



UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

A parser for text to First-order Logic conversion

Relatore:

Prof. Puliafito Antonio

Tesi di Laurea di:

Petix Marco

Correlatore:

Prof. Longo Francesco

Dott. Longo Carmelo Fabio

Anno Accademico 2018/2019

So long and thanks for all the fish

Contents

1	Introduction	6
2	Background	8
2.1	Natural Language Processing	8
2.1.1	Rule-based NLP: from the '50s to the late '80s	8
2.1.2	Statistical NLP: from the '80s to NOW()	10
2.2	First-Order Logic	11
2.2.1	Davidsonian Semantics	13
2.3	Tools	14
2.3.1	SpaCy	14
2.3.2	Neuralcoref	18
2.3.3	MongoDB	20
3	Design	21
3.1	Software architecture	21
3.2	Structure of a FOL predicate	24
3.3	Overview of the parser workflow	28
3.4	Database collections	31
4	Implementation	33
4.1	The Dependency Switcher	33
4.2	Recognition of FLAT-type and ISA-type sentences	43
4.3	Co-reference Resolution	46
4.4	Database Interaction	50

5	Evaluation	52
5.1	Semantic Ambiguities	53
5.2	Possible Applications	56
6	Conclusions	57
A	Scripts from <code>dependency_switcher.py</code>	58
B	Scripts from <code>library.py</code>	69

List of Figures

2.1	Example of phrase structure rules	9
2.2	Interface of ELIZA	10
2.3	Feature of SpaCy	14
2.4	Tokenization	15
2.5	Analysis of SpaCy's pipeline	15
2.6	Results of the lemmatization and part-of-speech tagging processes . . .	16
2.7	The dependency tree of a sentence as displayed by DisplaCy Dependency Visualizer	17
2.8	DisplaCy Named Entity Visualizer	17
2.9	An example of coreference clusters	18
2.10	Highlighting of mentions	18
2.11	Neuralcoref scoring model	19
2.12	MongoDB Documents	20
3.1	Component Diagram	22
3.2	Class Diagram	24
3.3	Table containing linguistic annotations	28
3.4	Results of a conversion	29
3.5	Workflow of the parser	30
3.6	Structure of Sentences and Terms	31

List of Tables

2.1	Concepts of Natural Language	12
2.2	Basic elements of First-order logic	12
4.1	Summary of semantic features returned by SpaCy and their management by the parser - 1	37
4.2	Summary of the semantic dependencies returned by SpaCy and their management by the parser - 2	38

Chapter 1

Introduction

The thesis describes the development and the features of a parser dedicated to the conversion of sentences, expressed through natural language, in predicates of first-order logic. The contents are divided into six chapters, with this introduction being the first and acting as at the same time as a summary and a guide to the thesis itself.

The second chapter deals with the theoretical background related to the software, to its components, and to its possible applications. It provides a brief introduction to the history and characteristics of natural language processing and first-order logic, it describes the libraries and third-party components used through the development as well.

The third chapter deals with the design behind the software. It provides various insights about its architecture and the interaction between its components. It also provides a detailed description of its conversion process.

The fourth chapter deals with the implementation of the software itself. It describe how its main features were implemented and provides samples of the code belonging to its main components.

The fifth chapter deals with the evaluation of the software's performance. It provides a brief description of the testing process utilized during the development and highlights the pitfalls of interacting with natural language.

The sixth chapter serves as a conclusion and contains considerations on the possible

fields of application of the software, on its limits and possible improvements.

The thesis is also accompanied by two appendices containing scripts from important modules of the software. The first one deals with a dictionary of methods responsible of analysing each token composing the sentences to be converted. The second one deals with a set of methods that manage the interaction with the software's main classes.

Chapter 2

Background

2.1 Natural Language Processing

NLP is a branch of artificial intelligence that deals with analyzing, understanding and generating the languages that humans use naturally in order to interface with computers, in both written and spoken contexts, using natural human languages instead of computer languages[1].

Natural Language Processing is considered a complex field of computer science. Clarity or accuracy are properties hardly associated with natural languages, both in written and spoken form. Understanding natural languages is therefore an issue of understanding "not only words, but the concepts and the way they are linked together to create meaning"[2]. The birth of NLP is generally dated around 1950, the year in which Alan Turing published an article entitled "Computing Machinery and Intelligence" whose scope was a game, which we ended up referring to as "Turing Test", performed through a conversation between men and machines.

The development of natural language processing can be summarized in two significant phases.

2.1.1 Rule-based NLP: from the '50s to the late '80s

The first one, dedicated to a systematic approach, deeply inspired by the Chomskyan theories of linguistics and made of mostly hand-coded rule-based systems.

Noam Chomsky, still considered "the father of modern linguistics", published, in 1957, his "Syntactic Structures", a book that would shape the young field of the natural language processing for the thirty years to follow. His work dealt with transformational generative grammar, a theory aiming to create a new formal approach to syntax. The book provided a set of rules regarding the separation and combination of sentences and sub-sentences in order to create a finite set of sentence's structures.

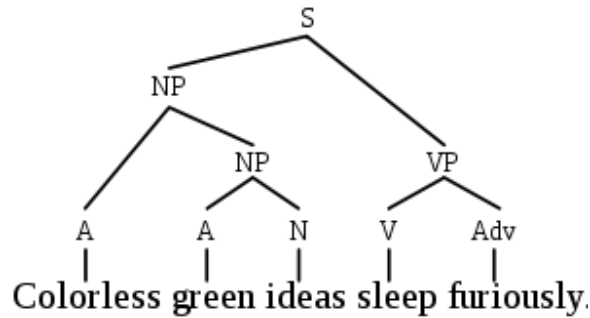


Figure 2.1: Example of phrase structure rules

This type of extremely formal rule systems dominated the field of natural language processing for a long time. The efforts of this first phase have been dedicated to the design and implementation of various projects such as:

- **automatic translation systems**, also called **machine translation systems**, whose development brought out effectively only after the advent of statistical means.
- **Natural Language Understanding programs**, focused of implementing **chatbot-like** interfaces based around a conversation between the user and the software, like **SHRDLU** and **ELIZA**. The first, dedicated to the manipulation of a virtual space and the objects contained within it while the second, aiming to give the illusion of dynamic understanding but without any means to contextualize events.
- **Conceptual ontologies**, whose objective, to structure information regarding the world that surrounds us in formats that can be understood by a computer system, is not unlike that of the software object of this thesis.

```
Welcome to

EEEEEE LL      IIII ZZZZZZZ AAAA
EE  LL      II      ZZ AA  AA
EEEEEE LL      II      ZZZ AAAAAAA
EE  LL      II      ZZ  AA  AA
EEEEEE LLLLLL IIII  ZZZZZZZ AA  AA

Eliza is a mock Rogerian psychotherapist.
The original program was described by Joseph Weizenbaum in 1966.
This implementation by Norbert Landsteiner 2005.

ELIZA: Is something troubling you ?
YOU:  Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU:  They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU:  Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU:  He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU:  It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:  ■
```

Figure 2.2: Interface of ELIZA

Towards the end of the 1980s, thanks also to the technological improvements related to storing capacity and computational power, an approach based on statistic models became widespread.

2.1.2 Statistical NLP: from the '80s to NOW()

The use of machine learning methods surely injected natural language processing with new life blood and potential. Feeding the models with large data sets led the software to naturally learn about which cases to turn their attention to and spared the developers the burden of generating specific rules for each possible sentence's structure. This, and the possibility of constantly improving the software's accuracy by providing more data while, in the meantime, increasing its robustness against unfamiliar inputs, have proved to be an invaluable advantage for natural language processing.

NLP's range of applications encompasses the vast panorama of analysis and manipulation of both syntax and semantics. These feature then get to be the bases for more complex tasks like speech recognition and human-like interaction. Right now it is rapidly scaling to lots of industries and thus achieving results in areas like healthcare, media, finance and human resources. Realities that are already part of our everyday life, such as being able to communicate, and be understood, by home automation assistants like Alexa and Google Home derive from the use of dozens of small fields of application of the natural language processing.

2.2 First-Order Logic

When a speaker points and says, “Look!” the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence “Look!” represents that fact.[3]

The versatility and expressiveness of natural languages has always proven them as the greatest asset for verbal communication. What makes them a bad candidate for representation and reasoning systems it’s their dependence from the context in which the sentences are spoken. A **formal system**, also called logic system, is an instrument to express statements through a systematic approach and to derive statements from a set of defined axioms[4]. Vector, matrix and tuple calculus are all considered formal systems, differing for the field of application and the degree of expressiveness.

Every formal system is composed of:

- A **finite alphabet** of symbols used to compose formulas;
- A **syntax** made by rules defining the structure of a proper formula;
- A **set of axioms** treated as primary rules of the system;
- A **set of inference rules** defining the proper approach to deduct new rules from a combination of the existing ones.[5]

First-Order Logic (FOL), also called First-Order Predicate Calculus, being composed by a set of such systems, and also being an extension of propositional logic, does qualify as a mean to represent knowledge. Just as propositional logic, FOL makes use of propositions to express facts about the world. The reason of its improved scope and expressiveness stays in his interaction with elements of the natural language’s syntax.

At the core of every phrase expressed through natural language are nouns, and noun phrases, that refer to objects and verbs, and verb phrases, that refer to relations among objects . Some of latter can be classified as functions, relations that associate a particular value with a given input. The following table provides examples for these fundamental elements of every natural language.

Objects	Entities such as people, things, places, ...
Relations (unary)	Property as red, mammal, first, tall, ...
Relations (n-ary)	Property as part of, has color, occurred after, owns, ...
Functions	sqrt, fatherOf, beginningOf, ...

Table 2.1: Concepts of Natural Language

Any given phrase can be expressed as a set of objects and properties or relations.
An example follows:

”Three rings (were crafted) for the elven-kings under the sky”

Objects: rings, kings, sky;

Relations: were crafted, for, under;

Properties: three, elven.

As a natural language, first-order logic is made up by both syntax and semantics.
The basic elements of its syntax, represented in the following table, can be used to craft atomic and complex sentences.

Constant	1, 2, A, John, Mumbai, cat, ...
Variables	x, y, z, a, b, ...
Predicates	Brother, Father, ...
Function	sqrt, LeftLegOf, ...
Connectives	\wedge , \vee , \neg , \Rightarrow , \Leftrightarrow
Equality	$=$
Quantifier	\forall , \exists

Table 2.2: Basic elements of First-order logic

First-order syntax implies the division between subject and predicate. Atomic sentences are therefore formed from a predicate symbol followed by a parenthesis with a sequence of terms representing the subject. They are represented as:

Predicate(term 1, term 2, ..., term n)

Phrases like "Charlie is a dog" could be expressed, through what is referred as "short-hand notation", by the formula:

$$\mathbf{Dog(Charlie)}$$

Complex sentences are instead formed by combining atomic sentences through connectives. A slightly more complex sentence like "Charlie is a white dog" would thus be represented by:

$$\mathbf{Dog(Charlie) \wedge White(Charlie)}$$

or even:

$$\mathbf{Dog(x) \wedge White(x) \text{ with } x = \mathbf{Charlie}}$$

2.2.1 Davidsonian Semantics

Even more complex sentences, with a particular attention to events, can be represented by means of Davidsonian semantics, which derives from the work of Donald Davidson. His theory focuses on the representation of the actions described by a sentence. He developed an event-specific variable, usually represented by the symbol \mathbf{e} .^[6]

By making use of said variable, the phrase "Brutus stabs Caesar" can be expressed through the predicate:

$$\mathbf{Stab(e, Caesar, Brutus)}$$

The real advantage of this structure, however, is the possibility of referring to an event, such as "stabs", from other predicates. The latter being representations of the optional modifiers influencing the event itself. A phenomenon highlighted by the conversion of the sentence as "Brutus stabs Caesar in the back with a knife":

$$\mathbf{Stab(e, Caesar, Brutus) \wedge in(e, back) \wedge with(e, knife)}$$

2.3 Tools

The following paragraphs give a description of the tools used during the development of the project subject of this thesis.

2.3.1 SpaCy

SpaCy is a open-source library for Natural Language Processing written in Python and Cython, the latter being a superset of the former with elements of C-inspired syntax.

Since its release in February 2015, SpaCy has proven to be a valid alternative to other NLP libraries like the Natural Language Toolkit[7]. While the toolkit is still widely used for research purpose SpaCy, being especially built for the production environment, maintains a firm grip over the commercial domain thanks to its high speed and accuracy[8].

The library provides pre-trained neural networks models as a mean to predict linguistic annotations. These models comes for a variety of languages and allow SpaCy to implement a set of NLP-related features.



Figure 2.3: Feature of SpaCy

Each model differs from the others for his size, speed, memory usage and accuracy. They all include **configuration options**, like the language and processing pipeline settings, but also a set of components powering up specific processes.

The **part-of-speech tagger**, **dependency parser** and **named entity recognizer**, whose functions are going to be explained shortly, rely on **binary weights** and **word vectors** in order to predict annotations from the context. The **lemmatizer**, responsible of the lemmatization process, depends on individual **data files** and **lexical entries**. They provide rules for said lemmatization and context-independent attributes for words.

