UNIT-II

WORD LEVEL AND SYNTACTIC ANALYSIS

1. WORD LEVEL ANALYSIS

- The language used to specify text search strings is called a regular expression (RE).
- Using a unique syntax that is stored in a pattern, RE aids us in matching or finding other strings or sets of strings.
- Both in UNIX and MS Word, regular expressions are used similarly to search text.

1.1.Regular Expressions

- A regular expression (RE) is a language for specifying text search strings.
- RE helps us to match or find other strings or sets of strings, using a specialized syntax held in a pattern.
- Regular expressions are used to search texts in UNIX as well as in MS WORD in identical way.
- We have various search engines using a number of RE features.

Properties of Regular Expressions

- Followings are some of the important properties of RE:
- American Mathematician Stephen Cole Kleene formalized the Regular Expression language.
- RE is a formula in a special language, which can be used for specifying simple classes of strings, a sequence of symbols.
- In other words, we can say that RE is an algebraic notation for characterizing a set of strings.
- Regular expression requires two things, one is the pattern that we wish to search and other is a corpus of text from which we need to search.

Mathematically, A Regular Expression can be defined as follows −

- ε is a Regular Expression, which indicates that the language is having an empty string.
- φ is a Regular Expression which denotes that it is an empty language.
- If X and Y are Regular Expressions, then
- X, Y
- X.Y(Concatenation of XY)
- X+Y (Union of X and Y)
- X*, Y* (Kleen Closure of X and Y)are also regular expressions.

If a string is derived from above rules then that would also be a regular expression.

Examples of Regular Expressions

The following table shows a few examples of Regular Expressions −

Regular Expressions	**Regular Set**

[abc]	**a,b or c**
[^abc]	**Any character except a,b or c**
[a-z]	**a to z**
[A-Z]	**A to Z**
[a-z,A-Z]	**a to z, A to Z**
[0-9]	**0 to 9**

a. Metacharacters in Regular Expressions

Metacharacters in regular expressions (regex) are special characters that represent particular operations or patterns. They are essential for pattern matching in natural language processing (NLP). Here's a breakdown of common metacharacters and their usage:

Metacharacter	Description	Example
.	Matches **any single character** except newline.	a.b matches acb, a1b, but not ab.
^	Matches the **beginning of a string**.	^Hello matches Hello at the start.
$	Matches the **end of a string**.	world$ matches world at the end.
\b	Matches a **word boundary**.	\bcat\b matches cat, not scatter.
\B	Matches a **non-word boundary**.	\Bcat matches scatter but not cat.

b. Quantifiers in Regular Expression

Metacharacter	Description	Example
`*`	Matches **zero or more** occurrences of the preceding.	`ab*` matches `a`, `ab`, `abb`.
`+`	Matches **one or more** occurrences of the preceding.	`ab+` matches `ab`, `abb` (not `a`).
`?`	Matches **zero or one** occurrences of the preceding.	`colou?r` matches `color` and `colour`.
`{n}`	Matches **exactly n** occurrences of the preceding.	`a{3}` matches `aaa`.
`{n,}`	Matches **n or more** occurrences of the preceding.	`a{2,}` matches `aa`, `aaa`, `aaaa`.
`{n,m}`	Matches between **n and m** occurrences of the preceding.	`a{1,3}` matches `a`, `aa`, `aaa`.

c. Character Class in Regular Expression

Metacharacter	Description	Example
`[]`	Matches any **character inside brackets**.	`[aeiou]` matches any vowel.
`[^]`	Matches any **character NOT inside brackets**.	`[^aeiou]` matches any non-vowel.
`\d`	Matches any **digit** (0-9).	`\d` matches `3`, `7`, etc.
`\D`	Matches any **non-digit** character.	`\D` matches `a`, `$`, etc.
`\w`	Matches any **word character** (alphanumeric + `_`).	`\w` matches `a`, `7`, `_`.
`\W`	Matches any **non-word character**.	`\W` matches `@`, `%`, ` ` (spaces).
`\s`	Matches any **whitespace character**.	`\s` matches ` `, `\t`, `\n`.
`\S`	Matches any **non-whitespace character**.	`\S` matches `a`, `!`, `1`.

d. Groups and Alternatives

Metacharacter	Description	Example
`( )`	Groups expressions for capturing.	`(cat
`	`	Acts as a logical **OR** between expressions.
`(?: )`	Matches group without capturing it.	`(?:cat

e. Escape Characters

Metacharacter	Description	Example
`\`	Escapes special characters or denotes a special sequence.	`\.` matches a literal `.`.
`\\`	Matches a literal backslash.	`\\` matches `\`.

f. Lookaheads and Lookbehinds

Metacharacter	Description	Example
`(?= )`	**Positive lookahead**: Match if followed by a pattern.	`\d(?=kg)` matches `5` in `5kg`.
`(?! )`	**Negative lookahead**: Match if NOT followed by a pattern.	`\d(?!kg)` matches `5` in `5m`, not in `5kg`.
`(?<= )`	**Positive lookbehind**: Match if preceded by a pattern.	`(?<=\$)\d+` matches `50` in `$50`.
`(?<! )`	**Negative lookbehind**: Match if NOT preceded by a pattern.	`(?<!\$)\d+` matches `50` in `cost 50`.

Regular Sets & Their Properties

It may be defined as the set that represents the value of the regular expression and consists specific properties.

Properties of regular sets

- If we do the union of two regular sets then the resulting set would also be regular.
- If we do the intersection of two regular sets then the resulting set would also be regular.
- If we do the complement of regular sets, then the resulting set would also be regular.
- If we do the difference of two regular sets, then the resulting set would also be regular.
- If we do the reversal of regular sets, then the resulting set would also be regular.
- If we take the closure of regular sets, then the resulting set would also be regular.
- If we do the concatenation of two regular sets, then the resulting set would also be regular.

****

2. Finite State Automata

The term automata, derived from the Greek word "αὐτόματα" meaning "self-acting", is the plural of automaton which may be defined as an abstract self-propelled computing device that follows a predetermined sequence of operations automatically. An automaton having a finite number of states is called a Finite Automaton (FA) or Finite State automata (FSA). Mathematically, an automaton can be represented by a 5-tuple (Q, Σ, δ, q0, F), where −

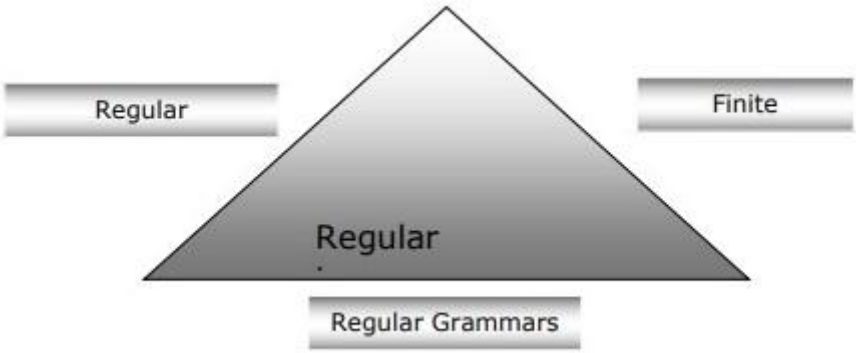
- Q is a finite set of states.
- Σ is a finite set of symbols, called the alphabet of the automaton.
- δ is the transition function
- q0 is the initial state from where any input is processed (q0 ∈ Q).
- F is a set of final state/states of Q (F ⊆ Q).

