Approach, we can infer that a sentence is a bag of words in which there exists only one predicate but there are situations where a sentence might have more than one predicate by using one or more conjunctions. For example consider the sentence below :

"*Andiamo a fare la spesa e poi ti incontrerò con i miei amici.* "

A simple way to reduce this issue is to break sentence into 2 chunks when an additional predicate is encountered as main verb. Thus our first rule file is created and also aim of the first rule engine is determined : *Breaking Sentences.* The figure 3.6.2 shows a simple rule for checking if the sentence has 2 predicates. It checks if there exists a conjunction and a main verb which comes after that conjunction. If this rule finds these two then it will print out "*A sentence with 2 main verbs is found, calling for separation....*". This rule can handle only 2 predicates so a question might be "what if we have more?". Well, then we need to write another rule which checks if a sentence has 3 predicates and later may be another rule for 4 predicates and so on. This depends on the language usage but if it needs more separations then same pattern can be used. Also after writing each rule for separation, say for 3 predicates, we need to clear one of the conjunctions thus now there are 2, otherwise the rule engine is going to trigger itself again and again infinitely. So "cleaning rules" are also play an important role for this particular rule set.

```
(defrule exists-a-verb-after-conjunction
    "this rule triggers the division rule for a sentence wiht 2 main verbs "
    (exists (and (Word {POS == "CONJ"}
                       (sourceNumber ?sNumber))
                 (Word {POS == "VERB"}
                       (sourceNumber ?sVerb &:(> ?sVerb ?sNumber)))))

    =>

    (printout t "A sentence with 2 main verbs is found, calling for separation....." crlf))
```

Figure 3.6.2: A simple rule for separating a sentence with two predicates

Another important information about this rule set is to distinguish each separated sentence. For this reason a *sentence number* is assigned to each word object of the sentence and each sentence's word is put in a Java object called R1Word for the next rule engine.

3.6.2 Second Rule File

After separating a sentence, depending on the number of main verbs it holds, now our second step is to start working on words in sentences. First it necessary to find the predicate of the sentence, then the subject of it. Also we know that there are 2 kinds of complements : Diretti and Indiretti. Complementi diretti are ones which are easy to find compared to indiretti because parser gives proper information of a word which is subject, verb and object more clearly. So the second step is to discover which word(s) forms the subject, predicate and object of the sentence. What we do in by creating this stage is we are inserting an intermediate stage between complementi diretti and indiretti thus we alleviate commotion of finding everting at one time. If we would use the approach of finding everything at once then readability of the code would reduce.

Let's take a look at the rule in figure 3.6.3 . This rule will be triggered when it encounters with a word which has "AUX+TENSE" as its relation and has a complement type "not-known-yet" and once it is fired it will assign "AUX-TENSE-VERB" to its complement type which was previously unknown. So this rule, like all defined rules, checks all the words in a sentence in order to match itself with it. An example which triggers this rule might be :

"Ho comprato una bambola per mia figlia."

If our described rule above and depicted below was working on a rule engine and if above sentence would be given as an input, it would start to check every word of that sentence and when it finds a proper word that matches with itself then it would assign to that word's complement type AUX-TENSE-VERB (or auxiliary verb). For this example it is word "Ho".

```
(defrule auxiliary-tense-verb
  "finds the auxiliary part of verb in an ACTIVE sentence.
   Example: HO COMPRATO UNA BAMBOLINA PER MIA FIGLIA. This rule will find the
   auxiliary part(s) of the top verb."
  ?active-verb <- (R1Word {relation == "AUX+TENSE"}
                  (complementType ?cACTIVE &:(eq ?cACTIVE "not-known-yet")))
  =>
  (bind ?active-verb.complementType AUX-TENSE-VERB))
```

Figure 3.6.3: Example rule for finding auxiliary verb

At the end of this stage each found verb, subject and object are put inside a proper java object: *ourVerb*, *ourSubject*, *ourObject* and those which left uncategorized are put in objects called *ourNotKnown*. These four Java classes are just children classes of *R1Word* class meaning that they do not have any newly defined property in their Java classes.

3.6.3 Third Rule File

The first rule set checks if there are more than one main verb exists in a sentence and when such condition holds it separates them. The second rule set identifies subject, verb and object of sentences and if there exists a word which is not compatible with those 3 types it is left as not know. So, our only mission in this rule set is to find this 4th category of words that are actually words forming complementi indiretti. This stage takes *ourVerb, ourObject, ourSubject* and *ourNotKnown* objects from the previous rule set. It might seem odd the necessity of ourVerb object type in this rule set but by taking ourVerb objects we actually simplify the control flow of the program. Let's observe the example below by using it with the rule depicted in figure 3.6.5.

"Il gatto di Marco è molto intelligente."

Let's assume that we executed all the previous rule sets and in the previous rule set it is found out that "Il gatto di Marco" is the subject of the sentence. But there is a complemento di specificazione in this subject which is "di Marco". The rule below is defined in a way that it only searches ourSubject objects and for this case ourSubject objects are composed of : Il, gatto, di, Marco. When the rule sees word's lemma as "di", the relation of the word as "prep-rmod"- a special string which came from parser - and preposition as the word's part of speech then it tries to combine this particular element with another subject element(or object) which comes after that "di" and if it succeeds it fires the rule and assigns COMPLEMENTO-DI-SPECIFICAZIONE as this word's complement type.

The rule depicted below finished its purpose by assigning the complement type but as you can see from the rule code snippet there are 3 binding operations. We already used one of them for that particular word which comes after "di". What about the

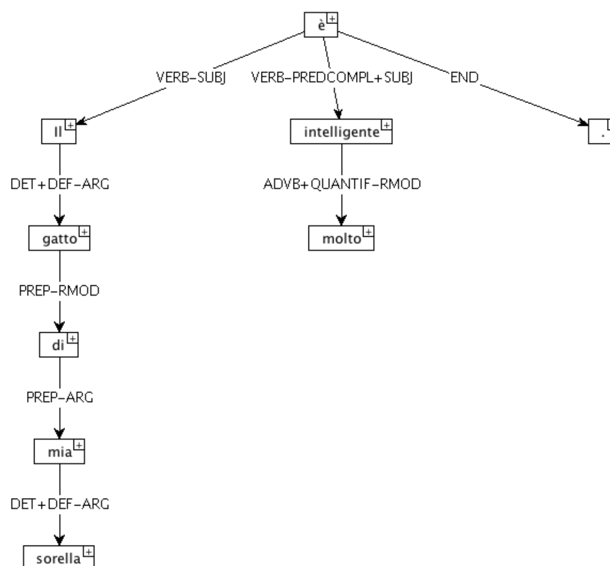

Figure 3.6.4: Dependency tree for “Il gatto di mia soralla”

other 2? The first binding changes the source number of the preposition “di”. This is necessary because we are taking information from a parse tree which gives word relations in a tree form meaning that they are linked to each other and if you want to reach a particular element you need to pass each element that are in between. More clearly, if we had a subject like :

“Il gatto di mia sorella “

If we want to reach “sorella” we need to first pass from word “mia” because it has the necessary information in order to reach it. This can be shown easily by using the figure 3.6.4. This “traversing” information is provided by source number and destination number data in the TUT file; see figure 2.1.4. So next time when this rule tries to match itself again for “di” and its dependents, this binding operation will serve it to take the “position” of the “mia” and give it to “di” hence it can see word “sorella”. Note that, we are not erasing word “mia”, we are just assigning its position to “di”.