When a text is sent to SpaCy in order to be processed it is firstly split into tokens contained in a Doc object. This iterative process is called **tokenization** and it's based on rules specific for each language.

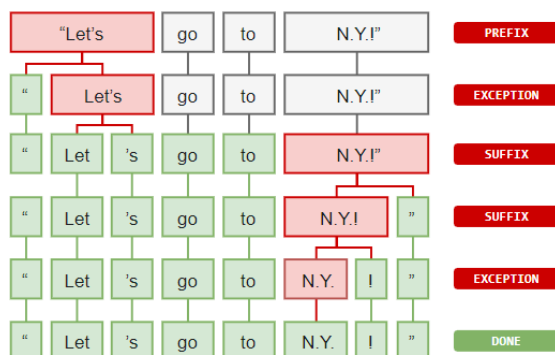


Figure 2.4: Tokenization

The object is then handled by several different, and modular, steps. The sequence of this processes is referred as **SpaCy's pipeline**.

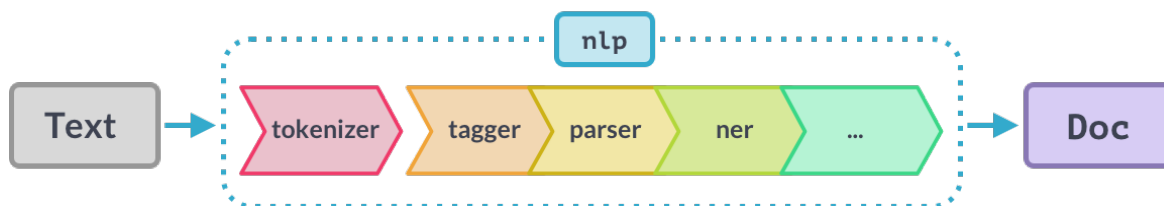


Figure 2.5: Analysis of SpaCy's pipeline

Every model comes with a default pipeline, most of them consists of a tagger, a parser and an entity recognizer but, as the following chapter demonstrates, extensions can be used to expand the pipeline's set of features.

After the tokenization SpaCy relies on the statistical model in order to predict which part-of-speech (POS) tag most likely applies to each token. This process is called

part-of-speech tagging and it's usually followed by the **lemmatization**. During the latter SpaCy does a rule-based deterministic mapping of each of the token's surface forms to a lemma independent from the context of the text in which the token reside.

An example of both the lemmatization and part-of-speech tagging processes are displayed in the following figure:

Text: A letter has been written, asking him to be released

Token	Lemma	Part-of-speech	Description
A	a	DT	determiner
letter	letter	NN	noun, singular or mass
has	have	VBZ	verb, 3rd person singular present
been	be	VBN	verb, past participle
written	write	VBN	verb, past participle
,	,	,	punctuation mark, comma
asking	ask	VBG	verb, gerund or present participle
him	-PRON-	PRP	pronoun, personal
to	to	TO	infinitival to
be	be	VB	verb, base form
released	release	VBN	verb, past participle

Figure 2.6: Results of the lemmatization and part-of-speech tagging processes

By parsing a text SpaCy can extract the structure of the parse tree linking all of the text's tokens. This process is called **dependency parsing** and it create a navigable tree whose arcs highlight the semantic dependency connecting pairs of tokens. The terms **child** and **head** are used to describe the tokens involved in a semantic dependency while the term **dep** describes the arc label and is used as an attribute containing the type of the dependency itself.

SpaCy provides a built-in visualizer, called **DisplaCy**, capable of displaying the results of both dependency parsing and part-of-speech tagging. Each semantic dependency is represented with an arrow, marked with the name of the dependency itself, starting from the token referred as "head" and pointing to the one referred as "child".

The following figure shows the results of the DisplaCy's elaboration of the sentence "You shall not pass!":

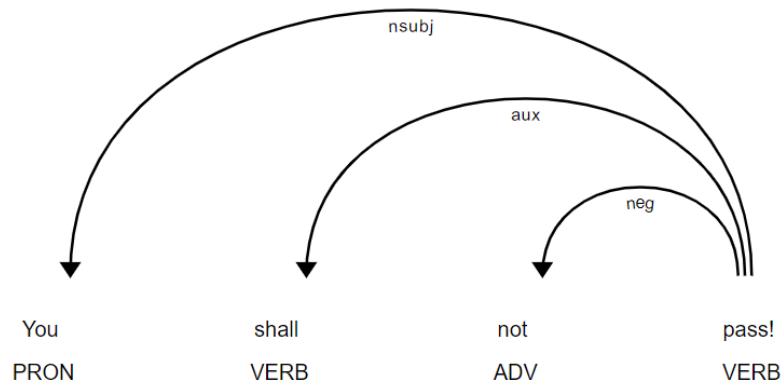


Figure 2.7: The dependency tree of a sentence as displayed by DisplaCy Dependency Visualizer

Most of SpaCy’s models possess a built-in **named entity recognition** system. The term **named entity** refers to an object from the real-world that is associated to a particular name. Entities come with various types, from persons and countries to products and companies, by interacting with a statistical model SpaCy is able to point out most of them. Given the models dependency from the data they were trained on the recognition could require some additional and specific training in order to achieve better performance[9].

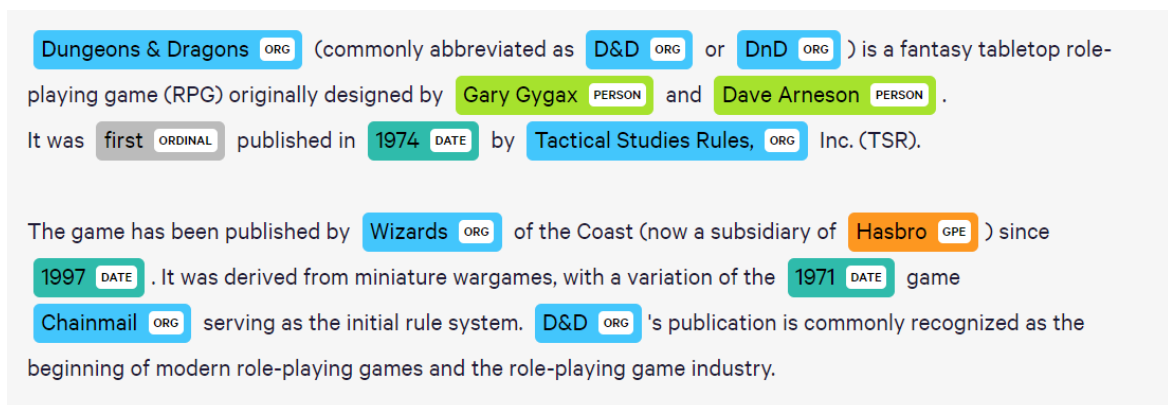


Figure 2.8: DisplaCy Named Entity Visualizer

2.3.2 Neuralcoref

Neuralcoref is a neural coreference resolution system that acts as a pipeline expansion for SpaCy. The main feature of this tool is to recognize and resolve coreference clusters by the use of a neural network[10].

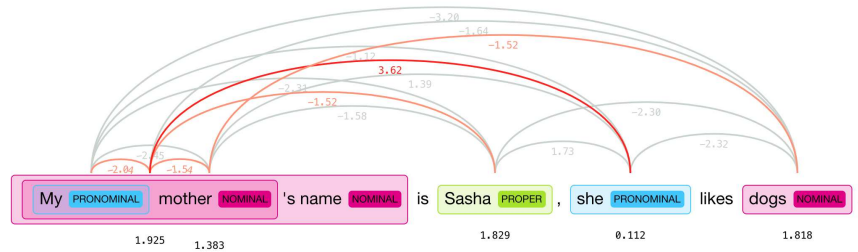


Figure 2.9: An example of coreference clusters

Coreference resolution algorithms tend to work by extracting a series of mentions from the text targeted, compute a set of features on them and then find the most likely antecedent for each mention, if there's one.

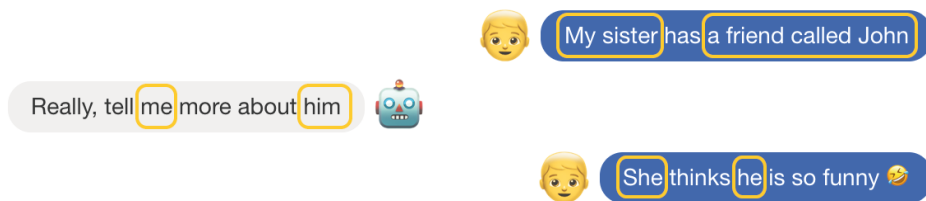


Figure 2.10: Highlighting of mentions

The aforementioned set of features used to be hand-crafted from linguistic rules and context. Neuralcoref, as others state-of-the-art systems, makes use of modern NLP techniques like word vectors and neural networks in order to reduce the volume of hand-crafted features while still maintaining a good accuracy. This process, while being still heavily dependent on the training set, does show positive results by analysing and altering the word vectors surrounding each mention and thus providing information about the context of the phrases.

Neuralcoref then continue feeding the gathered data to two neural networks, the first responsible of scoring possible pairs of mentions and antecedents and the second scoring the possibility of a mention having no antecedent.

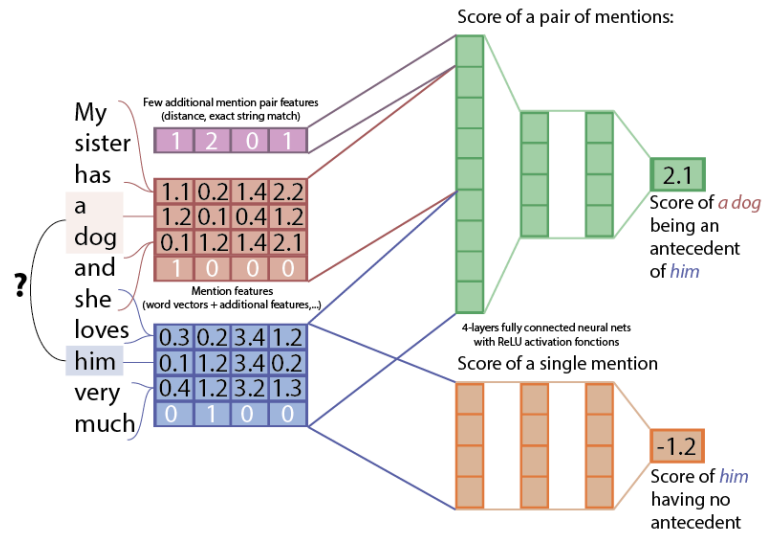


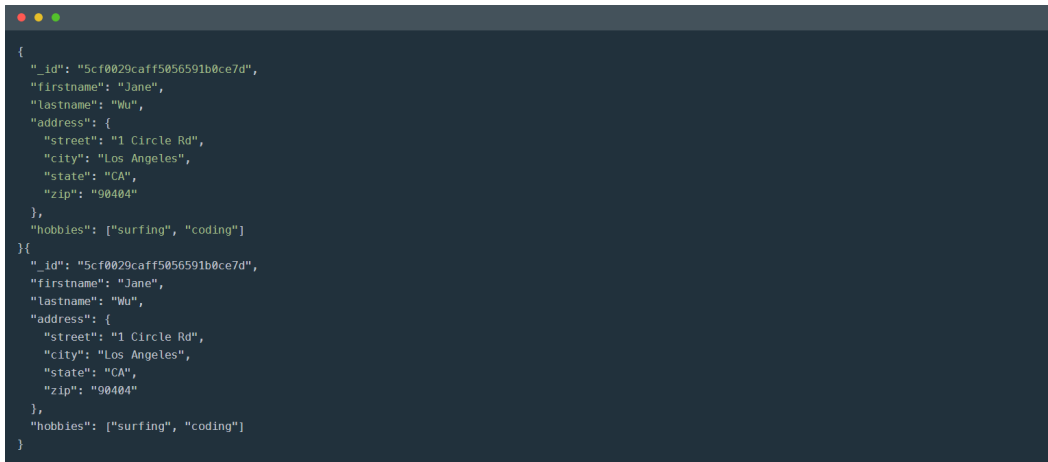
Figure 2.11: Neuralcoref scoring model

It then compares all the scores gathered and thus predict the most likely antecedent, if any, for each mention[11].

2.3.3 MongoDB

MongoDB is a NoSQL document-oriented database program developed by MongoDB Inc[12].

While the term “NoSQL” highlights the non-relational nature of the database, the term “document-oriented” comes from its use of JSON-like documents to store its data. MongoDB actually represents JSON documents in binary-encoded format called BSON and by doing this it improve its encoding and decoding efficiency.



```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": ["surfing", "coding"]
}
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": ["surfing", "coding"]
}
```

Figure 2.12: MongoDB Documents

Being a non-relational database, MongoDB has a flexible scheme and therefore supports documents without a rigid structure. It guarantees high availability through its replica sets. These are composed of several instances of MongoDB divided between primary and secondary replicas. The former are responsible for the interaction with the user and for the execution of reading and writing operations on the data, the latter instead aim to preserve an updated copy of them. In case of failure of the primary copy, a secondary copy automatically takes its place.

