**ACKNOWLEDGEMENTS**

I would like to thank my advisor and professor Dr. Gopal Gupta for giving me an opportunity to work with him. It was because of his continuous guidance and support that I was able to complete my thesis. I would also like to thank Dr. Vincent Ng and Dr. Vibhav Gogate for helping me with resources and encouraging me by being on my supervisory committee.

The ALPS lab along with its members provided me with an ideal and a productive work environment. The group meetings really helped me shape my ideas and mold them into constructive work. I would also like to thank all my colleagues, Dr. Zhuo Chen, Farhad Shakerin, Elmer Salazar and Sarat Chandra Varanasi, for all the interesting brainstorming sessions and helpful suggestions.

I would like to thank my parents Rajan Pendharkar and Vinita Pendharkar for their encouragement and support towards my master’s education. Lastly, I am grateful to all my friends at the University without whom this ride wouldn’t have been so much fun.

**ABSTRACT**

Knowledge Representation and Reasoning (KRnR) is a field of Artificial Intelligence that deals with converting information into knowledge patterns in a form that the computer understands. It applies concepts from the field of psychology, about how humans make rational decisions, to build formal rules that model the human cognitive processes. Using the generated knowledge bases the computer is then able to solve complex tasks like question answering, summarization, automated reasoning, medical diagnosis and many more. Many of these complex tasks, mentioned above, requires an understanding of natural language text.

A vast amount of knowledge that we have today comes from books and is in the form of natural language text. Such knowledge is in an unstructured form and is not easily interpretable by computers. Knowledge representation approaches try to convert this unstructured textual knowledge into a format that is meaningful to the computer, thus opening the doors for more knowledge sources.

In this thesis, I propose using an answer set programming (ASP) approach for knowledge generation from natural language text and reasoning using ASP-solvers like SaSP and clasp. Here, I have also explored different ways of modelling common sense reasoning using default logic patterns, hierarchical knowledge patterns and negation as failure. The thesis uses question answering as a task to represent the effectiveness of the generated knowledge from text. The proposed system consists of a query engine that accepts natural language questions and converts them into answer set queries automatically, which can be used by the solvers for question answering. To make the knowledge base richer in information and the querying engine more robust, we make use of knowledge resources like WordNet. This approach has been tested on the SQuAD dataset and has proved to be a promising proof of concept.

**CHAPTER 1**

**INTRODUCTION**

* 1. **Overview**
  2. **Related Work**
  3. **Contributions**
  4. **Structure of the Thesis**

In this section, a layout of the remained of the chapters is provided with a summary for each of them.

Chapter 2 deals with Answer Set Programming

Chapter 3 deals with System Architecture

Chapter 4 deals with Software Development Approach

Chapter 5 deals with Natural Language Resource

Chapter 6 deals with Knowledge Representation

Chapter 7 deals with Query Generation

Chapter 8 deals with Question Answering

Chapter 9 deals with Conclusions

Chapter 10 deals with Appendix

**CHAPTER 2**

**ANSWER-SET PROGRAMMING**

**2.1 Overview**

The system that has been designed uses an ASP-based approach to represent knowledge from natural language text. So, a basic understanding of answer-set programming is required to understand the remainder of the thesis. This chapter introduces the answer-set programming paradigm and further elaborates on some of the important definitions, concepts and patterns used in answer-set programming. At the end of this chapter, we will go over some of the systems that are developed to run ASP programs.

**2.1 What is Answer-Set Programming (ASP)**

Answer-Set Programming is a declarative problem-solving paradigm that uses both non-monotonic reasoning and logic programming. It is widely used in automatically solving problems relating to representation and reasoning tasks such as modeling reasoning agents, non-monotonic inferences, common sense reasoning, modeling preferences and priorities and many more. An answer set program is a collection of statements that describe the objects of a domain and model relations between them. The semantics of an ASP Program defines a set of possible beliefs that an agent has associated with the program. This set of beliefs is called as an answer-set. The basic constituents of an ASP program are the rules, facts and constraints that describe the problem. Such a program is then passed onto an answer-set solver, which generates answer-sets to the program, that are used to obtain solutions to the problem.

**2.2 Syntax**

In this subsection, we introduce the syntax of an ASP program.

**2.2.1 Atom**

The most basic constituent of the ASP program is an atom. An atomic statement or an atom, is an expression of the form *p (t1,…, tn)* where *p* is a predicate symbol of arity *n* and *t1…tn* are n terms belonging to the predicate *p*. Here *n >= 0* and the terms *ti* can be integers or strings of letters, numbers, or underscore that either begin with an underscore or a lower-case letter. If in an atomic statement *n = 0*, then the brackets are omitted. As an example, ‘*parent (mary, alice)*’ and ‘*alice*’ are both atoms, whereas ‘*parent (mary, girl(alice))*’ is not an atom.

**2.2.2 Literal**

A literal is an atom of the form *p (t1,…, tn)* or its negation *-p (t1,…, tn)*. Here, *-p (t1,…, tn)* is referred to as a negative literal. It means that *p (t1,…, tn)* is false. An atom is called as a ground literal if every term *ti* in the atom is ground. For example, ‘*parent (X, Y)*’ is a literal whereas ‘*parent (mary, alice)*’ is called as a ground literal.

**2.2.3 Rule or Clause**

An ASP Program consists of a collection of rules of the form

1. l0
2. li 🡨 li+1, …, lm, not lm+1,…, not ln

Here, the symbol ‘not’ is a logical connective and is called as a default negation or negation as failure. Its semantic is discussed later in the chapter. An ASP rule is divided into two parts viz. head and a body. A head is a literal on the left side of the rule and a body is a set of literals on the right side of the rule. The head or the body in a rule can be empty. A rule with an empty head is called as a constraint whereas a rule with an empty body is called as a fact.

**2.3 Semantics**

Using the earlier mentioned syntax, we create an ASP program as a collection of rules, facts, and constraints. In this section, we shall discuss about the meaning of these rules and how they are interpreted while reasoning using these rules. The following are a few semantic patterns commonly used in answer-set programs.

**2.3.1 Modelling Implication**

As we saw earlier, every rule (excluding facts) in ASP has two parts separated by the consequence operator “:-”. In such a rule the head of the rule is said to succeed only if every literal in the body of the rule succeeds. As an example, consider the rule

*p :- q, r.*

we can read this kind of rule as “if *q* and *r* succeed then *p* succeeds”. Such kind of a pattern is commonly used in ASP programs to show implications.

**2.3.2 Classical Negation**

Classical Negation is a pattern in which we use negative literals, to show the fact that the literal under consideration has been proved to be false. As an example, consider the following rule.

*-p(a) :- q(a)*

The above rule states that if *q(a)* is shown to succeed then *p(a)* is false or *-p(a)* is true. Classical negation is one of the ways to represent negations in ASP programs.

**2.3.3 Epistemic Disjunction**

We model epistemic disjunctions in ASP when we need to model the semantics for the statement, “Either *p(a)* succeeds or *q(a)* succeeds”. Epistemic disjunction is different from exclusive or, where both *p(a)* and *q(a)* might succeed at the same time. Thus, to model epistemic disjunction we can make use of even loops in the following manner.

*p(a) :- not q(a).*

*q(a) :- not p(a).*

If we solve the above ASP program using an answer-set solver we will get two answer sets {*p(a)*} and {*q(a)*}, i.e. either *p(a)* succeeds or *q(a)* succeeds.

**2.3.4 Constraints**

Constraints are applied in places where we know that certain rules are always false and should not be part of the answer-set. As an example, if we know that it is impossible for *p(a)* to succeed then we can model this constraint as follows

*:- p(a).*

The above rule states that *p(a)* is always false. Here we see that a constraint limits the sets of beliefs that an agent has but does not help to derive new information.

**2.3.5 Default Negation or Negation as Failure (NAF)**

Default Negation, also called as Negation as Failure is used to make conclusions based on the absence of information. This type of negation is used to conclude about default rules and assume defaults to be true in case of absence of enough information. As an example, consider the following example where we state that if we are not able to prove that *q(a)* succeeds then *p(a)* succeeds.

*p(a) :- not q(a).*

So, in the above rule we assumed that *p(a)* has succeeded based on the absence of information about *q(a)*. NAF is an important tool to model defaults in ASP programs. Negation as Failure assumes closed-world assumption (CWA), in which we assume, what is not currently known to be true, as false.

**2.4 Default Reasoning**

Default Reasoning or Representing Defaults is one of the advantages of using ASP. The concept of closed-world assumption discussed earlier is an example of default reasoning where we default the value of the literal to fail in the absence of the literal in the answer set. Default reasoning is very useful in modelling human reasoning as we can draw conclusions even in the absence of information by defaulting to the default rule. Default reasoning thus plays an important role in common sense reasoning and understanding. In case of ASP, a default d stated as “Normally elements of class C have property P” is represented as the following rule

*p(X) :- c(X),*

*not ab(d(X)),*

*not -p(X).*

Here, *ab(d(X))* can be read as “X is abnormal with respect to the default assumption d” and *not-p(X)* can be read as “We can’t successfully prove that *p(X)* is false” or “*p(X)* may be true”.

Default reasoning uses two kinds of exceptions viz Strong exceptions and weak exceptions. Weak exception makes the default inapplicable and stop the agent from making a default conclusion. For example, in the above-mentioned default rule we can apply a weak exception *e(X)* by adding the following rule to the program

*ab(d(X)) :- not -e(X).*

The exception states that X may not be applicable to d if *e(X)* may be true. Similarly, Strong Exceptions refute the defaults conclusion by allowing the agent to derive the opposite to be true. This can be demonstrated by adding the following rule to the program

*-p(X) :- e(X).*

The above rule states that *p(X)* is false if *e(X)* succeeds, which allows us to defeat d’s conclusion that normally class C elements have the property P.

**CHAPTER 3**

**SYSTEM ARCHITECTURE**

**3.1 Overview**

This chapter mainly focuses on the various parts of the system and how they interact with each other. It also describes the various sub-components and modules used in generating the knowledge base and goes through the various steps required to answer queries with the help of the generated ASP program.

**3.2 System Architecture**

The system is composed of two main components or sub systems viz. the Knowledge Generation System and the Query Generation System. Both these systems function independent of each other. The architecture comprises of a common resource framework that is shared by both these systems. This chapter will describe all these components in detail in the rest of the chapter.

**3.3 Components of the System**



As illustrated in the figure, the Knowledge Generation System, the Query Generation System, and the Common Resource Framework are the three components of the architecture. The Common Resource Framework consists of Natural Language Processing tools such as Stanford Core NLP Tools, WordNet API as well as modules for preprocessing incoming text. The Knowledge Generation System is mainly responsible for extracting knowledge from a natural language text. For extracting the knowledge from text, this component uses Stanford NLP tools like the POS Tagger, Stanford Dependency Parser, and the Stanford NER Tagger to gain more information about the input text. Apart from these resources it also taps into the vast information that is provided by WordNet and tries to extract information from the same. As currently there are a very few digital resources about verbs in the NLP domain, this component provides a flexible way to add custom information about verbs that would be reusable in many scenarios. Thus, the Knowledge Generation System takes in the natural language passage as input and produces rules in the form of three chunks of information, which can be aggregated together to form an ASP program representing all the extractable knowledge from the source text.

To help answer questions posed in Natural Language, the Query Generation System is used to automatically generate a set of queries that can be used to find solutions from the answer-sets generated by the ASP program. To ask queries to the ASP program we need to provide both the queries as well as the ASP program to an Answer-Set Solver like SaSP or Clasp. The Query Generation System generates multiple queries for a question and arranges them in the order of significance, keeping the more constraint queries before the less constraint ones. Hence, the kind of query that would lead to an answer is also a rough metric as to the quality of the answer. Now let’s dive deep into the various components in the architecture and talk about its sub modules and their interactions.

**3.4 Common Resources Framework**

The Common Resources Framework consists of the following modules as illustrated in the diagram.



**3.4.1 Text Preprocessing Module**

The style of writing in natural language text changes based on the domain, author, title of the text and many other factors. To automate text processing, becomes a very hard task when we must consider all these different writing styles. Thus, for this system we assume certain properties about the incoming natural language text. The text pre-processing module is the first module that the input text passes through and it makes sure that the input text conforms to these assumptions. Some of the assumptions that we make about the incoming text include concatenation of compound nouns and resolution of coreferences. National Aeronautics and Space Administration or NASA is an excellent example of a compound noun. In this example, we assume that the system detects and treats NASA to be a single concept (National\_Aeronautics\_and\_Space\_Administration) as opposed to separate words. Coreference resolution is the task of finding all expressions that refer to the same entity in a text. It plays an important role in higher level NLP tasks and so we assume that the coreferences in the incoming text have already been resolved. Many a times due to informal writing styles, humans miss certain words or assume certain words while reading and writing texts. Working with such informal style of English is hard, so we assume that the incoming text is written in formal English. As this system depends on many NLP tools for semantic resources, it is susceptible to any flaws in these tools. The preprocessing module tries to correct any mis-tagged entries in the text, due to ambiguous wording, with the help of relations given by higher level semantic tools.