Relation between Finite Automata, Regular Grammars and Regular Expressions

Following points will give us a clear view about the relationship between finite automata, regular grammars and regular expressions −

- Finite state automata are the theoretical foundation of computational work and regular expressions is one way of describing them.
- Any regular expression can be implemented as FSA and any FSA can be described with a regular expression.
- On the other hand, regular expression is a way to characterize a kind of language called regular language. Hence, we can say that regular language can be described with the help of both FSA and regular expression.
- Regular grammar, a formal grammar that can be right-regular or left-regular, is another way to characterize regular language.

Following diagram shows that finite automata, regular expressions and regular grammars are the equivalent ways of describing regular languages.



2.1.Types of Finite State Automation (FSA)

Finite state automation is of two types. Let us see what the types are.

2.1.1. **Deterministic Finite automation (DFA)**

- It may be defined as the type of finite automation wherein, for every input symbol we can determine the state to which the machine will move.
- It has a finite number of states that is why the machine is called Deterministic Finite Automaton (DFA). Mathematically, a DFA can be represented by a 5-tuple

  M=(Q, Σ, δ, q_0, F), where

- Q is a finite set of states.
- Σ is a finite set of symbols, called the alphabet of the automaton.(sigma)
- δ is the transition function where δ: Q × Σ → Q .(delta)
- q0 is the initial state from where any input is processed (q0 ∈ Q).
- F is a set of final state/states of Q (F ⊆ Q).

Whereas graphically, a DFA can be represented by diagraphs called state diagrams where −

- The states are represented by **vertices**.
- The transitions are shown by labeled **arcs**.
- The initial state is represented by an **empty incoming arc**.
- The final state is represented by **double circle**.

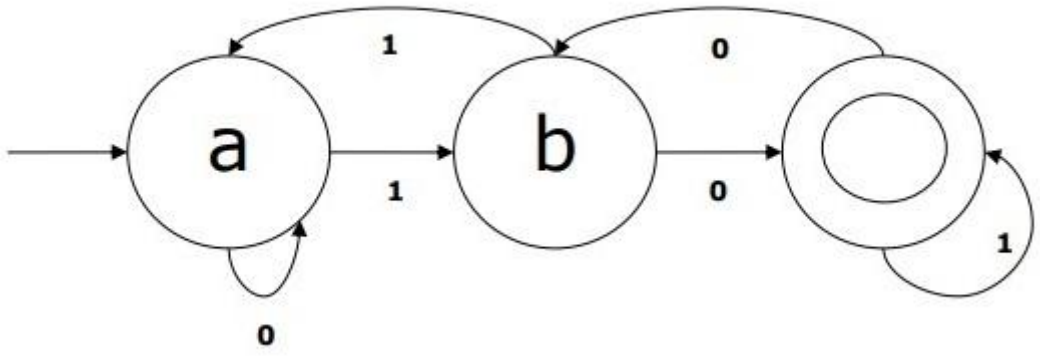
Example of DFA

Suppose a DFA be

- Q = {a, b, c},
- Σ = {0, 1},
- q_0 = {a},
- F = {c},
- Transition function δ is shown in the table as follows −

Current State	Next State for Input 0	Next State for Input 1
A	a	B
B	b	A
C	c	C

The graphical representation of this DFA would be as follows −



2.1.2. **Non-deterministic Finite Automation (NDFA)**

It may be defined as the type of finite automation where for every input symbol we cannot determine the state to which the machine will move i.e. the machine can move to any combination of the states. It has a finite number of states that is why the machine is called Non-deterministic Finite Automation (NDFA). Mathematically, NDFA can be represented by a 5-tuple (Q, Σ, δ, q0, F), where −

- Q is a finite set of states.
- Σ is a finite set of symbols, called the alphabet of the automaton.
- δ :-is the transition function where δ: $Q \times \Sigma \rightarrow 2^{Q}$.
- q0 :-is the initial state from where any input is processed ($q0 \in Q$).
- F :-is a set of final state/states of Q ($F \subseteq Q$).

Whereas graphically (same as DFA), a NDFA can be represented by diagraphs called state diagrams where −

- The states are represented by **vertices**.
- The transitions are shown by labeled **arcs**.
- The initial state is represented by an **empty incoming arc**.
- The final state is represented by double **circle**.

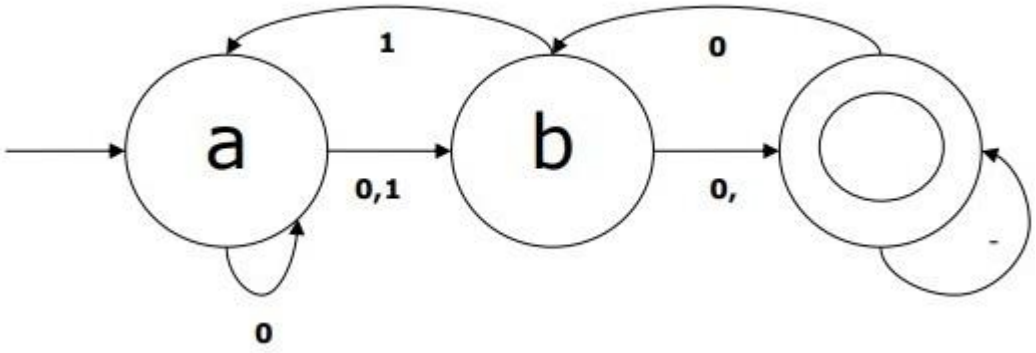
Example of NDFA

Suppose a NDFA be

- Q = {a, b, c},
- Σ = {0, 1},
- q0 = {a},
- F = {c},
- Transition function δ is shown in the table as follows −

Current State	**Next State for Input 0**	**Next State for Input 1**
A	a, b	B
B	C	a, c
C	b, c	C

The graphical representation of this NDFA would be as follows −



****

3. Morphological Parsing

- The term morphological parsing is related to the parsing of morphemes.
- We can define morphological parsing as the problem of recognizing that a word breaks down into smaller meaningful units called morphemes producing some sort of linguistic structure for it.
- For example, we can break the word *foxes* into two, *fox* and *-es*. We can see that the word *foxes*, is made up of two morphemes, one is *fox* and other is *-es*.