By being naturally able to coordinate multiple servers to store data MongoDB gets to be classified also as a distributed database. It also takes care of horizontal data scaling through an automatic sharding process. It splits the data over multiple instances and automatically balances the load in order to maintain the system running even in the event of hardware failure[13].

Chapter 3

Design

The main focus of the project subject of this thesis is the conversion of sentences, expressed in natural language, to first-order logic predicates. The entities mentioned in the original sentence, their attributes and the relations involving them must all be represented through the use of predicates and variables.

The following sections will discuss the design behind the software architecture, the predicates involved in the conversion, the workflow regarding the conversion process itself and the structure of the database dedicated to storing the conversion's results.

3.1 Software architecture

The software was developed in **Python** and among its various modules, which are going to be discussed shortly, two are dedicated to interacting with third-party components.

The NLP library **SpaCy** was employed in order to parse the text to be converted and therefor obtain various linguistic annotations. **Neuralcoref** was added to the default pipeline of SpaCy as a mean to clear the text to be converted from eventual coreferences. The non-relational DBMS **MongoDB** was employed in order to create two collections of documents which are going to be discussed in the following section. These have been utilized to store the sentences fed to the converter and the terms obtained from the conversion process.

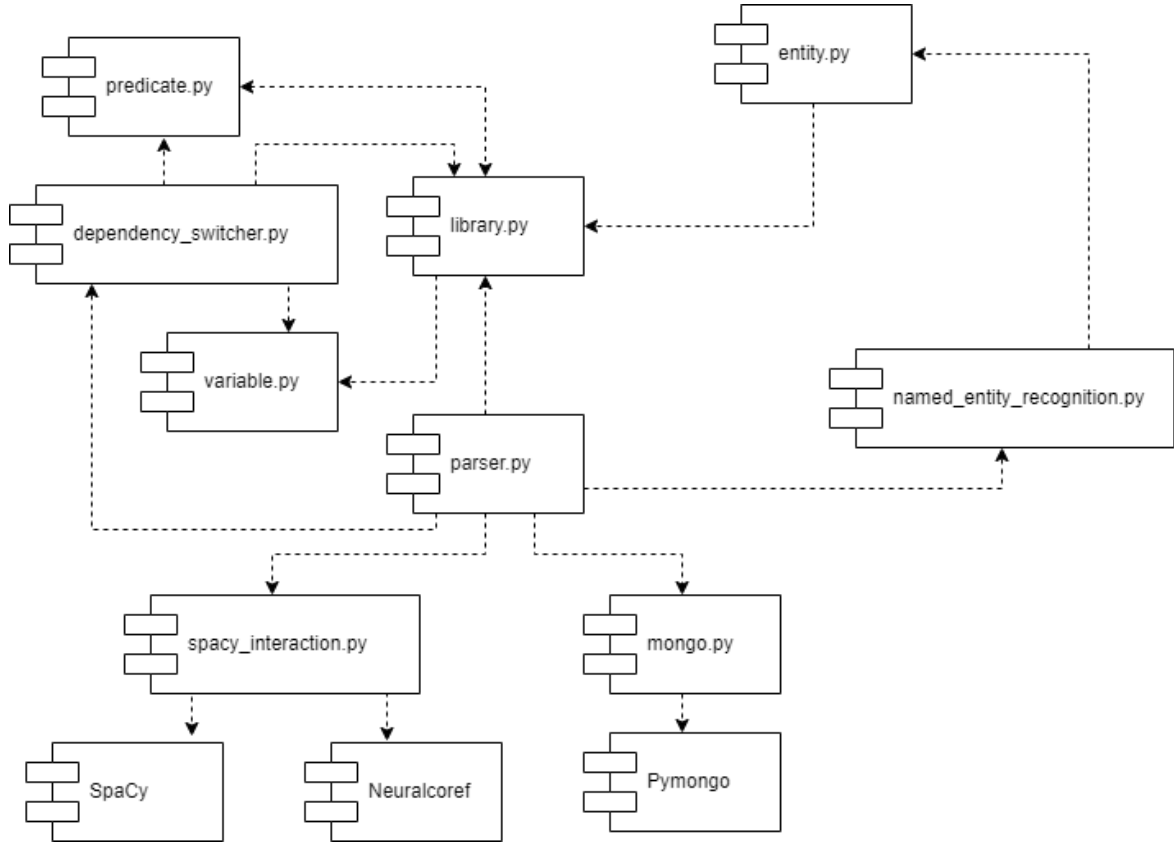


Figure 3.1: Component Diagram

A more precise overview of each of the parser's modules follows:

- **parser.py** is the main component of the software and is responsible of interaction with the rest of the modules managing the entire workflow.
- **predicate.py** and **variable.py** contain the classes Predicate and Variable. They're imported by **dependency_switcher.py** and **library.py** in order to create and add new predicates as the parser keeps analysing tokens.
- **dependency_switcher.py** is the module responsible of analysing every token that forms the phrase to be converted. It contains a dictionary of methods, each of which is dedicated to the management of a particular semantic dependency, or group of dependencies.
- **library.py** contains a set of methods dedicated to interacting with predicates, variables and tokens. Most of them are utilized during the creation and alteration of the terms obtained by analysing the tokens. It also contains methods to

print the annotation regarding the text parsed by SpaCy and the terms obtained through the conversion.

- **entity.py** and **named_entity_recognition.py** are imported by **parser.py** and **library.py** in order to implement the feature that highlights the entity present in the phrases.
- **spacy_interaction.py** is responsible of sending the text to be parsed to SpaCy. When requested it interacts with Neuralcoref in order to point out which coreference clusters, if any, are present in the phrase and resolve them.
- **mongo.py** is responsible of implementing the upload of phrases and predicates on the database. It presents means to upload proposition of both FLAT and ISA types whose details are discussed in chapter 4.2.

3.2 Structure of a FOL predicate

The following diagram shows all the classes defined and used throughout the parser. In order to properly document the interaction with the SpaCy library, the Token class was included in the diagram along with a small set of its attributes.

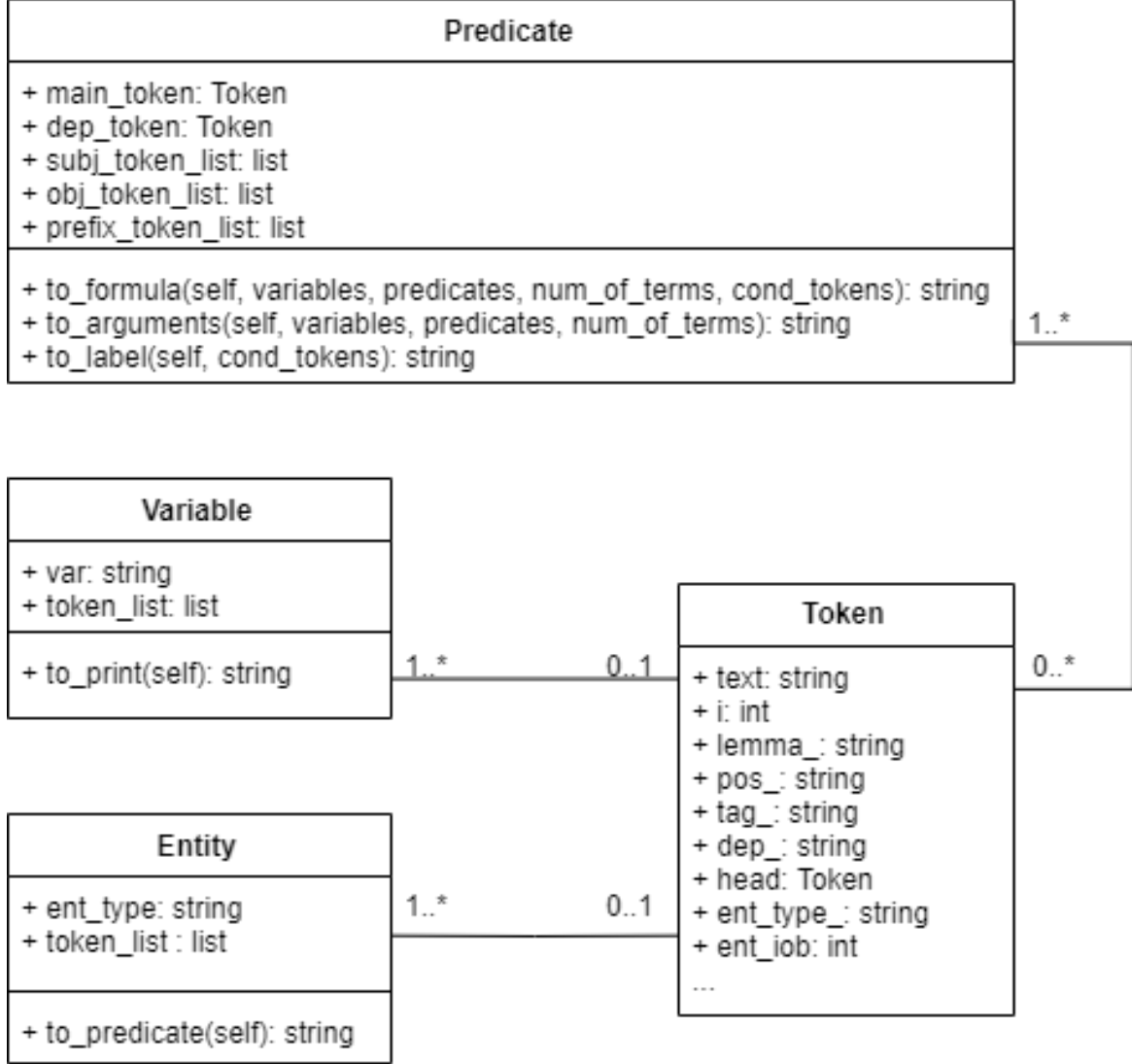


Figure 3.2: Class Diagram

Throughout the conversion process of a sentence, each concept, meaningful for the purpose of knowledge representation, is expressed through predicates and variables.

Variables are used to represent the entities and relations expressed through the original sentence and, beyond that, to refer to them from the various predicates. The davidsonian variable e_x is used to represent the events, expressed by verbs, described by

the sentences while normal variables in the form of x_1, x_2, \dots, x_n are used to represent the remaining entities. As shown by the Variable class field "token_list", each instance of the class is linked to one, or more, token. Each occurrence of a token from the aforementioned list within an instance of the Predicate class will be substituted by the associated variable during the actual creation of the predicate itself.

During the "token by token" analysis, each predicate is created but not yet irreversibly defined. Every new analysis could add a potential subject, object or prefix to an existing predicate thus modifying its label or arguments. Every instance of the Predicate class, once fully defined, is made up by a label and by one to three arguments, these being tokens represented by variables contained in brackets. Each label can again be divided into two parts, the lemma and the part-of-speech tag, separated by a colon ":".

predicate:NN(x_1)

Each field of the Predicate class refers either to a single token or to a set of them. For example in the phrase "Frodo and Sam come from the Shire.", the tokens "Frodo" and "Sam", being both the subject of the token "come", will share the variable x_1 .

Follows a more precise overview of each of the fields of the Predicate class including predicates whose underlined elements highlight their dependence on the described fields:

- the **main_token** is the token to which the parser refers when the label is created. The first half of the label is composed by the lemma of the main_token while the second half by its tag as part-of-speech. Both of this proprieties regarding the tokens are accessible, through attributes like **.lemma_** and **.tag_**, thanks to SpaCy's annotations.

predicate:NN(x_1)

- the **dep_token** is the token referred to by the parser during the creation of the first and main argument of the predicate. For predicates referring to entities and events this token is the same as the main_token and points out to the variables

associated with them. For predicates referring to tokens involved in other kinds of semantic dependencies, like modifiers or prepositions, the `dep_token` refers to the token target of said dependencies.

predicate:NN(x_1)

- the **subj_token_list** contains all of the tokens referred as subject, and thus second arguments, by a specific predicate, this usually representing either an event or a preposition.

You must be Igor.

You:PRP(x_1) & be:VB(e_1 , x_1 , x_2) & Igor:NNP(x_2)

Every token in the same `subj_list` share the same variable with the others subjects like in phrases as "Batman and Robin protect Gotham".

Batman and Robin protect Gotham.

**Batman:NNP(x_1) & Robin:NNP(x_1) & protect:VBD(e_1 , x_1 , x_2) &
Gotham:NNP(x_2)**

- the **obj_token_list** contains all of the tokens referred as object, and thus third and last arguments, by a specific predicate representing an event.

You must be Igor.

You:PRP(x_1) & be:VB(e_1 , x_1 , x_2) & Igor:NNP(x_2)

Just as the tokens sharing a `subj_token_list`, even the token sharing an `obj_token_list` are associated with the same variable. For example in "Smaug took the mountain and the treasure".

Smaug took the mountain and the treasure.