**3.4.2 Stanford NLP Core Tools**

Stanford Core NLP Tools is a set of linguistic tools that help in analyzing and understanding natural language text. It consists of a lot of different sub tools that can be pipelined one after the other to analyze a piece of text. It provides solutions to NLP tasks like POS Tagging, Parsing, NER Tagging, Coreference resolution and many more that play a vital role in higher level NLP tasks like text understanding. This system uses the Stanford-Core-NLP version 3.9.1 on the Java Platform and makes extensive use of its POS Tagger, NER Tagger, Stanford Dependency Parsing, and some other tools on the framework to process incoming text.

**3.4.3 WordNet Interface**

One of the important things in text understanding is being able to extract more information about concepts in the passage. This helps the system gain a deeper understanding into a concept. WordNet is one such digital resource that helps in gaining more knowledge about a concept. WordNet is a large lexical database of English. It consists of a large number of concepts grouped into sets of words that are synonyms i.e. synsets. WordNet has thus created a huge network of concepts by linking these synsets based on lexical relations and conceptual-semantics. WordNet's structure makes it a useful tool for computational linguistics and natural language processing. Java WordNet Interface or JWI is a Java library for interfacing with Wordnet created at MIT. With the help of JWI this system interfaces with WordNet and extracts semantic relations like hypernyms, hyponyms, meronyms etc. to gain more information on the passage.

**3.5 Knowledge Generation System**

The Knowledge Generation System deals with the generation of rules from text and extracting information from other sources like WordNet. This system is made up of 3 modules which are described as follows



**3.5.1 Knowledge Extraction from Text**

This module is responsible for generating rules and facts from the passage itself. It uses the various rules and patterns mentioned in Chapter 7 to generate part of the ASP program. The ASP rules generated by this module should contain all the information present in the input passage. The input file that is supplied to this module is assumed to be preprocessed according to the earlier mentioned assumptions.

**3.5.2 WordNet Ontology Generation**

To further understand the concepts mentioned in the input passage, the ontology generation module generates rules regarding disambiguation, hypernyms, and meronyms. Word sense disambiguation forms an integral part of NLP and is also tackled by this module using default reasoning. The Hypernym relation and the Meronym relation along with other relations like Antonyms and Synonyms help capture more semantic information into the ASP program. These topics would be touched upon in detail in the forthcoming chapters.

**3.5.3 Default Knowledge Base**

As mentioned earlier, there is very little digitalized information about the semantics of verbs in the NLP domain. Hence, to gain complete understanding of verbs and their usage, it is required to create rules, describing their complete meaning, manually. The default knowledge base makes it feasible to add knowledge about verbs and nouns by hand. Care must be taken to make sure that the knowledge being added is generic in nature and is reusable for other similar scenarios. With the help of such an increasing knowledge base the system can become more efficient and accurate.

**3.6 Query Generation System**



The Query Generation System is responsible for understanding the question asked in natural language text and converting it into a set of ranked queries, that could be understood by the ASP Solver to answer the question. It is comprised of the following 2 modules.

**3.6.1 Query Understanding**

Questions asked in natural language can be classified into multiple types based on various theories. To classify a question into a specific type requires complete understanding of the question along with the type of answer expected by the question. This module is tasked at finding the various components of the question including the kind of question, based on the ‘Wh’ word and the lexical type and kind of answer expected.

**3.6.2 Query Generation**

Using the information provided by the query understanding module, the query generation module first creates the most constraint query applicable for the question under consideration. This module then starts relaxing certain constraints in the query giving rise to lower quality queries or queries with lower confidence. In the later chapters, we discuss this approach in detail.

**CHAPTER 4**

**SOFTWARE DEVELOPMENT APPROACH**

**4.1 Overview**

This chapter describes the software development approach taken to build the system. I found out that Test Driven Development is one of the best software development approaches for building a rule-based system. This chapter will elaborate on the various stages involved in test driven development and how its principles help in building a stable rule-based system.

**4.2 Test Driven Development**

Using this approach, a paragraph was divided into multiple tests having a sentence each which was used to develop code for the Knowledge Generation System. Each sentence in the paragraph contributed to my understanding of the grammar rules and patterns in the input text. Test Driven Development, abbreviated as TDD, is a software development process that relies on the repetition of a very short development cycle: first the developer writes an automated test case that defines a desired improvement or new function, then produces a minimum amount of code to pass that test and finally refactors the new code to acceptable standards.

Test Driven Development is known to encourage simple designs that inspires confidence in the code developed under the technique. It is related to the test-first programming concepts of extreme programming. Programmers also apply it to improving and debugging legacy code developed with older techniques. A graphical representation of the typical development cycle can be shown as follows.



**4.3 Steps involved in TDD**

As shown in the above development cycle, there are 5 major stages in a single development cycle of the approach. A cycle in TDD follows the below mentioned steps in sequence.

**4.3.1 Add a test**

In test driven development, each new feature begins with writing a test. This test is supposed to fail as it is written before the feature has been implemented. If the test does not fail, then either the proposed feature is already implemented, or the test is defective. To write such a test the developer must completely understand the specifications and the requirements of the new feature. A developer can accomplish this through use cases and user stories to cover the requirements and exception conditions and can write the test in whatever testing framework is appropriate to the software environment. It could also be a modification of an already existing test case. This is a differentiating feature of test driven development which makes the user focus on the requirements before writing any code.

**4.3.2 Run all tests and see if the new test fails**

This is an important step in TDD, which makes sure that the test harness is working correctly, and the new test does not accidentally pass without requiring writing new code. This step also rules out the possibility that the new test always passes thus making the new test useless. Another important factor in this step is that the test should fail for the right reasons. This makes sure that the test is testing the intended condition and only passes when those conditions are met.

**4.3.3 Write some code**

The next step in this process is to write some code to pass this newly added test. The code written in this step may not be efficient and may pass the test inelegantly. This is acceptable, here as we are going to improve and hone the design in later stages. It is important to note here that the newly added code should only be designed to pass the current test, and no further functionality should be assumed or predicted.

**4.3.4 Run the automated tests all see them succeed**

Once all the automated tests pass with the inclusion of the newly added test and its corresponding code then the developer can be confident that the code meets all the current testing requirements. This is a good point from which we can begin the final step of the development cycle.

**4.3.5 Refactor code**

The entire code can now be refactored to accommodate any new updates and changes required in the design triggered by the addition of new code. By running the entire testing suite, the developer can guarantee that the refactoring has not hampered any functionality in the code. Removing duplicate and dead code is important in software development. This step gives the developer a chance to remove duplicates and improve the code design without affecting code.

**4.3.6 Repeat**

Now, the developer can start with another test case and repeat the cycle to improve the functionality of the system. It is recommended that the size of the edit should always be small, with as few as 1 to 10 edits between each test run. If the new code does not rapidly satisfy a new test or other tests start failing unexpectedly then it is recommended to undo the previous code change as compared to excessive debugging. In such cases, continuous integration helps by providing revertible checkpoints.

**4.4 Development Principles**

There are various principles that help while using test-driven development. These principles include “Keep it simple stupid” (KISS), “You are not going to need it” (YAGNI) and many more. Some of these principles along with their advantages has been discussed below.

By focusing on only writing code for passing a single test, the designs can be cleaner than other approaches. To achieve complex design patterns, tests can be written to generate the design pattern. This helps in keeping the changes short and simple to understand, which allows the developer to focus on what is important. Writing tests first before coding up the functionality has been claimed to have many benefits. It helps the developers think about testing from the outset instead of worrying about it later. Also, writing tests first creates a deeper understanding about the concept or feature in the developers mind which in turn helps in writing better code. Failing the test case first, before implementing its required feature, ensures that the test really works and can catch bugs. Test driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage in the development cycle reinforces the developers mental model of the code and boosts confidence in the code. Keeping the changes to be smaller has proved to have multiple benefits including reduced debugging effort and better understanding of code. These principles if followed correctly enable developers to build large scalable systems without adding to the complexity of debugging and maintaining the system.

**CHAPTER 5**

**NATURAL LANGUAGE RESOURCES**

**5.1 Overview**

This chapter gives an overview on the various natural language resources that are used to build the system. The Stanford Core NLP Toolset consists of many tools including the POS Tagger, Parser, Co-Reference Resolution, and Dependency Parser. This chapter would mainly deal with the details about the dependency parser, POS tagger, NER Tagger and the concepts relating to WordNet and its relations. We will make use of the concepts discussed in this chapter in the next chapter while discussing the different knowledge extraction techniques used.

**5.2 Resource Tools**

**5.2.1 Dependency Parser**

A dependency parser analyses the grammatical structure of a sentence and returns a set of relations between different words of the sentence. In general, one of these words is the independent word or the head word and the other is the dependent word in the relation. The dependent word modifies the independent word in the sentence using the relation. Consider the sentence “John gave Mary the book.”.



The figure shown above marks the various dependencies in the sentence. The dependencies can be given as follows

root(ROOT-0, gave-2), nsubj(gave-2, John-1), dobj(gave-2, book-5), iobj(gave-2, Mary-3), det(book-5, the-4), punct(gave-4, .-6).

We will discuss each of these dependencies and their meanings in detail later in the chapter. In the above-mentioned dependencies, the first word is the independent word, the second word is the dependent word and the predicate of the dependency describes the type of relation between the words. Consider the nominal subject dependency relation from the sentence

nsubj(gave-2, John-1)

Here, “gave” is the independent word, “John” is the dependent word and the relationship between these words is of the “nominal subject” or “nsubj”. Sometimes these relations have specifics mentioned along with them e.g the relation “nmod:poss” states that the relation is of a nominal modifier which shows a possessive relation.

**5.2.2 Part of Speech Tagger**

A Parts of Speech Tagger is responsible for assigning parts of speech to words in a sentence. The English language has eight parts of speech: noun, verb, pronoun, preposition, adverb, conjunction, particle, and article. Apart from these parts of speech categories tags are also applied to punctuations in a sentence. A tagging module uses certain predefined tag sets to tag various words. A tag set defines the various tags and their meanings that the POS Tagger outputs. The English tagger in the Stanford POS Tagger uses the Penn Treebank tag set.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Tag*** | ***Description*** | ***Tag*** | ***Description*** |
| CC | Coordinating conjunction | PRP$ | Possessive pronoun |
| CD | Cardinal number | RB | Adverb |
| DT | Determiner | RBR | Adverb, comparative |
| EX | Existential *there* | RBS | Adverb, superlative |
| FW | Foreign word | RP | Particle |
| IN | Preposition or subordinating conjunction | SYM | Symbol |
| JJ | Adjective | TO | to |
| JJR | Adjective, comparative | UH | Interjection |
| JJS | Adjective, superlative | VB | Verb, base form |
| LS | List item marker | VBD | Verb, past tense |
| MD | Modal | VBG | Verb, gerund or present participle |
| NN | Noun, singular or mass | VBN | Verb, past participle |
| NNS | Noun, plural | VBP | Verb, non-3rd person singular present |
| NNP | Proper noun, singular | VBZ | Verb, 3rd person singular present |
| NNPS | Proper noun, plural | WDT | Wh-determiner |
| PDT | Predeterminer | WP | Wh-pronoun |
| POS | Possessive ending | WP$ | Possessive wh-pronoun |
| PRP | Personal pronoun | WRB | Wh-adverb |

Let us take the previously mentioned example as a sentence and tags its parts of speech. You can understand the meaning of each tag from the above given table.



In general, many of the most common words used in English belong to more than one category of part of speech. As an example, the “book” can both be a verb and a noun. Thus, the task of parts of speech tagging is also required to disambiguate between the various possible tags that can be applied to a word. Following are some of the major problems that most taggers face, which affect tasks like information extraction that use these taggers as a source of information.

1. *Confusion between NN/NNP/JJ*

Proper Nouns, Nouns and Adjectives are predominantly hard to distinguish between as all of them form parts of the Nominal Phrase and can be reordered in multiple ways in English.

1. *Confusion between RP/RB/IN*

All the above parts of speech can occur immediately after the verb. It is especially hard to distinguish between particles and prepositions as both classes share certain words.