In other sense, we can say that morphology is the study of −

- The formation of words.
- The origin of the words.

- Grammatical forms of the words.
- Use of prefixes and suffixes in the formation of words.
- How parts-of-speech (PoS) of a language are formed.

Types of Morphemes

Morphemes, the smallest meaning-bearing units, can be divided into two types −

- Stems
- Word Order

Stems

It is the core meaningful unit of a word. We can also say that it is the root of the word. For example, in the word foxes, the stem is fox.

- **Affixes** − As the name suggests, they add some additional meaning and grammatical functions to the words. For example, in the word foxes, the affix is − es.

Further, affixes can also be divided into following four types −

- **Prefixes** − As the name suggests, prefixes precede the stem. For example, in the word unbuckle, un is the prefix.
- **Suffixes** − As the name suggests, suffixes follow the stem. For example, in the word cats, -s is the suffix.
- **Infixes** − As the name suggests, infixes are inserted inside the stem. For example, the word cupful, can be pluralized as cupsful by using -s as the infix.
- **Circumfixes** − They precede and follow the stem. There are very less examples of circumfixes in English language. A very common example is ‘A-ing’ where we can use -A precede and -ing follows the stem.

Word Order

The order of the words would be decided by morphological parsing. Let us now see the requirements for building a morphological parser −

Lexicon

The very first requirement for building a morphological parser is lexicon, which includes the list of stems and affixes along with the basic information about them. For example, the information like whether the stem is Noun stem or Verb stem, etc.

Morphotactics

It is basically the model of morpheme ordering. In other sense, the model explaining which classes of morphemes can follow other classes of morphemes inside a word. For example, the morphotactic fact is that the English plural morpheme always follows the noun rather than preceding it.

Orthographic rules

These spelling rules are used to model the changes occurring in a word. For example, the rule of converting y to ie in word like city+s = cities not citys.

****

4. Spelling Error Detection and correction

Dealing with spelling errors in Natural Language Processing (NLP) tasks is crucial for improving the accuracy of text-processing applications. Here are several techniques commonly used to handle spelling errors.

1. **Spell Checking**

- **Dictionary-Based Approaches**: Utilize a dictionary or lexicon to check if each word in the text is spelled correctly. If a word is not found in the dictionary, it is considered a potential spelling error.
- **Edit Distance Algorithms**: Algorithms such as Levenshtein distance or Damerau-Levenshtein distance measure the minimum number of edits (insertions, deletions, substitutions, or transpositions) required to transform one word into another. Words with small edit distances to known words in the dictionary can be suggested as corrections.

2. **Phonetic Matching:**

- **Soundex and Metaphone**: Phonetic algorithms map words to phonetic representations based on their pronunciation. Words with similar phonetic representations are likely to be spelled similarly, even if spelled differently. This technique helps in identifying spelling errors where words sound alike but are spelled differently.

3. **Language Models:**

- **Statistical Language Models**: Use statistical models trained on large text corpora to estimate the probability of a word sequence. Language models can help in identifying likely corrections for misspelled words based on the context of surrounding words.
- **Neural Language Models**: Modern neural language models like Transformer-based models (e.g., BERT, GPT) are effective at predicting and correcting spelling errors by considering the context of the entire sentence. Fine-tuning these models on spelling correction tasks can yield highly accurate results.

4. **Language Models:**

- **Statistical Language Models**: Use statistical models trained on large text corpora to estimate the probability of a word sequence. Language models can help in identifying likely corrections for misspelled words based on the context of surrounding words.
- **Neural Language Models**: Modern neural language models like Transformer-based models (e.g., BERT, GPT) are effective at predicting and correcting spelling errors by considering the context of the entire sentence. Fine-tuning these models on spelling correction tasks can yield highly accurate results.

5. **Rule-Based Approaches:**

- **Pattern Matching**: Apply regular expressions or pattern-matching rules to detect common types of spelling errors, such as repeated characters, missing characters, or transposed letters.

- **Language-Specific Rules**: Develop language-specific spelling correction rules based on common misspellings, phonetic patterns, or morphological rules.

6. **Ensemble Methods:**

- **Combining Multiple Approaches**: Combine the outputs of different spelling correction methods, such as spell checking, phonetic matching, and language models, using ensemble techniques to improve accuracy and robustness.

7. **User Feedback:**

- **Interactive Correction**: Allow users to provide feedback on suggested corrections and incorporate this feedback to improve the spelling correction system over time. This can be achieved through interactive interfaces or feedback mechanisms in applications.

8. **Domain-Specific Customization:**

- **Custom Dictionaries**: Create domain-specific dictionaries or lexicons containing relevant terms and vocabulary to improve the accuracy of spelling correction in specific domains or industries.

****

5. Words and Word Classes

Word Classes

- In grammar, a part of speech or part-of-speech (POS) is known as word class or grammatical category, which is a category of words that have similar grammatical properties.
- The English language has four major word classes: Nouns, Verbs, Adjectives, and Adverbs.
- Commonly listed English parts of speech are nouns, verbs, adjectives, adverbs, pronouns, prepositions, conjunction, interjection, numeral, article, and determiners.
- These can be further categorized into open and closed classes.

a. Closed Class

- Closed classes are those with a relatively fixed/number of words, and we rarely add new words to these POS, such as prepositions.
- Closed class words are generally functional words like of, it, and, or you, which tend to be very short, occur frequently, and often have structuring uses in grammar.
- Example of closed class-
- **Determiners**: a, an, the Pronouns: she, he, I, others Prepositions: on, under, over, near, by, at, from, to, with

b. Open Class

- Open Classes are mostly content-bearing, i.e., they refer to objects, actions, and features; it's called open classes since new words are added all the time.
- By contrast, nouns and verbs, adjectives, and adverbs belong to open classes; new nouns and verbs like iPhone or to fax are continually being created or borrowed. Example of open class-
- **Nouns:** computer, board, peace, school Verbs: say, walk, run, belong Adjectives: clean, quick, rapid, enormous Adverbs: quickly, softly, enormously, cheerfully

c. Tagset

- The problem is (as discussed above) many words belong to more than one word class.

- And to do POS tagging, a standard set needs to be chosen. We Could pick very simple/coarse tagsets such as Noun (NN), Verb (VB), Adjective (JJ), Adverb (RB), etc.
- But to make tags more dis-ambiguous, the commonly used set is finer-grained, University of Pennsylvania's "UPenn TreeBank tagset", having a total of 45 tags.