The third binding operation is assigning COMPLEMENTO-DI-SPECIFICAZIONE to “di” itself.

```
(defrule complemento-di-specificazione-for-Subject
  "This rule finds the complemento-di-specificazione for
  Subject.
  EXAMPLE : Il gatto di Marco è molto intelligente."

  ?word-di <- (ourSubject {lemma == "DI"}
                          {relation == "PREP-RMOD"}
                          {POS == "PREP"}
                          (sourceNumber ?sDI))

  ?dependent-word <- (ourSubject (destinationNumber ?dWord &:(eq ?dWord ?sDI)))

  =>

  (bind ?word-di.sourceNumber ?dependent-word.sourceNumber)
  (bind ?dependent-word.complementType COMPLEMENTO-DI-SPECIFICAZIONE)
  (bind ?word-di.complementType COMPLEMENTO-DI-SPECIFICAZIONE))
```

Figure 3.6.5: Example rule for finding complemento di specificazione for Subject

Another importing point for this rule, and generally for this rule file set, is if a complement type is matched it assigns same complement category to preposition itself. Let's take again same example :

"Il gatto di mia sorella"

After the above rule matches for assigning the proper complement type to "mia" it assigns also to "di". Next time when it matches for assigning complement type for "sorella" it again assigns to "di". This is a design choice too. There were 2 possibilities for writing complement rules. One is described and the other one is : it should be written another rule which assigns the proper complement type for each preposition. This second "idea" is not bright. First it messes the code and the second thing it slows down the execution process of the code.

3.6.4 Summary of Rule Files

For the purpose of describing aims of each rule files this subsection shows what happens when an example sentence is taken as an input. Our sentence for this subsection is :

"Il gatto di Marco è scappato."

First of all, when a sentence is taken all of words that are inside it are converted to Word object then sent to first engine; see figure 3.6.6 for the real view of Word objects in rule engine 1. The aim of this engine is to check if there is another main verb in the sentence. For the example above there is only one main verb so it will allow to pass all word objects to the next engine without touching any of them. From passing from rule engine 1 to rule engine 2, word objects are transformed to R1Word objects. This transformation step is essential for determining the *sentence number* of each word when there is another main verb in the sentence.

```
this is rule set 1
f-0   (MAIN::initial-fact)
f-1   (MAIN::MaxSent (maxSentence 0))
f-2   (MAIN::Word (POS "PRON") (class <Java-Object:java.lang.Class>) (destinationNumber 2.0) (destinationSet T
RUE) (destinationWord " vado  ") (lemma "IO") (relation "VERB-SUBJ") (sentenceNumber 0) (sourceNumber 1.0) (so
urceWord " Io  ") (types "PERS ALLVAL SING 1 LSUBJ") (OBJECT <Java-Object:com.myRuler.transformer.Word>))
f-3   (MAIN::Word (POS "VERB") (class <Java-Object:java.lang.Class>) (destinationNumber 0.0) (destinationSet T
RUE) (destinationWord " vado  ") (lemma "ANDARE") (relation "TOP-VERB") (sentenceNumber 0) (sourceNumber 2.0)
(sourceWord " vado  ") (types "MAIN IND PRES INTRANS 1 SING") (OBJECT <Java-Object:com.myRuler.transformer.Wor
d>))
f-4   (MAIN::Word (POS "PREP") (class <Java-Object:java.lang.Class>) (destinationNumber 2.0) (destinationSet T
RUE) (destinationWord " vado  ") (lemma "A") (relation "VERB-INDCOMPL-LOC+TO") (sentenceNumber 0) (sourceNumbe
r 3.0) (sourceWord " al  ") (types "MONO") (OBJECT <Java-Object:com.myRuler.transformer.Word>))
f-5   (MAIN::Word (POS "ART") (class <Java-Object:java.lang.Class>) (destinationNumber 3.0) (destinationSet TR
UE) (destinationWord " al  ") (lemma "IL") (relation "PREP-ARG") (sentenceNumber 0) (sourceNumber 3.1) (source
Word " al  ") (types "DEF M SING") (OBJECT <Java-Object:com.myRuler.transformer.Word>))
f-6   (MAIN::Word (POS "NOUN") (class <Java-Object:java.lang.Class>) (destinationNumber 3.1) (destinationSet T
RUE) (destinationWord " al  ") (lemma "CINEMA") (relation "DET+DEF-ARG") (sentenceNumber 0) (sourceNumber 4.0)
 (sourceWord " cinema  ") (types "COMMON M ALLVAL") (OBJECT <Java-Object:com.myRuler.transformer.Word>))
f-7   (MAIN::Word (POS "PREP") (class <Java-Object:java.lang.Class>) (destinationNumber 2.0) (destinationSet T
RUE) (destinationWord " vado  ") (lemma "CON") (relation "RMOD") (sentenceNumber 0) (sourceNumber 5.0) (source
Word " con  ") (types "MONO") (OBJECT <Java-Object:com.myRuler.transformer.Word>))
f-8   (MAIN::Word (POS "NOUN") (class <Java-Object:java.lang.Class>) (destinationNumber 5.0) (destinationSet T
RUE) (destinationWord " con  ") (lemma "MARCO") (relation "PREP-ARG") (sentenceNumber 0) (sourceNumber 6.0) (s
ourceWord " Marco  ") (types "PROPER M SING ££NAME") (OBJECT <Java-Object:com.myRuler.transformer.Word>))
f-9   (MAIN::Word (POS "PUNCT") (class <Java-Object:java.lang.Class>) (destinationNumber 2.0) (destinationSet
TRUE) (destinationWord " vado  ") (lemma "#\\\\.") (relation "END") (sentenceNumber 0) (sourceNumber 7.0) (sou
rceWord " .  ") (types "null") (OBJECT <Java-Object:com.myRuler.transformer.Word>))
For a total of 10 facts in module MAIN.
```

Figure 3.6.6: Objects in Rule Engine 1

Now, rule engine 2 takes all R1Word objects, finds the subject, verb and object of the sentence; see figure 3.6.7. For our example "Il gatto di Marco" is subject and "è scappato" forms the predicate of the sentence. After this phase finishes, like in the previous step, all objects are transformed. This time all R1Word objects are transmuted into 4 different object classes : ourSubject, ourVerb, ourObject and ourNotKnown. Each object class, as its name suggests, contains words that are found as subject, verb, object. All those words that left unknown and words groups that contain complementi indiretti are going to be processed by the next engine; rule engine 3. Here by saying word groups we mean a subject or an object which includes a complementi indiretti. For our example the subject "Il gatto di Marco" is inside this "words groups" class. Because "di Marco" forms Complemento di Specificazione which is complemento indiretto.