1. *Confusion between VBD/VBN/JJ*

Boundaries of noun phrases are determined using the above parts of speech and hence differentiating between them plays a crucial role in parsing.

**5.2.3 Named Entity Recognizer**

A Named Entity Recognizer is a module used to label a sequence of words in a sentence with predefined tags of Named Entities. Named Entities are names of things, such as person, company, organization, locations, cities and many more. Named Entity Taggers can be built for custom texts and passages with a rich predefined tag set. The Stanford Named Entity Tagger is trained on various models for the English Language. The various models are given as follows:

1. 3 class: *LOCATION*, *PERSON*, *ORGANIZATION*
2. 4 class: *LOCATION*, *PERSON*, *ORGANIZATION, MISC*
3. 7 class: *LOCATION*, *PERSON*, *ORGANIZATION, MONEY, PERCENT, DATE, TIME*

Let us take the sentence “John, who works at UTD, lives in Dallas.” as an example for the NER



The Named Entity Recognizer is one of the important sources of information for the information extraction task in NLP.

**5.3 Stanford Universal Dependencies**

The Universal Dependency (UD) Relation taxonomy divides the relations into structural categories as Nominals, Clauses, Modifier Words and Function Words. Similarly, the relations can be divided into functional categories with relation to the head as Core Arguments, Non-Core Dependents, and Nominal Dependents. We will study the meaning of some important dependency relations based on the functional categorization.

**5.3.1 Core Arguments**

Core Arguments are mostly relations that talk about the various participants that are involved in the event. These relations play an important role in understanding the sentence. Some of these core argument relations are given as follows:

5.3.1.1 *nsubj: nominal subject*

A nominal subject is a nominal that acts as the subject or agent of a clause. This nominal contains the do-er of the action. The head of such a nominal could be a noun, pronoun, or other things such as adjectives. Similarly, if the verb involved in the *nsubj* relation is a copula then the governor of the relation may not be a verb but now is a noun or an adjective related with the copula. In the case of passive sentences the relation *nsubj:pass* is used to indicate the subject instead.







5.3.1.2 *dobj: direct object*

A direct object is a noun phrase or a nominal that is the accusative object of the verb. The verb, here is the head of the verb phrase that governs this relation.



5.3.1.3 *iobj: indirect object*

A nominal phrase that is neither the subject nor the direct object of the verb but is a core argument of the verb is called the indirect object of the verb. In general, if there is only one object then its considered to be a direct object, but in the case of more than one objects, one of them is direct whereas all other are indirect objects of the verb.



5.3.1.4 *csubj: clausal subject*

A clausal subject is basically a clause that acts as the subject of the sentence. In general, the governor of this relation is a verb but, in the case, where the main verb is a copular verb the governor can be a non-verb. The dependent word for a *csubj* relation is the head verb of the subject clause. The *csubj* relation is also used for the passive verb or a verb group. We make use of the specific *csubj:pass* in passive transformations.







5.3.1.5 *ccomp: clausal complement*

The dependent clause which is the core argument of the verb is called as the clausal complement. Thus, such a clause behaves like a subordinate clause to the main verb i.e. the governor of this relation.



5.3.1.6 *xcomp: open clausal complement*

A clausal complement that does not contain its own subject is called as the open clausal complement. The subject is usually determined by a higher clause in the sentence.



**5.3.2 Non-Core Dependents**

5.3.2.1 *advcl: adverbial clause*

An adverbial clause is a clause that modifies a verb. An adverbial clause can include a temporal clause, condition, consequence, effect, purpose etc. Here the dependent entity must be a clause otherwise the relation becomes *advmod*. The dependent entity represents the head verb of the clause.



5.3.2.2 *advmod: adverbial modifier*

An adverbial modifier is an adverb or an adverbial phrase that modifies a predicate. Here, the adverbial phrase should not be a clause, otherwise it is marked as *advcl*. Here in general the dependent is an adverb whereas the governor is the head verb of the predicate that the adverb is modifying.



5.3.2.3 *aux: auxiliary*

An auxiliary is a function word in a clause that expresses categories like mood, tense, aspect, voice and evidentiality, Auxiliaries are often verbs, that may or may not have non-auxiliary uses, but many languages also have non-verbal auxiliaries. In general, auxiliaries used to create passive voice are also marked with the *aux* relation.



5.3.2.4 *cop: copula*

A copula relation is used to connect a subject to a non-verbal predicate. Although, copulas are verbs in general, there exist non-verbal copulas in certain languages. The copula “be” is not considered as the head of the clause but is a non-verbal predicate. A copula relation is not applied when the non-verbal predicate is used in the form of a clause.





5.3.2.5 *mark: marker*

The marker is a word that connects the subordinate clause to another clause. For a clausal complement a marker is a word like that or whether in English. Whereas for an adverbial clause the marker is a subordinating conjunction. The mark relation has the governor as the subordinate clause head and the dependent is the marker itself.



**5.3.3 Nominal Dependents**

5.3.3.1 *nmod: nominal modifier*

The nominal modifier relations describe different attributes or properties of verbs, adverbs, nouns. The dependents in such a relation are nouns or noun phrases and the governors can be verbs, nouns or other parts of speech. Nominal modifiers can have specifics like *nmod:poss*, describing possessive relations or *nmod:tmod*, describing temporal relations.





5.3.3.2 *appos: appositional modifier*

Appositional Modifiers are nominals that directly follow the noun that defines, modifies, or describes the noun*.* Appositional modifiers also include parenthesized example as well as abbreviations.





5.3.3.3 *nummod: numerical modifier*

Any number or a number phrase that describes more about a noun or a noun phrase in the form of a quantity is part of the numerical modifier relation. Here, indefinite quantifiers such as few, many, a lot etc. are marked as *det* and not *nummod*.



5.3.3.4 *acl: adjective clause*

Clauses that modify a nominal are marked as adjective clauses or *acl*. These are different from *advcl*, which modifies a predicate. The head of the relation is the nominal that is modified and the dependent of the relation is the head of the clause that modifies the relation.



5.3.3.5 *amod: adjective modifier*

An *amod* or adjective modifier is an adjective or an adjectival phrase that describes more about the noun or gives meaning to the noun. Here the head of the relation is the modified noun where as the dependent of the relation is the modifier.



5.3.3.6 *det: determiner*

The determiner relation holds between a nominal and its determiner. Generally, the words having the POS tag as DET belong to this relation. The head is the nominal and the dependent is the determiner in this relation.



5.3.3.7 *case: case marker*

The case relation is used to mark any element treated as a separate syntactic word like a preposition, a possessive altercation etc. Here the governor of the relation is the head of the nominal phrase where as the dependent is the syntactic word under consideration.





**5.3.4 Other Dependency Relations**

5.3.4.1 *conj: conjunct*

The conjunct relation is marked between words or elements the are connected via a coordinating conjunction like *and*, *or* etc. The conjunct relation is asymmetrical, here the head of the relation is first conjunction and the dependents are the other conjunctions that depend on the first conjunction through the *conj* relation. Such a relation is also marked if the conjunctions are omitted or replaced by commas or any other punctuation symbols.





5.3.4.2 *cc: coordinating conjunction*

The relation *cc* is marked between the conjunct and a preceding coordinating conjunction. The *cc* relation accompanies the *conj* relation mentioned before.



5.3.4.3 *compound: compound*

The compound relation is used for multiword expressions like joint ventures. These can also be applied to proper nouns like American Airlines and verbs that have particles like stand up. For particles the compound relation contains a specific and is marked as *compound:prt*.



5.3.4.4 *root: root*

The root relation marks the root of the sentence. A fake node ROOT-0 is created which governs the relation and the dependent is the head of the sentence, which in most cases is the main verb. The ROOT node starts at 0 as the rest of the sentence is indexed from 1.



**5.4 WordNet**

WordNet is one of the most commonly used resources in English. It is a lexical database consisting of sense relations between English words. WordNet consists of separate databases one each for nouns, verbs and a database for adjectives and adverbs. Databases are only created for open word classes in WordNet. WordNet contains a set of near synonyms called as synsets and marks relations between these synsets. We will discuss some of the important relations, present in WordNet, later in the chapter. In each of the databases, there consists a set of lemmas, each one annotated with a set of senses. WordNet can be accessed on the Web or downloaded and accessed locally. A typical entry for the noun “*lion*” in WordNet yields the following different senses.



Due to the presence of multiple senses for each word in WordNet, disambiguation is of the utmost importance when using WordNet as a source for any application.

**5.4.1 WordNet Relations**

WordNet contains various relations depending upon the word type. Some of the noun relations include Hypernym, Hyponym, Instance Hypernym, Instance Hyponym, Meronyms, and many more. Some of the verb relations include Hypernyms, Troponyms, Antonyms etc. We now discuss some of the above-mentioned category relations.

**5.4.1.1 Hypernyms**

One of the most commonly used WordNet relations is the Hypernym relation. It connects specific entities to their more general entities. This relation is also called as the IS\_A relation. This relation states that the category *vehicle* includes the *motor vehicle* which in turn includes the *car*. In this way all noun hierarchies eventually end in the root node, *entity*. The Hypernym relation is transitive in nature i.e. if a *car* is a *motor vehicle* and the *motor vehicle* is a *vehicle*, then we can conclude that the *car* is also a *vehicle*. WordNet can also distinguish between types of nouns versus instances of nouns. As an example, a car is a type of a vehicle, whereas Kenya is an instance of a country.

**5.4.1.2 Meronyms**

Meronyms indicate the part-whole relationship between any two concepts in WordNet. For example, A *horn*, *air bags*, and an *engine* are parts of a *car*. This relationship can be inherited from the super ordinates but not from the subordinates as some of the relations may be characteristic to certain concepts. As an example, all cars will have horns, but not all vehicles (submarines) will necessarily have a horn.

**5.4.1.3 Synonyms**

Synonyms are two different concepts having similar or nearly identical senses. Two words can also be said to be synonymous if they are substitutable for each other in the sentence. The synonym relation can be found in nouns, adjectives, adverbs as well as verb categories. For example, *car/automobile*, *eat/consume/take in*, *pretty/beautiful*, *quickly/rapidly* are all synonyms of each other.

**5.4.1.4 Antonyms**

Antonyms on the contrary to Synonyms are concepts or words with opposite meaning. Antonyms have various definitions, which make it hard to define antonyms. Antonyms are concepts that may be on the opposite end of a scale or a measurement i.e. *long/short*, *fast/slow*, or *positive/negative* which are all concepts that lie on the opposite sides of a scale. Another definition of antonyms describes some change in the direction or movement in opposite direction which can be given by *up/down, left/right*, and so on. Antonyms are somewhat like synonyms, as antonyms almost have similar meanings in all aspects except for the fact that they belong to the opposite sides on a scale. Thus, due to the cryptic definitions of antonyms it is often difficult to decide between synonyms and antonyms.

**5.4.2 WordNet Senses**

WordNet has defined 45 lexical categories for synsets during its development. Synsets were organized into these categories based on the syntactic category and logical groupings of the synsets. The syntactic categories based on which the synsets are divided are NOUN, VERB, ADJECTIVE and ADVERB. Let us consider some of the sense categories for nouns as generated by WordNet. With the help of these sense categories we will later discuss how we can generate ASP code to represent hypernym relations efficiently without blowing up the space requirements for the system. Following are the lexical categories for nouns along with their meaning.

|  |  |
| --- | --- |
| ***Name*** | ***Content*** |
| noun.Tops | unique beginner for nouns |
| noun.act | nouns denoting acts or actions |
| noun.animal | nouns denoting animals |
| noun.artifact | nouns denoting man-made objects |
| noun.attribute | nouns denoting attributes of people and objects |
| noun.body | nouns denoting body parts |
| noun.cognition | nouns denoting cognitive processes and contents |
| noun.communication | nouns denoting communicative processes and contents |
| noun.event | nouns denoting natural events |
| noun.feeling | nouns denoting feelings and emotions |
| noun.food | nouns denoting foods and drinks |
| noun.group | nouns denoting groupings of people or objects |
| noun.location | nouns denoting spatial position |
| noun.motive | nouns denoting goals |
| noun.object | nouns denoting natural objects (not man-made) |
| noun.person | nouns denoting people |
| noun.phenomenon | nouns denoting natural phenomena |
| noun.plant | nouns denoting plants |
| noun.possession | nouns denoting possession and transfer of possession |
| noun.process | nouns denoting natural processes |
| noun.quantity | nouns denoting quantities and units of measure |
| noun.relation | nouns denoting relations between people or things or ideas |
| noun.shape | nouns denoting two and three-dimensional shapes |
| noun.state | nouns denoting stable states of affairs |
| noun.substance | nouns denoting substances |
| noun.time | nouns denoting time and temporal relations |

**CHAPTER 6**

**KNOWLEDGE REPRESENTATION**

**6.1 Overview**

This chapter describes how using all the concepts mentioned in the previous chapter, the system generates facts and rules from text. This process involves creation of a semantic graph which acts like an event tree that binds various events in the text together. Using this semantic graph, we start generating fact in the form of some predefined predicates. Section 8.3 defines these predicates with examples. Apart from the knowledge that the system extracts from the text, it also makes use of WordNet and creates certain rules from it. Section 8.4 dives deep into how we can make use of the various WordNet synsets to understand text and how we can use ASPs preference patterns for sense disambiguation while using these synsets.