Tag	Description	Example	Tag	Description	Example
CC	Coordin. Conjunction	and, but, or	SYM	Symbol	+%, &
CD	Cardinal number	one, two, three	TO	"to"	to
DT	Determiner	a, the	UH	Interjection	ah, oops
EX	Existential 'there'	there	VB	Verb, base form	eat
FW	Foreign word	mea culpa	VBD	Verb, past tense	ate
IN	Preposition/sub-conj	of, in, by	VBG	Verb, gerund	eating
JJ	Adjective	yellow	VBN	Verb, past participle	eaten
JJR	Adj., comparative	bigger	VBP	Verb, non-3 sg pres	eat
JJS	Adj., superlative	wildest	VBZ	Verb, 3 sg pres	eats
LS	List item marker	1, 2, One	WDT	Wh-determiner	which, that
MD	Modal	can, should	WP	Wh-pronoun	what, who
NN	Noun, sing. or mass	llama	WP$	Possessive wh-	whose
NNS	Noun, plural	llamas	WRB	Wh-adverb	how, where
NNP	Proper noun, singular	IBM	$	Dollar sign	$
NNPS	Proper noun, plural	Carolinas	#	Pound sign	#
PDT	Predeterminer	all, both	"‘	Left quote	(or)
POS	Possessive ending	's	"	Right quote	(or)
PRP	Personal pronoun	I, you, he	(	Left parenthesis	([,(,{,<)
PRP$	Possessive pronoun	your, one's	)	Right parenthesis	(],),},>)
RB	Adverb	quickly, never	,	Comma	
RBR	Adverb, comparative	faster	.	Sentence-final punc	(. ! ?)
RBS	Adverb, superlative	fastest	:	Mid-sentence punc	(: ; ...)
RP	Particle	up, off			

****

6. Parts of Speech Tagging (PoS)

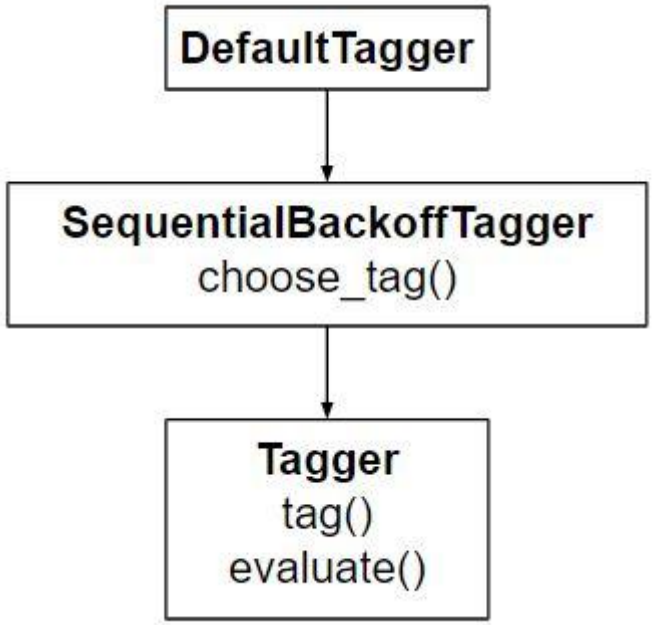
- One of the core tasks in Natural Language Processing (NLP) is Parts of Speech (PoS) tagging, which is giving each word in a text a grammatical category, such as nouns, verbs, adjectives, and adverbs.
- Through improved comprehension of phrase structure and semantics, this technique makes it possible for machines to study and comprehend human language more accurately.
- In many NLP applications, including machine translation, sentiment analysis, and information retrieval, PoS tagging is essential.
- PoS tagging serves as a link between language and machine understanding, enabling the creation of complex language processing systems and serving as the foundation for advanced linguistic analysis.

What is POS(Parts-Of-Speech) Tagging?

- Parts of Speech tagging is a linguistic activity in Natural Language Processing (NLP) wherein each word in a document is given a particular part of speech (adverb, adjective, verb, etc.) or grammatical category.
- Through the addition of a layer of syntactic and semantic information to the words, this procedure makes it easier to comprehend the sentence's structure and meaning.
- In NLP applications, POS tagging is useful for machine translation, named entity recognition, and information extraction, among other things.
- It also works well for clearing out ambiguity in terms with numerous meanings and revealing a sentence's grammatical structure.

Part of Speech	Tag
Noun	n
Verb	v
Adjective	a
Adverb	r

- **Default tagging** is a basic step for the part-of-speech tagging. It is performed using the DefaultTagger class.
- The DefaultTagger class takes 'tag' as a single argument. NN is the tag for a singular noun.
- DefaultTagger is most useful when it gets to work with most common part-of-speech tag. that's why a noun tag is recommended.



Example of POS Tagging

Consider the sentence: "The quick brown fox jumps over the lazy dog."

After performing POS Tagging:

- “The” is tagged as determiner (DT)
- “quick” is tagged as adjective (JJ)
- “brown” is tagged as adjective (JJ)
- “fox” is tagged as noun (NN)
- “jumps” is tagged as verb (VBZ)
- “over” is tagged as preposition (IN)
- “the” is tagged as determiner (DT)
- “lazy” is tagged as adjective (JJ)
- “dog” is tagged as noun (NN)
- By offering insights into the grammatical structure, this tagging aids machines in comprehending not just individual words but also the connections between them inside a phrase.
- For many NLP applications, like text summarization, sentiment analysis, and machine translation, this kind of data is essential.

Workflow of POS Tagging in NLP

The following are the processes in a typical natural language processing (NLP) example of part-of-speech (POS) tagging:

- **Tokenization**: Divide the input text into discrete tokens, which are usually units of words or subwords. The first stage in NLP tasks is tokenization.
- **Loading Language Models:** To utilize a library such as NLTK or SpaCy, be sure to load the relevant language model. These models offer a foundation for comprehending a language’s grammatical structure since they have been trained on a vast amount of linguistic data.
- **Text Processing**: If required, preprocess the text to handle special characters, convert it to lowercase, or eliminate superfluous information. Correct PoS labeling is aided by clear text.
- **Linguistic Analysis**: To determine the text’s grammatical structure, use linguistic analysis. This entails understanding each word’s purpose inside the sentence, including whether it is an adjective, verb, noun, or other.
- **Part-of-Speech Tagging:** To determine the text’s grammatical structure, use linguistic analysis. This entails understanding each word’s purpose inside the sentence, including whether it is an adjective, verb, noun, or other.
- **Results Analysis:** Verify the accuracy and consistency of the PoS tagging findings with the source text. Determine and correct any possible problems or mistagging.

