

MODULE – 2

Word Level Analysis & Syntactic Analysis

Word Level Analysis: Regular Expressions, Finite-State Automata, Morphological Parsing, Spelling Error Detection and Correction, Words and Word Classes, Part-of Speech Tagging.

Syntactic Analysis: Context-Free Grammar, Constituency, Top-down and Bottom-up Parsing, CYK Parsing.

Textbook 1: Ch. 3, Ch. 4.

Word Level Analysis

1. Introduction

Processing carried out at word level, including methods for characterizing word sequences, identifying morphological variants, detecting and correcting misspelled words, and identifying correct part-of-speech of a word.

1.1 The part-of-speech tagging methods:

1. Rule-based (linguistic).
2. Stochastic (data-driven).
3. Hybrid.

1.2 Regular expressions: standard notations for describing text patterns.

1.3 Implementation Regular expressions using finite-state automaton (FSA): applications in speech recognition and synthesis, spell checking, and information extraction.

1.4 Detecting and correcting errors.

2. Regular Expressions (regexes)

- Pattern-matching standard for string parsing and replacement.
- Powerful way to find and replace strings that take a defined format.
- They are useful tools for the design of language compilers.
- Used in NLP for tokenization, describing lexicons, morphological analysis, etc..
- Perl was the first language that provided integrated support for regular expressions.
 - It uses a slash “/” around each regular expression;
- A regular expression is an algebraic formula whose value is a pattern consisting of a set of strings, called the language of the expression. Example: /a/

Some simple regular expressions: First instance of each match is underlined in table

Regular expression	Example patterns
/book/	The world is a <u>book</u> , and those who do not travel read only one page.

/book/	Reporters, who do not read the stylebook, should not criticize their editors.
/face/	Not everything that is faced can be changed. But nothing can be changed until it is faced.
/a/	Reason, Observation, and Experience-the Holy Trinity of Science.

2.1 Character Classes

Characters are grouped by square brackets, matching one character from the class. For example, `/[abcd]/` matches a, b, c, or d, and `/[0123456789]/` matches any digit. A dash specifies a range, like `/[5-9]/` or `/[m-p]/`. The caret at the start of a class negates the match, as in `/[^x]/`, which matches any character except x. The caret is interpreted literally elsewhere.

Use of square brackets

RE	Match	Example patterns Matched
[abc]	Match any of a, b, and c	'Refresher course will start tomorrow'
[A-Z]	Match any character between A and Z (ASCII order)	the course will end on Jan. 10, 2006'
[^A-Z]	Match any character other than an uppercase letter	'TREC Conference'
[^abc]	Match anything other than a, b, and c	'TREC Conference'
[+ *?.]	Match any of +, *, ?, or the dot.	'3 +2 = 5'
[a^]	Match a or ^	'^has different uses.'

- Regular expressions are case-sensitive (e.g., `/s/` matches 's', not 'S').
- Use square brackets to handle case differences, like `/[sS]/`.
 - `/[sS]ana/` matches 'sana' or 'Sana'.
- The question mark (?) makes the previous character optional (e.g., `/supernovas?/`).
- The * allows zero or more occurrences (e.g., `/b*/`).
- `/[ab]*/` matches zero or more occurrences of 'a' or 'b'.
- The + specifies one or more occurrences (e.g., `/a+/`).
- `/[0-9]*/` matches a sequence of one or more digits.
- The caret (^) anchors the match at the start, and \$ at the end of a line.
 - `/^The nature\.$/` will search exactly for this line.
- The dot (.) is a wildcard matching any single character (e.g., `/./`).
 - Expression `/at/` matches with any of the string cat, bat, rat, gat, kat, mat, etc.

Special characters

RE	Description
.	The dot matches any single character.
\n	Matches a new line character (or CR+LF combination).

\t	Matches a tab (ASCII 9).
\d	Matches a digit [0-9].
\D	Matches a non-digit.
\w	Matches an alphanumeric character.
\W	Matches a non-alphanumeric character.
\s	Matches a whitespace character.
\S	Matches a non-whitespace character.
\	Use \ to escape special characters. For example, \. matches a dot, * matches a * and \\ matches a backslash.

- The wildcard symbol can count characters, e.g., /.....berry/ matches ten-letter strings ending in "berry".
- This matches "strawberry", "sugarberry", but not "blueberry" or "hackberry".
- To search for "Tanveer" or "Siddiqui", use the disjunction operator (), e.g., "Tanveer|Siddiqui".
- The pipe symbol matches either of the two patterns.
- Sequences take precedence over disjunction, so parentheses are needed to group patterns.
- Enclosing patterns in parentheses allows disjunction to apply correctly.

Example: Suppose we need to check if a string is an email address or not.

$^{[A-Za-z0-9_ \ .-]^{+}@[A-Za-z0-9_ \ .-]^{+}[A-Za-z0-9_][A-Za-z0-9_]\$}$

Pattern	Description
$^{[A-Za-z0-9_ \ .-]^{+}}$	Match a positive number of acceptable characters at the start of the string.
@	Match the @ sign.
$[A-Za-z0-9_ \ .-]^{+}$	Match any domain name, including a dot.
$[A-Za-z0-9_][A-Za-z0-9_]\$$	Match two acceptable characters but not a dot. This ensures that the email address ends with .xx, .xxx, .xxxx, etc.

- The language of regular expressions is similar to formulas of Boolean logic.
- Regular languages may be encoded as finite state networks.
- A regular expression can contain symbol pairs, e.g., /a:b/, which represents a relation between two strings.
- Regular languages can be encoded using **finite-state automata** (FSA), making it easier to manipulate and process regular languages, which in turn aids in **handling more complex languages like context-free languages**.

3. Finite-State Automata

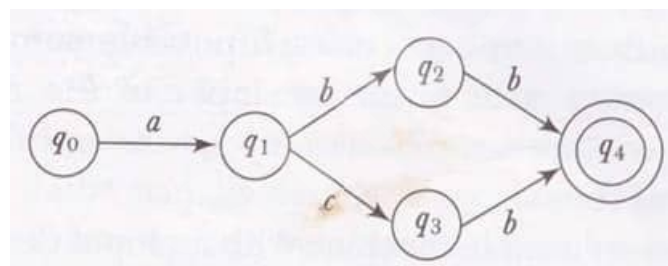
- **Game Description:** The game involves a board with pieces, dice or a wheel to generate random numbers, and players rearranging pieces based on the number. There's no skill or choice; the game is entirely based on random numbers.

- **States:** The game progresses through various states, starting from the initial state (beginning positions of pieces) to the final state (winning positions).
- **Machine Analogy:** A machine with input, memory, processor, and output follows a similar process: it starts in an initial state, changes to the next state based on the input, and eventually reaches a final state or gets stuck if the next state is undefined.
- **Finite Automaton:** This model, with finite states and input symbols, describes a machine that automatically changes states based on the input, and it's deterministic, meaning the next state is fully determined by the current state and input.

A finite automaton has the following properties:

1. A finite set of states, one of which is designated the initial or start state, and one or more of which are designated as the final states.
2. A finite alphabet set, Σ , consisting of input symbols.
3. A finite set of transitions that specify for each state and each symbol of the input alphabet, the state to which it next goes.

This finite-state automaton is shown as a directed graph, called transition diagram.



A deterministic finite -state automaton (DFA)

Let $\Sigma = \{a, b, c\}$, the set of states = $\{q_0, q_1, q_2, q_3, q_4\}$ with q_0 being the start state and q_4 the final state, we have the following rules of transition:

1. From state q_0 and with input a , go to state q_1 .
2. From state q_1 and with input b , go to state q_2 .
3. From state q_1 and with input c go to state q_3 .
4. From state q_2 and with input b , go to state q_4 .
5. From state q_3 and with input b , go to state q_4 .

A finite automaton can be deterministic or non-deterministic.

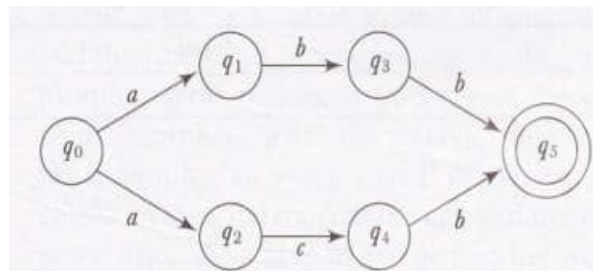
Deterministic Automata:

- The nodes in this diagram correspond to the states, and the arcs to transitions.
- The arcs are labelled with inputs.
- The final state is represented by a double circle.
- There is exactly one transition leading out of each state. Hence, this automaton is **deterministic**.
- Any regular expression can be represented by a finite automaton and the language of any finite automaton can be described by a regular expression.

- A **deterministic finite-state automaton (DFA)** is defined as a 5-tuple $(Q, \Sigma, \delta, S, F)$, where Q is a set of states, Σ is an alphabet, S is the start state, $F \subseteq Q$ is a set of final states, and δ is a transition function.
- The transition function δ defines mapping from $Q \times \Sigma$ to Q . That is, for each state q and symbol a , there is at most one transition possible

Non-Deterministic Automata:

- For each state, there can be more than one transition on a given symbol, each leading to a different state.
- This is shown in Figure, where there are two possible transitions from state q_0 on input symbol a .
- The transition function of a non-deterministic finite-state automaton (NFA) maps $Q \times (\Sigma \cup \{\epsilon\})$ to a subset of the power set of Q .



Non-deterministic finite-state automaton (NFA)

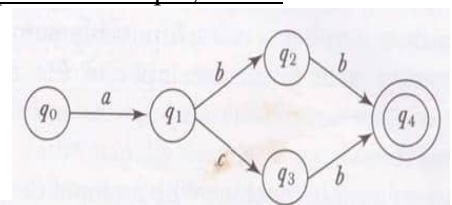
How it Works for Regex – NLP?

- A path is a sequence of transitions beginning with the start state.
- A path leading to one of the final states is a successful path.
- The FSAs encode regular languages.
- The language that an FSA encodes is the set of strings that can be formed by concatenating the symbols along each successful path.

Example:

1. Consider the deterministic automaton described in above example and the input, “***ac***”.

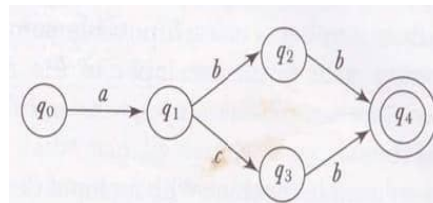
- We start with state q_0 and input symbol ***a*** and will go to state q_1 .
- The next input symbol is ***c***, we go to state q_3 .
- No more input is left and we have not reached the final state.
- Hence, the string ***ac*** is not recognized by the



automaton.

2. Now, consider the input

- we start with state q_0
- The next input symbol
- The next input symbol
- No more input is left and we have reached to final state.
- The string ***acb*** is a word of the language defined by the automaton.



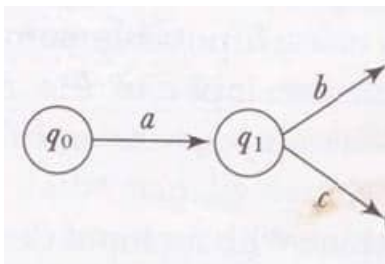
“acb”

and go to state q_1
is ***c***, so we go to state q_3 .
is ***b***, which leads to state q_4 .

State-transition table

- The rows in this table represent states and the columns correspond to input.
- The entries in the table represent the transition corresponding to a given state-input pair.

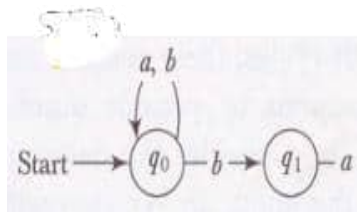
- A ϕ entry indicates missing transition.
- This table contains all the information needed by FSA.



Input			
State	a	b	c
Start: q0	q1	ϕ	ϕ
q1	ϕ	q2	q3
q2	ϕ	q4	ϕ
q3	ϕ	q4	ϕ
Final: q4	ϕ	ϕ	ϕ

Deterministic finite -state automaton (DFA) The state-transition table of DFA **Example**

- Consider a language consisting of all strings containing only **a's** and **b's** and ending with **baa**.
- We can specify this language by the regular expression **$\epsilon / (a|b)^*baa\$$** .
- The NFA implementing this regular expression is shown & state-transition table for the NFA is as shown below.