**6.2 Semantic and Event Graph**

One of the first steps after parsing the sentences in the text is to generate a semantic graph from the sentence. We create such a graph with the help of the Stanford Typed Dependencies representation mentioned in the earlier chapter. It gives a simple description of the grammatical relationships in a sentence that can easily be understood and effectively used to extract textual relations. Consider the following sentence as an example

*Example:* “NASA carried out the Apollo program.”

For this sentence some of the Stanford Dependency (SD) representations can be given as:

*nsubj* (carried-2, NASA-1) *root* (ROOT-0, carried-2)

*compound:prt* (carried-2, out-3) *det* (program-6, the-4)

*compound* (program-6, Apollo-5) *dobj* (carried-2, program-6)



These dependencies map straightforwardly onto a directed graph representation, in which words in the sentence are nodes in the graph and grammatical relations are edge labels. The Figure gives the graph representation for the example sentence above. The meanings of these type dependencies are explained in the previous chapter in detail. This semantic graph thus created has been used to extract the different facts from the sentence. In English, most event mentions correspond to verbs and most verbs are triggers to events. Although this is true in most cases there are other word groups that can trigger events as well. The different verbs in the sentence thus, define various events that take place in the sentence and how these events are connected to each other. In the above sentence there is only one event having the head verb “carried out”. We can demonstrate the use of the event graph by taking a more complicated example as follows.

*Example:* “Miitomo, which Nintendo introduced globally in 2016, features the company's, Mii, avatar-system and lets the users communicate by exchanging personal information such as favorite movies.”



The above graph shows how various words are connected to each other in the sentence. This example is a complex sentence consisting of multiple clauses. The main verbs of the sentence being “feature” and “let” connected by a coordinating conjunction. In the above diagram the verbs have been marked using double-bordered ovals, the prepositions have been shown using diamonds whereas all other words like adjectives, nouns and adverbs have be shown using rectangular boxes. The verbs in the diagram represent the head of events in the sentence and are marked using event IDs. The various color regions denote the rough boundaries of these event regions. Many of the concepts in the sentence also take part in multiple events, for example the word, Mittomo. It is with the help of these event regions that we can infer missing or implied information from the sentence. We will go over the details of how the semantic graph with these event regions are used to create facts and rules in the rest of this section.

**6.3 Predicate Generation**

In this section, we would go over the various predicates that have been defined to represent information. Some of the predicates are special cases like abbreviation, start\_time etc. whereas others are more generic like mod, event and so on. The more generic predicates we generate the easier it is to query about information from the ASP program. The generic predicates that have been generated try to convey information that is more explicitly present in the text. Whereas the special kinds of predicates can be used to model implied information in text. Implied reasoning forms an integral part of the human or common-sense reasoning, thus we can add more special predicates for better performance. Following are some of the important predicates that have been used by the system.

**6.3.1 Event Predicate**

The event predicate defines an event that happens in the sentence. The verb marks the head of the event predicate. Any event in general consists of certain actors and participants taking part in the event. The event predicate consists of the various actors (doers) and participants involved in the event. The signature of the event predicate can be given as follows

*event* (*event\_id*, *trigger\_verb*, *actor*, *participant*)

The signature of the event predicate consists of 4 terms. The *event\_id* is an integer that uniquely identifies that event in the sentence or paragraph. It is using this *event\_id* that we can differentiate various properties and relations in the text. The *event\_id* is tightly coupled with the events *trigger\_verb*. The *trigger\_verb* denoted in the event predicate is the lemma of the actual word used in the sentence. Using lemmas of trigger verbs help in easily querying for an event. Event predicates are created for all kinds of events i.e. both active versus passive events. Active events are those that are trigged by action verbs whereas events triggered by passive verbs like copulas or modal verbs are more passive in nature. The actors in the event predicate are the subjects found to the *trigger\_verb* in the sentence. Subjects in the sentence can be found with the help of dependencies like *nsubj and nsubj:xsubj*. Here *nsubj:xsubj* is a special controlling subject used to find relations between the main subject and the subordinate clause. Just as actors can be obtained from the subject of the sentence, the participants can be determined from the direct object dependency (*dobj*). Consider the following sentence from a ‘Super Bowl 50’ passage as an example along with the event predicates that are generated from it.

*Example:* “The American\_Football\_Conference's (AFC) champion team, Denver\_Broncos, defeated the National\_Football\_Conference's (NFC) champion team, Carolina\_Panthers, by 24\_10 to earn AFC third Super\_Bowl title”

*event (1, defeat, denver\_broncos, carolina\_panthers)*

*event (2, earn, afc, title)*

From the above predicates, we can understand that there are two main events occurring in the sentence, i.e. the first relating to a defeat and the other relating to earning of something. The system has marked the event “defeat” with ID 1, whereas the event of earning has been marked with the ID 2. This is how we can capture the information that ‘denver\_broncos defeated carolina\_panthers’ and ‘afc earned the title’. Here we can get richer information about the event if we know more about the actors and the participants in the event. Thus, duplicate event predicates are also created for each *modifier* for the actor as well as the participants involved in the event. To generate such event predicates, we can use the *amod* or *nummod* dependencies for the actors and the participants and create compound atoms from the modifiers and their governors. Such supplementary event predicates, although are duplicates but make querying easier. As an example, consider the same sentence as before and this duplicate event predicate that is generating shedding light upon the kind of title won.

*event (2, earn, afc, third\_super\_bowl\_title)*

The default value for the actor and the participant field is *null*. A *null* value indicates that either the term is absent for the event or the system was not able to determine it.

**6.3.2 Property Predicate**

The property predicate elaborates on the properties of the modified noun or verb. The modifier in this case is generally a prepositional phrase in the sentence. Here, the *modifier* is different from the modifiers that are encountered as adjectives or adverbs that are marked as *amod* and *advmod* respectively. A property predicate is coupled with an event and describes the modification only for that event. The signature for the property predicate can be given as follows

*\_property (event\_id, modified\_entity, preposition, modifier)*

This signature contains 4 terms that together define the modification. The *event\_id* of this predicate should belong to one of the events in the context. It is with the help of this term that we can constraint our query to an event and answer queries accurately with respect to the event under consideration. The *event\_id* used here is propagated to the *modified\_word* while creating the semantic graph. The *modified\_word* is the head of this predicate and can be a noun or a verb. The *preposition* is the dependent of the *case* relation with the *modifier* as the governor and the *modifier* can be found using the nominal modifier or the *nmod* relation from the dependency relations. The *preposition* in this relation helps in identifying the kind of *modifier* used in the relation. The following example gives example of both the *modified\_word* being a noun as well as a verb.

*Example:* “The game was played on February 7 2016, at Levis\_Stadium, in the San\_Francisco\_Bay\_Area, at Santa\_Clara in California”

*\_property (2, play, on, 'february\_7\_2016')*

*\_property (2, play, at, levis\_stadium)*

*\_property (2, play, in, san\_francisco\_bay\_area)*

*\_property (2, play, at, santa\_clara)*

*\_property (2, santa\_clara, in, california)*

From the above property predicates we can get more information about the *modified\_words*, play and *santa\_clara*. To give context let us also consider the event with ID 2 along with this example.

*event (2, play, null, game)*

From the above facts we understand that the game is/was/will be played ‘*on february\_7\_2016*’, ‘*at levis\_stadium*’, ‘*in san\_francisco\_bay\_area*’, and ‘*at santa\_clara*’. Here we can also conclude that santa\_clara is ‘*in california*’. Thus, with the help of the property predicates we can understand more about the event regarding its properties like time, place and many more. While generating property rules we exclude the *nmod* relations having the specifics like *poss*, and *such\_as* as these are handled as special cases.

**6.3.3 Modifier Predicate**

The modifier predicate is used to model the relationship between adjectives and the nouns they modify and between verbs and their modifying adverbs. The signature of the *mod* predicate can be given as follows.

*\_mod (modified\_word, modifier\_word)*

Here, the *modified\_word* can be a noun or a verb. In case of the noun the *modifier\_word* is an adjective given by the *amod* relation. Similarly, if the *modified\_word* is a verb then the *modifier\_word* is an adverb given by the *advmod* relation. Sometimes, nouns are also modified using the adjectives of quantity, which are marked using the numerical modifier relation or the *nummod* dependency. An example for the same can be shown as follows

*Example:* “The Amazon\_rainforest, also known in English as Amazonia or the Amazon\_Jungle, is a moist broadleafed forest that covers most of the Amazon\_basin of South\_America.”

*\_mod (forest, broadleafed)*

*\_mod (forest, moist)*

*\_mod (know, also)*

The above sentence generates the modifier predicate having both the noun as well as the verb as the *modified\_word*. Here the “*forest”* is modified by the adjectives “*broadleafed*” and “*moist*”, and the verb “*know*” is modified by the adverb “*also*”. Using this predicate, we can infer about the qualities of the nouns and the verbs in the sentence.

**6.3.4 Possessive Predicate**

The possessive predicate is used to model the genitive case in English. It is used to show possession or a possessive relation between two entities in the sentence. The possessive predicate given by *\_possess* has the following signature

*\_possess (possessor, possessed)*

Such a relation is generally present in nouns using the possessive case (‘s). The *possessor* as well as the *possessed* are nouns in the sentence. The *\_possess* relation can be extended towards appositional modifiers of the *possessed*. As defined previously appositional modifiers defines or describes a noun. An example that shows genitive case in English and the extension of the possessive case using appositional modifiers can be given as follows

*Example:* “The American\_Football\_Conference's (AFC) champion team, Denver\_Broncos, defeated the National\_Football\_Conference's (NFC) champion team, Carolina\_Panthers, by 24\_10 to earn AFC third Super\_Bowl title”

*\_possess (american\_football\_conference, team)*

*\_possess (national\_football\_conference, team)*

*\_possess (american\_football\_conference, denver\_broncos)*

*\_possess (national\_football\_conference, carolina\_panthers)*

From the above sentence we can generate the first two predicates, as they fall under the genitive case rule, using the *nmod:poss* dependency relation, but the next two facts need to be extended using the appositional modifier relation (*appos*). According to the first two facts, the “*american\_football\_conference*” and the “*national\_football\_conference*” each possess a team. The names for these team can be inferred using the *appos* relation given by the dependency relations “*appos (team, denver\_broncos)*”and“*appos (team, carolina\_panthers)*”. From the above two observations we can conclude the two facts that “the *american\_football\_conference* possesses *denver\_broncos*” and “the *national\_football\_conference* possesses *carolina\_panthers*”

**6.3.5 Instance Predicate**

Instance predicates are used to model the concept associated with being an instance of a concept. As an example, red is an instance of a color, so we need to model the fact that “red is a color” as follows *color (red)*. This fact states that red is an instance of the concept of color. Alternatively, we can also generate the predicate *\_is (red, color)*, which lets us query both the concept or the instance. This the signature of the *\_is* predicate can be given as follows

*\_is (instance, concept)*

Here the *instance* as well as the *concept* are nouns having the relationship of *instance\_of*. To model this relationship, we can use the *cop* dependency along with its related *nsubj*. As mentioned while describing the *nsubj* relation, if the verb in the sentence is a copula then the governor of the relation is the noun associated with the copula. In such a case we can make use of this relation with the copula to generate this predicate. An example of such a generation can be seen in the following sentence.

*Example:* “Nikola\_Tesla was a serbian-american inventor, electrical engineer, mechanical engineer, physicist, and futurist”

*\_is (nikola\_tesla, inventor).*

*\_is (nikola\_tesla, serbian\_american\_inventor).*

The above example, generates the following dependencies, *nsubj (inventor, nikola\_tesla)* and *cop (inventor, was)*. From these dependencies we can generate the above given instance predicate. Similarly, in the above example we see that the verb (*is*) is associated with other concepts like *engineer*, *physicist*, and *futurist* using the conjunction. Thus, we can extend the definition of the instance predicate to also include the following other facts.