Types of POS Tagging in NLP

- Assigning grammatical categories to words in a text is known as Part-of-Speech (PoS) tagging, and it is an essential aspect of Natural Language Processing (NLP).
- Different PoS tagging approaches exist, each with a unique methodology. Here are a few typical kinds:

1. **Rule-Based Tagging**

- Rule-based part-of-speech (POS) tagging involves assigning words their respective parts of speech using predetermined rules, contrasting with machine learning-based POS tagging that requires training on annotated text corpora.
- In a rule-based system, POS tags are assigned based on specific word characteristics and contextual cues.

- For instance, a rule-based POS tagger could designate the “noun” tag to words ending in “-tion” or “-ment,” recognizing common noun-forming suffixes.
- This approach offers transparency and interpretability, as it doesn’t rely on training data.
- Let’s consider an example of how a rule-based part-of-speech (POS) tagger might operate:
- **Rule:** Assign the POS tag “noun” to words ending in “-tion” or “-ment.”
- **Text:** “The presentation highlighted the key achievements of the project’s development.”

Rule based Tags:

- “The” – Determiner (DET)
- “presentation” – Noun (N)
- “highlighted” – Verb (V)
- “the” – Determiner (DET)
- “key” – Adjective (ADJ)
- “achievements” – Noun (N)
- “of” – Preposition (PREP)
- “the” – Determiner (DET)
- “project’s” – Noun (N)
- “development” – Noun (N)

In this instance, the predetermined rule is followed by the rule-based POS tagger to label words. “Noun” tags are applied to words like “presentation,” “achievements,” and “development” because of the aforementioned restriction. Despite the simplicity of this example, rule-based taggers may handle a broad variety of linguistic patterns by incorporating different rules, which makes the tagging process transparent and comprehensible.

2. **Transformation Based tagging**

- Transformation-based tagging (TBT) is a part-of-speech (POS) tagging method that uses a set of rules to change the tags that are applied to words inside a text.
- In contrast, statistical POS tagging uses trained algorithms to predict tags probabilistically, while rule-based POS tagging assigns tags directly based on predefined rules.
- To change word tags in TBT, a set of rules is created depending on contextual information.
- A rule could, for example, change a verb’s tag to a noun if it comes after a determiner like “the.” The text is systematically subjected to these criteria, and after each transformation, the tags are updated.
- When compared to rule-based tagging, TBT can provide higher accuracy, especially when dealing with complex grammatical structures.
- To attain ideal performance, nevertheless, it might require a large rule set and additional computer power.
- Consider the transformation rule: Change the tag of a verb to a noun if it follows a determiner like “the.”

Text: “The cat chased the mouse”.

Initial Tags:

- “The” – Determiner (DET)
- “cat” – Noun (N)
- “chased” – Verb (V)
- “the” – Determiner (DET)
- “mouse” – Noun (N)

Transformation rule applied:
Change the tag of "chased" from Verb (V) to Noun (N) because it follows the determiner "the."
Updated tags:
- "The" – Determiner (DET)
- "cat" – Noun (N)
- "chased" – Noun (N)
- "the" – Determiner (DET)
- "mouse" – Noun (N)

In this instance, the tag "chased" was changed from a verb to a noun by the TBT system using a transformation rule based on the contextual pattern. The tagging is updated iteratively and the rules are applied sequentially. Although this example is simple, given a well-defined set of transformation rules, TBT systems can handle more complex grammatical patterns.

3. **Statistical POS Tagging**

- Utilizing probabilistic models, statistical part-of-speech (POS) tagging is a computer linguistics technique that places grammatical categories on words inside a text.
- If rule-based tagging uses massive annotated corpora to train its algorithms, statistical tagging uses machine learning.
- In order to capture the statistical linkages present in language, these algorithms learn the probability distribution of word-tag sequences. CRFs (conditional random fields) and Hidden Markov Models (HMMs) are popular models for statistical point-of-sale classification.
- The algorithm estimates the chance of observing a specific tag given the current word and its context by learning from labeled samples during training.
- The most likely tags for text that hasn't been seen are then predicted using the trained model. Statistical POS tagging works especially well for languages with complicated grammatical structures because it is exceptionally good at handling linguistic ambiguity and catching subtle language trends.
- **Hidden Markov Model POS tagging:** Hidden Markov Models (HMMs) serve as a statistical framework for part-of-speech (POS) tagging in natural language processing (NLP).
- In HMM-based POS tagging, the model undergoes training on a sizable annotated text corpus to discern patterns in various parts of speech. Leveraging this training, the model predicts the POS tag for a given word based on the probabilities associated with different tags within its context.
- Comprising states for potential POS tags and transitions between them, the HMM-based POS tagger learns transition probabilities and word-emission probabilities during training.
- To tag new text, the model, employing the Viterbi algorithm, calculates the most probable sequence of POS tags based on the learned probabilities.
- Widely applied in NLP, HMMs excel at modeling intricate sequential data, yet their performance may hinge on the quality and quantity of annotated training data.

Advantages of POS Tagging

There are several advantages of Parts-Of-Speech (POS) Tagging including:

- **Text Simplification:** Breaking complex sentences down into their constituent parts makes the material easier to understand and easier to simplify.
- **Information Retrieval:** Information retrieval systems are enhanced by point-of-sale (POS) tagging, which allows for more precise indexing and search based on grammatical categories.

- **Named Entity Recognition:** POS tagging helps to identify entities such as names, locations, and organizations inside text and is a precondition for named entity identification.
- **Syntactic Parsing:** It facilitates syntactic parsing, which helps with phrase structure analysis and word link identification.

Disadvantages of POS Tagging

Some common disadvantages in part-of-speech (POS) tagging include:

- **Ambiguity:** The inherent ambiguity of language makes POS tagging difficult since words can signify different things depending on the context, which can result in misunderstandings.
- **Idiomatic Expressions:** Slang, colloquialisms, and idiomatic phrases can be problematic for POS tagging systems since they don't always follow formal grammar standards.
- **Out-of-Vocabulary Words:** Out-of-vocabulary words (words not included in the training corpus) can be difficult to handle since the model might have trouble assigning the correct POS tags.
- **Domain Dependence:** For best results, POS tagging models trained on a single domain should have a lot of domain-specific training data because they might not generalize well to other domains.