After word objects are transformed into 4 different categories, rule engine 3 takes all of them, see figure 3.6.8, and searches if any of them contains complementi indiretti. For our case it is "di Marco". After it consumes all words it sends type of found complements and all other information for each word in the sentence to Java for writing process to text file.

```
this is rule set 2
f-0   (MAIN::initial-fact)
f-2   (MAIN::R1Word (POS "ART") (class <Java-Object:java.lang.Class>) (complementType SUBJECT) (destinationNum
ber 6.0) (destinationSet TRUE) (destinationWord " scappato  ") (lemma "IL") (relation "VERB-SUBJ") (sentenceNu
mber 0) (sourceNumber 4.0) (sourceWord " Il  ") (types "DEF M SING") (OBJECT <Java-Object:com.myRuler.rules.ru
leSet1.R1Word>))
f-3   (MAIN::R1Word (POS "NOUN") (class <Java-Object:java.lang.Class>) (complementType SUBJECT) (destinationNu
mber 1.0) (destinationSet TRUE) (destinationWord " Il  ") (lemma "GATTO") (relation "DET+DEF-ARG") (sentenceNu
mber 0) (sourceNumber -1) (sourceWord " gatto  ") (types "COMMON M SING") (OBJECT <Java-Object:com.myRuler.rul
es.ruleSet1.R1Word>))
f-4   (MAIN::R1Word (POS "PREP") (class <Java-Object:java.lang.Class>) (complementType SUBJECT) (destinationNu
mber 2.0) (destinationSet TRUE) (destinationWord " gatto  ") (lemma "DI") (relation "PREP-RMOD") (sentenceNumb
er 0) (sourceNumber -1) (sourceWord " di  ") (types "MONO") (OBJECT <Java-Object:com.myRuler.rules.ruleSet1.R1
Word>))
f-5   (MAIN::R1Word (POS "NOUN") (class <Java-Object:java.lang.Class>) (complementType SUBJECT) (destinationNu
mber 3.0) (destinationSet TRUE) (destinationWord " di  ") (lemma "MARCO") (relation "PREP-ARG") (sentenceNumbe
r 0) (sourceNumber -1) (sourceWord " Marco  ") (types "PROPER M SING ££NAME") (OBJECT <Java-Object:com.myRuler
.rules.ruleSet1.R1Word>))
f-6   (MAIN::R1Word (POS "VERB") (class <Java-Object:java.lang.Class>) (complementType AUX-TENSE-VERB) (destin
ationNumber 6.0) (destinationSet TRUE) (destinationWord " scappato  ") (lemma "ESSERE") (relation "AUX+TENSE")
 (sentenceNumber 0) (sourceNumber 5.0) (sourceWord " è  ") (types "AUX IND PRES INTRANS 3 SING") (OBJECT <Java
-Object:com.myRuler.rules.ruleSet1.R1Word>))
f-7   (MAIN::R1Word (POS "VERB") (class <Java-Object:java.lang.Class>) (complementType VERB) (destinationNumbe
r 0.0) (destinationSet TRUE) (destinationWord " scappato  ") (lemma "SCAPPARE") (relation "TOP-VERB") (sentenc
eNumber 0) (sourceNumber 6.0) (sourceWord " scappato  ") (types "MAIN PARTICIPLE PAST INTRANS SING M") (OBJECT
 <Java-Object:com.myRuler.rules.ruleSet1.R1Word>))
f-8   (MAIN::R1Word (POS "PUNCT") (class <Java-Object:java.lang.Class>) (complementType "not-known-yet") (dest
inationNumber 6.0) (destinationSet TRUE) (destinationWord " scappato  ") (lemma "#\\\\\\\\.") (relation "END")
 (sentenceNumber 0) (sourceNumber 7.0) (sourceWord " .  ") (types "null") (OBJECT <Java-Object:com.myRuler.rul
es.ruleSet1.R1Word>))
```

Figure 3.6.7: Objects in Rule Engine 2

```
This is rule set 3
f-0   (MAIN::initial-fact)
f-2   (MAIN::ourVerb (POS "VERB") (class <Java-Object:java.lang.Class>) (complementType "AUX-TENSE-VERB") (des
tinationNumber 6.0) (destinationSet TRUE) (destinationWord " scappato  ") (lemma "ESSERE") (relation "AUX+TENS
E") (sentenceNumber 0) (sourceNumber 5.0) (sourceWord " è  ") (types "AUX IND PRES INTRANS 3 SING") (OBJECT <J
ava-Object:com.myRuler.rules.ruleSet2.ourVerb>))
f-3   (MAIN::ourVerb (POS "VERB") (class <Java-Object:java.lang.Class>) (complementType "VERB") (destinationNu
mber 0.0) (destinationSet TRUE) (destinationWord " scappato  ") (lemma "SCAPPARE") (relation "TOP-VERB") (sent
enceNumber 0) (sourceNumber 6.0) (sourceWord " scappato  ") (types "MAIN PARTICIPLE PAST INTRANS SING M") (OBJ
ECT <Java-Object:com.myRuler.rules.ruleSet2.ourVerb>))
f-4   (MAIN::ourSubject (POS "ART") (class <Java-Object:java.lang.Class>) (complementType "SUBJECT") (destinat
ionNumber 6.0) (destinationSet TRUE) (destinationWord " scappato  ") (lemma "IL") (relation "VERB-SUBJ") (sent
enceNumber 0) (sourceNumber 1.0) (sourceWord " Il  ") (types "DEF M SING") (OBJECT <Java-Object:com.myRuler.ru
les.ruleSet2.ourSubject>))
f-5   (MAIN::ourSubject (POS "NOUN") (class <Java-Object:java.lang.Class>) (complementType "SUBJECT") (destina
tionNumber 1.0) (destinationSet TRUE) (destinationWord " Il  ") (lemma "GATTO") (relation "DET+DEF-ARG") (sent
enceNumber 0) (sourceNumber 2.0) (sourceWord " gatto  ") (types "COMMON M SING") (OBJECT <Java-Object:com.myRu
ler.rules.ruleSet2.ourSubject>))
f-6   (MAIN::ourSubject (POS "PREP") (class <Java-Object:java.lang.Class>) (complementType COMPLEMENTO-DI-SPEC
IFICAZIONE) (destinationNumber 2.0) (destinationSet TRUE) (destinationWord " gatto  ") (lemma "DI") (relation
"PREP-RMOD") (sentenceNumber 0) (sourceNumber 4.0) (sourceWord " di  ") (types "MONO") (OBJECT <Java-Object:co
m.myRuler.rules.ruleSet2.ourSubject>))
f-7   (MAIN::ourSubject (POS "NOUN") (class <Java-Object:java.lang.Class>) (complementType COMPLEMENTO-DI-SPEC
IFICAZIONE) (destinationNumber 3.0) (destinationSet TRUE) (destinationWord " di  ") (lemma "MARCO") (relation
"PREP-ARG") (sentenceNumber 0) (sourceNumber 4.0) (sourceWord " Marco  ") (types "PROPER M SING ££NAME") (OBJE
CT <Java-Object:com.myRuler.rules.ruleSet2.ourSubject>))
f-8   (MAIN::ourNotKnown (POS "PUNCT") (class <Java-Object:java.lang.Class>) (complementType "not-known-yet")
(destinationNumber 6.0) (destinationSet TRUE) (destinationWord " scappato  ") (lemma "#\\\\\\\\\\\\\\\\.") (re
lation "END") (sentenceNumber 0) (sourceNumber 7.0) (sourceWord " .  ") (types "null") (OBJECT <Java-Object:co
m.myRuler.rules.ruleSet2.ourNotKnown>))
For a total of 8 facts in module MAIN.
```

Figure 3.6.8: Objects in Rule Engine 3

3.6.5 Scripting

During thesis, one of main considerations about rule file sets was that if any manipulations on the engine side is needed then the user should not interact with the source code in order to make improvements on rules. And by using rule system language

as a scripting language provides user to manipulate behavior of the program without touching any source code. This is done by the Jess' compatibility with Java.