**Smaug:NNP(x_1) & take:VBD(e_1 , x_1 , x_2) & mountain:NN(x_2) &
treasure:NN(x_2)**

- the **prefix_token_list** is usually utilized during the conversion of tokens involved in an open clausal complement (xcomp) dependency. The tokens in this list are used to create a sub-label that will be placed before the label produced by the main_token. A similar phenomenon is observable in the conversion of the phrase "Daenerys wants to rule the seven kingdoms.", where the token "rule" will be placed in the prefix_token_list of the predicate representing the token "want".

Daenerys wants to rule the seven kingdoms.

Daenerys:NNP(x_1) & rule:VB_want:VBP(e_1, x_1, x_2) & seven:CD(x_2)
& kingdom:NNS(x_2)

3.3 Overview of the parser workflow

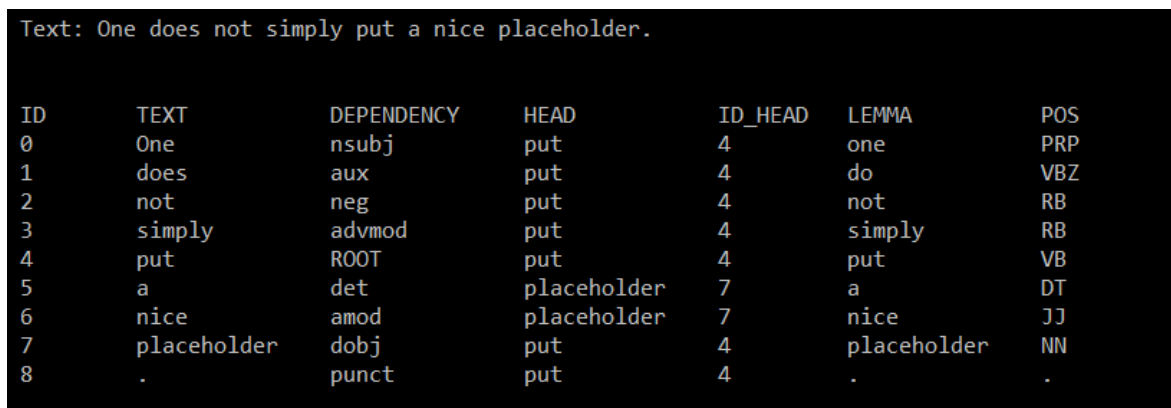
The job of the parser starts with the input of text, either typed by the user or contained in a file. Regardless of the number of sentences within the text, the latter is sent to SpaCy to be analysed and thus obtain various linguistic annotations regarding it.

If the number of sentences is more than one, however, the parser will use the meta-attributes originated by adding Neuralcoref to SpaCy's pipeline in order to identify and resolve any coreference present in it.

After the possible alteration of the text, this is divided into individual sentences which are sent again to SpaCy, this time individually.

SpaCy will start by tokenizing the text and thus splitting it into tokens, these will be stored in a doc object. The latter will therefore be involved in a set of processes through which SpaCy will be able to identify, and predict, many properties of the tokens themselves. Said processes consist in: lemmatization, part-of-speech tagging, dependency parsing and named entity recognition. For a detailed description of these see chapter 2.3.1.

With the return of the doc object by SpaCy the results of the analysis, and the annotations coming with them, will be displayed to the user through a table.



Text: One does not simply put a nice placeholder.

ID	TEXT	DEPENDENCY	HEAD	ID_HEAD	LEMMA	POS
0	One	nsubj	put	4	one	PRP
1	does	aux	put	4	do	VBZ
2	not	neg	put	4	not	RB
3	simply	advmod	put	4	simply	RB
4	put	ROOT	put	4	put	VB
5	a	det	placeholder	7	a	DT
6	nice	amod	placeholder	7	nice	JJ
7	placeholder	dobj	put	4	placeholder	NN
8	.	punct	put	4	.	.

Figure 3.3: Table containing linguistic annotations

Every token contained in the doc object will then be analyzed by the dependency switcher. This is a dictionary of functions that filters the tokens based on the semantic dependencies that involve them and uses them as arguments for the many methods it

contains. Each of these have been specifically developed to process a particular set of semantic dependencies. While some methods, like the one dedicated to the nominal subject of a verb (nsubj), refer to a single dependency, others try to refer to a broad category of them. The method processing the modifiers, for example, refers to many semantic dependencies including: the adjectival (amod), appositional (appos) or even numerical (nummod) modifiers.

Some dependencies are considered unnecessary for the purpose of representing the sentence through first-order predicates. The tokens involved in these dependencies will thus be ignored by the switcher. The articles preceding a noun (det) are an example of this phenomenon.

The methods contained in the switcher are responsible for the creation of each predicate and variable involved in the terms obtained through the conversion. Both the dependency switcher and the structure of the aforementioned predicates will be addressed in the following chapters.

After having cycled through all the tokens, the parser will proceed by displaying the results of the conversion, both predicates and variables, to the user and then by highlighting every named entity present in the text. Each variable will be paired up with the list of every token referred into its token_list and each of said tokens will be represented through its id and lemma.

The following picture displays the results of the conversion of the sentence referred previously:

```
PREDICATES:
put:VB(e1, x1, x2)
One:PRP(x1)
simply:RB(e1)
placeholder:NN(x2)
nice:JJ(x2)

VARIABLES:
e1: [4-put ]
x1: [0-one ]
x2: [7-placeholder ]
```

Figure 3.4: Results of a conversion

Eventually the sentence fed to the parser and the terms obtained from its conversion will be stored in the database. Whenever terms derived from ISA type sentences are identified and marked during conversion, the storing process will proceed to treat them accordingly.

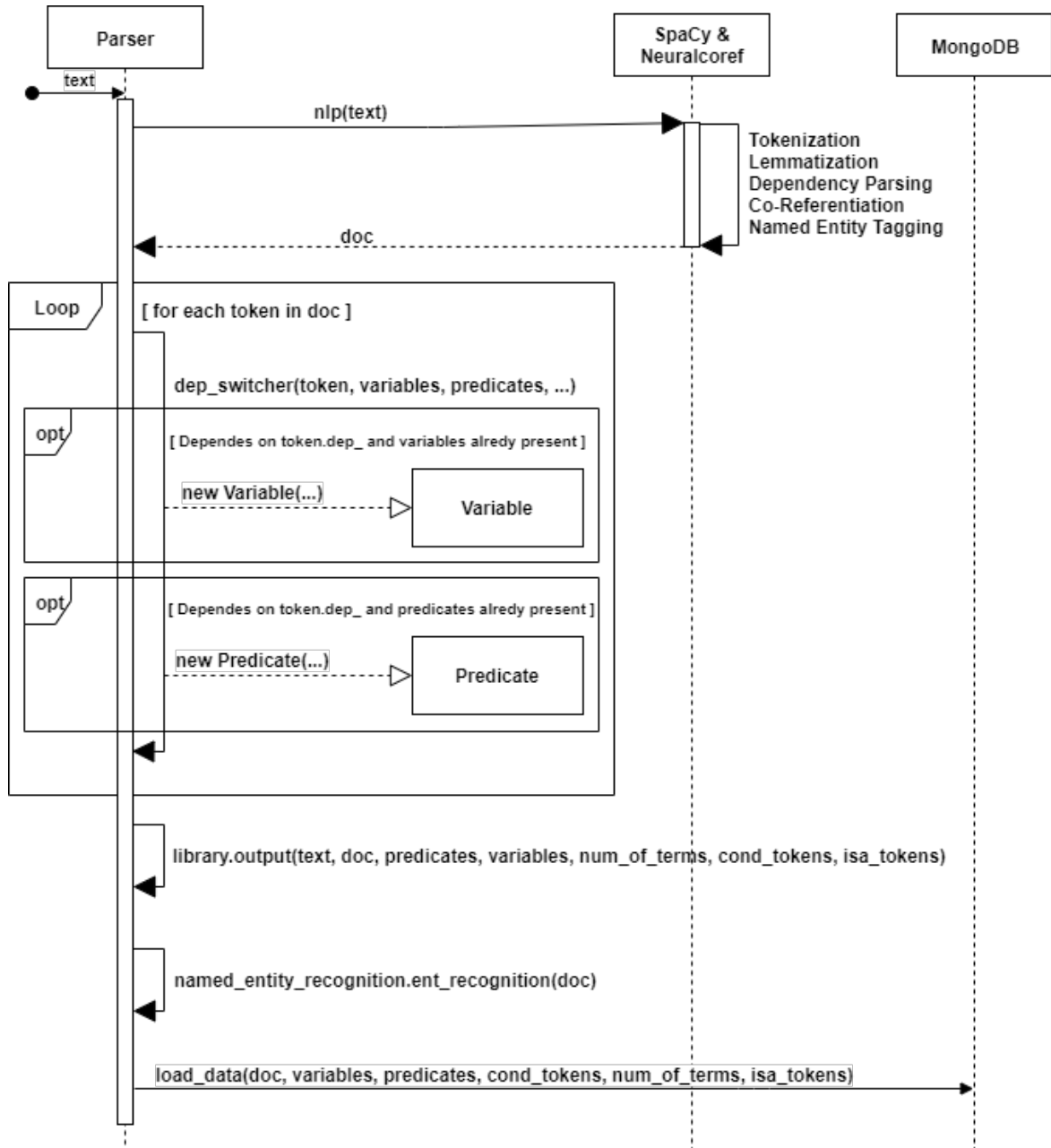


Figure 3.5: Workflow of the parser

3.4 Database collections

The collections Sentences and Terms have been created in order to contain the phrases converted by the parser and the predicates obtained by the conversion itself.

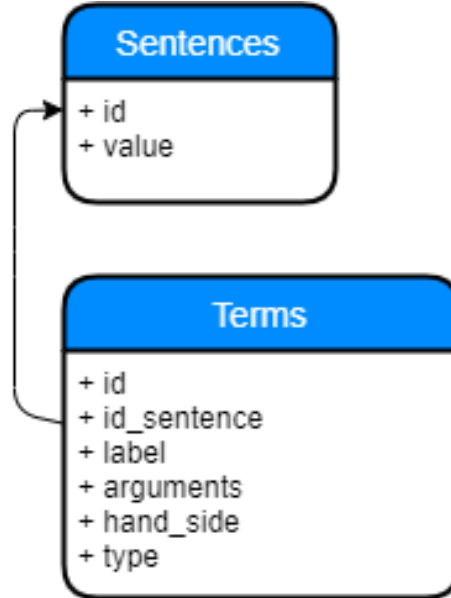


Figure 3.6: Structure of Sentences and Terms

The **id** field from both of the collections is automatically generated by MongoDB. The field **value** from the Sentence collection refers to the text that has been fed to the parser.

```
{
  "_id": "5de91fa32baf150dc5268217",
  "value": "42 is the Answer."
}
```

Source Code 3.1: Example of the document of a sentence

The **id** field from the Sentences collection is used as **id_sentence** in the Terms collection thus linking every sentence to the terms obtained by its parsing.

The field **label** is the label associated with each predicate, it originates from the main token of the predicate just as described in "Structure of a FOL predicate".

The field **arguments** is made up by the variables that act as arguments for each predicates contained in square brackets.

The field **hand_side** depends on the type of the sentence and on the relative position of the token. It's BLANK for every term of a FLAT sentence and L, M or R for every term of a ISA sentence.

The field **type** obviously indicates the type of sentence that originated the terms. It can be either FLAT or ISA.

The following documents represent an example of the upload of the sentence "42 is the Answer." and the terms originated from its conversion.

```
{
  "_id": "5de91fa32baf150dc5268219",
  "id_sentence": "5de91fa32baf150dc5268217",
  "label": "42:CD",
  "arguments": "[x1, _, _]",
  "hand_side": "L",
  "type": "ISA"
}
{
  "_id": "5de91fa32baf150dc5268218",
  "id_sentence": "5de91fa32baf150dc5268217",
  "label": "be:VBZ",
  "arguments": "[e1, x1, x2]",
  "hand_side": "M",
  "type": "ISA"
}
{
  "_id": "5de91fa32baf150dc526821a",
  "id_sentence": "5de91fa32baf150dc5268217",
  "label": "Answer:NN",
  "arguments": "[x2, _, _]",
  "hand_side": "R",
  "type": "ISA"
}
```

Source Code 3.2: Example of the documents of several terms

Chapter 4

Implementation

4.1 The Dependency Switcher

The dependency switcher is a dictionary of functions responsible for the analysis and manipulation of tokens contained in each sentence. There are forty-five possible semantic dependencies identifiable by SpaCy and the dependency switcher, through twenty different methods, is able to deal with the conversion of thirty-two of these.

Some dependencies, such as those referring to determinants, parataxis or punctuation, are ignored by the switcher because of their uselessness in creating a conversion in first-order logical predicates.

Others, such as the one referring to the adverbial clause modifier (*advcl*), have not been treated because of their complexity and unpredictability. A common behavior was not identified for the tokens involved in these dependencies and therefore it was not possible to construct a method dedicated to their management.

