**CHAPTER 4**

**ANSWER-SET PROGRAMMING**

**4.1 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.

**4.1 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.

**4.2 Syntax**

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

**4.2.1 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.

**4.2.2 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 where as ‘parent(mary, alice)’ is called as a ground literal.

**4.2.3 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.

**4.3 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.

**4.3.1 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.

**4.3.2 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.

**4.3.3 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.

**4.3.4 Constraints**

Constraints are applied in places where we know for a fact 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.

**4.3.5 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 know to be true, as false.

**4.4 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.

**4.5 Present systems SASP, CLASP**

TBD

**References:**

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*

**CHAPTER 5**

**SYSTEM ARCHITECTURE**

**5.1 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.

**5.2 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.

**5.3 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.

**5.4 Common Resources Framework**

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



**5.4.1 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.

**5.4.2 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.

**5.4.3 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.

**5.5 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



**5.5.1 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.

**5.5.2 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.

**5.5.3 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.

**5.6 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.

**5.6.1 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.

**5.6.2 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.

**References:**

1. <https://stanfordnlp.github.io/CoreNLP/>
2. 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)]
3. 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.
4. 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.
5. 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>
6. Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser Using Neural Networks. In Proceedings of EMNLP 2014.
7. George A. Miller (1995). WordNet: A Lexical Database for English. Communications of the ACM Vol. 38, No. 11: 39-41.
8. Christiane Fellbaum (1998, ed.) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.
9. 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.

**CHAPTER 6**

**SOFTWARE DEVELOPMENT APPROACH**

**6.1 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.

**6.2 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.



**6.3 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.

**6.3.1 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.

**6.3.2 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.

**6.3.3 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.

**6.3.4 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.

**6.3.5 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.

**6.3.6 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.

**6.4 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.

**6.5 Testing Framework**

TBD

**References:**

1. <https://martinfowler.com/bliki/TestDrivenDevelopment.html>
2. Test Driven Development: By Example: Kent Beck

**CHAPTER 7**

**NATURAL LANGUAGE RESOURCES**

**7.1 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.

**7.2 Resource Tools**

**7.2.1 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.

**7.2.2 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.

**7.2.3 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.

**7.3 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.

**7.3.1 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:

7.3.1.1 *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.







7.3.1.2 *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.



7.3.1.3 *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 where as all other are indirect objects of the verb.



7.3.1.4 *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.







7.3.1.5 *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.



7.3.1.6 *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.



**7.3.2 Non-Core Dependents**

7.3.2.1 *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.



7.3.2.2 *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.



7.3.2.3 *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.



7.3.2.4 *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.





7.3.2.5 *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.



**7.3.3 Nominal Dependents**

7.3.3.1 *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.





7.3.3.2 *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.





7.3.3.3 *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*.



7.3.3.4 *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.



7.3.3.5 *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.



7.3.3.6 *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.



7.3.3.7 *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.





**7.3.4 Other Dependency Relations**

7.3.4.1 *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.





7.3.4.2 *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.



7.3.4.3 *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*.



7.3.4.4 *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.



**7.4 WordNet**

**7.4.1 WordNet Relations**

**7.4.1.1 Hypernyms**

**7.4.1.2 Meronyms**

**7.4.1.3 Synonyms**

**7.4.1.4 Antonyms**

**7.4.2 WordNet Senses**

**References:**

1. Danqi Chen and Christopher Manning. 2014. [A Fast and Accurate Dependency Parser Using Neural Networks](http://cs.stanford.edu/~danqi/papers/emnlp2014.pdf). In Proceedings of EMNLP 2014.
2. Kristina Toutanova and Christopher D. Manning. 2000. [Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger](http://nlp.stanford.edu/~manning/papers/emnlp2000.pdf). In *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000)*, pp. 63-70.
3. Kristina Toutanova, Dan Klein, Christopher Manning, and Yoram Singer. 2003. [Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network](http://nlp.stanford.edu/~manning/papers/tagging.pdf). In *Proceedings of HLT-NAACL 2003*, pp. 252-259.
4. Penn Treebank English POS tag set: [1993*Computational Linguistics* article in PDF](http://acl.ldc.upenn.edu/J/J93/J93-2004.pdf), [AMALGAM page](http://www.comp.leeds.ac.uk/amalgam/tagsets/upenn.html), [Aoife Cahill's list](http://www.computing.dcu.ie/~acahill/tagset.html).
5. [**https://www.ling.upenn.edu/courses/Fall\_2003/ling001/penn\_treebank\_pos.html**](https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html)
6. **POS Confusion Matrix Franz(1996), Kupiec(1992), and Ratnaparkhi(1996)**
7. 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>
8. [**http://universaldependencies.org/u/dep/**](http://universaldependencies.org/u/dep/)
9. [**http://universaldependencies.org/u/dep/all.html**](http://universaldependencies.org/u/dep/all.html)