For instance if user wants to add new rules to the system then he/she only needs to open proper .clp[6]script file and write the necessary rule. This is applicable for all rule files.

3.6.6 Problem of Disordered Words and its Solution

Jess takes POJO objects and puts them in its working memory by assigning IDs. It tries to match these working memory things, called facts, with user defined rules and it does not care about the order. Although on one hand, this makes it perform operations much faster, on the other hand this condition forces Jess to give results unordered. As expected from a natural language tool, given results should be in an order otherwise, the end user gets confused easily. So a simple and efficient solution for this issue is to store IDs and word details in a HashMap. Since, Jess gives developers to access Jess variables inside Java source code, 2 lists can be created, which contain ID and word details, then these two variables can be passed to Java and inside Java, these two lists can be put inside a HashMap. And lastly, when time comes for getting results in order this HashMap structure will be used.

3.6.7 Queries

Queries play an important role in the design of the thesis program. When each rule set consumes all words that are in the working memory of Jess, the last step is to send found results to Java for object conversion. This "sending service" is carried out by using query structure of Jess.

Look at figure 3.6.9. This set of query is designed for the second rule file. Remember that the second rule file tries to find subject, verb, object. And it leaves exploration of complementi indiretti to the third rule file. Basically, these queries search words in the working memory and when they find a proper one they save it inside a list[7] and since a Jess list can be reached by Java all R1Word objects which their complement types are equal to a proper complement type like VERB, MODAL-VERB or SUBJECT, etc. can be put inside a new word object type like ourVerb or ourSubject, etc.

[6]extension format of Jess files
[7]name of the list here is the name of the query. For instance return-verbs or return-subjects.

```
(defquery return-verbs
  "this query simply returns the verb(s) of the sentence."
  ?word-VERB <- (R1Word (complementType ?ctVERB &:(or (eq ?ctVERB VERB)
                                                       (eq ?ctVERB MODAL-VERB)
                                                       (eq ?ctVERB AUX-TENSE-VERB)
                                                       (eq ?ctVERB AUX-PASSIVE-VERB)))))

(defquery return-subjects
  "this query will return the subjects of the sentence."
  ?word-SUBJ <- (R1Word (complementType ?ctSUBJ &:(or (eq ?ctSUBJ SUBJECT)
                                                       (eq ?ctSUBJ IMPLICIT-SUBJECT)))))
```

Figure 3.6.9: Query example

Chapter 4

Implementation

4.1 System Architecture

In the design section, it is mentioned shortly about suitable approaches for building a system which finds complementary parts of a sentence. And as a consequence, it is found out that one subsystem is dependent to other such that they form predecessor successor relationship. Thus forces program to be in sequential order. For implementing this consecutiveness Batch-Sequential architecture pattern is selected. A general information about A Batch-Sequential architectural pattern can be given as follows:

- It is formed by stages,
- each of its stages are independent of each other and a stage is only dependent to its input data,
- each stage interacts with the next component by only sending(passing) its output as input,
- the relation between stages form a graph which is Directed Acyclic Graph.
- each stage consumes all of its data first and than passes that data to the next one. This provides modifiability of a stage without touching others.

A General overview of system architecture can be described as follow:

A text file which contains natural language sentences is taken as an input to the system. Then system sends this text file to Tule Parser Server which is remote and waits the return of parse tree. After parse tree (or dependency tree) is returned from the Tule Server it directly sends to other system which takes Rule Scripts' Folder besides parse tree. This "new" system first transforms the dependency tree to object form and by using the rule engines and rule scripts it publishes found complements on a text file; figure 4.1.1 depicts general architecture.

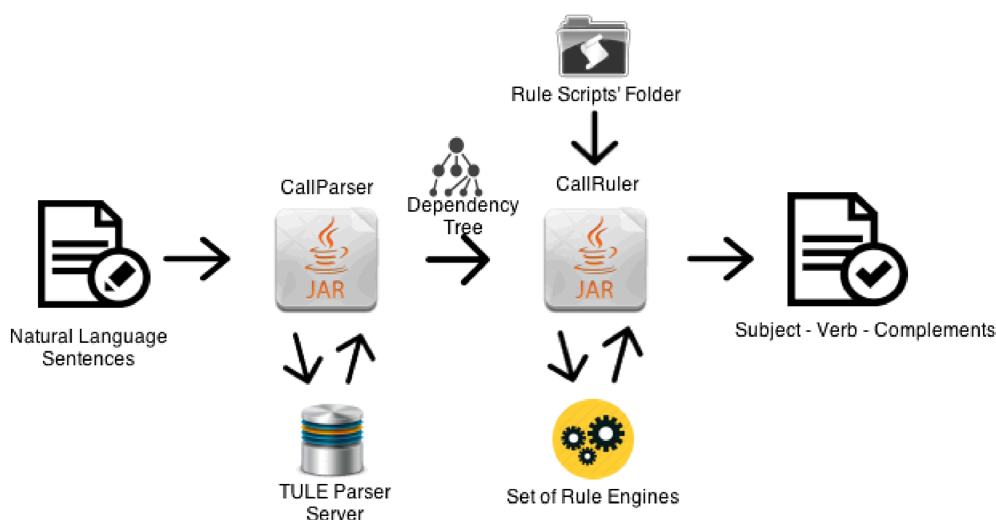

Figure 4.1.1: General System Architecture

4.2 Program vs. Tool Set

Until now, all of the work that has been done is shown as one single program. But, the most important difference which separates the implementation chapter from others is that in reality the whole system is composed by 2 distinct and consecutive programs. Thus form a tool set : *CallParser* & *CallRuler*.

1. *CallParser* calls TULE parser and gives the dependency tree as output.
2. *CallRuler* takes output of the CallParser and finds the relations between words then by using rule engines it gives complements of given sentences.

4.3 Splitting Tools

Using a dependency parser produces dependencies of each word in the given sentences but it is possible that parser does not give the appropriate trees, even though this situation exceeds the limit of this thesis, for future improvement of the tool set/thesis, this possibility can not be avoided. Otherwise, there will be situations with undesired results - and actually it happens with usage of specific prepositions; see Conclusion section. Hence, a flexible approach is needed which allows users to manipulate dependency tree, either by hand or by using an extra tool; figure 4.3.1. Also this is another reason why the architecture of the tool set is built with Batch & Sequential paradigm.