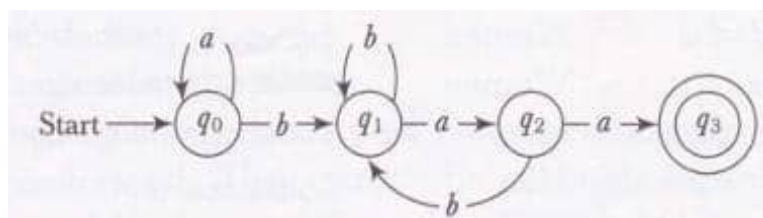


Input		
State	a	b
Start: q0	{q0}	{q0, q1}
q1	{q2}	ϕ
q2	{q3}	ϕ
Final: q3	ϕ	ϕ

NFA for **$\epsilon / (a|b)^*baa\$$**

State transition table

An NFA can be converted to an equivalent DFA and vice versa.



DFA for **$\epsilon / (a|b)^*baa\$$**

4. Morphological Parsing

- It is a sub-discipline of linguistics
- It studies word structure and the formation of words from **smaller units (morphemes)**.
- The goal of morphological parsing is to discover the morphemes that build a given word.
- A morphological parser should be able to tell us that the word 'eggs' is the plural form of the noun stem 'egg'.

Example:

The word 'bread' consists of a single morpheme.

'eggs' consist of two morphemes: the **egg** and **-s**

4.1 Two Broad classes of Morphemes:

1. Stems – **Main morpheme**, contains the central meaning.
2. Affixes – **modify the meaning** given by the stem.
 - Affixes are divided into prefix, suffix, infix, and circumfix.
 - 1. **Prefix** - morphemes which appear before a stem. (un-happy, be-waqt)
 - 2. **Suffix** - morphemes applied to the end. (ghodha-on, gurramu-lu, bidr-s, शीतलता)
 - 3. **Infixes** - morphemes that appear inside a stem.
 - **English slang word "abso-bloody-lutely."** The morpheme **"-bloody-"** is inserted into the stem "absolutely" to emphasize the meaning.
 - 4. **Circumfixes** - morphemes that may be applied to beginning & end of the stem.
 - *German word - **gespielt** (played) ⑦ ge+spiel+t*
Spiel – play (stem)

4.2 Three main ways of word formation: Inflection, Derivation, and Compounding

Inflection: a root word combined with a grammatical morpheme to **yield a word of the same class** as the original stem.

Ex. *play* (verb)+ *ed* (suffix) = *Played* (inflected form – past-tense)

Derivation: a root word combined with a grammatical morpheme to **yield a word belonging to a different class**.

Ex. *Compute* (verb)+*ion*=*Computation* (noun).

Care (noun)+ *ful* (suffix)= *careful* (adjective).

Compounding: The process of **merging two or more words** to form a new word.

Ex. Personal computer, desktop, overlook.

Morphological analysis and generation deal with inflection, derivation and compounding process in word formation and essential to many NLP applications:

1. Spelling corrections to machine translations.
2. In Information retrieval – to **identify the presence of a query word** in a document in spite of different morphological variants.

4.3 Morphological parsing:

It converts inflected words into their canonical form (**lemma**) with syntactical and morphological tags (e.g., tense, gender, number).

Morphological generation reverses this process, and both parsing and generation rely on a **dictionary of valid lemmas** and **inflection paradigms** for correct word forms.

A morphological parser uses following information sources:

1. **Lexicon:** A lexicon lists **stems and affixes** together with basic information about them.

2. **Morphotactics:** **Ordering** among the morphemes that constitute a word, It describes the way morphemes are arranged or touch each other. Ex. Rest-less-ness is a valid word, but not Rest-ness-less.
3. **Orthographic rules:** **Spelling rules** that specify the changes that occur when two given morphemes combine. Ex. 'easy' to 'easier' and not to 'easier'. (y → ier spelling rule)

Morphological analysis can be avoided if an exhaustive lexicon is available that lists features for all the word-forms of all the roots.

Ex. A sample lexicon entry:

Word form	Category	Root	Gender	Number	Person
Ghodhaa	Noun	GhoDaa	Masculine	Singular	3 _{rd}
Ghodhii	-do-	-do-	feminine	-do-	-do-
Ghodhon	-do-	-do-	Masculine	plural	-do-
Ghodhe	-do-	-do-	-do-	-do-	-do-

Limitations in Lexical entry:

1. It puts a heavy demand on memory.
2. Fails to show the relationship between different roots having similar word-forms.
3. Number of possible word-forms may be theoretically infinite (complex languages like Turkish).

4.4 Stemmers:

- The simplest morphological systems
- Collapse morphological variations of a given word (word-forms) to one lemma or stem.
- Stemmers do not use a lexicon; instead, they make use of rewrite rules of the form:
 - ier → y (e.g., earlier → early) ○
 - ing → ε (e.g., playing → play)
- Stemming algorithms work in two steps:

(i) Suffix removal: This step removes predefined endings from words.

(ii) Recoding: This step adds predefined endings to the output of the first step.

- Two widely used stemming algorithms have been developed by Lovins (1968) and Porter (1980).
 - Lovins's stemmer performs Suffix removal & Recoding sequentially
 - e.g. earlier ➔ first removes *ier* and recodes as *Early*
 - Porter's stemmer performs Suffix removal & Recoding simultaneously
 - e.g. ational → ate

To transform word such as 'rotational' into 'rotate'.

Limitations:

- It is difficult to use stemming with morphologically rich languages.
- E.g. Transformation of the word 'organization' into 'organ'
- It reduces only suffixes and prefixes; Compound words are not reduced. E.g. “toothbrush” or “snowball” can’t be broken.

A more efficient two-level morphological model – Koskenniemi (1983)

- Morphological parsing is **viewed as a mapping from the surface level into morpheme and feature sequences on the lexical level**.
- The surface level represents the actual spelling of the word.
- The lexical level represents the concatenation of its constituent morphemes.

e.g. '*playing*' is represented in the

Surface level ⑦ *play* + *V* + *PP*

Lexical level ⑦ '*play*' followed by the morphological information +*V*+*PP*

Surface Level ⑦

p	l	a	y	i	n	g
---	---	---	---	---	---	---

Lexical Level ⑦

Similarly, '*books*' is

p	l	a	y	+V	+PP
---	---	---	---	----	-----

represented in the lexical form as '*book* + *N* + *PL*'

This model is usually implemented with a kind of finite-state automata, called **finite-state transducer (FST)**.

Finite-state transducer (FST)

- FST maps an input word to its morphological components (root, affixes, etc.) and can also generate the possible forms of a word based on its root and morphological rules.
- An FST can be thought of as a two-state automaton, which recognizes or generates a pair of strings.

E.g. Walking

Analysis (Decomposition):

The **analyzer** uses a transducer that:

- Identifies the base form ("walk") from the surface form ("walking").
- Recognizes the suffix ("-ing") and removes it.

Generation (Synthesis):

The **generator** uses another transducer that:

- Identifies the base form ("walk") and applies the appropriate suffix to generate different surface forms, like "walked" or "walking".

A finite-state transducer is a 6-tuple $(\Sigma_1, \Sigma_2, Q, \delta, S, F)$, where Q is set of states, S is the initial state, and $F \subseteq Q$ is a set of final states, Σ_1 is input alphabet, Σ_2 is output alphabet, and δ is a function mapping $Q \times (\Sigma_1 \cup \{\epsilon\}) \times (\Sigma_2 \cup \{\epsilon\})$ to a subset of the power set of Q .

$$\delta: Q \times (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$$

Thus, an FST is similar to an NFA except in that transitions are made on strings rather than on symbols and, in addition, they have outputs. FSTs encode regular relations between regular languages, with the upper language on the top and the lower language on the bottom. For a transducer T and string s , $T(s)$ represents the set of strings in the relation. FSTs are closed under union, concatenation, composition, and Kleene closure, but not under intersection or complementation.

Two-step morphological parser

Two-level morphology using FSTs involves analyzing surface forms in two steps.

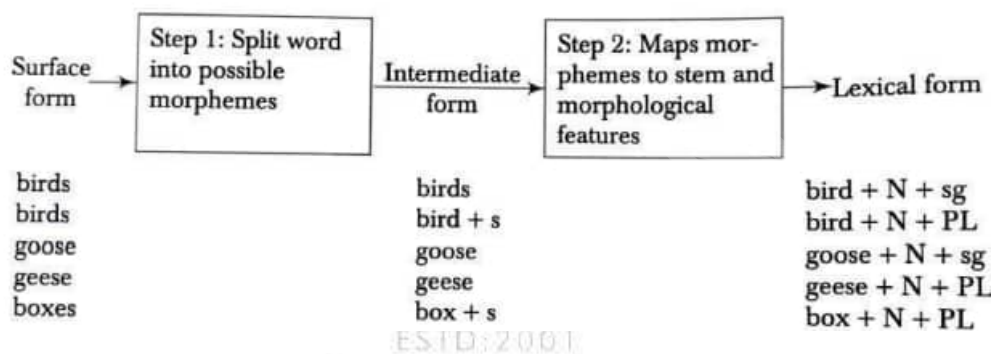


Fig. Two-step morphological parser

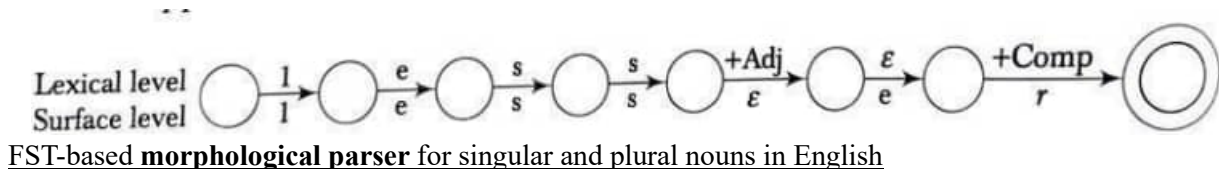
Step1: **Words are split into morphemes**, considering spelling rules and possible splits (e.g., "boxe + s" or "box + s").

Step2: The **output is a concatenation of stems and affixes**, with multiple representations possible for each word.

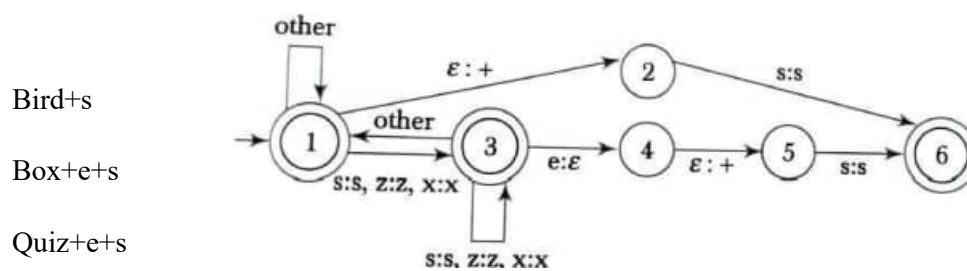
We need to build two transducers: one that maps the surface form to the intermediate form and another that **maps the intermediate form to the lexical form**.

A transducer maps the surface form "lesser" to its comparative form, where ϵ represents the empty string. This bi-directional FST can be used for both analysis (surface to base) and generation (base to surface).