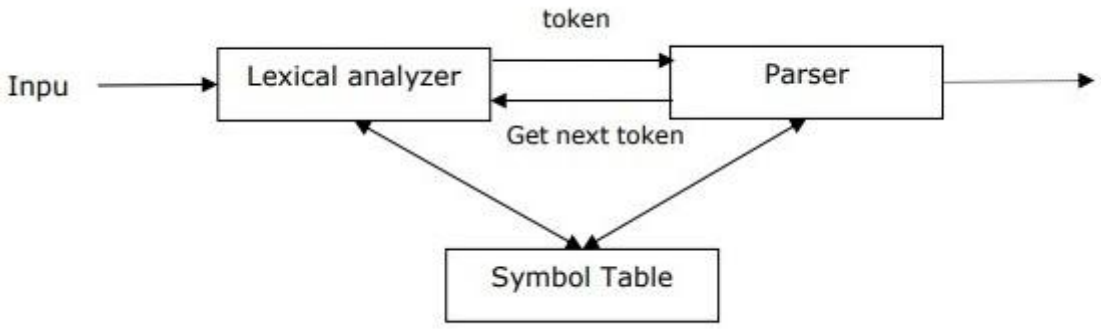
****

7. SYNTACTIC ANALYSIS

- Syntactic analysis or parsing or syntax analysis is the third phase of NLP. The purpose of this phase is to draw exact meaning, or you can say dictionary meaning from the text.
- Syntax analysis checks the text for meaningfulness comparing to the rules of formal grammar.
- For example, the sentence like "hot ice-cream" would be rejected by semantic analyzer.
- In this sense, syntactic analysis or parsing may be defined as the process of analyzing the strings of symbols in natural language conforming to the rules of formal grammar.
- The origin of the word 'parsing' is from Latin word 'pars' which means 'part'.

7.1.Concept of Parser

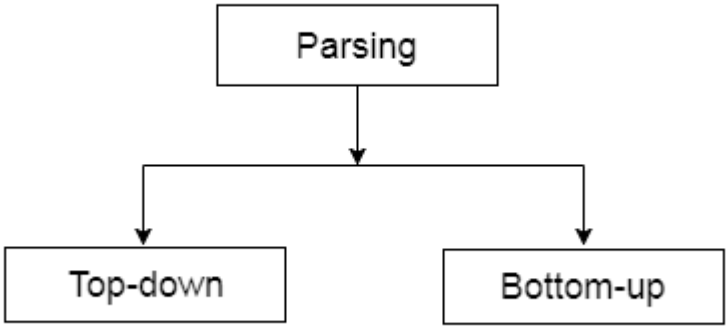
- It is used to implement the task of parsing. It may be defined as the software component designed for taking input data (text) and giving structural representation of the input after checking for correct syntax as per formal grammar.
- It also builds a data structure generally in the form of parse tree or abstract syntax tree or other hierarchical structure.



The main roles of the parse include −

- To report any syntax error.
- To recover from commonly occurring error so that the processing of the remainder of program can be continued.
- To create parse tree.
- To create symbol table.
- To produce intermediate representations (IR).

Types of Parsing
Derivation divides parsing into the followings two types



- Top-down Parsing
- Bottom-up Parsing

i. Top-down Parsing

- A top-down parser is a type of parser used in compiler design and language processing. It works by starting at the highest level of grammar (usually called the "start symbol") and breaking it down step by step into its components, trying to match the structure of the input.

What is a Top Down Parser?

- In the top-down technique, parse tree constructs from the top, and input will read from left to right.
- In a top-down parser, It will start the symbol to proceed to the string or input. It follows left most derivation.
- In a top-down parser, difficulty with top-down parser is if the variable contains more than one possibility selecting 1 is difficult.

Working of Top Down Parser

Let's consider an example where grammar is given and you need to construct a parse tree by using a top-down parser technique.

Example:

S -> aABe
A -> Abc | b
B -> d
Now, let's consider the input to read and construct a parse tree with top-down approach.

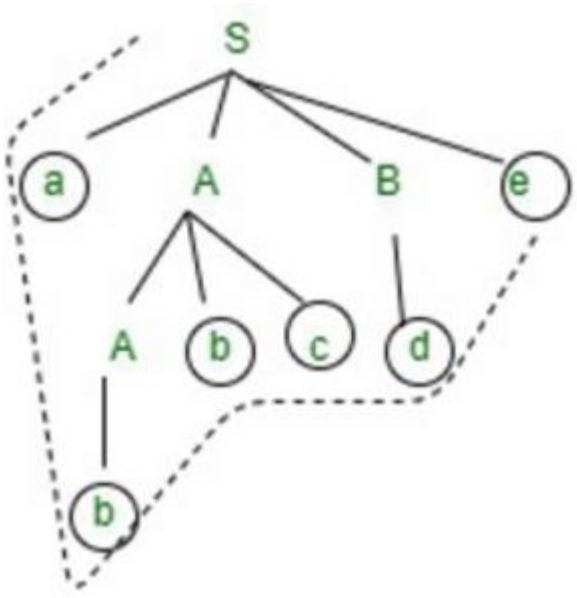
Input

abbcde

Now, you will see how top-down approach works. Here, you will see how you can generate a input string from the grammar for top top-down approach.

- First, you can start with S -> a A B e and then you will see input string **'a'** in the beginning and **'e'** in the end.
- Now, you need to generate ~~a~~bbcd~~e~~.
- Expand A-> Abc and Expand B-> d.
- Now, You have a string like aAbcde and your input string is abbcde.
- Expand A->b.
- Final string, you will get **abbcde**.

Given below is the Diagram explanation for constructing a top-down parse tree. You can see clearly in the diagram how you can generate the input string using grammar with top-down approach.



ii. **Bottom-up Parsing**

- In this kind of parsing, the parser starts with the input symbol and tries to construct the parser tree up to the start symbol.
- It will start from string and proceed to start. In Bottom-up parser, Identifying the correct handle (substring) is always difficult. It will follow rightmost derivation in reverse order.

Working of Bottom-up parser:

Let's consider an example where grammar is given and you need to construct a parse tree by using bottom-up parser technique.

Example –

S -> aABe
A -> Abc | b
B -> d

Now, let’s consider the input to read and to construct a parse tree with bottom-up approach.

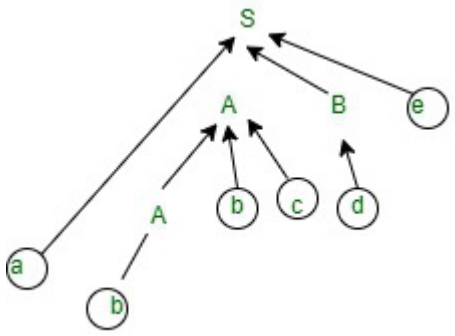
Input

abbcde$

Now, you will see that how bottom-up approach works. Here, you will see how you can generate a input string from the grammar for bottom-up approach.

- First, you can start with A -> b.
- Now, expand A -> Abc.
- After that Expand B-> d.
- In the last, just expand the S -> aABe
- Final string, you will get abbcde.