The tokens are not the only arguments requested by the switcher, the rest being:

- **variables** and **predicates**, two lists containing all of the variables and predicates actually defined by the parser;
- **cond_tokens**, a list referencing each token expressing a condition in the form of "when...", they're identified by the parser through the analysis of the marker (mark) dependency and the part-of-speech WRB;
- **passive_tokens**, a list containing all of the tokens referring to a verb in the

passive form, they're identified through the analysis of the passive auxiliary (auxpass) dependency;

- **num_of_terms**, a list utilized during the creation of each new variable, it contains the counters related to the number of variables x_n and e_n already defined by the parser;
- **isa_tokens**, a list containing all of the tokens that occupy a central position within an ISA-type sentence, they're identified through the analysis of tokens involved in root and prepositional (prep) dependencies.

The code of the methods responsible of dealing with the tokens involved in the nominal subject (nsubj) and the direct object (dobj) dependencies are going to be presented as examples, the code regarding the rest of the methods referred by the switcher is contained in the Appendix A.

```
1 def dep_dobj(token, variables, predicates, cond_tokens,
   ↪ passive_tokens, num_of_terms, isa_tokens):
2 target_token = token.head
3 if target_token.dep_ is "xcomp":
4     target_token = library.until_not("xcomp", target_token)
5 if not library.add_to_obj_list(token, target_token, variables,
   ↪ predicates, num_of_terms):
6     library.make_predicate(predicates, target_token, target_token,
   ↪ None, token)
7 library.make_mono_predicate(token, predicates)
```

dep_dobj()

What this method essentially does, without referring to a particular case like the one involving the open clausal complement (xcomp) dependencies, is:

1. Search among the existing predicates for the one associated with the verb of which the analyzed token is object. If the predicate is found then the method adds the object token into the obj_token_list of said predicate.

2. If the predicate associated with the verb is not found, it create said predicate and adds the object token into his obj_token_list.
3. Create the predicate referencing the object token.

The following method, responsible of dealing with token that are subjects to verbs, exhibit a workflow similar to the one just describe, of course considering the inversion between objects and subjects.

```
1 def dep_nsubj(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     target_token = token.head
3     if target_token.dep_ is not "parataxis":
4         if target_token.dep_ is "ccomp":
5             if not library.add_to_subj_list(token, target_token,
  ↪ variables, predicates, num_of_terms):
6                 library.make_predicate(predicates, target_token,
  ↪ target_token, token)
7         elif target_token.dep_ is "relcl":
8             if not library.add_to_subj_list(target_token.head,
  ↪ target_token, variables, predicates, num_of_terms):
9                 library.make_predicate(predicates, target_token,
  ↪ target_token, target_token.head)
10        else:
11            if not library.add_to_subj_list(token, target_token,
  ↪ variables, predicates, num_of_terms):
12                library.make_predicate(predicates, target_token,
  ↪ target_token, token)
13        library.make_mono_predicate(token, predicates)
```

dep_nsubj()

Both methods, as every other method referenced by the dependency switcher, make wide use of the methods belonging to library.py. The latter being a library containing

a set of methods built in order to simplify the interaction between the switcher and the Variable and Predicate classes.

The following tables provide an overview of each type of semantic dependency identifiable by SpaCy. Each of these is accompanied by a sentence acting as an example and by the name of the method responsible of its treatment, if present.

Each of the aforementioned sentences present two highlighted tokens describing the behaviour of a specific dependency. These represent the main token involved in the said dependency, in **blue**, and the token targeted by it, in **red**.

dep	Semantic Dependency	Example [child → head]	Treated by
acl	Clausal modifier of noun	There are many online sites offering booking facilities.	dep_acl
acomp	Adjectival complement	You are so beautiful	dep_dobj
advcl	Adverbial clause modifier	She came to see me	untreated
advmod	Adverbial modifier	Bob hastily run into the room.	dep_advmod
agent	Agent	The car was bought by Sam	(implicitly by dep_pobj)
amod	Adjectival modifier	It's a bright new day.	dep_modifier
appos	Appositional modifier	John, my brother, is a student	dep_modifier
attr	Attribute	He is a student	dep_dobj
aux	Auxiliary	I will meet her tomorrow	untreated
auxpass	Auxiliary (passive)	Placido was taken very far away.	dep_auxpass
case	Case marker	John's car	untreated
cc	Coordinating conjunction	Batman and Robin watch over Gotham	untreated
ccomp	Clausal complement	He says you like to swim.	dep_ccomp
compound	Compound modifier	Barack Obama was a fine president.	dep_compound
conj	Conjunct	Batman and Robin watch over Gotham	dep_conj
csubj	Clausal subject	What she said makes sense.	dep_csubj
csubjpass	Clausal subject (passive)	That she lied was suspected by everyone.	dep_csubjpass
dative	Dative	Give me your word	dep_modifier
dep	Unclassified dependent	—	untreated
det	Determiner	The US military	untreated
dobj	Direct Object	She bought these books for me	dep_dobj

dep	Semantic Dependency	Example [child → head]	Treated by
expl	Expletive	There was an explosion	untreated
intj	Interjection	Well, it is my birthday	untreated
mark	Marker	The owls wake up when the sun goes down.	dep_advmod
meta	Meta modifier	Applause Thank you	untreated
neg	Negation modifier	I am never coming back	untreated
nsubj	Nominal subject	Han shot first	dep_nsubj
nsubjpass	Nominal subject (passive)	Placido was taken away.	dep_nsubjpass
nummod	Number modifier	Seven million dollars	dep_modifier
oprd	Object predicate	He was judged guilty .	dep_oprd
parataxis	Parataxis	She, I mean , Mary was here	untreated
pcomp	Complement of preposition	They heard about you missing classes.	dep_pcomp
pobj	Object of preposition	The lords of the seas .	dep_pobj
poss	Possession modifier	I bought his car	dep_modifier
preconj	Pre-correlative conjunction	Either Batman or Robin watch over Gotham	untreated
predet	Pre-determiner	All the books we read	dep_modifier
prep	Prepositional modifier	Placido was the greatest ranger of his universe.	dep_prep
prt	Particle	Shut down the machine	dep_modifier
punct	Punctuation	This is an example.	untreated
quantmod	Modifier of quantifier	More than five	dep_modifier
relcl	Relative clause modifier	I saw the book which you bought .	dep_relcl
root	Root	—	dep_root
xcomp	Open clausal complement	I want to rule the world.	dep_xcomp

The list of methods belonging to **library.py**, whose bodies are found in appendix B, is included as a reference.

```
1 def same_lemma_tag(token_a, token_b):
2
3 def get_lemma(token, cond_tokens):
4
5 def get_predicate(target_token, predicates):
6
7 def make_mono_predicate(token, predicates):
8
9 def make_predicate(predicates, main_token, dep_token, subj_token =
  ↳ None, obj_token = None, prefix_token = None):
10
11 def name_var(token, num_of_terms):
12
13 def get_var(token, variables, predicates, num_of_terms):
14
15 def make_var(token, variables, predicates, num_of_terms):
16
17 def add_to_same_var_list(token_added, token_to_add, variables,
  ↳ predicates, num_of_terms):
18
19 def add_to_subj_list(token_to_add, target_token, variables,
  ↳ predicates, num_of_terms):
20
21 def add_to_obj_list(token_to_add, target_token, variables, predicates,
  ↳ num_of_terms):
22
23 def add_to_prefix(token_to_add, target_token, predicates):
24
25 def until_not(dep, token):
26
```



```
27 def print_table(doc):
28
29 def print_predicates(variables, predicates, num_of_terms,
    ↪ cond_tokens):
30
31 def print_variables(variables):
32
33 def print_isa(isa_tokens):
34
35 def output(text, doc, predicates, variables, num_of_terms, cond_tokens,
    ↪ isa_tokens):
36
37 def clear_lists(predicates, variables, cond_tokens, passive_tokens,
    ↪ num_of_terms, isa_tokens):
```

Methods from library.py

Follows the code belonging to the dictionary responsible of pairing up every token to the method dedicated to the dependency with which it is involved:

```
1 def dep_switcher(token, variables, predicates, cond_tokens,
2   ↪ passive_tokens, num_of_terms, isa_tokens):
3     switcher = {
4         "acl": dep_acl,
5         "acomp": dep_dobj,
6         "advmod": dep_advmod,
7         "amod": dep_modifier,
8         "appos": dep_modifier,
9         "attr": dep_dobj,
10        "auxpass": dep_auxpass,
11        "ccomp": dep_ccomp,
12        "compound": dep_compound,
13        "conj": dep_conj,
14        "csubj": dep_csubj,
15        "csubjpass": dep_csubjpass,
16        "dative": dep_modifier,
17        "dobj": dep_dobj,
18        "mark": dep_advmod,
19        "npadvmod": dep_modifier,
20        "nmod": dep_modifier,
21        "npmod": dep_modifier,
22        "nsubj": dep_nsubj,
23        "nsubjpass": dep_nsubjpass,
24        "nummod": dep_modifier,
25        "oprd": dep_oprd,
26        "pcomp": dep_pcomp,
27        "pobj": dep_pobj,
28        "poss": dep_modifier,
29        "predet": dep_modifier,
30        "prep": dep_prep,
```

```

30     "prt": dep_modifier,
31     "quantmod": dep_modifier,
32     "relcl": dep_relcl,
33     "ROOT": dep_root,
34     "xcomp": dep_xcomp
35 }
36 func = switcher.get(token.dep_, lambda token, variables,
    ↪ predicates, cond_tokens, passive_tokens, num_of_terms,
    ↪ isa_tokens: dep_scarta)
37 func(token, variables, predicates, cond_tokens, passive_tokens,
    ↪ num_of_terms, isa_tokens)

```

dep_switcher()

4.2 Recognition of FLAT-type and ISA-type sentences

Every sentence structured as "X is/are Y" or "X is made of/by Y" is considered an ISA-type sentence. "The platypus is a semiaquatic mammal" and "A pack is made up by dozens of wolves" are ISA-type sentences, all other sentences are considered having FLAT-types.

The purpose of converting ISA-type sentences is to build the bases for a possible replacement rule that would link the tokens preceding the main verb of the sentence with those that follow it. At the end of the conversion phase, the tokens inserted into the isa_token list are associated with a hand_side equals to M, which stands for "middle". The one preceding them get classified with a L, for "left" and, obviously, those following it with a R for "right".

```
1 def get_hand_side(predicate, isa_tokens):
2     if predicate.main_token in isa_tokens:
3         return "M"
4     elif predicate.main_token.i < isa_tokens[0].i:
5         return "L"
6     else:
7         return "R"
```

get_hand_side()

Considering, for example, the conversion of an ISA-type sentence like:

The platypus is a semiaquatic mammal.

Text: The platypus is a semiaquatic mammal.						
ID	TEXT	DEPENDENCY	HEAD	ID_HEAD	LEMMA	POS
0	The	det	platypus	1	the	DT
1	platypus	nsubj	be	2	platypus	NN
2	is	ROOT	be	2	be	VBZ
3	a	det	mammal	5	a	DT
4	semiaquatic	amod	mammal	5	semiaquatic	JJ
5	mammal	attr	be	2	mammal	NN
6	.	punct	be	2	.	.

$$\text{platypus:NN}(x_1) \ \& \ \text{be:VBZ}(e_1, x_1, x_2) \ \& \ \text{mammal:NN}(x_2) \ \& \ \text{semiaquatic:JJ}(x_2)$$

with `hand_side` respectively equals to:

- M for "be";
- L for "platypus";
- R for "mammal" and "semiaquatic".

The eventual replacement rule highlighted by the parser would be:

$$\text{platypus:NN}(x_1) \Rightarrow \text{mammal:NN}(x_2) \ \& \ \text{semiaquatic:JJ}(x_2)$$

Among all of the methods contained in the dependency switcher, those dedicated to the management of the ROOT and prep dependencies are also those dedicated to the identification of tokens belonging to an ISA type sentence. While the prep dependency is associated with prepositions, the root dependency is associated with the tokens at the base of the dependency tree, the tree linking all the tokens in a sentence through their semantic dependency. Each sentence has just one token involved in a root dependency.

```

1 def dep_root(token, variables, predicates, cond_tokens, passive_tokens,
  ↪ num_of_terms, isa_tokens):
2     if token.lemma_ == "be":
3         for child in token.children:
4             if child.dep_ == "attr":
5                 isa_tokens.append(token)
6                 break
7     elif token.lemma_ == "make":
8         for child in token.children:
9             if child.text == "of" or child.text == "by":
10                 isa_tokens.append(token)
11                 break

```

`dep_root()`

What the method responsible for their analysis actually does is searching for tokens with a lemma equals to "be" or "make". Once these tokens are found the method proceeds by checking if the "be" token is referenced by an attribute (attr) dependency or by doing an additional test regarding the involvement of the "make" token with any prepositions and, finally, inserting them in the list.

4.3 Co-reference Resolution

Thanks to the interaction with Neuralcoref, the parser is able to resolve any coreference present in the sentences to be converted. The substitutions of the mentions highlighted by the tool, displayable through the meta-attribute `doc._.coref_clusters`, are implicitly already performed, by Neuralcoref itself, and applied to the altered text stored in the meta-attribute `doc._.coref_resolved`.