E.g. Lesser

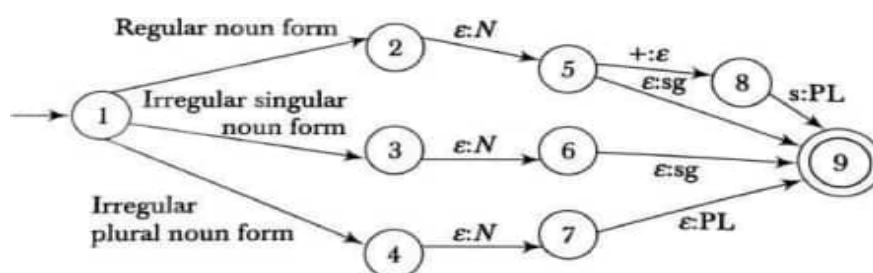


- The plural form of regular nouns usually ends with -s or -es. (not necessarily be the plural form – class, miss, bus).
- One of the required translations is the deletion of the 'e' when introducing a morpheme boundary.
 - E.g. Boxes, This deletion is usually required for words ending in xes, ses, zes.
- This is done by below transducer – Mapping English nouns to the intermediate form:



Mapping English nouns to the intermediate form

- The next step is to develop a transducer that does the mapping from the intermediate level to the lexical level. The input to transducer has one of the following forms:
 - Regular noun stem, e.g., bird, cat
 - Regular noun stem + s, e.g., bird + s
 - Singular irregular noun stem, e.g., goose
 - Plural irregular noun stem, e.g., geese
 - In the first case, the transducer has to map all symbols of the stem to themselves and then output N and sg.
 - In the second case, it has to map all symbols of the stem to themselves, but then output N and replaces PL with s.
 - In the third case, it has to do the same as in the first case.
 - Finally, in the fourth case, the transducer has to map the irregular plural noun stem to the corresponding singular stem (e.g., geese to goose) and then it should add N and PL.



Transducer for Step 2

The mapping from State 1 to State 2, 3, or 4 is carried out with the help of a transducer encoding a lexicon. The transducer implementing the lexicon maps the individual regular and irregular noun stems to their correct noun stem, replacing labels like regular noun form, etc.

This lexicon maps the surface form geese, which is an irregular noun, to its correct stem goose in the following way:

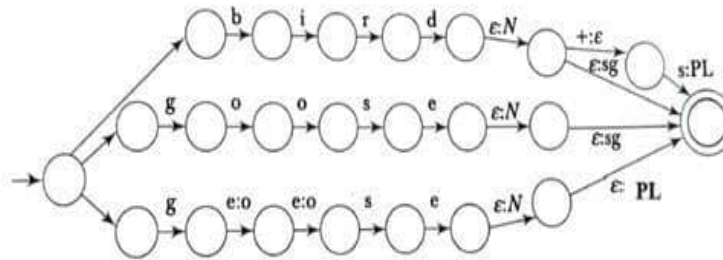
g:g e:o e:o s:s e:e

Mapping for the regular surface form of bird is b:b i:i r:r d:d. Representing pairs like a:a with a single letter, these two representations are reduced to g e:o e:o s e and b i r d respectively.

Composing this transducer with the previous one, we get a single two-level transducer with one input tape and one output tape. This maps plural nouns into the stem plus the morphological marker + pl and singular nouns into the stem plus the morpheme + sg. Thus a surface word form birds will be mapped to bird + N + pl as follows.

b:b i:i r:r d:d + ε:N + s:pl

Each letter maps to itself, while & maps to morphological feature +N, and s maps to morphological feature pl. Figure shows the resulting composed transducer.



A transducer mapping nouns to their stem and morphological features

1. Spelling Error Detection and Correction

Typing mistakes: single character omission, insertion, substitution, and reversal are the most common typing mistakes.

- Omission: When a single character is missed - 'concpt'
- Insertion error: Presence of an extra character in a word - 'error' is misspell as 'errorm'
- Substitution error: When a wrong letter is typed in place of the right one - 'errpr'
- Reversal: A situation in which the sequence of characters is reversed - 'aer' instead of 'are'.

OCR errors: Usually grouped into five classes: substitution, multi-substitution (or framing), space deletion or insertion, and failures.

Substitution errors: Caused due to visual similarity (single character) such as $c \rightarrow e$, $l \rightarrow i$, $r \rightarrow n$.

The same is true for **multi-substitution (two or more chars)**, e.g., $m \rightarrow rn$.

Failure occurs when the **OCR algorithm fails to select a letter** with sufficient accuracy. Solution:

These errors can be corrected using 'context' or by using linguistic structures.

Phonetic errors:

- Speech recognition matches spoken utterances to a dictionary of phonemes.
- Spelling errors are often phonetic, with incorrect words sounding like correct ones.
- Phonetic errors are harder to correct due to more complex distortions.
- Phonetic variations are common in translation

Spelling errors: are classified as non-word or real-word errors.

- Non-word error:
 - Word that **does not appear in a given lexicon** or is **not a valid orthographic word** form.
 - The two main techniques to find non-word errors: n-gram analysis and dictionary lookup.
- Real-word error:
 - It occurs due to **typographical mistakes or spelling errors**.

- E.g. piece for peace or meat for meet.
- May cause local syntactic errors, global syntactic errors, semantic errors, or errors at discourse or pragmatic levels.
- Impossible to decide that a word is wrong without some contextual information

Spelling correction: consists of detecting and correcting errors. Error detection is the process of finding misspelled words and error correction is the process of suggesting correct words to a misspelled one.

These sub-problems are addressed in two ways:

1. Isolated-error detection and correction
2. Context-dependent error detection and correction

Isolated-error detection and correction: Each word is checked separately, independent of its context.

- It requires the existence of a lexicon containing all correct words.
- Would take a long time to compile and occupy a lot of space.
- It is impossible to list all the correct words of highly productive languages.

Context dependent error detection and correction methods: Utilize the context of a word. This requires grammatical analysis and is thus more complex and language dependent. the list of candidate words must first be obtained using an isolated-word method before making a selection depending on the context.

The spelling correction algorithm:

Broadly categorized by Kukich (1992) as follows:

Minimum edit distance The minimum edit distance between two strings is the minimum number of operations (insertions, deletions, or substitutions) required to transform one string into another.

Similarity key techniques The basic idea in a similarity key technique is to change a given string into a key such that similar strings will change into the same key.

n-gram based techniques n-gram techniques usually require a large corpus or dictionary as training data, so that an n-gram table of possible combinations of letters can be compiled. In case of real-word error detection, we calculate the likelihood of one character following another and use this information to find possible correct word candidates.

Neural nets These have the ability to do associative recall based on incomplete and noisy data. They can be trained to adapt to specific spelling error patterns. Note: They are computationally expensive.

Rule-based techniques In a rule-based technique, a set of rules (heuristics) derived from knowledge of a common spelling error pattern is used to transform misspelled words into valid words.

1.1 Minimum Edit Distance:

The minimum edit distance is the number of insertions, deletions, and substitutions required to change one string into another.

For example, the minimum edit distance between 'tutor' and 'tumour' is 2: We substitute 'm' for 't' and insert 'u' before 'r'.

Edit distance can be viewed as a *string alignment problem*. By aligning two strings, we can measure the degree to which they match. There may be more than one possible alignment between two strings.

Alignment 1:

t u **t** o - r t u **m** o u r

The best possible alignment corresponds to the minimum edit distance between the strings. The alignment shown here, between tutor and tumour, has a distance of 2.

A dash in the upper string indicates insertion. A substitution occurs when the two alignment symbols do not match (shown in bold).

The Levenshtein distance between two sequences is obtained by **assigning a unit cost to each operation**, therefore distance is 2.

Alignment 2:

Another possible alignment for these sequences is

t u t - o - r t u - m o u r

which has a cost of 3.

Dynamic programming algorithms can be quite useful for finding minimum edit distance between two sequences. (table-driven approach to solve problems by combining solutions to sub-problems).

The dynamic programming algorithm for minimum edit distance is implemented by creating an edit distance matrix.

- This matrix has one row for each symbol in the source string and one column for each matrix in the target string.
- The (i, j)th cell in this matrix represents the distance between the first i character of the source and the first j character of the target string.
- Each cell can be computed as a simple function of its surrounding cells. Thus, by starting at the beginning of the matrix, it is possible to fill each entry iteratively.
- The value in each cell is computed in terms of three possible paths.
- The substitution will be 0 if the ith character in the source matches with jth character in the target.
- The minimum edit distance algorithm is shown below.

$$dist[i, j] = \begin{cases} dist[i-1, j] + insert_cost, \\ dist[i-1, j-1] + subst_cost[source_i, target_j] \\ dist[i, j-1] + delete_cost \end{cases}$$

- How the algorithm computes the minimum edit distance between tutor and tumour is shown in table.

Input: Two strings, X and Y
Output: The minimum edit distance between X and Y
 $m \leftarrow \text{length}(X)$ $n \leftarrow \text{length}(Y)$ for $i = 0$ to m do $\text{dist}[i,0]$
 $\leftarrow i$
for $j = 0$ to n do $\text{dist}[0,j]$
 $\leftarrow j$
for $i = 0$ to m do for $j = 0$ to n do $\text{dist}[i,j] = \min\{ \text{dist}[i-1,j] + \text{insert_cost}, \text{dist}[i-1,j-1] + \text{subst_cost}(X_i, Y_j), \text{dist}[i,j-1] + \text{delet_cost} \}$

	#	t	u	m	o	u	r
#	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
u	2	1	0	1	2	3	4
t	3	2	1	1	2	3	4
o	4	3	2	2	1	2	3
r	5	4	3	3	2	2	2

Minimum edit distance algorithms are also useful for determining accuracy in speech recognition systems.

2. Words & Word Classes

- Words** are **classified** into categories called **part-of-speech**.
- These are sometimes called word classes or lexical categories.
- These lexical categories are usually defined by their syntactic and morphological behaviours.
- The most common lexical categories are nouns and verbs. Other lexical categories include adjectives, adverbs, prepositions, and conjunctions.

NN	noun	Student, chair, proof, mechanism
VB	verb	Study, increase, produce
ADJ	adj	Large, high, tall, few
JJ	adverb	Carefully, slowly, uniformly
IN	preposition	in, on, to, of
PRP	pronoun	I, me, they
DET	determiner	the, a, an, this, those

Table shows some of the word classes in English. Lexical categories and their properties vary from language to language.

Word classes are further categorized as **open and closed word classes**.

- Open word classes** constantly acquire new members while **closed word classes** do not (or only infrequently do so).

- Nouns, verbs (except auxiliary verbs), adjectives, adverbs, and interjections are **open word classes**.
e.g. *computer, happiness, dog, run, think, discover, beautiful, large, happy, quickly, very, easily*
- Prepositions, auxiliary verbs, delimiters, conjunction, and Interjections are **closed word classes**.
e.g. *in, on, under, between, he, she, it, they, the, a, some, this, and, but, or, because, oh, wow, ouch*

3. Part-of-Speech Tagging

- The process of assigning a part-of-speech (such as a noun, verb, pronoun, preposition, adverb, and adjective), to each word in a sentence.
- Input to a tagging algorithm: **Sequence of words** of a natural language sentence and **specified tag sets**.
- Output: single best part-of-speech tag for each word.
- Many words may belong to more than one lexical category:
 - *I am reading a good book* ⑦ *book*: Noun
 - *The police booked the snatcher* ⑦ *book*: verb ○ 'sona' may mean 'gold' (noun) or 'sleep' (verb)

Determine the correct lexical category of a word in its context

Tag set:

- The collection of tags used by a particular tagger is called a tag set.
- Most part-of-speech tag sets make use of the same basic categories, i.e., noun, verb, adjective, and prepositions.
- Most tag sets capture morpho-syntactic information such as singular/plural, number, gender, tense, etc.
- Tag sets differ in how they define categories and how finely they divide words into categories.

Consider,

Zuha eats an apple daily.

Aman ate an apple yesterday.

They have eaten all the apples in the basket.

I like to eat guavas.

The word eat has a distinct grammatical form in each of these four sentences.

Eat is the base form, ate its past tense, and the form eats requires a third person singular subject.

Similarly, eaten is the past participle form and cannot occur in another grammatical context.

Number of tags:

- The number of tags used by different taggers varies substantially (20 tags and over 400 tags).

- Penn Treebank tag set contains 45 tags & C7 uses 164
- TOSCA-ICE for the International Corpus of English with 270 tags (Garside 1997).
- TESS with 200 tags.
- English, which is not morphologically rich, the C7 tagset is too big ⑦ yield too many mistagged words.