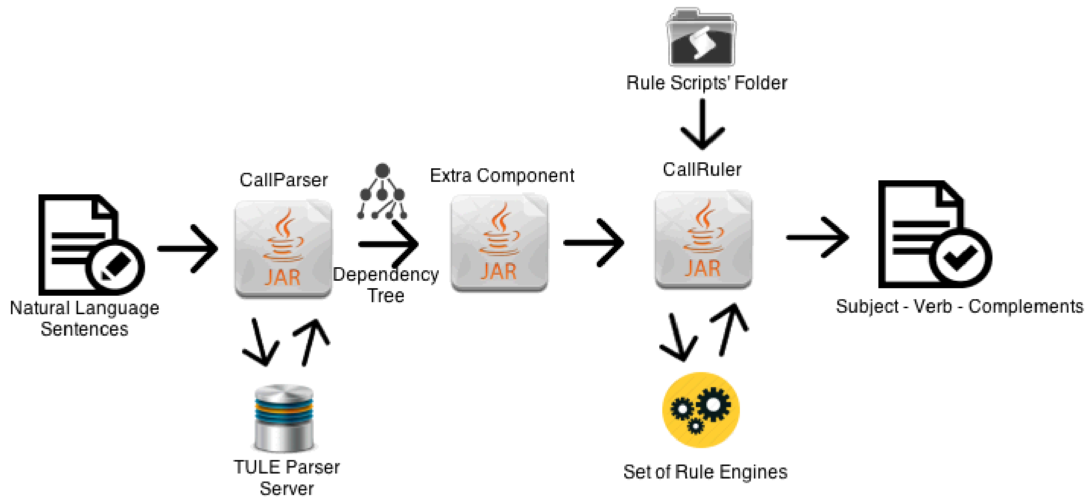

Figure 4.3.1: System architecture with an Extra Component

4.4 Details of CallParser

The *CallParser tool* is composed of 4 different packages that are : *Command Package*, *File Package*, *Parser Package* and *ReflectVariables Package*.

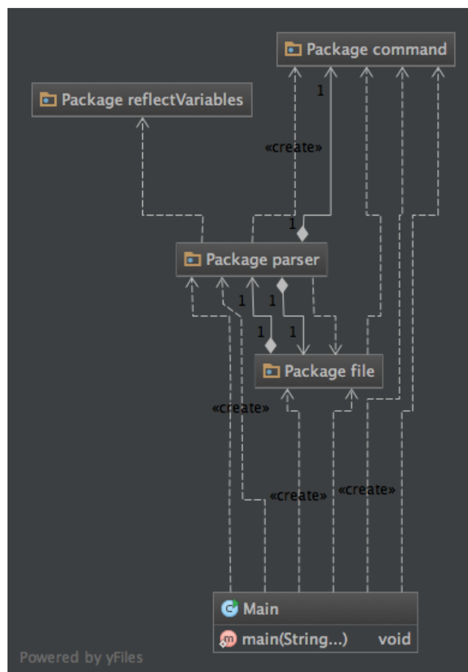

Figure 4.4.1: CallParser Packages

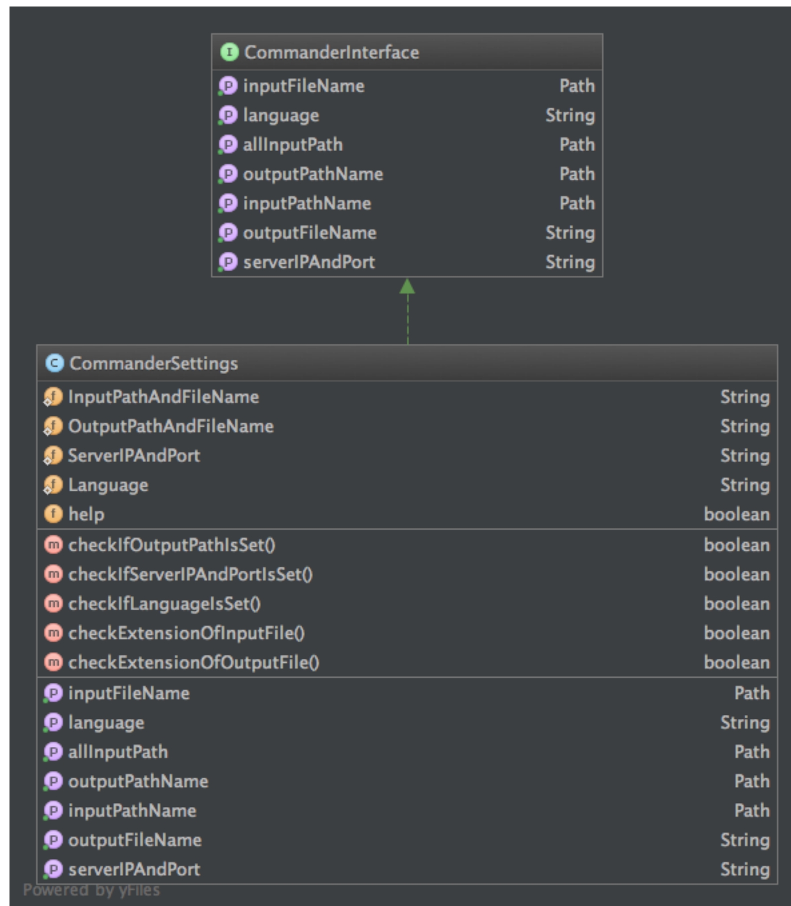

Figure 4.4.2: Command UML

1. Command Package : This package holds input parameters which are given by user and helps the system interacts according to these input.

2. File Package : This package helps to receive input file and create output file in the paths that user has determined by using the command line; see figure 4.4.3.

3. Parser Package : This package serves to connect to Tule Parser server locally and get dependency trees for the sentences It also helps for determining language of parser server and its IP address and port number which can be specified by command line; see figure 4.4.4.

4. ReflectVariables Package : This package helps to manipulate static variables of used .jar libraries. Hence it helps us to change the IP address, Port number of the server dynamically.