*\_is (nikola\_tesla, electrical\_engineer).*

*\_is (nikola\_tesla, engineer).*

*\_is (nikola\_tesla, futurist).*

*\_is (nikola\_tesla, mechanical\_engineer).*

*\_is (nikola\_tesla, physicist).*

Adding these facts makes the knowledge base stronger and is now able to infer many other things about the passage. Such other facts can be generated with the help of the *conj* dependency that connects concepts like *engineer, physicist*, *and futurist* to the first concept of an *inventor*. Another case, where we can generate the instance predicate is in cases where multi-word expressions like such as, or like are used to compare two concepts to be equivalent. Consider the following sentence as an example for the same.

“Miitomo, which Nintendo introduced globally in 2016, features the company's, Mii, avatar-system and lets the users communicate by exchanging personal information such as favorite movies.”,

*\_is (movie, personal\_information)*

*\_is (favorite\_movie, personal\_information)*

In the above example we use the expression “*such as*” to compare two concepts to be equivalent in the sense that one concept is a likely equivalent of the other. In such cases the generated instance predicate is shown above. Such predicates can be generated using the *nmod*, *mwe*, and the *case* dependencies.

**6.3.6 Relation Predicate**

The relation predicate is used to connect two concepts in events. This predicate is generated to model mainly two relations i.e. dependent clauses and conjunctions. The signature of the relation predicate can be given as follows.

*\_relation (independent\_entity, dependent\_clause\_id, relation\_type)*

Dependent clauses can be of two types depending upon their governors i.e. adjective clauses or adverb clauses. Adjective clauses are dependent clauses that modify a noun, whereas adverb clauses modify a verb. The *independent\_entity* defined in the signature can be either the noun that is modified or the event ID of the verb that is modified. The *dependent\_clause\_id* is always the ID of the verb that is the head of the dependent clause. The *relation\_type* define the relation between the dependent and the independent clause. This system recognizes 3 types of relations viz. *\_clause*, *\_clcomplement* and *\_conj*. All these relations are discussed below with examples.

As mentioned above the *\_clause* relation is generated for adjective and adverb clauses. Examples for both these clauses is given as follows.

*Example:* “The American\_Football\_Conference's (AFC) champion team, Denver\_Broncos, defeated the National\_Football\_Conference's (NFC) champion team, Carolina\_Panthers, by 24\_10 to earn AFC third Super\_Bowl title”

*\_relation (1, 2, \_clause)*

*event (1, defeat, denver\_broncos, carolina\_panthers)*

*event (2, earn, afc, title)*

In the above given sentence, there are two clauses, one headed by the verb “*defeat*” and the other by the verb “*earn*”. Here, the clause headed by “*defeat*” is the main clause whereas the adverb clause headed by “*earn*” is the dependent clause. Such a relation can be found out using the dependency *advmod*. Similarly, consider the adjective clause in the following sentence

“The American\_Broadcasting\_Company (ABC), stylized in the network's logo as ABC since 1957, is an American commercial broadcast television network”

*\_relation (american\_broadcasting\_company, 1, \_clause)*

*event (1, stylize, null, null)*

In this sentence, the noun “*American\_Broadcasting\_Company*” is modified by the dependent clause headed by the verb “*stylize*”. Such a predicate can be modeled using the *acl* dependency relation. One of the other relations is *\_clcomplement*. This type of relation models both the clausal complement (*comp*) as well as the open clausal complement (*xcomp*). Such a relation is used to search for the subject of the dependent clause. An example of such a relation is given as follows.

“The ideal thermodynamic cycle used to analyze the process is called the Rankine\_Cycle”

*\_relation (1, 2, \_clcomplement)*

event (1, use, null, null)

*event (2, analyze, null, process)*

Conjunctions are words that connect two or more clauses. The relation predicate is also used to model this relation between any two clauses. An example of such a predicate is given below.

“Water heats and transforms into steam within a boiler operating at a high pressure”

*\_relation (2, 3, \_conj)*

*event (2, heat, water, null)*

*event (3, transform, water, null)*

In the above example we use the *\_conj* relation type to model the conjunction relation between the verbs “heat” and “transform”. As seen above the *\_conj* relation type uses ID’s for verb conjunctions whereas actual nouns for noun conjunctions.

**6.3.7 Named Entity Predicates**

As mentioned in the Chapter 5, the system uses the Named Entity Tagger to get information about entities in the text. The Named Entity Tagger marks various classes like *LOCATION*, *PERSON*, *ORGANIZATION, MONEY, PERCENT, TIME* in the text. We make use of these tags to generate facts of the form *concept(instance)*. These facts together with the rules generated by the ontology help in reasoning about the text. We will discuss about how the predicates generated using the Named Entity Tagger look like and examples where they may be useful.

**6.3.7.1 Time Predicate**

The time predicate deals with recognizing a date, timestamp, time of the day, month, or year in the sentence. Thus, a time predicate is used to understand that the term inside the predicate represents temporal knowledge. The signature of the time predicate can be given as

*time (term)*

Most of the models in the Stanford Named Entity Tagger mark time entities using the tag ‘DATE’.



This predicate proves useful in constraining queries regarding time. Questions that look for a date, month or a year can apply this predicate as an additional constraint to improve accuracy. Some of the examples using this predicate include ‘*time (1745)*’, ‘*time ('7\_january\_1943')*’.

**6.3.7.2 Location Predicate**

The location predicate is used for identifying specific locations. These may include cities, states, regions, countries and many more. The Named Entity Tagger marks locations with the tag LOCATION.



The signature for this predicate can be given as

*location (instance)*

The *instance* represents the name of the location. Locations can be used to constraint Where queries. Furthermore, with the help of knowledge from WordNet, we could infer more about the event under consideration.

**6.3.7.3 Organization Predicate**

The organization predicate is used to recognize organizations like groups, companies, products etc. With the help of this predicate we can better answer questions regarding entities like companies, their ventures, subsidiaries etc. The Named Entity Tagger marks organizations with the tag ORG.



The signature of the Organization predicate can be given as

*organization (instance)*

**6.3.7.4 Person Predicate**

The person predicate marks all the people in the given text. It is used to identify persons in the text so that we could infer more about those people with the help of WordNet. The NER would annotate the person with the tag PERSON as shown below.



The signature of the Person predicate can be given as

*person (instance)*

Using this predicate, we can apply generalization rules about persons on the instance. For example, if we have a generic rule that “All persons can run”, then now if we can prove that ‘Nikola Tesla’ is a person then we can automatically infer that “Nikola Tesla can run”.

**6.3.8 Special Predicates**

Special predicates have been used in the system, to model concepts that are patterns and are understood by humans implicitly. As grammatical relations in the sentence do not convey the meaning of these concepts, they must be extracted out specially. Some of them include abbreviations, time spans and so on. The more of these patterns are learned by the system the better it can reason like humans. We will discuss some of them in this chapter.

**6.3.8.1 Time Span Predicates**

Time spans are sometimes expressed in text using the bracketed notion. The meaning of these time spans depends upon the noun that they follow. If the noun is a person, then it may mean that the time span indicates the birth and the death date of the person. On the contrary if the noun is an organization or a product or a project then the time span indicates the start and end date for the organization or the project. A timespan is generally of the format “(DATE - DATE)”. The first date is considered as the start date and the second is considered as the end date in the format. The signature for the start and the end date can be given as

*\_start\_date (entity, date)*

*\_end\_date (entity, date)*

Here the entity refers to the noun that precedes this timestamp form and the date would indicate either the start or the end date extracted from the format. Some examples for these are given in the following sentences.

*Example:* “Project\_Mercury was followed by the two-man Project\_Gemini (1962 – 1966)”

*\_start\_date (project\_gemini, 1962)*

*\_end\_date (project\_gemini, 1966)*

*Example:* “Martin\_Luther (10 November 1483 – 18 February 1546) was a German\_professor of theology, composer, priest, former\_monk and a seminal\_figure in the Protestant\_Reformation.”

*\_start\_date(martin\_luther, '10\_november\_1483')*

*\_end\_date (martin\_luther, '18\_february\_1546')*

In the first example the timespan represents the start and the end time for project gemini and in the second example the same denotes the life span of Martin Luther.

**6.3.8.2 Date Part Predicates**

The time predicates that have been detected from the Named Entity Tagger, can be utilized to get the day, month, and year parts from the time. This information about the various parts of time is understood implicitly by humans. We can use simple information extraction techniques to find out the year, month, and day predicates from the time predicate. The signature of these predicates can be given as follows

*day (time, day)*

*month (time, month)*

*year (time, year)*

Here, the *time* refers to the entity tagged as TIME by the NER and the second term refers to the extracted day, month and year represented in the time. This is useful information for answering specific queries relating to time. Consider the time predicate and its parts mentioned below.

time ('10\_november\_1483')

day ('10\_november\_1483', 10)

month ('10\_november\_1483', november)

year ('10\_november\_1483', 1483)

From the previous section we know that the time mentioned is the birth date of Martin Luther, so using these date parts we can now query the day, month, or year that Martin Luther was born.

**6.3.8.3 Concept Predicates for Appositional Modifiers**

Appositional modifiers directly follow the noun they describe. We could use this information to model the fact that most appositional modifiers follow the *instance\_of* pattern. This is used to generate concept predicates from the appositional modifiers. Consider the following example for creating a concept predicate.

*Example:* “Jim's brother, Sam, is coming to town today.”

*brother(Sam)*

*Example:* “Jim met Sam, Jim's brother, today.”

*brother(Sam)*

In the above sentences, we have a relation of an appositional modifier between the words “*brother*” and “*Sam*”. This tells us that these words may share an *instance\_of* relation between each other, which leads us to generate the above facts from the appositional modifier. While generating the predicate it is important to look at who is acting as the governor and the dependent in the *appos* relation. Adding more concept predicates to the knowledge base opens the possibility of being able to infer better.

**6.3.8.4 Abbreviation Predicate**

Sometimes, texts contain abbreviation of words. It becomes very important for a system to understand the meaning of these abbreviations for querying. This is because the queries asked can sometimes contain short forms for organizations and projects rather than their long forms. Abbreviations can be detected by either doing pattern matching i.e. looking for the pattern “CONCEPT (ABBREVIATION)” or by using the *appos* dependency relation. The signature for the abbreviation predicate is as follows

*\_abbreviation (short\_form, long\_form)*

In the above definition, the *short\_form* refers to the abbreviation whereas the *long\_form* refers to the abbreviated concept. This can be shown using the following example

*Example:* “The Apollo program was the third United\_States human spaceflight program carried out by the National\_Aeronautics\_and\_Space\_Administration (NASA)”

\_abbreviation (nasa, national\_aeronautics\_and\_space\_administration)

Capturing the abbreviations of various concepts in the text makes the inferring system more robust.

**6.3.8.5 Number Predicate**

The number predicate is used to identify numbers. Identifying a word to be a number helps in constraining the domain for certain queries. Other than that, for questions like “how many” and “how much” it is more accurate to give a number as the answer. The signature for the number predicate can be given as

*number (value)*

Here the value is the actual number occurring in text. Finding if a word is a number can be either done using regex matching or by relying on the POS tagger and using the Cardinal Number (CD) POS Tag. The following sentence can be used as an example

*Example:* “Kenya covers 581,309 km2 and had a population of approximately 45 million people in July 2014.”

*number ('581,309').*

*number (45).*

*number (million).*

**6.4 Supplementary Knowledge Generation**

The knowledge that we extract using the above sections comes directly from the input text. Apart from this, we can make use of various knowledge sources like WordNet to gain supplementary information about the concepts in the passage. Such supplementary knowledge plays an important role in common sense reasoning. We as humans use this knowledge, often referred to as “Common World Knowledge”, subconsciously while reasoning. This system uses the Hypernym relation from WordNet to build its ontology. The following sections dive deep into how these relations are used for reasoning.

**6.4.1 Ontology Representation**

The ontology section of the generated ASP program contains rules generated automatically from the Hypernym relations and rules used for Word Sense Disambiguation. To generate these rules, we use set patterns from answer-set programming like the preference pattern and the default reasoning pattern. Once rules and facts are generated from the sentences, we extract all the nouns from the passage for further processing. These nouns are used for building the ontology for the passage. Before building the ontology all these nouns are converted into concepts. In general the concept is represented using the following signature and we will continue using this signature to represent concepts all through the rest of this chapter.