Tags from Penn Treebank tag set

VB	Verb, base form Subsumes imperatives, infinitives, and subjunctives
VBD	Verb, past tense Includes the conditional form of the <i>verb to be</i>
VBG	Verb, gerund, or present participle
VCN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present

Possible tags for the word to eat

eat	VB
ate	VBD
eaten	VCN
eats	VBP

Example of a tagged sentence:

Speech/NN sounds/NNS were/VBD sampled/VBN by/IN a/DT microphone/NN.

Another tagging possible

Speech/NN sounds/VBZ were/VBD sampled/VBN by/IN a/DT microphone/NN

It leads to semantic incoherence. We resolve the ambiguity using the context of the word. The context is also utilized by automatic taggers.

Part-of-speech tagging methods:

1. Rule-based (linguistic)
2. Stochastic (data-driven)
3. Hybrid

Rule-based taggers use **hand-coded rules** to assign tags to words. These rules use a lexicon to obtain a list of candidate tags and then use rules to discard incorrect tags.

Stochastic taggers have **data-driven** approaches in which frequency-based information is automatically derived from corpus and used to tag words. Probability that a word occurs with a particular tag. E.g. Hidden Markov model (HMM).

Hybrid taggers combine features of both these approaches. Like rule-based systems, they **use rules to specify tags**. Like stochastic systems, they use **machine-learning to induce rules** from a tagged training corpus automatically. E.g. Brill tagger.

3.1 Rule-based Tagger

- A two-stage architecture.

- **The first stage:** A dictionary look-up procedure, which returns a set of potential tags (parts-of-speech) and appropriate syntactic features for each word.
- **The second stage:** A set of hand-coded rules to discard contextually illegitimate tags to get a single part-of-speech for each word.

E.g Consider the noun-verb ambiguity in the following sentence:

“The show must go on”

Show ❷ ambiguity {VB, NN}

Following are the rules to resolve this ambiguity:

IF preceding word is determiner THEN eliminate VB tag.

In addition to contextual information, many taggers use morphological information to help in the disambiguation process:

IF word ends in -ing and preceding word is a verb THEN label it a verb (VB).

Rule-based taggers use capitalization to identify unknown nouns and typically require supervised training. Rules can be induced by running untagged text through a tagger, manually correcting it, and feeding it back for learning.

TAGGIT (1971) tagged 77% of the Brown corpus using 3,300 rules. ENGTWOL (1995) is another rule-based tagger known for speed and determinism.

Advantages & disadvantages:

While rule-based systems are fast and deterministic, they require significant effort to write rules and need a complete rewrite for other languages. Stochastic taggers are more flexible, adapting to new languages with minimal changes and retraining. Thus, rule-based systems are precise but labor-intensive, while stochastic systems are more adaptable but probabilistic.

3.2 Stochastic Tagger

- The standard stochastic tagger algorithm is the HMM tagger.
- Applies the simplifying assumption that the probability of a chain of symbols can be approximated in terms of its parts or n-grams.

The unigram model requires a tagged training corpus to gather statistics for tagging data. It assigns tags based solely on the word itself. For example, the tag **JJ** (Adjective) is frequently assigned to "fast" because it is more commonly used as an adjective than as a noun, verb, or adverb. However, this can lead to incorrect tagging, as seen in the following examples:

1. **She had a fast** — Here, "fast" is a noun.
2. **Muslims fast during Ramadan** — Here, "fast" is a verb.

3. **Those who were injured in the accident need to be helped fast** — Here, "fast" is an adverb.

In these cases, a more accurate prediction could be made by considering additional context. A **bi-gram tagger** improves accuracy by incorporating both the current word and the tag of the previous word. For instance, in sentence (1), the sequence "DT NN" (determiner, noun) is more likely than "DT JJ" (determiner, adjective), so the bi-gram tagger would correctly tag "fast" as a noun. Similarly, in sentence (3), a verb is more likely to be followed by an adverb, so the bi-gram tagger assigns "fast" the tag **RB** (adverb).

In general, **n-gram models** consider both the current word and the tags of the previous n-1 words. A **tri-gram model**, for example, uses the previous two tags, providing even richer context for more accurate tagging. The context considered by a tri-gram model is shown in Figure, where the shaded area represents

Tokens	w_{n-2}	w_{n-1}	w_n	w_{n+1}
tags	t_{n-2}	t_{n-1}	t_n	t_{n+1}

the contextual window.

How the HMM tagger assigns the most likely tag sequence to a given sentence:

We refer to this model as a **Hidden Markov Model (HMM)** because it has two layers of states:

- A visible layer corresponding to the input words.
- A hidden layer corresponding to the tags.

While tagging input data, we can observe the words, but the tags (states) are hidden. The states are visible during training but not during the tagging process.

As mentioned earlier, the HMM uses **lexical** and **bi-gram probabilities** estimated from a tagged training corpus to compute the most likely tag sequence for a given sentence. One way to store these probabilities is by constructing a **probability matrix**. This matrix includes:

- The probability that a specific word belongs to a particular word class.
- The n-gram analysis (for example, in a bi-gram model, the probability that a word of class X follows a word of class Y).

During tagging, this matrix is used to guide the HMM tagger in predicting the tags for an unknown sentence. The goal is to determine the most probable tag sequence for a given sequence of words.

Let W be the sequence of words.

$$W = w_1, w_2, \dots, w_n$$

The task is to find the tag sequence

$$T = t_1, t_2, \dots, t_n$$

which maximizes $P(T|W)$, i.e.,

$$T' = \operatorname{argmax}_T P(T|W)$$

Applying Bayes Rule, $P(T|W)$ can be estimated using the expression:

$$P(T|W) = P(W|T) * P(T)/P(W)$$

$P(W)$, remains the same for each tag sequence, we can drop it. The expression for the most likely tag sequence becomes:

$$T' = \operatorname{argmax}_T P(W|T) * P(T)$$

A tag sequence can be estimated as the product of the probability of its constituent n-grams, i.e.,

$$P(T) = P(t_1) * P(t_2|t_1) * P(t_3|t_1 t_2) \dots * P(t_n|t_1 \dots t_{n-1})$$

$P(W|T)$ is the probability of seeing a word sequence, given a tag sequence.

For example, it is asking the probability of seeing '*The egg is rotten*' given '*DT NNP VB JJ*'. We make the following two assumptions:

- The words are independent of each other.
- The probability of a word is dependent only on its tag.

$$P(W/T) = P(w_1/t_1) * P(w_2/t_2) \dots P(w_i/t_i) * \dots P(w_n/t_n)$$

i.e.,
$$P(W/T) \approx \prod_{i=1}^n P(w_i/t_i)$$

So,
$$P(W/T) * P(T) = \prod_{i=1}^n P(w_i/t_i) \times P(t_1) \times P(t_2|t_1) * P(t_3|t_1 t_2) \times \dots \times P(t_n|t_1 \dots t_{n-1})$$

Approximating the tag history using only the two previous tags, the transition probability, $P(T)$, becomes

$$P(T) = P(t_1) \times P(t_2|t_1) \times P(t_3|t_1 t_2) \times \dots \times P(t_n|t_{n-2} t_{n-1})$$

Hence, $P(T/W)$ can be estimated as

$$\begin{aligned} P(W/T) * P(T) &= \prod_{i=1}^n P(w_i/t_i) \\ &\times P(t_1) \times P(t_2|t_1) * P(t_3|t_2 t_1) \times \dots \times P(t_n|t_{n-2} t_{n-1}) \\ &= \prod_{i=1}^n P(w_i/t_i) \times P(t_1) \times P(t_2|t_1) \times \prod_{i=3}^n P(t_i|t_{i-2} t_{i-1}) \end{aligned}$$

We estimate these probabilities from relative frequencies via Maximum Likelihood Estimation.

$$P(t_i|t_{i-2} t_{i-1}) = \frac{c(t_{i-2}, t_{i-1}, t_i)}{c(t_{i-2}, t_{i-1})}$$

$$P(w_i|t_i) = \frac{c(w_i, t_i)}{c(t_i)}$$

where $c(t_{i-2}, t_{i-1}, t_i)$ is the number of occurrences of t_i followed by $t_{i-2} t_{i-1}$.

Example Consider the sentence “The *bird can fly*”. and

the tag sequence DT NNP MD VB

Using bi-gram approximation, the probability

$$P \begin{pmatrix} \text{DT} & \text{NNP} & \text{MD} & \text{VB} \\ | & | & | & | \\ \text{The} & \text{bird} & \text{can} & \text{fly} \end{pmatrix}$$

can be computed as

$$= P(DT) \times P(NNP|DT) * P(MD|NNP) \times P(VB|MD) \times P(the|DT) \times P(bird|NNP) \times P(can|MD) \times P(fly|VB)$$

3.3 Hybrid Taggers

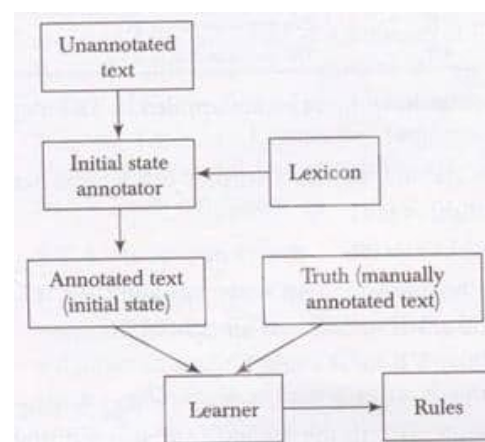
Hybrid approaches to tagging combine the strengths of both rule-based and stochastic methods.

- These approaches use rules to assign tags to words.
- While leveraging machine learning techniques to automatically generate rules from data.

An example is **Transformation-Based Learning (TBL)**, or **Brill tagging**, introduced by E. Brill in 1995. TBL has been applied to tasks like part-of-speech tagging and syntactic parsing.

Figure illustrates the TBL process, which is a **supervised learning** technique. The algorithm starts by assigning the most likely tag to each word using a lexicon. Transformation rules are then applied iteratively, with the rule that improves tagging accuracy most being selected each time. The process continues until no significant improvements are made.

The output is a ranked list of transformations, which are applied to new text by first assigning the most frequent tag and then applying the transformations.



TBL tagging algorithm

INPUT: Tagged corpus and lexicon (with most frequent information) Step 1: Label every word with most likely tag (from dictionary) Step 2: Check every possible transformation and select one which most improves tagging Step 3: Re-tag corpus applying the rules Repeat 2-3: Until some stopping criterion is reached RESULT Ranked sequence of transformation rules
--

Example: Assume that in a corpus, fish is most likely to be a noun.

$$P(\text{NN}/\text{fish}) = 0.91$$

$$P(\text{VB}/\text{fish}) = 0.09$$

Now consider the following two sentences and their initial tags.

I/PRP like/VB to/TO eat/VB fish/NNP.

I/PRP like/VB to/TO fish/NNP.

As the most likely tag for fish is NNP, the tagger assigns this tag to the word in both sentences. In the second case, it is a mistake. After initial tagging when the transformation rules are applied, the tagger learns a rule that applies exactly to this mis-tagging of fish:

Change NNP to VB if the previous tag is TO. As

the contextual condition is satisfied, this rule will change fish/NN to fish/VB:

like/VB to/TO fish/NN $\rightarrow$ like/VB to/TO fish/VB

Scope; Advantages and disadvantages:

The algorithm can be made more efficient by indexing words in a training corpus using potential transformations. Recent work has applied finite state transducers to compile pattern-action rules, combining them into a single transducer for faster rule application, as demonstrated by Roche and Schabes (1997) on Brill's tagger.

Most part-of-speech tagging research focuses on English and European languages, but the lack of annotated corpora limits progress for other languages, including Indian languages. Some systems, like Bengali (Sandipan et al., 2004) and Hindi (Smriti et al., 2006), combine morphological analysis with tagged corpora.

Tagging Urdu is more complex due to its right-to-left script and grammar influenced by Arabic and Persian. Before Hardie (2003), little work was done on Urdu tag sets, with his research part of the EMILLE project for South Asian languages.