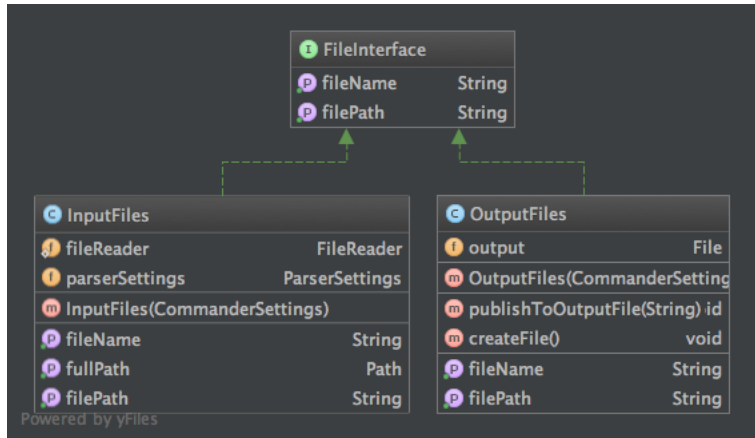

Figure 4.4.3: File Package UML

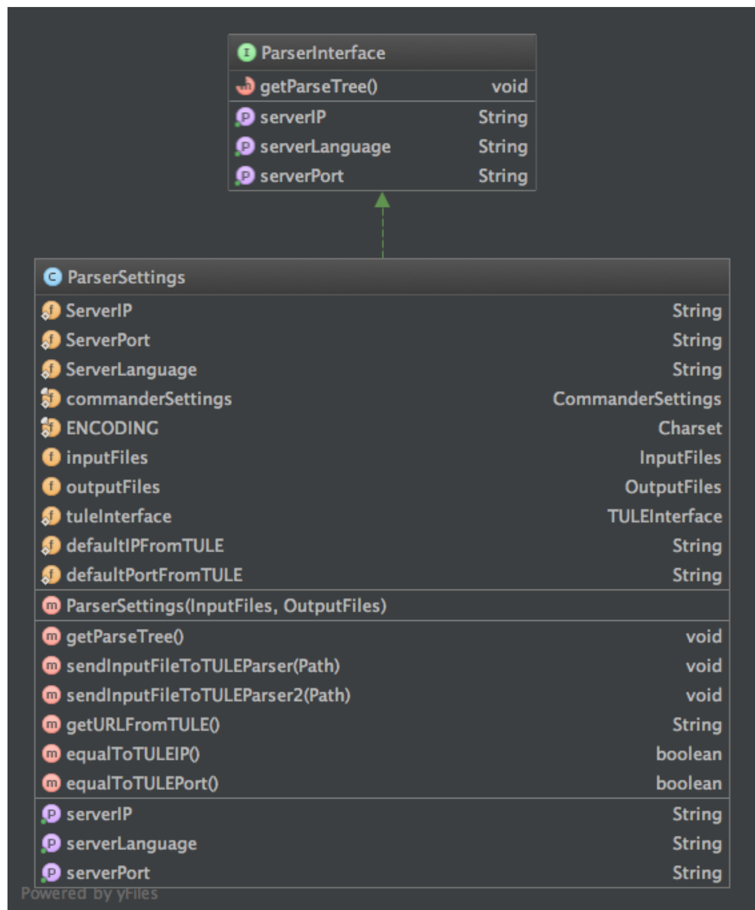

Figure 4.4.4: Parser Package UML

In next sections, description of CallRuler tool will be shared. For finding complementary parts of a sentence it might be seen that all important work is done by the CallRuler tool hence its implementation details are more important. Still this is partially correct, during the implementation phase of the CallParser tool some important techniques are used which might be useful and interesting for developers : *Decompilation*, *Reflection*.

4.4.1 Decompilation

Decompilation of Java source code is easy. And this feature can be used in order to integrate your code with someone else's code with using its code as a library. This property is followed during the implementation of CallParser. Since Tule Parser has already been built and written in Java. It can be used as a usual .jar library. Other than this, reading someone else code is the only thing that has to be done. During this phase a Java Decompiler tool is used which has plugins both for Eclipse and IntelliJ IDEA.

4.4.2 Reflection

Reflection is another used feature during the implementation of CallParser tool. Reflection serves to manipulate Java classes that are designed to be *"untouchable"* at run time. For our case variables like IP address of TULE server in the TULE's ".jar" file are statically determined and can not be neither changed nor touched. Even though due to performance reasons Reflection usage is discouraged, in situations like it is negligible. Thus now a user can enter what ever IP address he/she assigned to Tule Server.

```
public static void setInstanceValue(final Object classInstance,final String
    fieldName,final Object newValue) throws NoSuchFieldException,
    IllegalAccessException {
        final Field field = classInstance.getClass().getDeclaredField(
            fieldName);
        field.setAccessible(true);
        field.set(classInstance,newValue);
    }
```

The description of the above code can be given as :

First in order to use this method it is required to create an instance of that particular class. Hence we have an object of that class.This means that, that particular object contains the default values which are untouchables. What does this method do? It takes that created object, it takes the field name that has to be changed, and the value that we want to replace with the original field name. At the end it replaces the old value with our value.

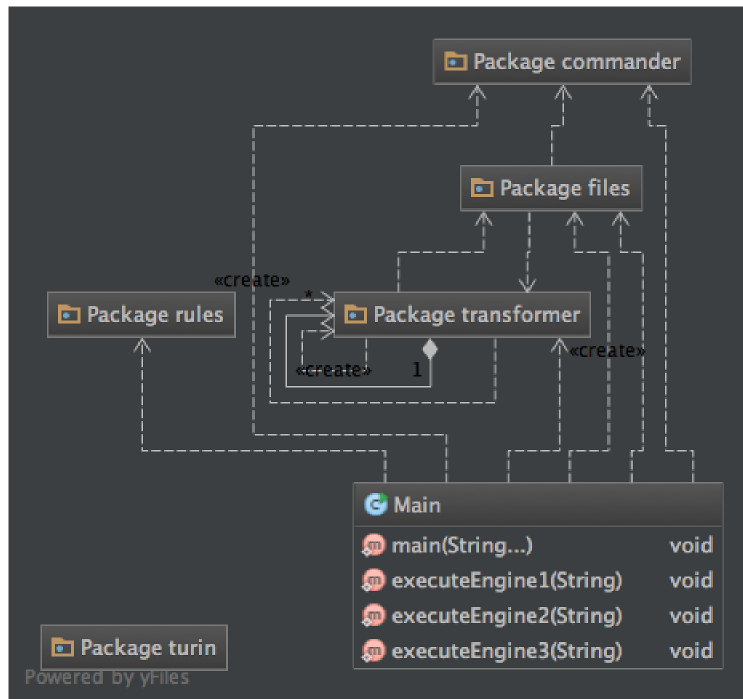

Figure 4.5.1: CallRuler Packages

4.5 Details of CallRuler

The CallRuler tool is composed of 5 packages that are : *Commander Package, Files Package, Transformer Package, Turin Package and Rules Package.*

1. Commander Package : This package, as in the CallParser tool, holds input parameters which are taken from user and helps the system interaction between user and the tool by sending these information to the appropriate packages; see figure 4.5.2 .

2. Files Packages: Different than the CallParser's File Package, *Files* Packages has 3 different classes that serve to give output for particular information: *Base-Facts*, *RelationsText* and *RuleText;* figure 4.5.3 shows classes of this package.

   - RelationsText class gives the output of dependency parser in TUT format in a path determined by the user.
   - BaseFacts class gives the output of dependency relations of words in a sentence. Output path is determined by the user.
   - RuleText class takes rule files from the user and sends them to appropriate rule engines.

CommandArgs	
InputPathAndFileName	String
OutputPathAndFileNameForBaseFacts	String
rulesPath	List<String>
help	boolean
checkExtensionOfInputFile()	boolean
checkIfOutputPathIsSet()	boolean
checkExtensionForBaseFactFile()	boolean
inputFileName	Path
allInputPath	Path
outputPathName	Path
ruleInputFileNames	List<String>
inputPathName	Path
rulePathName	String
outputFileName	String

Powered by yFiles

Figure 4.5.2: Commander Package

BaseFacts	
fileName	String
pathName	Path
output	File
BaseFacts(CommandArgs)	
createFile(String)	void
discoverWordRelations()	void
writeDiscoveredBaseFactsToAFile()	void
fileName	String
filePath	String

InputText	
pathName	Path
InputText(CommandArgs)	
fileName	String
fullPath	Path
pathName	String
file	File

RulesText	
fileName	List<String>
pathName	String
output	File
RulesText(CommandArgs)	
getFullPathOfTheRuleFileForEngine(int)	String
createFile(String)	void
writeFoundsOnAFile(String, ArrayList<String>	

RelationsText	
FilePath	String
output	File
RelationsText(String)	
getFilePath()	String
getFullPath()	String
createFile(String)	void
writeRelationsToFile(String)	void

Powered by yFiles

Figure 4.5.3: Files Package

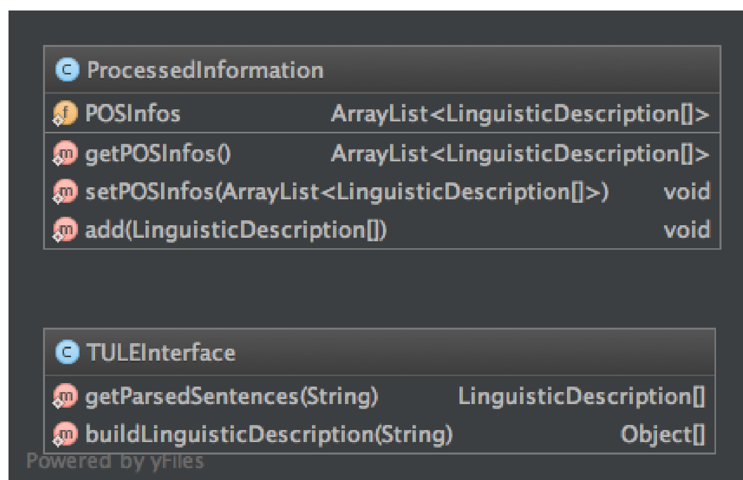

Figure 4.5.4: Turin Package

3. Turin Package : This package holds the data converted from dependency tree to TUT format. Transformer package will use these information for creating word objects.