Given below is the Diagram explanation for constructing bottom-up parse tree. You can see clearly in the diagram how you can generate the input string using grammar with bottom-up approach.



7.2.Concept of Derivation

- Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:
- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.
- We have two options to decide which non-terminal to be replaced with production rule.

Types of Derivation

In this section, we will learn about the two types of derivations, which can be used to decide which non-terminal to be replaced with production rule −

a. Left-most Derivation

In the left-most derivation, the sentential form of an input is scanned and replaced from the left to the right. The sentential form in this case is called the left-sentential form.

Example:
Production rules:

S = S + S
S = S - S
S = a | b |c

Input:

a - b + c

The left-most derivation is:

S = S + S
S = S - S + S
S = a - S + S
S = a - b + S
S = a - b + c

b. Right-most Derivation

In the left-most derivation, the sentential form of an input is scanned and replaced from right to left. The sentential form in this case is called the right-sentential form.

Example:
S = S + S
S = S - S
S = a | b |c

Input:

a - b + c

The right-most derivation is:

S = S - S
S = S - S + S
S = S - S + c
S = S - b + c
S = a - b + c

7.3.Concept of Parse Tree

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It is the graphical representation of symbol that can be terminals or non-terminals.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

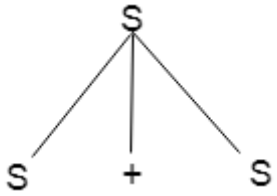
Example:

Production rules:
T= T + T | T * T
T = a|b|c

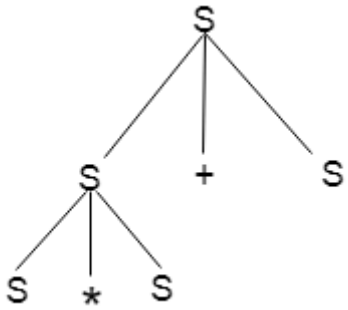
Input:

a * b + c

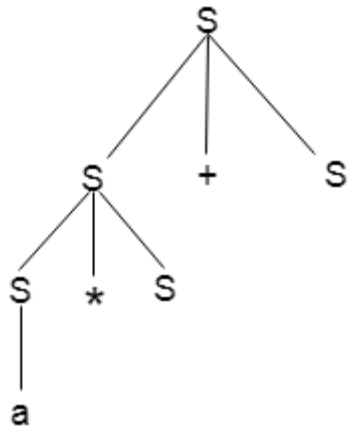
Step 1:



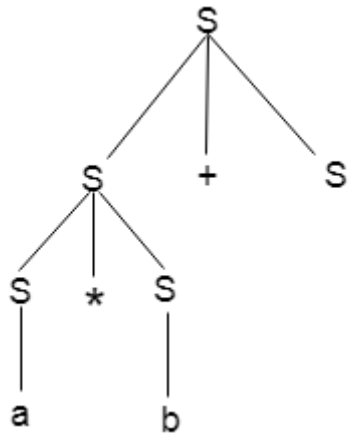
Step 2:



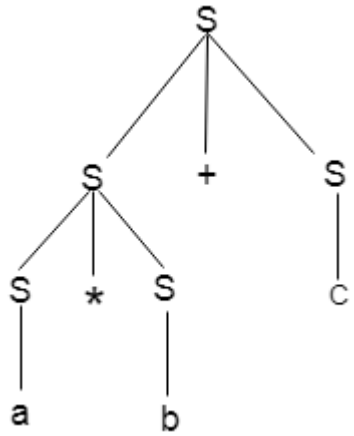
Step 3:



Step 4:



Step 5:



****

8. Grammar in NLP

- **Grammar** is a formal system that defines a set of rules for generating valid strings within a language.

- It serves as a blueprint for constructing syntactically correct sentences or meaningful sequences in a formal language.

Grammar is basically composed of two basic elements:

- **Terminal Symbols:** Terminal symbols are those that are the components of the sentences generated using grammar and are represented using small case letters like a, b, c, etc.
- **Non-Terminal Symbols:** Non-terminal symbols are those symbols that take part in the generation of the sentence but are not the component of the sentence. Non-Terminal Symbols are also called Auxiliary Symbols and Variables. These symbols are represented using a capital letters like A, B, C, etc.

Representation of Grammar

Any Grammar can be represented by 4 tuples – <N, T, P, S>

- **N –** Finite Non-Empty Set of Non-Terminal Symbols.
- **T –** Finite Set of Terminal Symbols.
- **P –** Finite Non-Empty Set of Production Rules.
- **S –** Start Symbol (Symbol from where we start producing our sentences or strings).

Production Rules

- A production or production rule in computer science is a rewrite rule specifying a symbol substitution that can be recursively performed to generate new symbol sequences.
- It is of the form α-> β where α is a Non-Terminal Symbol which can be replaced by β which is a string of Terminal Symbols or Non-Terminal Symbols.

Example-1: Consider Grammar G1 = <N, T, P, S>

```
T = {a,b}    #Set of terminal symbols
P = {A->Aa,A->Ab,A->a,A->b,A-> ϵϵ}    #Set of all production rules
S = {A}    #Start Symbol
```

As the start symbol is S then we can produce Aa, Ab, a,b,ϵ ϵ which can further produce strings where A can be replaced by the Strings mentioned in the production rules and hence this grammar can be used to produce strings of the form (a+b)*.

Derivation of Strings

```
A->a    #using production rule 3
OR
A->Aa    #using production rule 1
Aa->ba    #using production rule 4
OR
A->Aa    #using production rule 1
Aa->AAa    #using production rule 1
AAa->bAa    #using production rule 4
bAa->ba    #using production rule 5
```

Example-2: Consider Grammar G2 = <N, T, P, S>

```
N = {A}    #Set of non-terminals Symbols
T = {a}    #Set of terminal symbols
P = {A->Aa, A->AAa, A->a, A->ϵϵ}    #Set of all production rules
S = {A}    #Start Symbol
```

As the start symbol is S then we can produce Aa, AAa, a,ϵ ϵ which can further produce strings where A can be replaced by the Strings mentioned in the production rules and hence this grammar can be used to produce strings of form (a)*.