3.4 Unknown words:

Unknown words, which do not appear in a dictionary or training corpus, pose challenges during tagging. Solutions include:

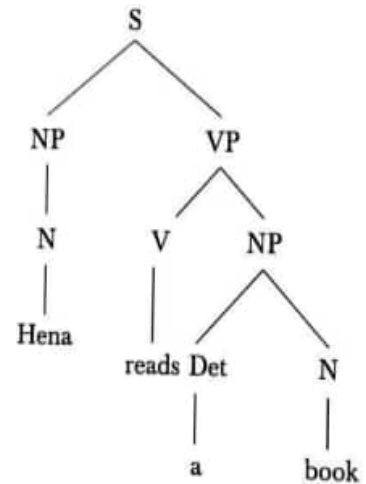
- Assigning the most frequent tag from the training corpus or initializing unknown words with an open class tag and disambiguating them using tag probabilities.
- Another approach involves using morphological information, such as affixes, to predict the tag based on common suffixes or prefixes in the training data, similar to Brill's tagger.

Syntactic Analysis

1. Introduction:

- Syntactic parsing deals with the syntactic structure of a sentence.
- 'Syntax' refers to the grammatical arrangement of words in a sentence and their relationship with each other.
- The objective of **syntactic analysis is to find the syntactic structure** of the sentence.
- This structure is usually depicted as a tree, as shown in Figure.
 - Nodes in the tree represent the phrases and leaves correspond to the words.
 - The root of the tree is the whole sentence.

- Identifying the **syntactic structure** is useful in determining the **meaning** of the sentence.
- Syntactic parsing can be considered as the **process of assigning 'phrase markers'** to a sentence.
- Two important ideas in natural language are those of constituency and word order.
 - Constituency is about how words are grouped together.
 - Word order is about how, within a constituent, words are ordered and also how constituents are ordered with respect to one another.



- A widely used mathematical system for **modelling constituent structure** in natural language is **context-free grammar (CFG)** also known as **phrase structure grammar**.

2. Context-free Grammar:

- Context-free grammar (CFG) was first defined for natural language by Chomsky (1957).
- Consists of four components:
 1. A set of non-terminal symbols, N
 2. A set of terminal symbols, T
 3. A designated start symbol, S, that is one of the symbols from N.
 4. A set of productions, P, of the form: $A \rightarrow \alpha$ Where $A \in N$ and α is a string consisting of terminal and non-terminal symbols.
 - The rule $A \rightarrow \alpha$ says that constituent A can be rewritten as α . This is also called the phrase structure rule. It specifies which elements (or constituents) can occur in a phrase and in what order.
 - For example, the rule $S \rightarrow NP VP$ states that S consists of NP followed by VP, i.e., a sentence consists of a noun phrase followed by a verb phrase.

CFG as a generator:

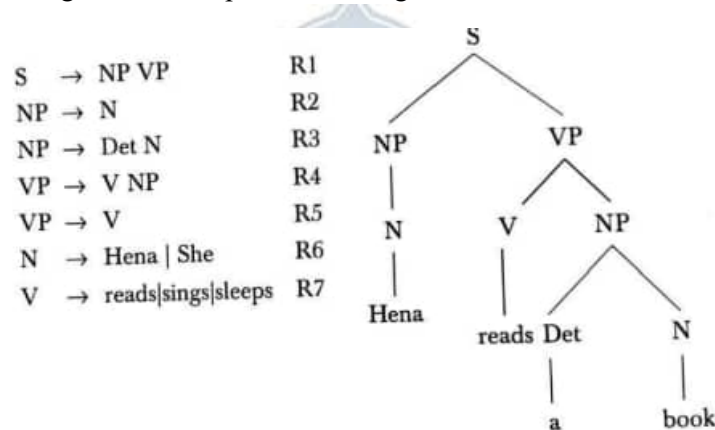
- A CFG can be used to **generate a sentence** or to **assign a structure to a given sentence**.
- When used as a generator, the arrows in the production rule may be read as 'rewrite the symbol on the left with symbols on the right'.
- Consider the toy grammar: shown in Figure.
- The symbol S can be rewritten as NP VP using Rule 1, then using rules R2 and R4, NP and VP are rewritten as N and V NP respectively. NP is then rewritten as Det N (R3). Finally, using rules R6 and R7, we get the sentence:

Hena reads a book.

- Above can be derived from S. The representation of this derivation is shown in Figure.
- Sometimes, a more compact bracketed notation is used to represent a parse tree.

$[_s [_{NP} [_N \text{Hena}]] [_{VP} [_V \text{reads}]] [_{NP} [_{Det} \text{a}]] [_N \text{book}]]]$

- The parse tree in Figure can be represented using this notation as follows:



3. Constituency:

- Words in a sentence are not tied together as a sequence of part-of-speech.
- Language puts constraints on word order.
- Words group together to form constituents** (often termed phrases), each of which acts as a single unit. They combine with other constituents to form larger constituents, and eventually, a sentence.
- Constituents combine with others to form a sentence constituent.
- For example: the noun phrase, ***The bird***, can combine with the verb phrase, ***flies***, to form the sentence, ***The bird flies***.
- Different types of phrases have different internal structures.

3.1 Phrase Level Constructions

Noun Phrase, Verb phrase, Prepositional Phrase, Adjective Phrase, Adverb Phrase

Noun Phrase:

- A noun phrase is a phrase whose head is a noun or a pronoun, optionally accompanied by a set of modifiers. It can function as subject, object, or complement.
- The modifiers of a noun phrase can be determiners or adjective phrases.
- Phrase structure rules are of the form: $A \rightarrow B C$ $NP \rightarrow Pronoun$

$NP \rightarrow Det Noun$

$NP \rightarrow Noun$

$NP \rightarrow Adj Noun$

$NP \rightarrow Det Adj Noun$

- We can combine all these rules in a single phrase structure rule as follows:

$NP \rightarrow (Det) (Adj) Noun | Pronoun$

- A noun phrase may include post-modifiers and more than one adjective. $NP \rightarrow (Det) (AP) Noun (PP)$

Few examples of noun phrases:

They

NP

The foggy morning	Adj Noun
Chilled water	Adj Noun
A beautiful lake in Kashmir	Det Adj Noun PP
Cold banana shake	Adjective followed by a sequence of nouns .

- Adjective followed by a sequence of nouns ⑦ A noun sequence is termed as **nominal**. We modify our rules to cover this situation.

NP → (Det) (AP) **Nom** (PP)

Nom → Noun | Noun Nom

- A noun phrase can act as a subject, an object, or a predicate.

Example:

The foggy damped weather disturbed the match. ⑦ noun phrase acts as a subject I would like a nice cold banana shake. ⑦ noun phrase acts as an object Kula botanical garden is a beautiful location. ⑦ noun phrase acts as predicate

Verb Phrase:

- Headed by a verb
- The verb phrase organizes various elements of the sentence that depend syntactically on the verb.

Examples of verb phrases:

Khushbu slept .	VP → Verb
The boy kicked the ball	VP → Verb NP
Khushbu slept in the garden.	VP → Verb PP
The boy gave the girl a book .	VP → Verb NP NP
The boy gave the girl a book with blue cover.	Verb NP NP PP

In general, the number of NPs in a VP is limited to two, whereas it is possible to add more than two PPs.

VP → Verb (NP) (NP) (PP)*

Things are further complicated by the fact that objects may also be entire clauses as in the sentence, I know that Taj is one of the seven wonders. Hence, we must also allow for an alternative phrase statement rule, in which NP is replaced by S.

VP → Verb S

Prepositional Phrase:

Prepositional phrases are headed by a preposition. They consist of a preposition, possibly followed by some other constituent, usually a noun phrase.

We played volleyball on the beach.

We can have a preposition phrase that consists of just a preposition.

John went outside.

The phrase structure rule that captures the above eventualities is as follows.

PP → Prep (NP)

Adjective Phrase:

The head of an adjective phrase (AP) is an adjective. APs consist of an adjective, which may be preceded by an adverb and followed by a PP.

Here are few examples.

Ashish is clever.

The train is very late.

My sister is fond of animals.

The phrase structure rule for adjective phrase is

$$AP \rightarrow (\text{Adv}) \text{Adj} (\text{PP})$$

Adverb Phrase:

An adverb phrase consists of an adverb, possibly preceded by a degree adverb. Here is an example.

Time passes very quickly.

$$\text{AdvP} \rightarrow (\text{Intens}) \text{Adv}$$

3.2 Sentence Level Constructions

A sentence can have varying structure.

The four commonly known structures are declarative structure, imperative structure, yes-no question structure, and wh-question structure.

1. **Declarative structure:** Makes a statement *or expresses an idea*.

Example: *I like horse riding*

Structure: will have a subject followed by a predicate.

The subject is noun phrase and the predicate is a verb phrase.

Grammar rule: $S \rightarrow NP VP$

2. **Imperative structure:** Gives a *command, request, or suggestion*.

Example: *Please pass the salt, Look at the door, Show me the latest design.*

Structure: usually begin with a verb phrase and lack subject.

Grammar rule: $S \rightarrow VP$

3. **Yes-no question structure:** *Asks a question* that expects a yes or no answer.

Example: *Do you have a red pen?*

Did you finish your homework?

Is the game over?

Structure: usually begin with an auxiliary verb, followed by a subject NP, followed by a VP.

Grammar rule: $S \rightarrow \text{Aux NP VP}$

4. **Wh-question structure:** *Asks for specific information* using words like who, what, or where.

Example: *Where are you going?*

Which team won the match?

Structure: May have a wh-phrase as a subject or may include another subject.

Grammar rule: $S \rightarrow \text{Wh-NP VP}$

Another type of wh-question structure is one that involves more than one NP, , the auxiliary verb comes before the subject NP, just as in yes-no question structures.

Example: *Which cameras can you show me in your shop?*

Grammar rule: $S \rightarrow \text{Wh-NP Aux NP VP}$

Table. Summary of grammar rules

$S \rightarrow \text{NP VP}$
$S \rightarrow \text{VP}$
$S \rightarrow \text{Aux NP VP}$
$S \rightarrow \text{Wh-NP VP}$
$S \rightarrow \text{Wh-NP Aux NP VP}$
$\text{NP} \rightarrow (\text{Det}) (\text{AP}) \text{Nom} (\text{PP})$
$\text{VP} \rightarrow \text{Verb} (\text{NP}) (\text{NP}) (\text{PP})^*$
$\text{VP} \rightarrow \text{Verb S}$
$\text{AP} \rightarrow (\text{Adv}) \text{Adj} (\text{PP})$
$\text{PP} \rightarrow \text{Prep} (\text{NP})$
$\text{Nom} \rightarrow$

Note:

- Grammar rules are not exhaustive.
- There are other sentence-level structures that cannot be modelled by the rules.
- Coordination, Agreement and Feature structures

Coordination:

Refers to conjoining phrases with conjunctions like 'and', 'or', and 'but'.

For example,

A **coordinate noun phrase** can consist of two other noun phrases separated by a conjunction.

I ate [NP [NP an apple] and [NP a banana]].

Similarly, *verb phrases and prepositional phrases* can be conjoined as follows:

It is [VP [VP dazzling] and [VP raining]].

Not only that, even a *sentence can be conjoined*.

[S [S I am reading the book] and [S I am also watching the movie]]

Conjunction rules for NP, VP, and S can be built as follows:

$\text{NP} \rightarrow \text{NP and NP}$

$VP \rightarrow VP \text{ and } VP$

$S \rightarrow S \text{ and } S$

Agreement:

Most *verbs use two different forms* in present tense-one for third person, singular subjects, and the other for all other kinds of subjects. Subject and verb must agree.

Examples: Demonstrate how the subject NP affects the form of the verb.

Does [_{NP} Priya] sing?

Do [_{NP} they] eat?

The -es form of 'do', i.e. 'does' is used. The second sentence has a plural NP subject. Hence, the form 'do' is being used. Sentences in which subject and verb do not agree are ungrammatical.

The following sentences are ungrammatical:

[Does] they eat?

[Do] she sings?