To make the replacement of a mention containing a possessive pronoun more evident, the *"spacy_parsing_coref_poss"* method, responsible for adding a Saxon genitive ('s) to the end of each set of tokens involved in these substitutions, was developed. The phenomenon is observable by considering the text:

"Neil Gaiman is a competent writer. His books are quite captivating."

```
Text: Neil Gaiman is a competent writer. His books are quite captivating.
```

ID	TEXT	DEPENDENCY	HEAD	ID_HEAD	LEMMA	POS
0	Neil	compound	Gaiman	1	Neil	NNP
1	Gaiman	nsubj	be	2	Gaiman	NNP
2	is	ROOT	be	2	be	VBZ
3	a	det	writer	5	a	DT
4	competent	amod	writer	5	competent	JJ
5	writer	attr	be	2	writer	NN
6	.	punct	be	2	.	.
7	His	poss	book	8	-PRON-	PRP\$
8	books	nsubj	be	9	book	NNS
9	are	ROOT	be	9	be	VBP
10	quite	advmod	captivating	11	quite	RB
11	captivating	acomp	be	9	captivating	JJ
12	.	punct	be	9	.	.

Which, by letting Neuralcoref carry out his default substitution of mentions, would turn in:

"Neil Gaiman is a competent writer. Neil Gaiman books are quite captivating."

```
Text: Neil Gaiman is a competent writer. Neil Gaiman books are quite captivating.
```

ID	TEXT	DEPENDENCY	HEAD	ID_HEAD	LEMMA	POS
0	Neil	compound	Gaiman	1	Neil	NNP
1	Gaiman	nsubj	be	2	Gaiman	NNP
2	is	ROOT	be	2	be	VBZ
3	a	det	writer	5	a	DT
4	competent	amod	writer	5	competent	JJ
5	writer	attr	be	2	writer	NN
6	.	punct	be	2	.	.
7	Neil	compound	Gaiman	8	Neil	NNP
8	Gaiman	compound	book	9	Gaiman	NNP
9	books	nsubj	be	10	book	NNS
10	are	ROOT	be	10	be	VBP
11	quite	advmod	captivating	12	quite	RB
12	captivating	acomp	be	10	captivating	JJ
13	.	punct	be	10	.	.

Only slightly different from the text version returned by the `spacy_parsing_coref_poss`

method:

Neil Gaiman is a competent writer. Neil Gaiman's books are quite captivating.

```
Text: Neil Gaiman is a competent writer. Neil Gaiman's books are quite captivating.
```

ID	TEXT	DEPENDENCY	HEAD	ID_HEAD	LEMMA	POS
0	Neil	compound	Gaiman	1	Neil	NNP
1	Gaiman	nsubj	be	2	Gaiman	NNP
2	is	ROOT	be	2	be	VBZ
3	a	det	writer	5	a	DT
4	competent	amod	writer	5	competent	JJ
5	writer	attr	be	2	writer	NN
6	.	punct	be	2	.	.
7	Neil	compound	Gaiman	8	Neil	NNP
8	Gaiman	poss	book	10	Gaiman	NNP
9	's	case	Gaiman	8	's	POS
10	books	nsubj	be	11	book	NNS
11	are	ROOT	be	11	be	VBP
12	quite	advmod	captivating	13	quite	RB
13	captivating	acom	be	11	captivating	JJ
14	.	punct	be	11	.	.

This method, however, can be considered functional only with text written in English and therefore avoids the generality required for the converter.

The whole coreference resolution process has proven itself not suitable for use on single sentences, as well as being deeply dependent on the model used and the "greediness" configured for it, therefore it is not included in the set of standard operations performed on the input sentences.

The following are the scripts contained in **spacy_interaction.py**, the module responsible for communicating with the SpaCy library and implementing the coreference resolution.

```
1 import spacy
2 import neuralcoref
3
4 nlp = spacy.load("en_core_web_lg")
5 neuralcoref.add_to_pipe(nlp)
6
7 def spacy_parsing(text):
8     return nlp(text)
9
10 def spacy_parsing_coref_poss(text):
11     doc = nlp(text)
12     if doc._.has_coref:
13         poss_clusters = []
14         for x in doc._.coref_clusters:
15             poss_mention = []
16             i = 0
17             for y in x.mentions:
18                 if y is not x.main:
19                     token = doc[y.end - 1]
20                     if token.dep_ is "poss":
21                         poss_mention.append(i+1)
22                     i += 1
23             poss_clusters.append([x.main.text, poss_mention])
24     new_doc = str(doc._.coref_resolved)
25     for cluster in poss_clusters:
26         for occurrence in cluster[1]:
27             # Finding nth occurrence of substring
28             val = -1
```

```
29         for i in range(0, occurrence):
30             val = new_doc.find(cluster[0], val + 1)
31             # Printing nth occurrence ends at
32             index = val + len(cluster[0])
33             new_doc = new_doc[:index] + "'s" + new_doc[index:]
34         return nlp(new_doc)
35     else:
36         return doc
```

methods from spacy_interaction.py

4.4 Database Interaction

The interaction between the parser and MongoDB has been implemented through the import of the Python distribution PyMongo. Every phrase converted by the parser is stored in the Sentences collection while the predicates obtained by their conversion are stored in the Terms collection.

```
1 import pymongo
2
3 myclient = pymongo.MongoClient("mongodb://localhost:27017/")
4 mydb = myclient["nlp"]
5 col_sentences = mydb["Sentences"]
6 col_terms = mydb["Terms"]
7
8 def load_data(doc, variables, predicates, cond_tokens, num_of_terms,
9 ↪ isa_tokens, isa_tokens = None):
10     sentence_to_add = { "value": doc.text }
11     x = col_sentences.insert_one(sentence_to_add)
12     sentence_id = x.inserted_id
13
14     for predicate in predicates:
15         term_to_add = {}
16         term_to_add["id_sentence"] = sentence_id
17         term_to_add["label"] = predicate.to_label(cond_tokens)
18         term_to_add["arguments"] = predicate.to_arguments(variables,
19 ↪ predicates, num_of_terms)
20
21     if isa_tokens != []:
22         term_to_add["hand_side"] = get_hand_side(predicate,
23 ↪ isa_tokens)
24         term_to_add["type"] = "FLAT"
25     else:
26         term_to_add["hand_side"] = "BLANK"
27         term_to_add["type"] = "FLAT"
```

25

26

```
col_terms.insert_one(term_to_add)
```

methods from mongo.py

Just as the methods responsible for conversion are able to identify and mark the type of converted sentences, the methods responsible for uploading to the database are also designed to be able to determine the hand_side of the terms belonging to sentences of both types FLAT and ISA.

Chapter 5

Evaluation

The conversion process provided by the converter has been tested using a set of specific sentences. This set was built by including sentences with very different semantic patterns in order to push the converter to interact with the largest number of different semantic dependencies possible. A correct and complete translation has been obtained for each of the sentences included in the set but this does not guarantee the same conversion performance for sentences of a greater complexity.

The great variety of semantic dependencies, combined with the virtually infinite number of combinations, makes the development of "a perfect converter" a complex goal. The "perfect conversion" is also opposed by possible errors in the linguistic annotations provided by third-party components such as SpaCy. However, being, these components, usually powered by statistical models, it is possible to improve their performance through supplemented training sessions with custom datasets.

An important factor to consider in the development of similar software is the generality expected from the context to which it is addressed. Limiting the variety of syntax that can be used by the user when entering sentences, for example through a set of syntactic patterns, limits the variety of semantic dependencies to be managed and simplifies the conversion process. This however not without having a negative impact on the range of application of the converter. The development of such software therefore remains at the center of a constant compromise between generality and conversion performance.

5.1 Semantic Ambiguities

The correctness of the conversions provided by the parser has proved to be uncertain with respect to particular patterns that can be found in the sentences fed by the users. Often, as displayed in the following example, the performance of the converter has been conditioned by the ambiguity of the sentence itself, a fairly common problem when dealing with natural languages. In the sentence:

John went on a trip and attended a concert with his friends.

Text: John went on a trip and attended a concert with his friends.						
ID	TEXT	DEPENDENCY	HEAD	ID_HEAD	LEMMA	POS
0	John	nsubj	go	1	John	NNP
1	went	ROOT	go	1	go	VBD
2	on	prep	go	1	on	IN
3	a	det	trip	4	a	DT
4	trip	pobj	on	2	trip	NN
5	and	cc	go	1	and	CC
6	attended	conj	go	1	attend	VBD
7	a	det	concert	8	a	DT
8	concert	dobj	attend	6	concert	NN
9	with	prep	attend	6	with	IN
10	his	poss	friend	11	-PRON-	PRP\$
11	friends	pobj	with	9	friend	NNS
12	.	punct	go	1	.	.

it's not clear if John was with his friends during the whole trip or exclusively during the concert and thus the converter is unable to effectively handle the sentence.

Another problem related to the conversion process based on the analysis of the tree of dependencies is the transparency of the converter with respect to the transitivity or intransitivity of a verb. A non-negligible defect when compared to phrases presenting a single token, involved in a direct object dependency, pointing to a group of verbs linked by a conjunction.

This case is embodied by the following sentences, of which the first one is composed of two transitive verbs, both referring to the same object, and the second one by an intransitive and a transitive one.

Bruno loves to watch and play football.

Text: Bruno loves to watch and play football.

ID	TEXT	DEPENDENCY	HEAD	ID_HEAD	LEMMA	POS
0	Bruno	nsubj	love	1	Bruno	NNP
1	loves	ROOT	love	1	love	VBZ
2	to	aux	watch	3	to	TO
3	watch	xcomp	love	1	watch	VB
4	and	cc	watch	3	and	CC
5	play	conj	watch	3	play	VB
6	football	dobj	play	5	football	NN
7	.	punct	love	1	.	.

Bruno loves to swim and play football.

Text: Bruno loves to swim and play football.

ID	TEXT	DEPENDENCY	HEAD	ID_HEAD	LEMMA	POS
0	Bruno	nsubj	love	1	Bruno	NNP
1	loves	ROOT	love	1	love	VBZ
2	to	aux	swim	3	to	TO
3	swim	xcomp	love	1	swim	VB
4	and	cc	swim	3	and	CC
5	play	conj	swim	3	play	VB
6	football	dobj	play	5	football	NN
7	.	punct	love	1	.	.

As shown by the following tables, there are no differences in the dependencies involving their tokens thus the linguistic annotations provided by SpaCy regarding the dependency tree of the sentences are basically the same.

Even when working with two transitive verbs, a defect of the converter has proven to be his lack of "common sense" and, therefore, his inability to make decisions dictated by a context taken for granted for a human being but not derivable from the sole tree of dependencies of a sentence.

A phenomenon that can be observed by confronting the different possible meanings of the sentence:

Bruno loves to eat and to pet dogs.

Text: Bruno loves to eat and to pet dogs.						
ID	TEXT	DEPENDENCY	HEAD	ID_HEAD	LEMMA	POS
0	Bruno	nsubj	love	1	Bruno	NNP
1	loves	ROOT	love	1	love	VBZ
2	to	aux	eat	3	to	TO
3	eat	xcomp	love	1	eat	VB
4	and	cc	eat	3	and	CC
5	to	conj	eat	3	to	TO
6	pet	amod	dog	7	pet	VB
7	dogs	pobj	to	5	dog	NNS
8	.	punct	love	1	.	.

A sentence, from the point of view of the tree of the dependencies, indistinguishable from those already presented in this chapter.

Another requisite necessary in order to obtain good performance from the conversion process is the respect of formal grammar rules regarding the sentences to be parsed. An easily forgettable detail is the use of multiple passive auxiliaries in the case of sentences with multiple verbs in passive form. The lack of a passive auxiliary referred to a verb eliminates the only clue concerning its nature and thus alters the process of attribution of a subject or object to the same verb.

The following sentence does present a pattern that would lead to this very same error, where the absence of an auxiliary referring to the token "taken" makes impossible the association of "Donald Trump" as an object for his predicate.

Donald Trump was judged guilty and taken to prison.

Text: Donald Trump was judged guilty and taken to prison.						
ID	TEXT	DEPENDENCY	HEAD	ID_HEAD	LEMMA	POS
0	Donald	compound	Trump	1	Donald	NNP
1	Trump	nsubjpass	judge	3	Trump	NNP
2	was	auxpass	judge	3	be	VBD
3	judged	ROOT	judge	3	judge	VRB
4	guilty	oprd	judge	3	guilty	JJ
5	and	cc	judge	3	and	CC
6	taken	conj	judge	3	take	VRB
7	to	prep	take	6	to	IN
8	prison	pobj	to	7	prison	NN
9	.	punct	judge	3	.	.

5.2 Possible Applications

Ultimately the software fits into the context of natural language processing as a flexible tool dedicated to building a knowledge base, a role that places it at the base of cutting-edge systems like intelligent assistants, chatbots and other software related to question-answering. This last option has been explored by many teams whose management of logical predicates presents parallels with that implemented in the software object of the thesis. One of the systems resulting from said exploration is described in **Transforming Dependency Structures to Logical Forms for Semantic Parsing**[14].