4. Transformer Package: This package has necessary methods for converting strings of data, taken from TUT format, to java objects. It creates *Word* objects and put those word objects to an array structure. An important note about Word class is that it is a POJO[1] class. The characteristic of this object paradigm is for each its defined property in the class it has a setter and a getter method. By using this object paradigm we can interact with Jess. Another important feature of Word Class is that it is the *"mother"* of all other object classes that are interacting with Jess engine such as R1Word, ourSubject, ourVerb, ourObject and ourNotKnown classes; see figure 4.5.5.

5. Rules Package : This package serves to send Word objects and its children class objects to separate rule engines which are positioned sequentially. It also serves to send .clp files to rule engines for processing Word objects and its variants. At the end of each processing, it returns query results from rule engines; see figure 4.5.6.

[1] Plain Old Java Object

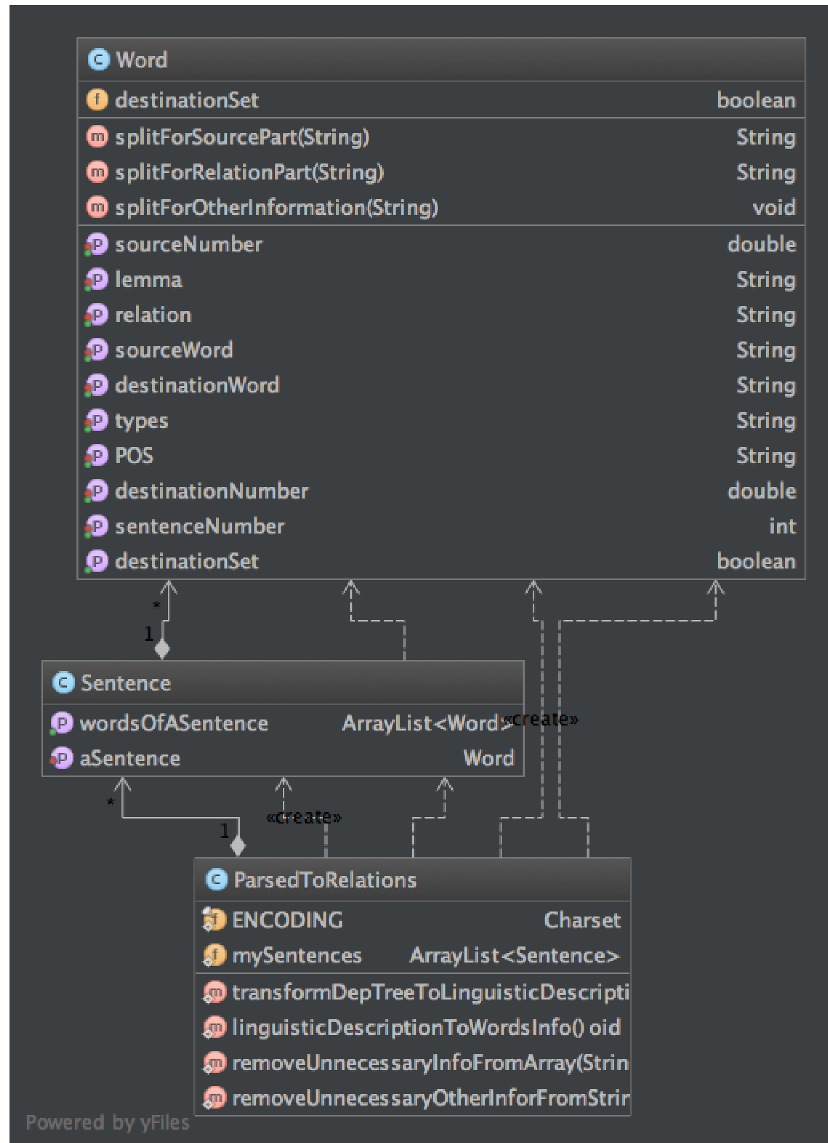

Figure 4.5.5: Transformer Package

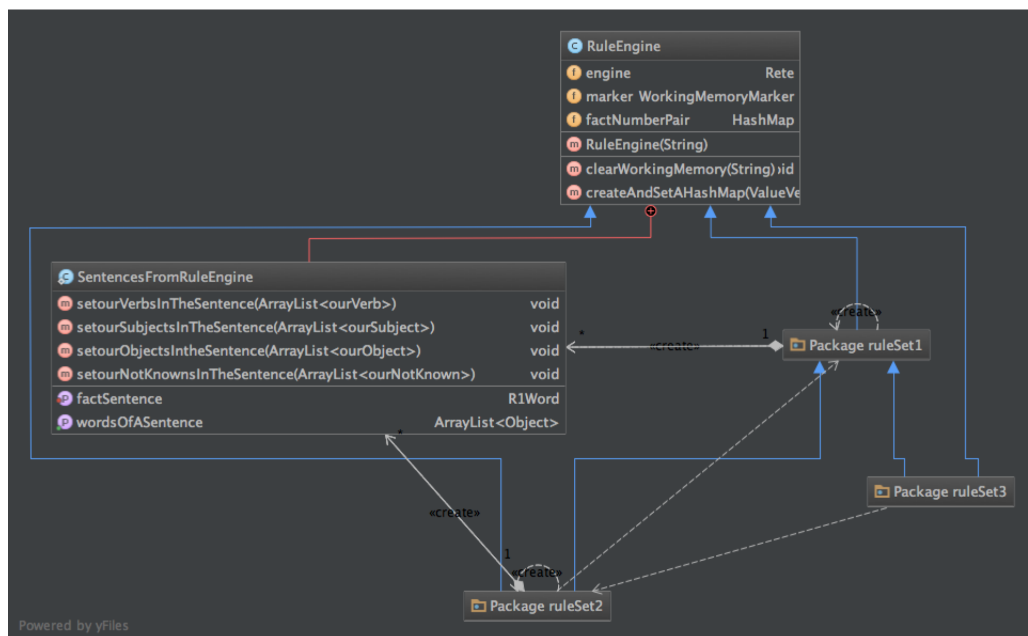

Figure 4.5.6: Rules Package

Chapter 5

Conclusion and Future Work

The aim of this thesis was finding the proper complementary parts of a given verbal frame in a natural language, in this case in Italian. In order to do so, two separate tools were built by using an Italian parser and a rule based system that works dependently to the results of the parser. The design and implementation sections describe the systems behavior in detail.

The results have shown that the data taken from the parser is sometimes enough but sometimes it is not. Also, this data is always dependent to the verb of the sentence. Hence, this outcome proves that the used approach and methods in this thesis are in fact correct. However the lack of the information between verbs and its possible complements might causes inaccurate consequences when given a sentence.