Rules that handle the yes-no questions: $S \rightarrow \text{Aux NP VP}$

To take care of the subject-verb agreement, we replace this rule with a pair of rules as follows:

$S \rightarrow 3\text{sgAux } 3\text{sgNP VP}$

$S \rightarrow \text{Non}3\text{sgAux Non}3\text{sgNP VP}$

We could add rules for the lexicon like these:

$3\text{sg Aux} \rightarrow \text{does} \mid \text{has} \mid \text{can}$

$\text{Non}3\text{sg Aux} \rightarrow \text{do} \mid \text{have} \mid \text{can}$

Similarly, rules for 3sgNP and Non3sgNP need to be added. So we replace each of the phrase structure rules for noun phrase by a pair of rules as follows:

$3\text{sgNP} \rightarrow \rightarrow (\text{Det}) (\text{AP}) \text{SgNom} (\text{PP})$

$\text{Non}3\text{sgNP} \rightarrow (\text{Det}) (\text{AP}) \text{PINom} (\text{PP})$

$\text{SgNom} \rightarrow \text{SgNoun} \mid \text{SgNoun SgNom}$

$\text{PINom} \rightarrow \text{PINoun} \mid \text{PINoun PINom}$

$\text{SgNoun} \rightarrow \text{Priya} \mid \text{lake} \mid \text{banana} \mid \text{sister} \mid \dots$

$\text{PINoun} \rightarrow \text{Children} \mid \dots$

Note: These results in an explosion in the number of grammar rules and loss of generality.

Solution: Feature structures

Feature Structures

Feature structures are able to capture grammatical properties without increasing the size of the grammar.

Feature structures are sets of feature-value pairs.

Features are simply symbols representing properties that we wish to capture.

For example, the number property of a noun phrase can be represented by NUMBER feature. The value that a NUMBER feature can take is SG (for singular) and PL (for plural).

Feature structures are represented by a matrix-like diagram called attribute value matrix (AVM).

$$\begin{bmatrix} \text{FEATURE}_1 & \text{VALUE}_1 \\ \text{FEATURE}_2 & \text{VALUE}_2 \\ \dots & \dots \\ \text{FEATURE}_n & \text{VALUE}_n \end{bmatrix}$$

The feature structure can be used to encode the grammatical category of a constituent and the features associated with it. For example, the following structure represents the third person singular noun phrase.

$$\begin{bmatrix} \text{CAT} & \text{NP} \\ \text{NUMBER} & \text{SG} \\ \text{PERSON} & 3 \end{bmatrix}$$

Similarly, a third person plural noun phrase can be represented as follows:

$$\begin{bmatrix} \text{CAT} & \text{NP} \\ \text{NUMBER} & \text{PL} \\ \text{PERSON} & 3 \end{bmatrix}$$

The CAT and PERSON feature values remain the same in both structures, illustrating how feature structures support generalization while maintaining necessary distinctions. Feature values can also be other feature structures, not just atomic symbols. For instance, combining NUMBER and PERSON into a single AGREEMENT feature makes sense, as subjects must agree with predicates in both properties.

This allows a more streamlined representation.

$$\begin{bmatrix} \text{CAT} & \text{NP} \\ \text{AGREEMENT} & \begin{bmatrix} \text{NUMBER} & \text{PL} \\ \text{PERSON} & 3 \end{bmatrix} \end{bmatrix}$$

4. Parsing

- A phrase structure tree constructed from a sentence is called a parse.
- The syntactic parser is thus responsible for **recognizing a sentence and assigning a syntactic structure to it**.
- The task that uses the rewrite rules of a grammar to either generate a particular sequence of words or reconstruct its derivation (or phrase structure tree) is termed parsing.
- It is possible for many **different phrase structure trees to derive the same sequence of words**.
- Sentence can have multiple parses ⑦ This phenomenon is called syntactic ambiguity.
- Processes input data (usually in the form of text) and converts it into a format that can be easily understood and manipulated by a computer.
 - *Input*: The first constraint comes from the words in the input sentence. A valid parse is one that covers all the words in a sentence. Hence, these words must constitute the leaves of the final parse tree.

- *Grammar*: The second kind of constraint comes from the grammar. The root of the final parse tree must be the start symbol of the grammar.

Two most widely used search strategies by parsers,

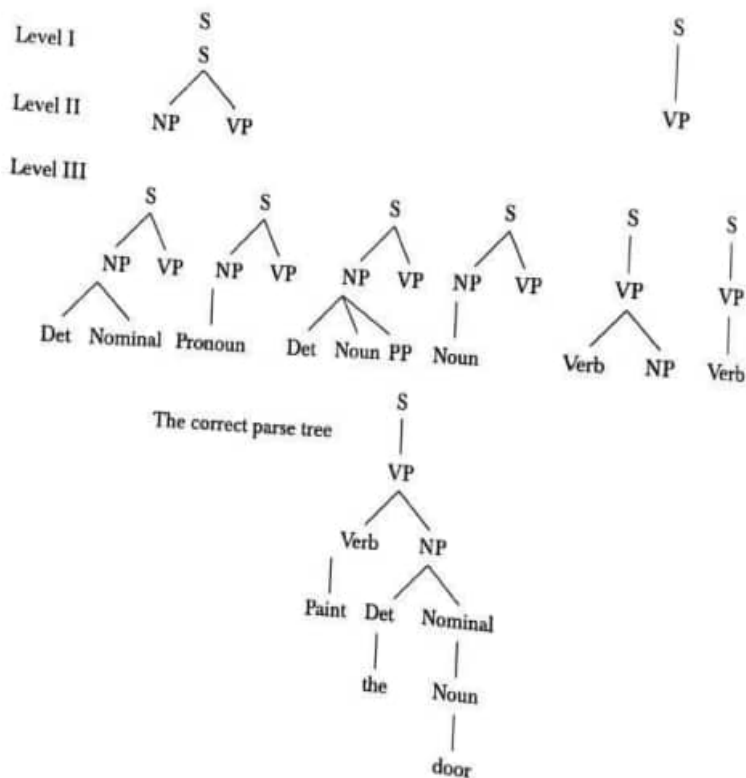
1. Top-down or goal-directed search.
2. Bottom-up or data-directed search.

4.1 Top-down Parsing

- Starts its search from the root node S and works downwards towards the leaves.
- Find all sub-trees which can start with S: Expand the root node using all the grammar rules with S on their left-hand side.
- Likewise, each non-terminal symbol in the resulting sub-trees is expanded next using the grammar rules having a matching non-terminal symbol on their left-hand side.
- The right-hand side of the grammar rules provide the nodes to be generated, which are then expanded recursively.
- The **tree grows downward** and eventually reaches a state where the bottom of the tree consists only of part-of-speech categories.
- A successful parse corresponds to a tree which matches exactly with the words in the input sentence.

Example: Consider the grammar shown in *Table* and the sentence “*Paint the door*”.

$S \rightarrow NP VP$	$VP \rightarrow Verb NP$
$S \rightarrow VP$	$VP \rightarrow Verb$
$NP \rightarrow Det Nominal$	$PP \rightarrow Preposition NP$
$NP \rightarrow Noun$	$Det \rightarrow this \mid that \mid a \mid the$
$NP \rightarrow Det Noun PP$	$Verb \rightarrow sleeps \mid sings \mid open \mid saw \mid paint$
$Nominal \rightarrow Noun$	$Preposition \rightarrow from \mid with \mid on \mid to$
$Nominal \rightarrow Noun Nominal$	$Pronoun \rightarrow she \mid he \mid they$



1. The first level (ply) search tree consists of a single node labelled S.
2. The grammar in Table has two rules with S on their left hand side.

S → NP VP & S → VP

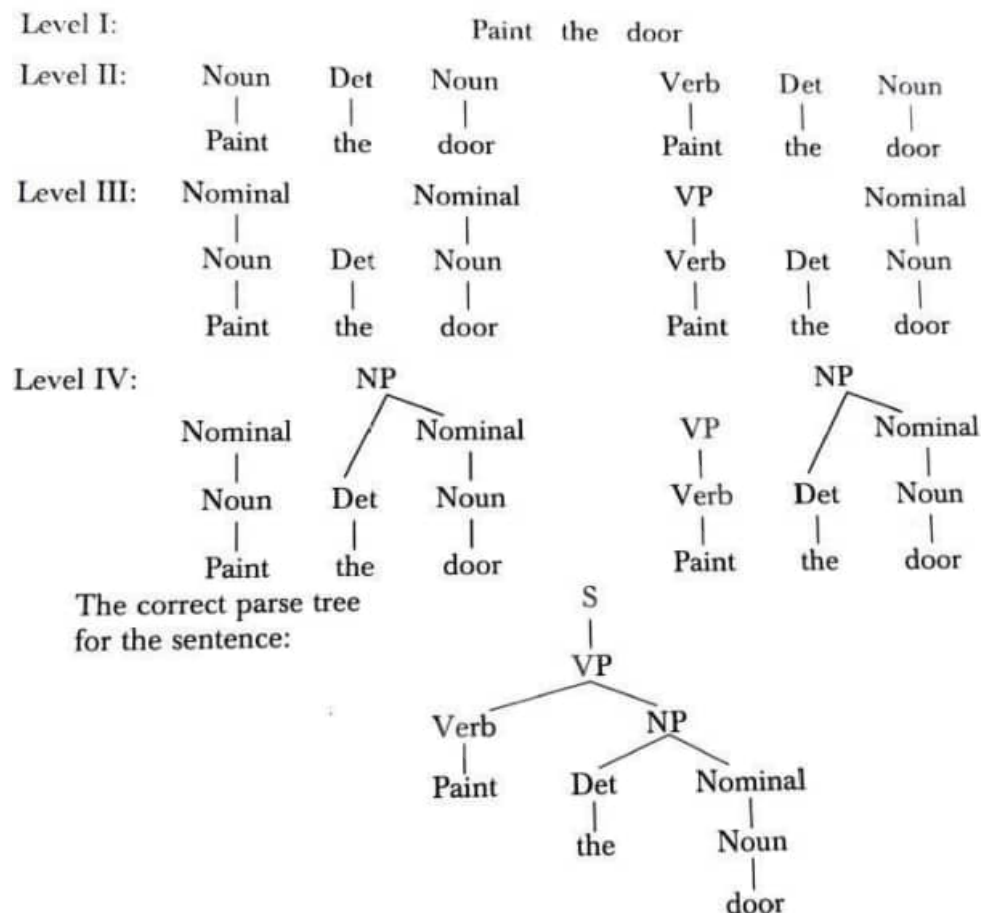
3. These rules are used to expand the tree, gives us two partial trees at the second level search.
4. The third level is generated by expanding the non-terminal at the bottom of the search tree in the previous.

4.2 Bottom-Up Parsing

A bottom-up parser starts with the words in the input sentence and attempts to construct a parse tree in an upward direction towards the root.

- **Start with the input words** – Begin with the words in the sentence as the leaves of the parse tree.
- **Look for matching grammar rules** – Search for rules where the right-hand side matches parts of the input.
- **Apply reduction using the left-hand side** – Replace matched portions with non-terminal symbols from the left-hand side of the rule.
- **Construct the parse tree upwards** – Build the parse tree by moving upward toward the root.

- **Repeat until the start symbol is reached** – Continue reducing until the entire sentence is reduced to the start symbol.
- **Successful parse** – The parsing is successful if the input is fully reduced to the start symbol, completing the parse tree.



Advantages & disadvantages:

- **Top-Down Parsing:** Starts from the start symbol and generates trees, avoiding paths that lead to a different root, but it may waste time exploring inconsistent trees before seeing the input.
- **Bottom-Up Parsing:** Starts with the input and ensures only matching trees are explored, but may waste time generating trees that won't lead to a valid parse tree (e.g., incorrect assumptions about word types).
- **Top-Down Drawback:** It can explore incorrect trees that eventually do not match the input, resulting in wasted computation.

Basic Search Strategy: Combines top-down tree generation with bottom-up constraints to filter out bad parses, aiming to optimize the parsing process.

4.3 A Basic Top-Down Parsing

A depth first, left to right search.

- **Start with Depth-First Search (DFS):** Use a depth-first approach to explore the search tree incrementally.
- **Left-to-Right Search:** Expand nodes from left to right in the tree.