Derivation of Strings

A->a #using production rule 3
OR
A->Aa #using production rule 1
Aa->aa #using production rule 3
OR
A->Aa #using production rule 1
Aa->AAa #using production rule 1
AAa->Aa #using production rule 4
Aa->aa #using production rule 3

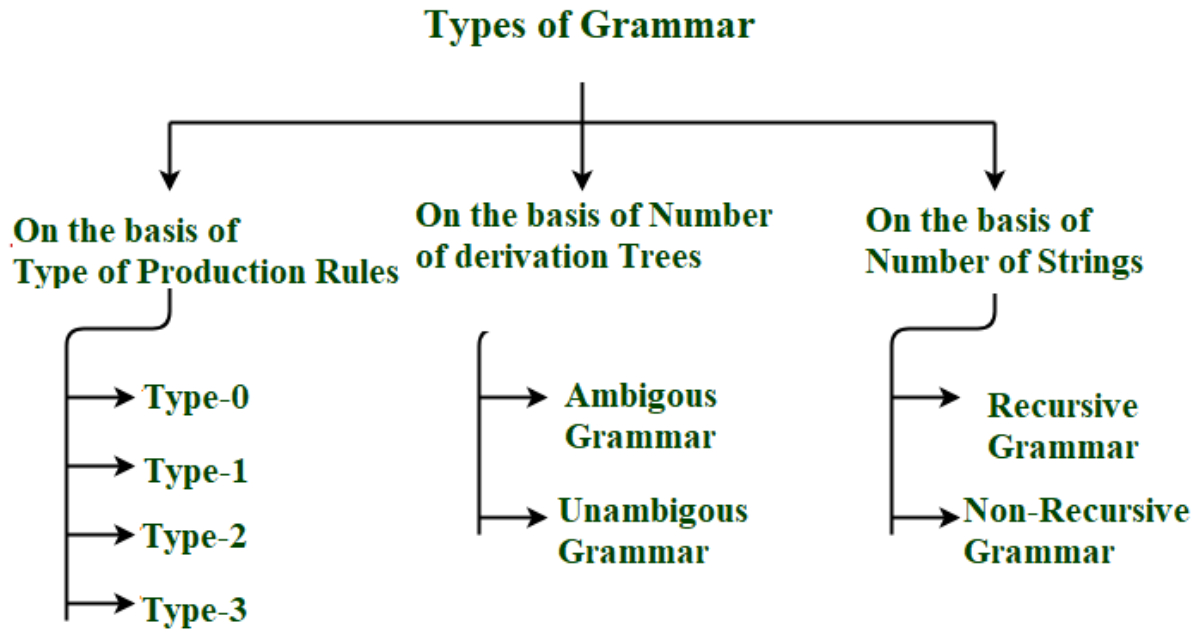
Equivalent Grammars

Two grammars are said to be equivalent if they generate the same language. For instance, if Grammar 1 and Grammar 2 both generate strings of the form $(a+b)*$, they are considered equivalent.

Types of Grammars

There are several types of Grammar. We classify them on the basis mentioned below.

- **Type of Production Rules:** The form and complexity of the production rules define the grammar type, such as context-free, context-sensitive, regular, or unrestricted grammars.
- **Number of Derivation Trees**: The number of ways a string can be derived from the grammar. Ambiguous grammars have multiple derivation trees for the same string.
- **Number of Strings**: The size and nature of the language generated by the grammar.



****

9. CONTEXT FREE GRAMMAR

A **context-free grammar (CFG)** is a formal system used to describe a class of languages known as **context-free languages (CFLs)**. purpose of context-free grammar is:

- To list all strings in a language using a set of rules (production rules).
- It extends the capabilities of regular expressions and finite automata.

A grammar is said to be the Context-free grammar if every production is in the form of:
G -> (V∪T), where G ∈ V*

V (Variables/Non-terminals): *These are symbols that can be replaced using production rules. They help in defining the structure of the grammar. Typically, non-terminals are represented by uppercase letters (e.g., S, A, B).*
T (Terminals): *These are symbols that appear in the final strings of the language and cannot be replaced further. They are usually represented by lowercase letters (e.g., a, b, c) or specific symbols.*
The left-hand side can only be a Variable, it cannot be a terminal.
But on the right-hand side here it can be a Variable or Terminal or both combination of Variable and Terminal.

The above equation states that every production which contains any combination of the 'V' variable or 'T' terminal is said to be a context-free grammar.

Core Concepts of CFGs

A CFG is defined by:

1. **Nonterminal symbols (variables):** Represent abstract categories or placeholders (e.g., E,SE, SE,S).
2. **Terminal symbols (alphabet):** The actual characters or tokens in the language (e.g., a,b,+,∗,(,)a, b, +, *, (,)a,b,+,∗,(,)).
3. **Production rules:** Specify how nonterminals can be replaced with other nonterminals or terminals (e.g., E→E+EE → E + EE→E+E).
4. **Start symbol:** A special nonterminal from which derivations begin.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

CFG vs. Other Models

Model	Description
Finite Automata	Accept strings via computation (accept/reject).
Regular Expressions	Match strings by describing their structure.
CFG	Generate strings via recursive replacement.

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

S → aSa
S → bSb
S → c

Now check that abbcbba string can be derived from the given CFG.

S ⇒ aSa
S ⇒ abSba

S ⇒ abbSbba
S ⇒ abbcbba
By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

Capabilities of CFG

There are the various capabilities of CFG:

- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficient parser can be constructed automatically.
- Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

****

10. PROBABILISTIC PARSING IN NLP

- Probabilistic parsing in Natural Language Processing (NLP) refers to the process of analysing a sentence's syntactic structure using probability models.
- It involves assigning probabilities to different possible parse trees (grammatical structures) for a sentence and selecting the most likely one based on statistical models.

Key Concepts of Probabilistic Parsing

1. **Probabilistic Context-Free Grammar (PCFG)**
   - PCFG is an extension of Context-Free Grammar (CFG), where each production rule is assigned a probability.
   - The probability of a parse tree is the product of the probabilities of the rules used in generating it.
2. **Parse Trees & Probability Assignment**
   - Given a sentence, multiple parse trees can be generated.
   - The probability of a parse tree is computed as: P(T)=∏P(rule)
   - Where each rule contributes to the overall likelihood.
3. **Maximum Likelihood Estimation (MLE)**
   - Rule probabilities are estimated from a treebank (a corpus of parsed sentences) using MLE: $P(A \rightarrow BC) = \frac{Count(A \rightarrow BC)}{\sum Count(A \rightarrow XY)}$
   - This approach helps determine which parse trees are more common.

Types of Probabilistic Parsers

1. **Probabilistic CYK (Cocke–Younger–Kasami) Parser**
   - A dynamic programming algorithm that uses PCFG to find the most probable parse.
2. **Inside-Outside Algorithm**
   - A probabilistic parsing technique used for training PCFGs in an unsupervised manner.
3. **Chart Parsing with Probabilities**
   - Uses a chart data structure to store intermediate probabilities, improving efficiency.
4. **Dependency Parsing with Probabilities**

- Instead of phrase structures, assigns probabilities to dependency relations between words.

Applications of Probabilistic Parsing

- Machine translation
- Speech recognition
- Information extraction
- Named entity recognition
- Question answering systems

****