*concept (concept\_instance, instance\_sense)*

In the above signature the *concept* represents the noun under consideration, *concept\_instance* is the instance of the concept also a noun from the passage, and *instance\_sense* is one of the lexical category mentioned in Chapter 7 Section 7.4.2. An example of a concept can be given as follows

*lion (lion1, noun\_animal)*

Here, *lion1* is an instance of a lion, with the sense *noun\_animal*. The following subsections will go through the procedure of generating the ontology rules from the concepts that are retrieved from the passage.

**6.4.1.1 Hypernym Processing**

Hypernyms can be used to infer about various properties and functions about concepts. Hypernyms make use of the generalization principle to transfer properties from more general concepts to their specific concepts. To do this, we first need to identify the concepts or entities from the passage and generate a hypernym graph from those concepts. These individual graphs can then be aggregated to join common base concepts. This aggregated graph is now ready to be converted into rules that transfer properties from one concept to the other. Let us consider the concept “*lion*” to demonstrate this process. From the figure shown in Chapter 7 section 7.4, we know that there are 4 different senses of the word lion viz. animal, person, person, location. Let us explore the *noun.animal* sense of the concept *lion* and see how the hypernym rules can be generated.



From the above figure, we can understand that if we are able to prove that some concept *lion1*, which is an instance of a lion, having the sense *noun\_animal* is true i.e. the fact *lion (lion1, noun\_animal)* holds then we can infer that “*lion1 is an animal*” [*animal(lion1, noun\_tops)*] or “*lion1 is a living thing*” [*living\_thing(lion1, noun\_tops)*]. The above figure just shows rules generated for one of the senses of the concept “*lion*”. Let us now discuss in detail how these rules can be generated for all the senses of a concept in the following sub-sections.

**6.4.1.1 Generate Hypernym Concept Graph**

The first step to generate Hypernym rules is to generate a hypernym graph from the concepts found in the passage and create a concept bag containing all the hypernyms required for the passage. As hypernyms in WordNet also depend on the different senses of the word, we need to get hypernyms for the different senses of the word as well. As an example, let us consider the concept of a “*car*”. The word “*car*” has 5 distinct senses. A car can mean the following things

*C.1* A motor vehicle with four wheels; usually propelled by an internal combustion engine

*C.2* A wheeled vehicle adapted to the rails of railroad (rail car)

*C.3* A conveyance for passengers or freight on a cable railway (cable car)

*C.4* A compartment that is suspended from an airship and that carries personnel and the cargo and the power plant (gondola)

*C.5* A compartment where passengers ride up and down (elevator car)

Here, sense 1 used for a motor vehicle is the most frequently used sense for the word “*car*”. Now, as “*vehicle*” is a hypernym for sense *C.1* of “*car*” we can apply the properties of a vehicle like vehicle has capacity, it has mass, it is driven and many more to the “*car*”, but apart from this meaning, mentioned below, vehicle also has other 3 meanings.

*V.1* A conveyance that transports people or objects

*V.2* A medium for the expression or achievement of something (A vehicle for political views)

*V.3* A substance that facilitates the use of a drug or pigment or other material that is mixed with it

*V.4* Any inanimate object that can transmit infectious agents from one person to another

From all the above meanings of vehicle only properties of sense *V.1* can be applied to that of car in sense *C.1*, thus also considering the sense of the entity while extracting knowledge from WordNet is crucial. Below is the algorithm mentioned to collect hypernyms of all the concepts in the text. In the algorithm, we go over all the senses of the word under consideration and add its hypernyms to the bag of concepts. We will then use this concept bag to generate the graph through aggregation as described in the next section.

****

The *GetHypernyms* procedure used in the above procedure is a recursive method that keeps on retrieving Hypernyms for the concept till it reaches ‘*entity*’, which is the root of all nouns in the WordNet Ontology. In this procedure, hypernyms are selected from a set of all hypernyms that either belong to the current sense or are hypernyms belonging to the “noun.Tops” group. The current concept is then connected with its ‘*parentConcept’* and added to the concept bag. This is continued until WordNet reaches entity or no more hypernyms are found. Using such a conservative approach helps to not blow up the information obtained exponentially and helps increase the speed of the reasoning systems by pruning the information at the source.



The above algorithm can be explained better with an example of the concept of “*lion*”. As we mentioned earlier, the word “*lion*” has 4 senses, if we track the 4 senses of the word to ‘entity’ we get the following hypernym relations

Sense 1: (*animal*)

lion 🡪 big\_cat 🡪 feline 🡪 carnivore 🡪 placental 🡪 mammal 🡪 vertebrate 🡪 chordate 🡪 animal 🡪 organism 🡪 living\_thing 🡪 object 🡪 physical\_entity 🡪 entity

Sense 2: (*person*)

lion 🡪 celebrity 🡪 important\_person 🡪 adult 🡪 person 🡪 organism 🡪 living\_thing 🡪 object 🡪 physical\_entity 🡪 entity

Sense 3: (*person*)

lion 🡪 person 🡪 organism 🡪 living\_thing 🡪 object 🡪 physical\_entity 🡪 entity

Sense 4: (*location*)

lion 🡪 sign\_of\_the\_zodiac 🡪 region 🡪 location 🡪 object 🡪 physical\_entity 🡪 entity

The next step is the aggregate all the duplicate entries in these hypernym links and create a common graph that represents the knowledge extracted from WordNet.

**6.4.1.2 Aggregation of Concepts**

In this step, we look at all the concepts in the concept bag and aggregate the concepts that have the same base word but have multiple senses under the same umbrella term. By doing this we can now represent a graph where all the concepts form nodes of the graph and the edges of the graph indicate different senses. The edges in this graph are directed edges going from a more specific concept to a more general concept or the hypernym for a sense of the word. The following graph is formed after merging the 4 senses mentioned above.



From the above graph we can see how the concepts are connected to each other under the hypernymy relation. In this graph we can see the closer the senses are to each other, the earlier they get merged into a higher-level concept. As an example, the senses of “*animal* and *person*”are closer to each other as compared to the senses “*person* and *location*” or “*animal* and *location*”. Using this graph, we can generate rules directly based on certain pre-defined patterns.

**6.4.1.3 Generate Hypernym Rules**

There are two different ways to represent the hypernymy relationship using answer set programming. We will discuss both patterns along with their use cases. One of the patterns uses the ‘*class*’ whereas the other one directly uses the ‘*instance*’ of the class in the relationship. Let us consider the pattern using the *class* first.

The *is\_a* relationship between the concepts of *lion* and *big\_cat* and other concepts in its chain can be represented as facts as follows

*is\_superclass (big\_cat, lion),*

*is\_superclass (feline, big\_cat),*

*is\_superclass (carnivore, feline), … ,*

*is\_superclass (entity, physical\_entity)*

We can then infer from these facts using the *superclass* predicate defined as follows

*superclass (X, Y) :- is\_superclass (X, Y).*

*superclass (X, Y) :- is\_superclass (Z, Y), superclass (X, Z).*

These kinds of modelling for the inheritance relationship is typically used in ASP. Although, this model helps us to understand that the classes *lion* and *animal* are related with the relationship “*A lion is an animal*”, it is not able to infer anything about the instances of a lion in general. This kind of representation helps when the domain for classes is infinite. In the current application it would be better to represent the hypernymy relations with the help of the instances of the class instead as the number of classes used are finite. We have seen an example of such a representation in Section 6.4.1.1. Let us consider a part of its simplified representation here as an example

*big\_cat (X) :- lion (X).*

*feline (X) :- big\_cat (X).*

*…*

*animal (X) :- vertebrate (X).*

Thus, given the sentence “*Sam is a lion*”, if we can prove that the predicate “*lion (sam)*” holds then using the above rules we can prove that “*animal (sam)*”. Doing this kind of inference helps us know more about “*sam*” and can help in complex tasks like question answering. When extracting relationships from WordNet we need to deal with multiple senses. This introduced the concept of *sense* in the hypernym relations as well. Thus, the above relations become

*big\_cat (X, noun.animal) :- lion (X, noun.animal).*

*feline (X, noun.animal) :- big\_cat (X, noun.animal).*

*…*

*animal (X, noun.tops) :- vertebrate (X, noun.animal).*

Adding the sense term to the predicate makes the hypernym rule representation more accurate, as now only if we can prove that the instance of the *lion* belongs to the animal lexical category, then only we would be able to prove that the instance also belongs to the animal class. We will discuss how we implement the word sense disambiguation using these senses in the next section.

**6.4.2 Word Sense Disambiguation**

Simply put, Word Sense Disambiguation is a task of selecting the best sense out of a collection of senses applicable to a concept. Word Sense Disambiguation or WSD is one of the important sub tasks in most natural language tasks. When queried from WordNet we get a list of senses for a specific concept ordered from the most used to the least used. The following sections will discuss more on how ASP patterns are used to select the most correct sense for every concept using default and preference patterns.

**6.4.2.1 Representation of Word Senses**

Word senses are represented using two different logic patterns. The first of those will be discussed in this sub section and the later i.e. the preference pattern will be introduced in the next section. Using both these patterns in union, senses are selected for the various concepts in the text which activate their hypernym relations. The pattern discussed here, tries to prove a sense for a concept. Its template can be given as follows

*c (X, si) :- c (X),*

*properties\_si (X),*

*not -c (X, si).*

In the above template, we are trying to prove that “*X is an instance\_of concept c with si*”. Here every concept *c* has one or more senses denoted by *si*. The above rule states that *X*  is an instance of the concept *c*  with the sense *si*only if we can prove that *X* is some instance of concept *c*, *X* has all the properties required to be of sense *si*, and we cannot prove that *X*is definitely not an instance of concept *c*with sense *si*. Such, rules are generated for every sense *si* of the concept in the order of senses from the most use sense to the least used sense. Let us consider each term of the rule body and understand its importance. The first term given by ‘*c (X)*’ is responsible to short circuit and fail the rule it the instance *X* does not belong to the class *c*. The second term, ‘*properties\_si(X)*’, is the main predicate which tries to prove that *X* shows the properties of having sense *si*. It is this predicate that can be added either manually or using other sources to prove the sense. The third term of the body is a strong exception against the head of the rule, that fails the rule if an exception is found against the application of the sense.

These kinds of rules help the system prove that a certain sense applies if such information exists. In general, these kinds of rules are created so that users of the system are able to prove certain sense using external manually coded rules and properties. As an example, let us take the example of a *tree* concept. The WordNet gives 3 different senses for the *tree* concept as given below



Here, *tree* has three senses S = { *plant, diagram, person* }, ordered according to their frequency of use. Thus, using the above-mentioned template for the sense the three rules generated for the *tree* concept can be given as follows

*tree (X, plant) :- tree (X),*

*properties\_plant (X),*

*not -tree (X, plant).*

*tree (X, diagram) :- tree (X),*

*properties\_diagram (X),*

*not -tree (X, diagram).*

*tree (X, person) :- tree (X),*

*properties\_person (X),*

*not -tree (X, person).*

**6.4.2.2 Preference Patterns for senses**

From the above section we know that every concept has one or more senses. One of the principles that we as humans follow when performing word sense disambiguation in our day to day life is use a preference pattern over these senses and choose the most preferential pattern. This section describes, how to system models this preferential pattern to decide a default sense to every concept, if a sense is not provided or it can’t be proved. The preferential pattern can be given as follows

Consider a concept *c* having three senses *s1*, *s2* and *s3* ordered according to the frequency of their use from the most used to the least used. Then, we first assume the sense to be *s1*, unless we know that *s1* is not the sense from some other source. Then we move on to the next sense *s2* unless we know that both *s1* and *s2* can’t be the senses. And then finally we choose *s3*. This process of elimination of senses and choosing senses according to preferences can be modeled in the following template.

*c (X, sp) :- c (X),*

*not -c (X, sp),*

*-c (X, s1), -c (X, s2), … -c (X, sp-1),*

*not c (X, sp+1), not c (X, sp+2), …, not c (X, sn).*

The above skeleton is applied for all the senses of concept *c* in order of preference. The above template represents the rule generated for the pth sense of the concept *c* such that *1 > p > n*. Here *n* is the total number of senses of concept *c*. If *p = 1* then the template omits the classical negation terms as follows

*c (X, s1) :- c (X),*

*not -c (X, s1),*

*not c (X, s2), not c (X, s3), …, not c (X, sn).*

Similarly, if *p = n*, then the template omits the negation as failure terms, given as

*c (X, sn) :- c (X),*

*not -c (X, sn),*

*-c (X, s1), -c (X, s2), … -c (X, sn-1),*

In the above templates the combination of classical negation and negation as a failure play an important role in modelling the preferential pattern. Let us discuss the 4 terms in the general template of this rule. The first term ‘*c (X)*’ does the function of a guard clause as mentioned before. The second term given by ‘*not -c (X, sp)*’ act like a strong exception against the sense *sp*. The third set of terms representing classical negation make sure that senses are applied according to preference i.e. the *pth* sense is only considered if all other senses before *p* have been proved to be false. The fourth set of terms representing negation as a failure, make sure that no other sense falling in the lower order can be proved to be valid before assuming *sp* to be the correct sense. We can see this pattern in action with the *tree* concept as an example.