- Incremental Expansion: Expand the search space one state at a time.
- Select Left-most Node for Expansion: Always select the left-most unexpanded node for expansion.
- Expand Using Grammar Rules: Expand nodes based on the relevant grammar rules.
- Handle Inconsistent State: If a state is inconsistent with the input, it is flagged.
- Return to Recent Tree: The search then returns to the most recently unexplored tree to continue.

Top-down, depth-first parsing algorithm

1. Initialize agenda
2. Pick a state, let it be curr_state, from agenda
3. If (curr_state) represents a successful parse then return parse tree else if curr_stat is a POS then
 if category of curr_state is a subset of POS associated with curr_word then
 apply lexical rules to current state
 else reject
 else generate new states by applying grammar rules and push them into agenda
4. If (agenda is empty) then return failure else select a node from agenda for expansion and go to step 3.

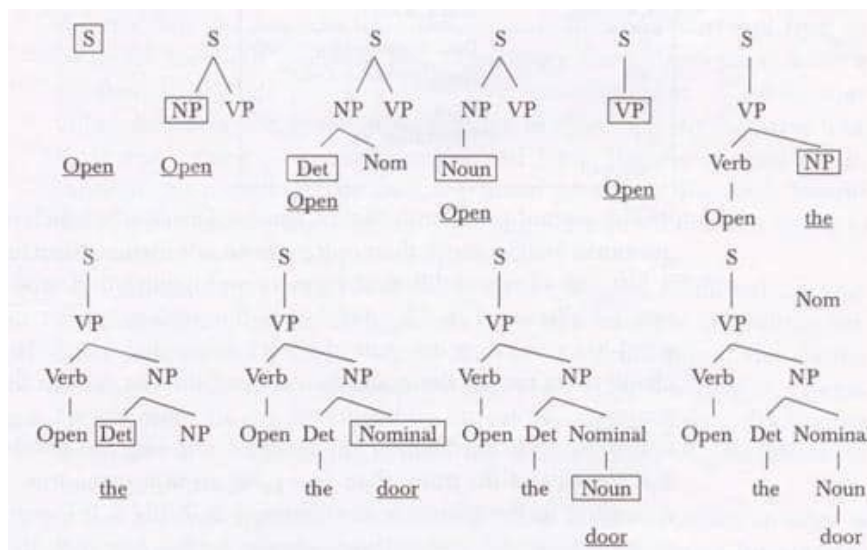


Figure shows the trace of the algorithm on the sentence, *Open the door*.

- The algorithm begins with the node S and input word "Open."
- It first expands S using the rule $S \rightarrow NP VP$, then expands NP with $NP \rightarrow Det Nominal$.
- Since "Open" cannot be derived from Det, the parser discards this rule and tries $NP \rightarrow noun$, which also fails.
- The next agenda item corresponds to $S \rightarrow VP$.
- Expanding VP using $VP \rightarrow Verb NP$ matches the first input word successfully.
- The algorithm then continues in a depth-first, left-to-right manner to match the remaining words.

Left corner for each grammar category

Category	Left Corners
S	Det, Pronoun, Noun, Verb
NP	Noun, Pronoun, Det
VP	Verb
PP	Preposition
Nominal	Noun

Disadvantages:

1. **Inefficiency:** It **may explore many unnecessary branches** of the parse tree, especially if the input does not match the grammar well, leading to high computational overhead.
2. **Backtracking:** If a rule fails, the parser often needs to backtrack to a previous state and try alternative expansions, which can significantly **slow down parsing**.
3. **Left Recursion Issues:** Top-down parsers **struggle with left-recursive grammars** because they can lead to **infinite recursion**.
4. **Lack of Lookahead:** Basic top-down parsers generally lack lookahead capabilities, meaning they might make incorrect decisions early on without enough information, leading to errors.
5. **Ambiguity Handling:** They may have difficulty handling ambiguities in the grammar, often exploring all possible alternatives without any way of pruning inefficient branches.
6. **Limited Error Recovery:** Basic top-down parsers typically have poor error recovery and can fail immediately when encountering an unexpected input.

Dynamic programming algorithms can solve these problems. These algorithms construct a table containing solutions to sub-problems, which, if solved, will solve the whole problem.

There are three widely known dynamic parsers-the Cocke-Younger-Kasami (CYK) algorithm, the Graham-Harrison-Ruzzo (GHR) algorithm, and the Earley algorithm.

Probabilistic grammar can also be used to disambiguate parse trees.

4.4 Earley Parser

- Efficient parallel top-down search using dynamic programming.
- It builds a table of sub-trees for each of the constituents in the input (eliminates the repetitive parse and reduces the exponential-time problem).
- Most important component of this algorithm is Earley chart.
 - The chart contains *a set of states for each word position* in the sentence.
 - The algorithm makes a *left to right scan* of input to fill the elements in this chart.
 - It *builds a set of states*, one for each position in the input string.

- The states in each entry provide the following information.
 - A sub-tree corresponding to a grammar rule.
 - Information about the progress made in completing the sub-tree.
 - Position of the sub-tree with respect to input.

Earley Parsing

Input: Sentence and the Grammar

Output: Chart $\text{chart}[0] + S' \rightarrow S, [0,0]$ $n \leftarrow \text{length}(\text{sentence})$ //
 number of words in the sentence for $i = 0$ to n do for each
 state in $\text{chart}[i]$ do
 if (incomplete (state) and next category is not a part of speech) then
 predictor (state)
 else if (incomplete (state) and next category is a part of speech)
 scanner (state)
 else completer
 (state)
 end-if
end-if
end for end
for

return:

Procedure **predictor** ($A \rightarrow X_1 \dots \bullet B \dots X_m, [i, j]$)

for each rule ($B \rightarrow \alpha$) in G do insert the state B

$\rightarrow \bullet \alpha, [j, j]$ to chart $[j]$ End

Procedure **scanner** ($A \rightarrow X_1 \dots \bullet B \dots X_m [i, j]$)

If B is one of the part of speech associated with word $[j]$ then

Insert the state $B \rightarrow \text{word } [j] \bullet, [j, j + 1]$ to chart $[j + 1]$

End

Procedure **Completer** ($A \rightarrow X_1 \dots \bullet A \dots, [j, k]$)

for each $B \rightarrow X_1 \dots \bullet A \dots, [i, j]$ in $\text{chart}[j]$ do

insert the state $B \rightarrow X_1 \dots A \dots, [i, k]$ to $\text{chart}[k]$

End

Steps:

Earley's algorithm works in three main steps:

1. Prediction

- If the dot ($\bullet$) is before a non-terminal in a rule, **add all rules expanding that non-terminal** to the state set.
- The predictor generates new states representing potential expansion of the non-terminal in the left-most derivation.
- A predictor is applied to every state that has a non-terminal to the right of the dot.
- Results in the creation of as many new states as there are grammar rules for the non-terminal

Their start and end positions are at the point where the generating state ends. If

$$A \rightarrow X_1 \dots \bullet B \dots X_m, [i, j]$$

Then for every rule of the form $B \rightarrow \alpha$, the operation adds to chart $[j]$, the state

$$B \rightarrow \bullet \alpha, [j, j] \quad \% \text{ Rule}$$

For example, when the generating state is $S \rightarrow \bullet \text{ NP VP}, [0,0]$, the predictor adds the following states to chart $[0]$:

$\text{NP} \rightarrow \bullet \text{ Det Nominal}, [0,0]$

$\text{NP} \rightarrow \bullet \text{ Noun}, [0,0]$

$\text{NP} \rightarrow \bullet \text{ Pronoun}, [0,0]$

$\text{NP} \rightarrow \bullet \text{ Det Noun PP}, [0,0]$

2. Scanning

- A scanner is used when *a state has a part-of-speech category to the right of the dot*.
- The scanner examines the input to see if the part-of-speech appearing to the right of the dot matches one of the part-of-speech associated with the current input.
- If yes, then it creates a new state using the rule that allows generation of the input word with this part-of-speech.
- If the dot ($\bullet$) is before a terminal that matches the current input symbol, move the dot to the right.

Example:

When the state $\text{NP} \rightarrow \bullet \text{ Det Nominal}, [0,0]$ is processed, the parser finds a part-of-speech category next to the dot.

It checks if the category of the current word (curr_word) matches with the expectation in the current state.

If yes, then it adds the new state $\text{Det} \rightarrow \text{curr_word} \bullet, [0,1]$ to the next chart entry.

3. Completion

- If the dot reaches the end of a rule, find and update previous rules that were waiting for this rule to complete.
- The completer identifies all previously generated states that expect this grammatical category at this position in the input and creates new states by advancing the dots over the expected category.

Example:

Let's consider a simple CFG:

```
S → NP VP
NP → Det N | John
VP → V NP
Det → the
N → dog
V → sees
```

"John sees the dog"

We want to parse the sentence:

Chart [0] (Start State)

We start with $S \rightarrow \bullet NP VP$:

```
(0) S → • NP VP [0]
(1) NP → • Det N [0]
(2) NP → • John [0]
```

Chart [1] ("John")

```
(3) NP → John • [0] (Completed NP)
(4) S → NP • VP [0] (Move dot)
(5) VP → • V NP [1] (Predict VP)
```

Since **John** is a valid NP, we scan it.

The next word is "sees", which matches V.

Chart [2] ("sees")

```
(6) VP → V • NP [1] (Scanned "sees")
(7) NP → • Det N [2]
(8) NP → • John [2]
```

We scan "the".

Chart [3] ("the")

```
(9) NP → Det • N [2] (Scanned "the")
```

We scan "dog".

Chart [4] ("dog")

```
(10) NP → Det N • [2] (Completed NP)
(11) VP → V NP • [1] (Completed VP)
(12) S → NP VP • [0] (Completed S)
```

The sequence of states for *“Paint the door”* created by the parser is shown in Figure

Chart [0]	S0	$S' \rightarrow \bullet S$	[0,0]
Dummy		start	state
	S1	$S \rightarrow \bullet NP VP$	
	S2	$S \rightarrow \bullet VP$	
	S3	$NP \rightarrow \bullet Det Nominal$	
	S4	$NP \rightarrow \bullet Noun$	
	S5	$NP \rightarrow \bullet Pronoun$	
	S6	$NP \rightarrow Det Noun PP$	
	S7	$VP \rightarrow \bullet Verb NP$	
	S8	$VP \rightarrow \bullet Verb$	
Chart [1]	S9	$Noun \rightarrow paint \bullet$	[0,1]
	S10	$Verb \rightarrow paint \bullet$	[0,1]
	S11	$NP \rightarrow Noun \bullet$	[0,1]
	S12	$VP \rightarrow Verb \bullet NP$	[0,1]
	S13	$VP \rightarrow Verb \bullet$	[0,1]
	S14	$S \rightarrow NP \bullet VP$	[0,1]
	S15	$NP \rightarrow \bullet Det Nominal$	[1,1]
	S16	$NP \rightarrow \bullet Noun$	[1,1]
	S17	$S \rightarrow VP \bullet$	[0,1]
	S18	$VP \rightarrow \bullet Verb NP$	[1,1]
	S19	$VP \rightarrow \bullet Verb$	[1,1]
Chart [2]	S20	$Det \rightarrow the \bullet$	[1,2]
	S21	$NP \rightarrow Det \bullet Nominal$	[1,2]
	S22	$Nominal \rightarrow \bullet Noun$	[2,2]
	S23	$Nominal \rightarrow \bullet Noun Nominal$	[2,2]
Chart [3]	S24	$Noun \rightarrow door \bullet$	[2,3]
	S25	$Nominal \rightarrow Noun \bullet$	[2,3]
	S26	$NP \rightarrow Det Nominal \bullet$	[1,3]
	S27	$S \rightarrow NP \bullet VP$	[0,3]
	S28	$VP \rightarrow Verb NP \bullet$	[0,3]
	S29	$VP \rightarrow \bullet Verb NP$	[3,3]
	S30	$VP \rightarrow \bullet Verb$	[3,3]

4.5 CYK Parser

- CYK (Cocke-Younger-Kasami) is a dynamic programming parsing algorithm.
- Follows a bottom-up approach in parsing.
- It builds a parse tree incrementally. Each entry in the table is based on previous entries. The process is iterated until the entire sentence has been parsed.
- Checks particular words is a part or member of particular grammar.
- The CYK parsing algorithm assumes the grammar to be in Chomsky normal form (CNF). A CFG is in CNF if all the rules are of only two forms: $A \rightarrow B C$ or $A \rightarrow w$, where w is a word.