Said system, although akin to that described in this thesis, differs, among other things, for a conversion order based on a detailed hierarchy between the semantic dependencies present in the sentence to be converted and for the use of neo-Davidsonian logical forms, an extension of the Davidsonian ones used by the software object of the thesis. Both these strategies, as well as the others described by the aforementioned paper, serve as inspiration for any improvements applicable to the software in question.

Chapter 6

Conclusions

The software at the heart of this thesis has proved to be capable of generating predicates of the first order starting from sentences expressed in natural language. The performance of the software, intended as the correctness of the conversions obtained, was considered satisfactory in relation to the results obtained during the testing phase.

The conversion process currently implemented is not able to interact with sentences of particularly high complexity while maintaining acceptable results. However, the architecture underlying the system was built to adapt to the frequent alteration, and evolution, of individual conversion methods. Therefore the software lends itself to future improvements in order to improve its conversion potential. The application of the same to a specific context would lead to the definition of constraints referring the context itself and to the plausible selection of the set of semantic dependencies on which to focus the conversion process.

The current lack of an inference system involving the predicates obtained through the conversion process provides a clear direction regarding project developments. Even the transition to neo-Davidsonian logical forms or the creation of event-related predicates of nouns, strategies implemented in the system described in Transforming Dependency Structures to Logical Forms for Semantic Parsing, could lead to an increase in both performance and robustness.

Appendix A

Scripts from `dependency_switcher.py`

This appendix contains the methods that make up the dependency switcher, the code of each method is accompanied by the name of the semantic dependency to which it is dedicated and a sentence as an example.

Each sentence present two colored tokens, the first, in blue, is the token involved in the dependency that is the subject of the method, the second, in red, is the target of this semantic dependence.

For example, in the sentence "Marco is working on a project", "Marco", as subject of the verb work, is involved in a `nsubj` dependency targeting "working".

```
1 def dep_acl(token, variables, predicates, cond_tokens, passive_tokens,
  ↪ num_of_terms, isa_tokens):
2     target_token = token.head
3     if not library.add_to_subj_list(target_token, token, variables,
  ↪ predicates, num_of_terms):
4         library.make_predicate(predicates, token, token, target_token)
```

dep_acl()

```

1 def dep_advmod(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     if not library.get_predicate(token, predicates):
3         if token.tag_ == "WRB":
4             cond_tokens.append(token)
5             library.make_predicate(predicates, token, token.head)
6         else:
7             dep_modifier(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens)

```

dep_advmod()

```

1 def dep_auxpass(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     target_token = token.head
3     if target_token not in passive_tokens:
4         passive_tokens.append(target_token)

```

dep_auxpass()

```

1 def dep_modifier(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     if not library.get_predicate(token, predicates):
3         target_token = token.head
4         if target_token.dep_ in dep_modifiers:
5             while target_token.dep_ in dep_modifiers:
6                 target_token = target_token.head
7             elif target_token.dep_ is "xcomp":
8                 target_token = library.until_not("xcomp", target_token)
9                 library.make_mono_predicate(target_token, predicates)
10                library.make_predicate(predicates, token, target_token)

```

dep_modifier()

```

1 def dep_ccomp(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     target_token = token.head
3     library.make_mono_predicate(token, predicates)
4     if not library.add_to_obj_list(token, target_token, variables,
  ↪ predicates, num_of_terms):
5         library.make_predicate(predicates, target_token, target_token,
  ↪ None, token)

```

dep_ccomp()

```

1 def dep_compound(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     target_token = token.head
3     if target_token.dep_ is "poss":
4         library.make_predicate(predicates, token, target_token)
5     elif target_token.dep_ is "conj":
6         conj_head = library.until_not("conj", token.head)
7         if conj_head.dep_ in dep_modifiers:
8             conj_head_predicate = library.get_predicate(conj_head,
  ↪ predicates)
9             library.make_predicate(predicates, token.head,
  ↪ conj_head_predicate.dep_token, None, None, token)
10    else:
11        dep_modifier(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens)

```

dep_compound()

```

1 def dep_conj(token, variables, predicates, cond_tokens, passive_tokens,
  ↪ num_of_terms, isa_tokens):
2     target_token = token.head
3     token_predicate = library.get_predicate(token, predicates)
4     target_predicate = library.get_predicate(target_token, predicates)
5     conj_head = library.until_not("conj", token)
6     if conj_head.dep_ is "xcomp":
7         if not library.add_to_prefix(conj_head.head, token,
  ↪ predicates):
8             head_predicate =
  ↪ library.get_predicate(library.until_not("xcomp",
  ↪ conj_head), predicates)
9             library.make_predicate(predicates, token, token,
  ↪ head_predicate.subj_token_list, None, conj_head.head)
10 elif conj_head.dep_ == "ROOT" or conj_head.dep_ is "relcl" or
  ↪ conj_head.dep_ is "advcl" or conj_head.tag_ in
  ↪ library.pos_verbs:
11     for cond_token in cond_tokens:
12         cond_predicate = library.get_predicate(cond_token,
  ↪ predicates)
13         if cond_predicate.dep_token == conj_head:
14             library.make_predicate(predicates, cond_token, token)
15             break
16     if token_predicate is not None:
17         if target_token in passive_tokens and
  ↪ token_predicate.obj_token_list == [] and
  ↪ token_predicate.subj_token_list == []:
18             token_predicate.obj_token_list =
  ↪ target_predicate.obj_token_list

```

```

19         elif target_token not in passive_tokens and token not in
           ↪ passive_tokens and token_predicate.subj_token_list ==
           ↪ []:
20             token_predicate.subj_token_list =
           ↪ target_predicate.subj_token_list
21     else:
22         if target_token in passive_tokens:
23             library.make_predicate(predicates, token, token, None,
           ↪ target_predicate.obj_token_list)
24         else:
25             library.make_predicate(predicates, token, token,
           ↪ target_predicate.subj_token_list, None)
26     elif conj_head.dep_ is "attr" or target_token.dep_ is "nsubj" or
           ↪ target_token.dep_ is "nsubjpass" or target_token.dep_ is
           ↪ "dobj" or target_token.dep_ is "acomp":
27         library.add_to_same_var_list(target_token, token, variables,
           ↪ predicates, num_of_terms)
28     elif conj_head.dep_ in dep_modifiers:
29         if not library.get_predicate(token, predicates):
30             target_predicate = library.get_predicate(target_token,
           ↪ predicates)
31             library.make_predicate(predicates, token,
           ↪ target_predicate.dep_token)
32     elif conj_head.dep_ is "pobj":
33         prep_token = conj_head.head
34         if prep_token.dep_ is "agent":
35             prep_head = prep_token.head
36             library.add_to_subj_list(token, prep_head, variables,
           ↪ predicates, num_of_terms)

```

dep_conj()

```

1 def dep_csubj(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     target_token = token.head
3     if target_token.dep_ is "aux" or target_token.dep_ is "auxpass":
4         target_token = target_token.head
5     token_predicate = library.get_predicate(token, predicates)
6     target_predicate = library.get_predicate(target_token, predicates)
7     if target_predicate is not None:
8         target_predicate.subj_token_list =
  ↪ token_predicate.obj_token_list
9     else:
10        library.make_predicate(predicates, target_token, target_token,
  ↪ token_predicate.obj_token_list)

```

dep_csubj()

```

1 def dep_csubjpass(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     target_token = token.head
3     if target_token.dep_ is "aux" or target_token.dep_ is "auxpass":
4         target_token = target_token.head
5     token_predicate = library.get_predicate(token, predicates)
6     target_predicate = library.get_predicate(target_token, predicates)
7     if target_predicate is not None:
8         target_predicate.subj_token_list =
  ↪ token_predicate.subj_token_list
9     else:
10        library.make_predicate(predicates, target_token, target_token,
  ↪ token_predicate.subj_token_list)

```

dep_csubjpass()


```

1 def dep_dobj(token, variables, predicates, cond_tokens, passive_tokens,
  ↪ num_of_terms, isa_tokens):
2     target_token = token.head
3     if target_token.dep_ is "xcomp":
4         target_token = library.until_not("xcomp", target_token)
5     if not library.add_to_obj_list(token, target_token, variables,
  ↪ predicates, num_of_terms):
6         library.make_predicate(predicates, target_token, target_token,
  ↪ None, token)
7     library.make_mono_predicate(token, predicates)

```

dep_dobj()

```

1 def dep_oprd(token, variables, predicates, cond_tokens, passive_tokens,
  ↪ num_of_terms, isa_tokens):
2     target_token = token.head
3     target_predicate = library.get_predicate(target_token, predicates)
4
5     if target_token in passive_tokens:
6         if not library.get_predicate(token, predicates) and
  ↪ target_predicate.obj_token_list != []:
7             library.make_predicate(predicates, token,
  ↪ target_predicate.obj_token_list[0])
8     else:
9         if not library.get_predicate(token, predicates) and
  ↪ target_predicate.subj_token_list != []:
10            library.make_predicate(predicates, token,
  ↪ target_predicate.subj_token_list[0])

```

dep_oprd()

```

1 def dep_nsubj(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     target_token = token.head
3     if target_token.dep_ is not "parataxis":
4         if target_token.dep_ is "ccomp":
5             if not library.add_to_subj_list(token, target_token,
  ↪ variables, predicates, num_of_terms):
6                 library.make_predicate(predicates, target_token,
  ↪ target_token, token)
7         elif target_token.dep_ is "relcl":
8             if not library.add_to_subj_list(target_token.head,
  ↪ target_token, variables, predicates, num_of_terms):
9                 library.make_predicate(predicates, target_token,
  ↪ target_token, target_token.head)
10        else:
11            if not library.add_to_subj_list(token, target_token,
  ↪ variables, predicates, num_of_terms):
12                library.make_predicate(predicates, target_token,
  ↪ target_token, token)
13        library.make_mono_predicate(token, predicates)

```

dep_nsubj()

```

1 def dep_nsubjpass(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     target_token = token.head
3     if not library.add_to_obj_list(token, target_token, variables,
  ↪ predicates, num_of_terms):
4         library.make_predicate(predicates, target_token, target_token,
  ↪ None, token)

```

dep_nsubjpass()

```

1 def dep_pobj(token, variables, predicates, cond_tokens, passive_tokens,
  ↪ num_of_terms, isa_tokens):
2     if token.dep_ is "conj":
3         pobj_token = library.until_not("conj", token)
4         prep_token = pobj_token.head
5     else:
6         prep_token = token.head
7     if prep_token.dep_ is "prt":
8         target_token = prep_token.head
9         if target_token.dep_ is "xcomp":
10             target_token = library.until_not("xcomp", target_token)
11         if not library.add_to_obj_list(token, target_token, variables,
  ↪ predicates, num_of_terms):
12             library.make_predicate(predicates, target_token,
  ↪ target_token, None, token)
13         library.make_mono_predicate(token, predicates)
14     else:
15         if prep_token.dep_ is "conj":
16             prep_head = library.until_not("conj", prep_token)
17             prep_head = prep_head.head
18         else:
19             prep_head = prep_token.head
20         if prep_head.dep_ is "appos":
21             prep_head = library.until_not("appos", prep_head)
22         elif prep_head.dep_ is "xcomp":
23             prep_head = library.until_not("xcomp", prep_head)
24         if prep_token.dep_ is "agent":
25             library.add_to_subj_list(token, prep_head, variables,
  ↪ predicates, num_of_terms)
26         while prep_head.dep_ is "conj" and prep_head.head in
  ↪ passive_tokens:

```

```

27         library.add_to_subj_list(token, prep_head.head,
    ↪     variables, predicates, num_of_terms)
28     prep_head = prep_head.head
29
30     elif library.get_predicate(prepare_token, predicates) is None:
31         library.make_mono_predicate(token, predicates)
32         library.make_mono_predicate(prepare_head, predicates)
33         library.make_predicate(predicates, prepare_token, prep_head,
    ↪     token)

```

dep_pobj()

```

1 def dep_prep(token, variables, predicates, cond_tokens, passive_tokens,
    ↪     num_of_terms, isa_tokens):
2     if token.head in isa_tokens and (token.text == "of" or token.text
    ↪     == "by"):
3         isa_tokens.append(token)

```

dep_prep()

```

1 def dep_pcomp(token, variables, predicates, cond_tokens,
    ↪     passive_tokens, num_of_terms, isa_tokens):
2     prep_token = token.head
3     prep_head = prep_token.head
4     if prep_head.dep_ is "xcomp":
5         prep_head = library.until_not("xcomp", prep_head)
6     if library.get_predicate(prepare_token, predicates) is None:
7         library.make_mono_predicate(token, predicates)
8         library.make_predicate(predicates, prepare_token, prep_head,
    ↪     token)

```

dep_pcomp()

```

1 def dep_relcl(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     if library.get_predicate(token, predicates) is None:
3         library.make_predicate(predicates, token, token, token.head)
4     else:
5         library.add_to_subj_list(token.head, token, variables,
  ↪ predicates, num_of_terms)

```

dep_relcl()

```

1 def dep_xcomp(token, variables, predicates, cond_tokens,
  ↪ passive_tokens, num_of_terms, isa_tokens):
2     library.add_to_prefix(token, token.head, predicates)

```

dep_xcomp()

```

1 def dep_root(token, variables, predicates, cond_tokens, passive_tokens,
  ↪ num_of_terms, isa_tokens):
2     if token.lemma_ == "be":
3         isa_tokens.append(token)
4     elif token.lemma_ == "make":
5         for child in token.children:
6             if child.text == "of" or child.text == "by":
7                 isa_tokens.append(token)
8                 break

```

dep_root()

Appendix B

Scripts from library.py

This appendix contains the methods that make up the library used by the parser to interact with the instances of the Variable, Predicate and Token classes.