*tree (X, plant) :- tree (X),*

*not -tree (X, plant),*

*not tree (X, diagram),*

*not tree (X, person).*

*tree (X, diagram) :- tree (X),*

*not -tree (X, diagram),*

*-tree (X, plant),*

*not tree (X, person).*

*tree (X, person) :- tree (X),*

*not -tree (X, person),*

*-tree (X, plant),*

*-tree (X, diagram).*

This pattern is responsible for assigning at least on sense for every concept in the text. The preferential pattern along with the property pattern mentioned in the above section help to solve the task of Word Sense Disambiguation with the help of assumptions based on common world reasoning.

**6.4.2.3 Sources for Word Sense Disambiguation**

Word Sense Disambiguation is a very common problem and there are many statistical and dictionary-based solutions to it. This framework also makes it possible for such methods to be used as inputs to the program. Dictionary based algorithms like the Lesk Algorithm, that uses the context of the ambiguous word, or statistical classifiers like the Naïve Bayes classifier, that uses statistical methods, are effective methods that can provide an accurate sense of a concept to the program.

We can override the preferential pattern mentioned above using these methods by generating facts of the type ‘*concept (c, s)*’ and placing them before the ontology patterns in the ASP execution. In the above fact, we know that *c* is the instance of the *concept* with the sense *s*. Due to the use of negation as failure terms in the pattern, if we are able to prove any one sense with the help of facts as above, then it can be applied thus avoiding the default pattern altogether.

**CHAPTER 7**

**QUERY GENERATION**

**7.1 Overview**

One of the parts in the system architecture is query generation from a natural language question. This chapter will deal with the idea behind and the steps required to query generation. First, we will go over the various schools of thought regarding classification of questions. Out of the many ways that questions can be classified, the system classifies the questions into categories based on the question word or the ‘Wh-word’ in the question. Once the question is posed to the system in natural language the system goes through some steps to convert it into a set of queries, that are ranked according to confidence. This chapter dives deep into the steps of conversion, discusses the various confidence classes and relaxation strategies to make the system more robust. Finally, the chapter will go over the various supplementary rules and predicates that are used for question answering.

**7.2 Question Classification**

In general question classification can be based on many different of schools of thought. In this section we will learn about four techniques for question classification and will describe in detail three of them. The fourth question classification technique for questions based on the question word has been used by the system for query generation and will be discussed in detail in the next section. The different techniques for question classification can be broadly classified into the following

1. Based on Bloom’s Taxonomy
2. Based on reading comprehensions
3. Based on the purpose of the question
4. Based on the question word

We will discuss the first three methods in some detail in this section.

1. Based on Blooms Taxonomy

Blooms taxonomy provides an important framework to focus on the higher order of thinking. This taxonomy is divided into 6 different levels. These levels consist of keywords that define a critical level of thinking. These 6 levels are as follows



These levels go from 1 to 6 in increasing order of difficulty. Level 1 deals with the task of remembering and tests the user’s ability to memorize, recall terms, facts, and details without necessarily understanding the concept. Level 2 is based on comprehension and deals with understanding the concepts and being able to describe them in their own words or being able to summarize these concepts. Level 3 wants the users to be able to apply or transfer the knowledge gained in their own experiences or to a different context. Level 4 deals with analyzing concepts like finding out relations between multiple concepts, being able to differentiate and distinguish between two entities. Level 5 deals with evaluation or judging that asks users to form their own opinions and make decisions. The final level i.e. Level 6 is based on creating or being able to create something completely new based on the ideas from various sources.

B. Based on reading comprehensions

According the reading comprehensions the questions are classified into 4 types. Literal questions, Cause and Effect questions, Inferential questions, and Vocabulary questions. Literal questions are fact-based question and the answers to those questions are present in the passage. Cause and Effect questions are why-based questions, that ask about the effects of an action or the cause of the action. Inferential questions require non-trivial inferential techniques to answer them. Vocabulary questions deal with grammar and require the user to understand the meaning of a word in the context to answer such questions.

C. Based on the purpose of the question

These types of questions are classified based on the purpose of the question. The various classes in this type of categorization as Factual, Convergent, Divergent, Evaluation and Combinations of all the classes. Factual questions are reasonably simple, straightforward questions usually based on obvious facts or awareness. These require the lowest form of cognition. Convergent questions can be answered with a finite level of accuracy. These may be at different levels of cognition but the answers to these questions lie in the passage in an indirect form. Convergent questions include questions on reasoning and contemplation. Divergent questions ask users to create different variations, alternate answers, and scenarios. Correctness of these kinds of questions are usually based upon context. Contrapositives and counterintuitive questions are examples of divergent questions. Evaluation questions require a highly sophisticated level of cognitive thinking and emotional intelligence. Answers to such questions are analyzed at multiple levels before reaching to a conclusion.

**7.3 Classification Based on Question Word**

Questions that are classified according to their question word are of three types: Questions starting with a Wh-word, starting with how or have, and starting with “Is”. Let us go over each of these types in detail

**7.3.1 Starting with ‘Wh-word’**

These kinds of questions contain Wh-word like what, why, which, where, when and who. Each of these question types are looking for a different type of answer

7.3.1.1 What Questions

These kinds of questions ask for the determination of cause, judgement, or properties. Questions like what caused something to happen (antecedent) or what caused something to happen (consequence) belong to the ‘what’ questions. Other than this, ‘what’ questions also ask for the kind of properties of an object or concept. What question as generally asked when the answer domain set is unknown or is infinite and hence, are harder to answer as they can result in a lot of different type of answers.

e.g. What color is the bus? What caused the accident?

7.3.1.2 Why Questions

These kinds of questions are looking for a cause-effect relationship and ask about goals or explanations from the user. These types of questions may require a higher level of cognitions and can’t be always answered using the paragraph.

e.g. Why did he leave? Why are you shivering?

7.3.1.3 Which Questions

Which questions ask for an identification of a person, place, object, or event. These kinds of questions may be answered with instances of concept that are asked along with the question. ‘Which’ questions indicate that the number of answer are finite, and the answer domain is completely known. If this is not the case, then the question changes to a ‘What’.

e.g. Which box is damaged? Which boy is the tallest?

7.3.1.4 Where Questions

Where questions are one of the easier questions to answer. These questions are mainly looking for places or locations as answers.

e.g. Where do you live? Where should we go to eat?

7.3.1.5 When Questions

When questions are asked to know about the time of an event or a process.

e.g. When is your next meeting? When did you reach here?

7.3.1.6 Who Questions

Who questions ask for an instance of a person or a group of people.

e.g. Who called you here? Who owns this company?

**7.3.2 Starting with ‘How’ or ‘Have’**

7.3.2.1 How Questions

How questions are posed to usually ask for a procedure or a quantity. How questions can be further extended to questions like “how much”, “how many”, “how far” and so on.

e.g. How much is price for cotton? How many people were injured?

7.3.2.2 Have Questions

Have questions ask for a yes or no response. These kinds of questions the modal verb as the main question word.

e.g. Have you finished the work? Have you turned off the lights?

**7.3.3 Starting with ‘Is’**

Questions that start with Is usually ask for verification or permission from someone.

e.g. Is this machine working? Is this seat taken?

**7.4 Steps in Query Generation**

The query generation module proposes a method for converting natural language questions to answer set queries. As a question is also a sentence, it is processed in the same way that any other sentence would as mentioned before in the thesis. This means that a semantic graph is generated for every question and event regions are created within the question assigning event ids to different parts of the question. In this section, we will go through the various rules that this module uses for query generation. The question understanding phase tries to analyze the question and uses this analysis for generating query predicates in the next phase. Currently this module is only built to deal with simple interrogative sentences and can be extended to deal with more complex questions.

**7.4.1 Understanding Question**

For analyzing the question, we try to get four kinds of information from the question viz. the question word, the question type as mentioned in the previous section, the answer word or the focus of the question and the answer type.

The question word is the Wh-word found in the question. In case there are more than one Wh-words in the question then we select the one that is closest to the root of the sentence. As an example, in the sentence, “*Who was climbing the tree?*” there is only one Wh-word, ‘*Who*’, that is acting as the subject to the root of the sentence i.e. *climbing*. But in the sentence “*Who was shouting when I walked into the class?*” there are two Wh-words one in the main clause with the head *shouting* and one in the subordinate clause headed by *walked*. In this case the question word is the Wh-word in the main clause that is the closest to the root of the sentence given by *shouting*. In general, Wh-words can be found out by looking at the following POS Tags on the words: WDT, WP, WP$ and WRB. If none of the tags are found, then the questions may have the copulas as their question word or modals their question words.

Once the question word is found then, we can set the question type with the help of the question word. The question type contains a set of predefined question types that help in further processing. These include *WHAT, WHERE, WHO, WHICH, WHEN, HOW\_MANY, HOW\_MUCH, HOW LONG, HOW\_FAR,* and *UNKNOWN*. Once the main question word is found, we can use the word and its relations to determine the exact question type. Most of the Wh-word questions map to a question type but in case of other questions we need to look at the relations of the question word to infer more. As an example, in the question “*How much do these apples cost?*” after determining the question word to be ‘*How*’, we need to look at the *advmod* relation to determine that the question type is *HOW\_MUCH*. These categories can be extended to add more question types to improve the system.

The third type of information we try to extract is the answer word or the focus of the question. The answer word is the word in a question that tells us what kind of answer is expected from the question. Not all types of questions require an answer word, so in those cases this information is *null*. As an example, questions having the question type *WHAT* and *WHICH* are the ones that require an answer type. For these questions, the answer word can be found by looking at the *det* relationship where the governor is the answer type and the Wh-word, or the question word is the dependent. As an example, in the question, “*What city is the Eifel tower located in?*”, the expected answer is some city or a location. Similarly, the question “*Which flavor of ice cream do you like?*” expects a flavor name as the answer.

The fourth and the last information that we extract from the question is the answer type. The answer type depends on the question type. For most of the question types the required answer type is predefined. The answer types supported by the system are as follows *SUBJECT, OBJECT, PLACE, PERSON, TIME, YEAR, DAY, MONTH, NUMBER and UNKNOWN*. The following table talks about some expected answer types depending on question types.

|  |  |
| --- | --- |
| ***Question Type*** | ***Expected Answer Type*** |
| WHERE | PLACE |
| WHO | PERSON |
| WHEN | TIME |
| HOW\_MANY | NUMBER |
| HOW\_MUCH | NUMBER |
| HOW\_LONG | NUMBER [*length*] |
| HOW\_FAR | NUMBER [*distance*] |
| WHAT | \*\**variable\*\** |
| WHICH | \*\**variable\*\** |
| UNKNOWN | UNKNOWN |

In the case of *WHAT* and *WHICH* question types, the answer type depends on the answer word that we extracted earlier. In the sentence, “*On what months is school closed?*”, the base word for the answer word is “*month*” which makes the answer type *MONTH*. Thus, for *WHAT* and *WHICH* questions different answer types like *YEAR, MONTH, DAY, NUMBER, TIME* etc. can be used. The answer types *SUBJECT* and *OBJECT* are also used with these question types. The *SUBJECT* answer type tells the system that the answer that it is looking is in the subject position of the main clause, whereas the *OBJECT* answer type directs the system to look for the object in the event. As an example, in the sentence “*What company owns YouTube?*” the answer word *company* lies in the subject relations from the main verb *owns*, thus indicating that the plausible answer lies in the subject of an event. Similarly, in the case of the sentence “*Whom are you calling?*” the *dobj* relation between the governor *calling* and the dependent Wh-word “*whom*” indicate that the answer lies in the modifier of some event. With the help of this basic analysis, we start generating the predicates for the query generation.

**7.4.2 Generating Query Predicates**

Using the information that we gathered above, we start generating predicate rules for all the words in the question. We will use predicates mentioned in Section 6.3 as a reference to generate predicates from the question. In general, verbs in the question participates in event predicates and property predicates whereas the nouns participate in property, possessive, and modifier predicates. All the predicates that are generated are combined to apply better constraints on the query. Let us study the *event* and *property* predicate generation separately. The rest of the predicates like the *possess*, *mod*, and named entity predicates are generated in a similar manner as mentioned in Section 6.3.