To avoid unwanted results there is the need of an auxiliary system. This auxiliary component should hold semantic information of verbs and their usages in sentences. So this way, the accuracy of the outcome will increase. Such systems have been developed previously for English as in VerbNet or FrameNet but there are none for Italian and very few for other languages. The development and implementation of a system like this requires time, effort and most importantly a community of engineers who can collaborate with experts in linguistics. Hopefully, this thesis will show others how important this problem is and will promote them to work for progression of such systems.

Bibliography

[1] O.L. Oliveira, AM. Monteiro, and N. Trevisan Roman. Can natural language be utilized in the learning of programming fundamentals? In *Frontiers in Education Conference, 2013 IEEE*, pages 1851–1856, Oct 2013. 5

[2] V. Punyakanok and D. Roth. The use of classifiers in sequential inference. In *NIPS*, pages 995–1001. MIT Press, 2001. 8

[3] Illinois shallow parser. URL: `http://cogcomp.cs.illinois.edu/demo/shallowparse/?id=7` [cited 18/09/2014]. 8

[4] Hoifung Poon and Pedro Domingos. Unsupervised semantic parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, pages 1–10. Association for Computational Linguistics, 2009. 8

[5] Stanford online parser. URL: `http://nlp.stanford.edu:8080/parser/` [cited 18/09/2014]. 9

[6] Cristina Bosco, Manuela Sanguinetti, and Leonardo Lesmo. The parallel-tut: a multilingual and multiformat treebank. In *LREC*, pages 1932–1938, 2012. 11

[7] Michael Lee Scott. *Programming language pragmatics*. Morgan Kaufmann, 2000. 13

[8] Ernest Friedman-Hill. *JESS in Action*. Manning Greenwich, CT, 2003. 13

[9] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982. 15

[10] Adrian A Hopgood. *Intelligent systems for engineers and scientists*. CRC press, 2011. 16

[11] Complementi diretti in detail [online]. URL: `http://www.zanichellibenvenuti.it/wordpress/?p=6162` [cited 18/09/2014]. 20

[12] Erin McKean. *The new oxford american dictionary*, volume 2. Oxford University Press New York, NY:, 2005. 21, 22

[13] A CERRINA. Elementi di sintassi strutturale, 2001. 22

Appendix A

User's Manual

As mentioned before there are two programs for this thesis : CallParser and CallRuler. They are 2 dependent tools. The usage of CallRuler is dependent to the output of CallParser. Both of these tools are designed to be work in any JRE installed environment. And both tools are in .jar files so that if these tools are needed by someone else in his/her application they can be included as libraries[1].

A.1 CallParser User Manual

First of all, in order to get the dependency tree from the TULE Parser you first need to install the TULE Parser Server to your local host, otherwise CallParser.jar file will send you connection errors.

After you downloaded the CallParser.jar file open a terminal window, go to the folder in which the CallParser.jar is located and then type one of the following:

1. **Help Option** : Gives you the all possible input parameters by their description.

   ```
   java -jar CallParser.jar -h
   ```

   or

   ```
   java -jar CallParser.jar --help
   ```

2. **Input a File** : This command is mandatory in order to give a text file to the parser.

   ```
   java -jar CallParser.jar -i /your_input_path/where_your_input_file_is.txt
   ```

[1] The code is well documented

This command line means call the CallParser.jar given the .txt file as input(-i). And by default it will generate the dependency tree in the same path(/your_input_path/) as an output file called *Output.txt*.

3. **Output Path** : This command is optional.

```
java -jar CallParser.jar -i /your_input_path/
    where_your_input_file_is.txt -o /your_output_path/
```

This command includes the first command and gives you the option to specify the output path. And the program will automatically generate the output text file as Output.txt by default.

4. **Another way for Output** : This command is optional.

```
java -jar CallParser.jar -i /your_input_path/
    where_your_input_file_is.txt -o /your_output_path/
    you_name_the_output.txt
```

This command line includes the first command andy the second command. And will generate the output file as you named it in your_output_path.

5. **Determining the IP and PORT number:** This command is optional.

```
java -jar CallParser.jar -i /your_input_path/
    where_your_input_file_is.txt -ip 192.168.1.109:5000
```

This command will allow you to insert an ip address and port number for connecting to the TULE Parser Server. By default ip and port number is set to : 192.168.1.103:5000.

6. **Determining the Language of the Server :**

```
java -jar CallParser.jar -i /your_input_path/
    where_your_input_file_is.txt -lang english
```

This command will allow you to specify the language of the server. By default the server language is set to **ITALIAN**. And if somehow you decide to insert English sentences into the parser then you need to open the dispatcher for that particular language. This particular step is described in the "For setting the Tule Parser" section below.

A.2 CallRuler User Manual

In order to run CallRuler Tool, you need to give a sentence to the CallParser tool and use the output of that tool as the input of CallRuler tool. As in the previous case, you need to download .jar file to your computer and also the *rule scripts* which you can find them as .zip files on the repository. After these steps you can open a terminal window, go to the folder where the CallRuler is located. This time you do not need to have a server connection. Just type one of the following:

1. **Help Option** : Gives you the all possible input parameters by their description.

   ```
   java -jar CallRuler.jar -h
   ```

   or

   ```
   java -jar CallRuler.jar --help
   ```

2. **Input a File** : This command is mandatory for giving the dependency tree to the rule engine.

   ```
   java -jar CallRuler.jar -i /your_input_path/
       where_your_input_dependency-tree_file_is.txt
   ```

   This command will take the input file from the path that you gave. Attention! You can only insert the dependency tree of the given text file to Parser. Otherwise, nothing will work. And as always the text file should end with .txt . If you try to execute this command by itself you should receive an **error message.**

3. **Insert Rule Files** :

   ```
   java -jar CallRuler.jar -i /your_input_path/
       where_your_input_dependency-tree_file_is.txt -R /
       your_input_path_of_Rules_FOLDER/ X_1.clp X_2.clp X_3.clp
   ```

   Takes the .clp scripts and runs it on the given text file. This option can only 3 .clp files as an input. And the first argument should be the path of the .clp files! Meaning it should be the folder and should end with '/' ! Also the order of the rule scripts is important! Because each of rule scripts has different scopes.

A.3 A Crucial Note About the Terminal Commands

Please note that the terminal commands might change in the future; in a positive way. So please always check the read me file or the repository for further improvements.

A.4 Setting Tule Server

Since Tule Parser is not downloadable from online, it is mandatory for me to describe the steps in order to set it up on your local or on a virtual machine.

First of all, in the CallParser repository in the downloads section there is a file called *TULEDISTR.zip* which is for installing the server. And there is another file which is called *viewerTULETUT.jar*. This file is for seeing the dependency tree in a graph from. This second file is not mandatory but it is a handy tool.

A.4.1 Steps for Installing TULE Server

1. Open the folder and click to Clisp folder.
2. Click to CLisp.exe. If your local host does not have common lisp interpreter version 2.35, you need to install this specific version. Otherwise, it will not work.
3. Then go to the Server Folder which is under the Root folder.
4. And first click to compile.bat then dispatcher.bat
5. Lastly, click the parser type which you want to take the dependency tree. (ex parserIt.bat)

The server should now be ready to receive inputs from the CallParser.