Some methods, like **make_predicate()** and **make_var()**, concern the creation of such instances. Others, like **add_to_subj_list()** and **add_to_same_var_list()**, concern their manipulation.

```
1 def make_mono_predicate(token, predicates):
2     if get_predicate(token, predicates) is None and (token.tag_ not in
    ↪ pos_verbs or (token.tag_ == "VBG" and token.dep_ in
    ↪ dep_modifiers)):
3         predicates.append(Predicate.Predicate(token, token))
```

make_mono_predicate() - utilized to create predicates with a single arguments like the ones referring to nouns

```
1 def make_predicate(predicates, main_token, dep_token, subj_token =
    ↪ None, obj_token = None, prefix_token = None):
2     predicates.append(Predicate.Predicate(main_token, dep_token,
    ↪ subj_token, obj_token, prefix_token))
```

make_predicate() - utilized to create complex predicates

```

1 def get_predicate(target_token, predicates):
2     for predicate in predicates:
3         if target_token == predicate.main_token:
4             return predicate
5     else:
6         return None

```

get_predicate() - utilized to search and eventually get a predicate from the predicates list

```

1 def make_var(token, variables, predicates, num_of_terms):
2     var = name_var(token, num_of_terms)
3     variables.append(Variable.Variable(token, var))
4     if token.tag_ not in pos_verbs or (token.tag_ == "VBG" and
5         ↪ token.dep_ in dep_modifiers):
6         make_mono_predicate(token, predicates)

```

make_var() - utilized to exclusively create a define a new variable referring to specific token

```

1 def name_var(token, num_of_terms):
2     if token.tag_ in pos_verbs and token.dep_ is not "pobj":
3         num_of_terms[1] += 1
4         return "e" + str(num_of_terms[1])
5     else:
6         num_of_terms[0] += 1
7         return "x" + str(num_of_terms[0])

```

name_var() - utilized to define a new variable

```

1 def get_var(token, variables, predicates, num_of_terms):
2     if token.dep_ is "xcomp":
3         return get_var(token.head, variables, predicates, n_noun,
4             ↪ n_verb)
5     for variable in variables:
6         if token in variable.token_list:
7             return variable.var
8     else:
9         var = name_var(token, num_of_terms)
10        variables.append(Variable.Variable(token, var))
11        if token.tag_ not in pos_verbs or (token.tag_ == "VBG" and
12            ↪ token.dep_ in dep_modifiers):
13            make_mono_predicate(token, predicates)
14    return var

```

get_var() - utilized to get the variable referring to a specific token or defining a new one

```

1 def get_lemma(token, cond_tokens):
2     lemma = ""
3     if token in cond_tokens:
4         lemma = "COND"
5     elif token.tag_ == "PRP" or token.tag_ == "PRP\$" or (token.tag_
6         ↪ == "VBG" and token.dep_ in dep_modifiers):
7         lemma = token.text
8     else:
9         lemma = token.lemma_
10    return lemma

```

get_lemma() - utilized during the creation of the labels


```

1 def add_to_same_var_list(token_added, token_to_add, variables,
  ↪ predicates, num_of_terms):
2     if variables:
3         for variable in variables:
4             if token_to_add not in variable.token_list:
5                 for token in variable.token_list:
6                     if token_added == token:
7                         variable.token_list.append(token_to_add)
8                         make_mono_predicate(token_to_add, predicates)
9                         return
10            else:
11                return
12    make_var(token_added, variables, predicates, num_of_terms)
13    add_to_same_var_list(token_added, token_to_add, variables,
  ↪ predicates, num_of_terms)

```

add_to_same_var_list() - utilized in order to associate a first token to var already defined and associated with a second one

```

1 def add_to_subj_list(token_to_add, target_token, variables,
  ↪ predicates, num_of_terms):
2     for predicate in predicates:
3         if target_token == predicate.main_token:
4             if predicate.subj_token_list != []:
5                 add_to_same_var_list(predicate.subj_token_list[0],
  ↪ token_to_add, variables, predicates, num_of_terms)
6                 predicate.subj_token_list.append(token_to_add)
7                 return True
8     return False

```

add_to_subj_list() - utilized to add a token as a subject to a predicate referring to a specific token

```

1 def add_to_obj_list(token_to_add, target_token, variables, predicates,
  ↪ num_of_terms):
2     for predicate in predicates:
3         if target_token == predicate.main_token:
4             if predicate.obj_token_list != []:
5                 add_to_same_var_list(predicate.obj_token_list[0],
  ↪ token_to_add, variables, predicates, num_of_terms)
6                 predicate.obj_token_list.append(token_to_add)
7                 return True
8     return False

```

add_to_obj_list() - utilized to add a token as an object to a predicate referring to a specific token

```

1 def add_to_prefix(token_to_add, target_token, predicates):
2     for predicate in predicates:
3         if target_token == predicate.main_token:
4             predicate.prefix_token_list.append(token_to_add)
5             return True
6     return False

```

add_to_prefix() - utilized to add a token as a prefix to the label of predicate referring to a specific token

```

1 def until_not(dep, token):
2     result_token = token
3     while result_token.dep_ is dep:
4         result_token = result_token.head
5     return result_token

```

until_not() - utilized to trace the arcs of dependencies in order to get the first token not involved in the specified dependency

```

1 def clear_lists(predicates, variables, cond_tokens, passive_tokens,
  ↪ num_of_terms, isa_tokens):
2     if predicates:
3         for predicate in predicates:
4             predicate.subj_token_list.clear()
5             predicate.obj_token_list.clear()
6             predicate.prefix_token_list.clear()
7
8         predicates.clear()
9
10    if variables:
11        for variable in variables:
12            variable.token_list.clear()
13
14        variables.clear()
15
16    if cond_tokens:
17        cond_tokens.clear()
18
19    if passive_tokens:
20        passive_tokens.clear()
21
22    if isa_tokens:
23        isa_tokens.clear()
24
25    num_of_terms = [0, 0]

```

clear_lists() - utilized in order to wipe clean the information regarding a previous conversion and ready the parser to begin a new one

```

1 def same_lemma_tag(token_a, token_b):
2     return token_a.lemma_ == token_b.lemma_ and token_a.tag_ ==
    ↪ token_b.tag_

```

same_lemma_tag() - utilized to check if two tokens have equals lemma and part-of-speech tag

```

1 def print_table(doc):
2     print("\n")
3     print("{:<10}{:<15}{:<15}{:<15}{:<10}{:<15}{:<10}".format(" ID",
    ↪ "TEXT", "DEPENDENCY", "HEAD", "ID_HEAD", "LEMMA", "POS"))
4     for token in doc:
5         print("{:<10}{:<15}{:<15}{:<15}{:<10}{:<15}{:<10}".format(" "
    ↪ + str(token.i), token.text, token.dep_, token.head.lemma_,
    ↪ token.head.i, token.lemma_, token.tag_))

```

print_table() - utilized in order to print the linguistic annotations regarding a specific text

```

1 def print_predicates(variables, predicates, num_of_terms,
    ↪ cond_tokens):
2     print("\n")
3     print(" PREDICATES:\n")
4     for x in predicates:
5         print(" " + x.to_formula(variables, predicates, num_of_terms,
    ↪ cond_tokens))

```

print_predicate() - utilized in order to print the predicates obtained through the conversion process of a sentence

```

1 def print_variables(variables):
2     print("\n")
3     print(" VARIABLES:\n")
4     for x in variables:
5         print(" " + x.to_print())

```

print_variables() - utilized in order to print the variables defined during the conversion process of a sentence

```

1 def print_isa(isa_tokens):
2     print("\n")
3     print(" ISA TOKENS:\n")
4     for x in isa_tokens:
5         print(" " + x.lemma_ + str(x.i))

```

print_isa() - utilized in order to print the tokens classified as isa_tokens during the conversion process of a sentence

```

1 def output(text, doc, predicates, variables, num_of_terms, cond_tokens,
↪ isa_tokens):
2     print("\n{:<7}{:<150}".format(" Text: ", text))
3     print_table(doc)
4     print_predicates(variables, predicates, num_of_terms, cond_tokens)
5     print_variables(variables)
6     print_isa(isa_tokens)
7     print("\n")

```

output() - utilized in order to print all of the information regarding the conversion process of a sentence

Bibliography

- [1] Ronak Vijay. *A Gentle Introduction to Natural Language Processing*. Aug. 2019. URL: <https://towardsdatascience.com/a-gentle-introduction-to-natural-language-processing-e716ed3c0863>.
- [2] Jason Brownlee. *What Is Natural Language Processing?* Aug. 2019. URL: <https://machinelearningmastery.com/natural-language-processing/>.
- [3] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson India Education Services Pvt. Ltd., 2018.
- [4] *Formal Systems*. URL: <https://cs.lmu.edu/~ray/notes/formalsystems/>.
- [5] *Formal system*. Oct. 2019. URL: https://en.wikipedia.org/wiki/Formal_system.
- [6] Asad Sayeed. *Neo-Davidsonian semantics A systematic exploration of the ways Brutus can do violence to Caesar*.
- [7] *Introducing spaCy · Blog · Explosion*. URL: <https://explosion.ai/blog/introducing-spacy>.
- [8] Jinho D. Choi, Joel Tetreault, and Amanda Stent. “It Depends: Dependency Parser Comparison Using A Web-based Evaluation Tool”. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, July 2015, pp. 387–396. DOI: 10.3115/v1/P15-1038. URL: <https://www.aclweb.org/anthology/P15-1038>.
- [9] *spaCy 101: Everything you need to know · spaCy Usage Documentation*. URL: <https://spacy.io/usage/spacy-101>.

- [10] Huggingface. *huggingface/neuralcoref*. Oct. 2019. URL: <https://github.com/huggingface/neuralcoref>.
- [11] Thomas Wolf. *State-of-the-art neural coreference resolution for chatbots*. Oct. 2017. URL: <https://medium.com/huggingface/state-of-the-art-neural-coreference-resolution-for-chatbots-3302365dcf30>.
- [12] *What Is MongoDB?* URL: <https://www.mongodb.com/what-is-mongodb>.
- [13] *What is MongoDB? Introduction, Architecture, Features and Example*. URL: <https://www.guru99.com/what-is-mongodb.html>.
- [14] Siva Reddy et al. “Transforming Dependency Structures to Logical Forms for Semantic Parsing”. In: *Transactions of the Association for Computational Linguistics* 4 (2016), pp. 127–140. DOI: 10.1162/tac1_a_00088. URL: <https://www.aclweb.org/anthology/Q16-1010>.

Ringraziamenti

Ringrazio il **Prof. Antonio Puliafito** per aver reso possibile la realizzazione di questo progetto.

Ringrazio il **Dott. Carmelo Fabio Longo** per avermi fatto da guida in un territorio sconosciuto ma incredibilmente affascinante.

E ringrazio di cuore il **Prof. Francesco Longo** per la sua infinità disponibilità ed il suo costante incoraggiamento che ha spesso fatto la differenza.

Ringrazio la mia famiglia che ha continuato a supportarmi, anche dopo essermi tagliato i capelli, e ha sempre fatto per me ben più del possibile.

A mia **Madre** che prova per me abbastanza di quel suo amore testardo da lasciarmi andar via ed a mio **Padre** e **Simona** che stanno costruendo insieme qualcosa di nuovo ma con uno spazio anche per me.

A **Giulia** che prova quotidianamente ad essere un'amica prima che una sorella.

A mia **Nonna** per essere quel “singolo tocco” di bontà, talvolta ingenua ma così pura da essere disarmante.

A mio **Nonno**, che noto sempre più nei piccoli dettagli.

Ai miei **Zii**, che sono entrambi un costante esempio di amore profondo verso il prossimo.

A **Bruno**, che fin da quando ho memoria significa per me più di quanto possa esprimere a parole.

Ad **Alessio**, il compagno di avventure e riflessioni che vorrei solo potesse vedere se stesso così come lo vedo io.

A **Carmelo**, che nonostante le tante vicissitudini trova il tempo di essere il mio bestman oggi come ogni giorno.

L'elenco potrebbe, e dovrebbe, continuare ma la fretta instillata dall'**uomo della cartoleria** mi concede solo altri trenta secondi, sta letteralmente aspettando la mail con la tesi, non avete idea.

Tutto ciò che posso dire con il tempo che mi resta è che voglio essere in grado di diventare un'espressione dell'immensa gratitudine che provo per ciascuno di voi. Grazie.