**7.4.2.1** **Event Predicate**

The event predicate in general can be given as follows

*event* (*event\_id*, *trigger\_verb*, *actor*, *participant*)

Here, we do not know the *event\_id* of the event in the question so we replace that using a variable. The *trigger\_verb* is in the question that triggers the predicate generation. The actor acts like the subject of the verb and the participant is the object or the modifier. The participants can be obtained from the direct object (*dobj*) relations of the verb. Let us now look at the various ways that we can obtain the actor or the subject in any event. Here, we consider three ways that help us find the subject in an event. The first is with the help of the *nsubj* relation, the second one is using the *nmod:agent* relation modeled using the property predicate with the *\_by* preposition and the third one is finding the main clause subject for the subordinate clause trigger word. There may be other methods to search for the subject and can be easily added as patterns. Thus, for any question, we have these three variations mentioned above that help model the *event* predicate.

The three patterns discussed above for the event predicate triggered by the *verb* are given as follows

*event (Ek, verb, Sk, Ok)*

*event (Ek, verb, \_, \_), \_relation (Sk, Ek, \_clause)*

*event (Ek, verb, \_, Ok), \_property (Ek, verb, \_by, Sk)*

While, generating the predicates we use the verb id as the event or the verb index. The index *k* in the above predicates comes from the verb indices that are calculated from the semantic graph of the question. It is this index that helps in constraining the predicates while reasoning about different predicates clubbed together. While generating these predicate patterns if we find out that the answer type is either a *SUBJECT* or an *OBJECT*, then we replace the subject (*Sk*) or the object tag (*Ok*) by the answer tag marked as ‘*Xk*’.

Here, we are using subject and object tags in the event predicates instead of using explicit atoms that are obtained from the question. This is done to make the querying process robust. We apply the subject and object constraints mentioned in the question on the generated *event* predicates with the help of the *\_similar* predicate. With every subject and object tag generated we may have a *\_similar* tag associated that would apply the respective constraint on the tag from the question. We will discuss how the similar tag works in a later section in this chapter.

Consider the question “*what company owns the walt\_disney?*” as an example for the *event* predicate generation. The question suggests that we have an object constraint ‘*walt\_disney*’ on the verb ‘*owns’* andthe answer type is *SUBJECT,* so the following different query parts are generated.

*event (E1, own, X1, O1), \_similar (walt\_disney, O1).*

*event (E1, own, \_, O1), \_property (E1, own, \_by, X1), \_similar (walt\_disney, O1).*

*event (E1, own, \_, \_), \_relation (X1, E1, \_clause), \_similar (walt\_disney, O1).*

**7.4.2.2** **Property Predicate**

Property predicates are generated from verbs and nouns that are modified by nominal phrases in the sentence. A general property constraint has the following template

*\_property (event\_id, modified\_entity, preposition, modifier)*

Here, like the event predicate we are unaware of the *event\_id* and hence it will be a variable. The *modified\_entity* is the noun or the verb that triggered the predicate generation. The *preposition* here can either be obtained from the *case* relation or can be left blank (\_). The modifier is the head of the nominal modifier that can be used to constraint the query. If the modifier is the answer word, then we replace the word with the answer tag (*Xk*). Consider the sentence “*On what streets is the ABC’s headquarter located?*”, that shows the *\_property* predicate as a constraint.

*\_property (located, on, X2).*

**7.4.2.3 Special Conditions and Predicates**

**7.4.3 Applying Base Constraints**

Base constraints are generated at the end after all the constraints from the question have been applied. These constraints refer to the constraint that depend on the answer type of the question. Answer types have already been described in Section 7.4.1, so here we will discuss the generation of some of the base constraints. Let us consider generation of the base constraints one by one.

For cases where the answer type is TIME, DAY, YEAR or MONTH we generate the base constraints as follows.

TIME 🡪 *time (Xk)*

DAY 🡪 *day (Tk, Xk), time (Tk)*

MONTH 🡪 *month (Tk, Xk), time (Tk)*

YEAR 🡪 *year (Tk,* *Xk), time (Tk)*

For Where-questions the answer type is PLACE, which can be constraint by applying the following constraint on the query

PLACE 🡪 *location (Xk) or location (Xk, noun\_location)*

For questions asking for people or Who-questions the answer type is PERSON

PERSON 🡪 *person (Xk)* or *person (Xk, noun\_person)*

In case of the answer type being *UNKNOWN*, we may be expecting the answer to be a specific concept represented by the answer word as used in What and Which questions. Thus, in such cases the base constraint comes from the answer word itself. It is represented by

UNKNOWN 🡪 *concept (Xk)* e.g. *company (Xk)* or *city (Xk)*

**7.4.4 Combining constraints**

After generation of the constraints from Section 7.4.2 and the base constraints from Section 7.4.3 we combine all the constraints to create query. This can be best explained with examples, so let us take a few questions and go over how the query has been generated from the different parts of the question.

Consider the following examples for query generation

1. In what borough of New York City is ABC headquartered?

*\_property (E2, borough, of, new\_york\_city), \_property (E2, headquarter, \_by, S2), \_property (E2, headquarter, in, X2), \_similar (abc, S2), event (E2, headquarter, \_, O2), organization (abc), borough (X2, \_).*

*\_property (E2, borough, of, new\_york\_city), \_property (E2, headquarter, in, X2), \_relation (S2, E2, \_clause), \_similar (abc, S2), event (E2, headquarter, \_, \_), organization(abc), borough (X2, \_).*

*\_property (E2, borough, of, new\_york\_city), \_property (E2, headquarter, in, X2), \_similar (abc, S2), event (E2, headquarter, S2, O2), organization(abc), borough (X2, \_).*

In the above question we have three types of subject event constraints, property constraints enforcing the constraint “*borough of New York City*”, named entity constraint forcing “*abc*” to be an organization and a base constraint forcing “*abc to be headquartered in a borough*”.

2. Since what year did ABC stylize its logo, as ‘ABC’?

*\_possess (abc, logo), \_property (E2, stylize, \_by, S2), \_property (E2, stylize, since, X2), \_similar (abc, S2), \_similar (logo, O2), event (E2, stylize, \_, O2), organization(abc), time(T2), year (T2, X2).*

*\_possess (abc, logo), \_property (E2, stylize, since, X2), \_relation (S2, E2, \_clause), \_similar (abc, S2), event (E2, stylize, \_, \_), organization (abc), time (T2), year (T2, X2).*

*\_possess (abc, logo), \_property (E2, stylize, since, X2), \_similar (abc, S2), \_similar (logo, O2), event (E2, stylize, S2, O2), organization (abc), time (T2), year (T2, X2).*

In the above question we have the similar constraints forcing the constraint that some entity like ABC has stylized an entity like logo. Apart from that we have subject constraints, named entity constraint forcing “*abc*” to be an organization and a base constraint forcing “*styling to be since a year*”.

3. When was Nikola Tesla born?

*event (E2, bear, S2, O2), \_similar (nikola\_tesla, S2), property (E2, bear, on, X2), time(X2).*

*event (E2, bear, \_, O2), \_property (E2, bear, \_by, S2), \_similar (nikola\_tesla, S2), property (E2, bear, on, X2), time(X2).*

*event (E2, bear, \_, \_), \_relation (S2, E2, \_clause), similar (nikola\_tesla, S2), property (E2, bear, on, X2), time(X2).*

*\_start\_date (S2, X2), \_similar (nikola\_tesla, S2), time(X2).*

In this sentence, we have the special predicates *\_start\_date* applying the constraints of timespans on an entity like ‘Nikola Tesla’. All the above are examples of how queries can be generated from natural language questions.

**7.5 Query Confidence Classes and Relaxing Constraints**

**7.6 Supplementary Predicates in Query Generation**

**\_similar, query, question**

**CHAPTER 8**

**APPLICATIONS IN QUESTION ANSWERING**

**8.1 Overview**

The system that has been designed uses an ASP-based approach to represent knowledge from natural language text. So, a basic understanding of answer-set programming is required to understand the remainder of the thesis. This chapter introduces the answer-set programming paradigm and further elaborates on some of the important definitions, concepts and patterns used in answer-set programming. At the end of this chapter, we will go over some of the systems that are developed to run ASP programs.

**8.2 SQuAD Dataset**

**8.3 Categorization of Domains**

**8.4 Results**

Results about Question Answering

% Success

**8.5 Analysis**

**--** Positives and Negatives

-- Able to answer questions from passage

-- Able to produce multiple answers

-- Not able to answer complex questions

-- Verb and Noun similarity not implemented

-- Not enough automated information about verbs

-- Implementation of temporal reasoning

**CHAPTER 9**

**CONCLUSION AND FUTURE WORK**

**9.1 Overview**

The system that has been designed uses an ASP-based approach to represent knowledge from natural language text. So, a basic understanding of answer-set programming is required to understand the remainder of the thesis. This chapter introduces the answer-set programming paradigm and further elaborates on some of the important definitions, concepts and patterns used in answer-set programming. At the end of this chapter, we will go over some of the systems that are developed to run ASP programs.

**9.2 Conclusion**

-- System can be improved by adding more knowledge

-- Can make use of more NLP tools to improve performance

-- Can give multiple answers if exists

-- Will help in utilizing the vast amount of knowledge in text

**9.3 Future Work**

-- Temporal reasoning

-- Adding Rules for more knowledge patterns

-- Adding Rules to handle more domains

-- Modelling semantic similarity for answering questions

ie synonyms, similarity score between words

-- Knowledge base can be improved by using other sources like Wikipedia

-- Facts can be marked with confidence depending on their source

-- Relaxing of Constraints have various approaches for a query like using \_ and removing predicates in various order etc…

-- Answering cross sentence questions

**CHAPTER 10**

**APPENDIX**

1. Knowledge Representation, Reasoning, and the Design of Intelligent Agents (The Answer-Set Programming Approach), Micheal Gelfond, Yulia Kahl
2. Dissertation – Goal Directed Answer Set Programming, Kyle Marple
3. Answer-Set Programming – A Primer, Thomas Eiter, Giovambattista Ianni,and Thomas Krennwallner
4. <https://stanfordnlp.github.io/CoreNLP/>
5. Manning, Christopher D., Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. [The Stanford CoreNLP Natural Language Processing Toolkit](http://nlp.stanford.edu/pubs/StanfordCoreNlp2014.pdf) In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55-60. [[pdf](http://nlp.stanford.edu/pubs/StanfordCoreNlp2014.pdf)] [[bib](http://nlp.stanford.edu/pubs/StanfordCoreNlp2014.bib)]
6. Kristina Toutanova and Christopher D. Manning. 2000. Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. In Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000), pp. 63-70.
7. Kristina Toutanova, Dan Klein, Christopher Manning, and Yoram Singer. 2003. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In Proceedings of HLT-NAACL 2003, pp. 252-259.
8. Jenny Rose Finkel, Trond Grenager, and Christopher Manning. 2005. Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics (ACL 2005), pp. 363-370. <http://nlp.stanford.edu/~manning/papers/gibbscrf3.pdf>
9. Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser Using Neural Networks. In Proceedings of EMNLP 2014.
10. George A. Miller (1995). WordNet: A Lexical Database for English. Communications of the ACM Vol. 38, No. 11: 39-41.
11. Christiane Fellbaum (1998, ed.) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.
12. Finlayson, Mark Alan (2014) [Java Libraries for Accessing the Princeton Wordnet: Comparison and Evaluation.](http://projects.csail.mit.edu/jwi/download.php?f=finlayson.2014.procgwc.7.78.pdf) In H. Orav, C. Fellbaum, & P. Vossen (Eds.), [Proceedings of the 7th International Global WordNet Conference](http://gwc2014.ut.ee/proceedings_of_GWC_2014.pdf) (GWC 2014) (pp. 78-85). Tartu, Estonia.
13. <https://martinfowler.com/bliki/TestDrivenDevelopment.html>
14. Test Driven Development: By Example: Kent Beck
15. Penn Treebank English POS tag set: 1993Computational Linguistics article in PDF, AMALGAM page, Aoife Cahill's list.
16. https://www.ling.upenn.edu/courses/Fall\_2003/ling001/penn\_treebank\_pos.html
17. POS Confusion Matrix Franz(1996), Kupiec(1992), and Ratnaparkhi(1996)
18. http://universaldependencies.org/u/dep/
19. http://universaldependencies.org/u/dep/all.html
20. https://wordnet.princeton.edu/
21. <https://wordnet.princeton.edu/documentation/lexnames5wn>
22. http://faculty.academyart.edu/faculty/teaching-topics/teaching-curriculum/enhancing-teacher-student-interaction/different-types-questions-blooms-taxonomy.html