Consider the following simplified grammar in CNF:

$S \rightarrow NP VP$ $Verb \rightarrow wrote$
 $VP \rightarrow Verb NP$ $Noun \rightarrow girl$
 $NP \rightarrow Det Noun$ $Noun \rightarrow essay$
 $Det \rightarrow an \mid the$

The sentence to be parsed is: *The girl wrote an essay.*

Table contains entries after a complete scan of the algorithm. The entry in the $[1, n]^{\text{th}}$ cell contains a start symbol which indicates that $S^* \Rightarrow W_{in}$ i.e., the parse is successful.

Create a triangular table where:

- Rows represent start positions in the sentence.

- Columns represent substrings of increasing length.

	1	2	3	4	5
1	Det → The	NP → Det Noun			S → NP VP
2	Noun → Girl				
3	Verb → wrote		VP → Verb NP		
4	Det → an	NP → Det Noun			
5	Noun → essay				

- Fill Base Case (Single Words): Find matching grammar rules for each word
- Fill Table for Larger Substrings: Now, we combine smaller segments.
- Check for Start Symbol (S): Since S appears in T[1,5], the sentence is valid under this grammar!

Algorithm:

```

Let w = w1 w2 w3 wi ... wj ... wn
    and wij = wi ... wi+j-1 //
Initialization step for i :=
1 to n do for all rules A →
wi do chart[i,1] = {A}
// Recursive step for j =
2 to n do for i = 1 to n-
j+1 do
begin chart[i,j] = ∅
    for k = 1 to j-1
    do
        chart[i,j] := chart[i,j] ∪ {A | A → BC is a production and
            B ∈ chart[i,k] and C ∈ chart[i+k,j-k]}
    end
end
if S ∈ chart[1,n] then accept else reject

```

Step 1: Initialization

- For each single-character substring w_i , find all non-terminals A such that $A \rightarrow w_i$ exists in the grammar.
- Store these A values in $\text{chart}[i,1]$.

Step 2: Recursive Filling

- For each substring of increasing length j (starting from 2 up to n):
 - Consider each possible starting position i .
 - Iterate through possible split points k where the substring is divided into two parts.
 - If there is a rule $A \rightarrow BC$ where:
 - B is in $\text{chart}[i, k]$
 - C is in $\text{chart}[i+k, j-k]$
 - Then, add A to $\text{chart}[i, j]$.

Step 3: Final Check

- If the start symbol S is in $\text{chart}[1, n]$, accept the string.
- Otherwise, reject the string.

5. Probabilistic Parsing

- Statistical parser, requires a corpus of hand-parsed text.

- The Penn tree-bank is a large corpus – consists Penn tree-bank tags, parsed based on simple set of phrase structure rules, Chomsky's government and binding syntax.
- The parsed sentences are represented in the form of properly bracketed trees.

Given a grammar G , **sentence s** , and a set of **possible parse trees of s** which we denote by $\mathcal{C}(s)$, a probabilistic parser finds the **most likely parse ' ϕ '** of s as follows:

$$\phi = \operatorname{argmax}_{\phi \in \mathcal{C}(s)} P(\phi | s) \quad \% \text{ where } \phi \text{ belongs to a feasible set } T(s) - \text{conditional probability.}$$

$$= \operatorname{argmax}_{\phi \in \mathcal{C}(s)} P(\phi, s) \quad \% \phi \text{ within the feasible set } T(s) \text{ that maximizes the joint probability } P(\phi, s).$$

$$= \operatorname{argmax}_{\phi \in \mathcal{C}(s)} P(\phi) \quad \% \phi \text{ within the feasible set } T(s) \text{ that maximizes the prior probability } P(\phi).$$

To construct a statistical parser:

We have to first **find all possible parses of a sentence**, then **assign probabilities** to them, and finally **return the most probable parse** ⑦ **probabilistic context-free grammars (PCFGs)**.

Benefits of statistical parser:

- A probabilistic parser helps resolve parsing ambiguity (multiple parse trees) by assigning probabilities to different parse trees, allowing selection of the most likely structure.
- It improves efficiency by narrowing the search space, reducing the time required to determine the final parse tree.

Probabilistic context-free grammar (PCFG):

- Every rule is assigned a probability. $A \rightarrow \alpha [p]$
 - Where p gives the probability of expanding a constituent using the rule: $A \rightarrow \alpha$.
- A PCFG is defined by the pair (G, f) , where G is a CFG and f is a positive function defined over the set of rules such that, the **sum of the probabilities associated with the rules expanding a particular non-terminal is 1**.

$$\sum_{\alpha} f(A \rightarrow \alpha) = 1$$

Example: PCFG is shown in Table, for each non-terminal, the sum of probabilities is 1.

$S \rightarrow NP VP$	0.8	$Noun \rightarrow door$	0.25
$S \rightarrow VP$	0.2	$Noun \rightarrow bird$	0.25
$NP \rightarrow Det Noun$	0.4	$Noun \rightarrow hole$	0.25
$NP \rightarrow Noun$	0.2	$Verb \rightarrow sleeps$	0.2
$NP \rightarrow Pronoun$	0.2	$Verb \rightarrow sings$	0.2
$NP \rightarrow Det Noun PP$	0.2	$Verb \rightarrow open$	0.2
$VP \rightarrow Verb NP$	0.5	$Verb \rightarrow saw$	0.2
$VP \rightarrow Verb$	0.3	$Verb \rightarrow paint$	0.2
$VP \rightarrow VP PP$	0.2	$Preposition \rightarrow from$	0.3
$PP \rightarrow Preposition NP$	1.0	$Preposition \rightarrow with$	0.25

Det→this	0.2	Preposition→on	0.2
Det→that	0.2	Preposition→to	0.25
Det→a	0.25	Pronoun→she	0.35
Det→the	0.35	Pronoun→he	0.35
Noun→paint	0.25	Pronoun→they	0.25

$$\begin{aligned}
& f(S \rightarrow NP VP) + f(S \rightarrow VP) = 1 \\
& f(NP \rightarrow Det Noun) + f(NP \rightarrow Noun) + f(NP \rightarrow Pronoun) \\
& + f(NP \rightarrow Det Noun PP) = 1 \\
& f(VP \rightarrow Verb NP) + f(NP \rightarrow Verb) + f(VP \rightarrow VP PP) = \\
& 1.0 \quad f(Det \rightarrow this) \quad + f(Det \rightarrow that) + f(Det \rightarrow a) + f(Det \rightarrow the) = 1.0 \\
& f(Noun \rightarrow paint) + f(Noun \rightarrow door) + f(Noun \rightarrow bird) + f(Noun \rightarrow hole) = 1.0
\end{aligned}$$

5.1 Estimating Rule Probabilities

- How are probabilities assigned to rules? (As shown in PCFG table)
- Manually construct a corpus of a parse tree for a set of sentences, and then estimate the probabilities of each rule being used by counting them over the corpus.
- The MLE estimate for a rule $A \rightarrow \alpha$ is given by the expression.

$$P_{MLE}(A \rightarrow \alpha) = \frac{\text{Count}(A \rightarrow \alpha)}{\sum_{\alpha} \text{Count}(A \rightarrow \alpha)}$$

If our training corpus consists of **two parse trees** (as shown in Figure), we will get the estimates as shown in Table for the rules.

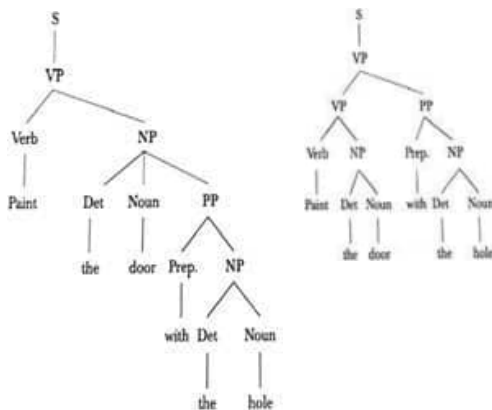


Figure: Two Parse trees

Rule	Count ($A \rightarrow \alpha$)	Count A	MLE estimates
$S \rightarrow VP$	2	2	1
$NP \rightarrow Det Noun PP$	1	4	0.25
$NP \rightarrow Det Noun$	3	4	0.75
$VP \rightarrow Verb NP$	2	3	0.66
$VP \rightarrow VP PP$	1	3	0.33
$Det \rightarrow the$	2	2	1
$Noun \rightarrow hole$	2	4	0.5
$Noun \rightarrow door$	2	4	0.5
$Prep \rightarrow with$	1	1	1
$Verb \rightarrow Paint$	1	1	1

Table: MLE for grammar rules considering two parse trees

Table: probabilistic context-free grammar (PCFG)

$S \rightarrow NP VP$	0.8	$Noun \rightarrow door$	0.25
$S \rightarrow VP$	0.2	$Noun \rightarrow bird$	0.25
$NP \rightarrow Det Noun$	0.4	$Noun \rightarrow hole$	0.25
$NP \rightarrow Noun$	0.2	$Verb \rightarrow sleeps$	0.2

NP→Pronoun	0.2	Verb→sings	0.2
NP→Det Noun PP	0.2	Verb→open	0.2
VP→Verb NP	0.5	Verb→saw	0.2
VP→Verb	0.3	Verb→paint	0.2
VP→VP PP	0.2	Preposition→from	0.3
PP→Preposition NP	1.0	Preposition→with	0.25
Det→this	0.2	Preposition→on	0.2
Det→that	0.2	Preposition→to	0.25
Det→a	0.25	Pronoun→she	0.35
Det→the	0.35	Pronoun→he	0.35
Noun→paint	0.25	Pronoun→they	0.25

What do we do with these probabilities?

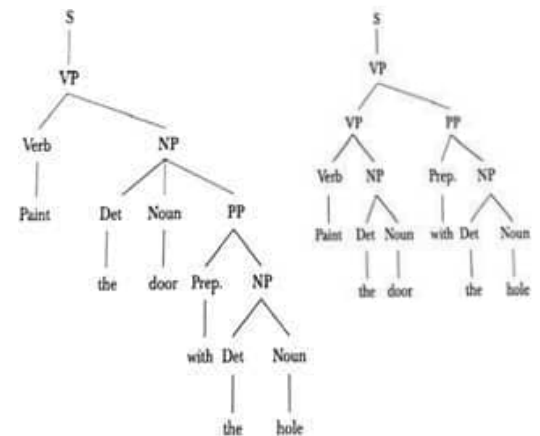
- We assign a probability to each parse tree ϕ of a sentence s .
- The probability of a complete parse is calculated by multiplying the probabilities for each of the rules used in generating the parse tree:

$$P(\phi, s) = \prod_{n \in \phi} p\{r(n)\}$$

Where n is a node in the parse tree ϕ and r is the rule used to expand n .

The probability of the two parse trees of the sentence *Paint the door with the hole* (shown in Figure) using PCFG table can be computed as follows:

S→NP VP	0.8	Noun→door	0.25
S→VP	0.2	Noun→bird	0.25
NP→Det Noun	0.4	Noun→hole	0.25
NP→Noun	0.2	Verb→sleeps	0.2
NP→Pronoun	0.2	Verb→sings	0.2
NP→Det Noun PP	0.2	Verb→open	0.2
VP→Verb NP	0.5	Verb→saw	0.2
VP→Verb	0.3	Verb→paint	0.2
VP→VP PP	0.2	Preposition→from	0.3
PP→Preposition NP	1.0	Preposition→with	0.25
Det→this	0.2	Preposition→on	0.2
Det→that	0.2	Preposition→to	0.25
Det→a	0.25	Pronoun→she	0.35
Det→the	0.35	Pronoun→he	0.35
Noun→paint	0.25	Pronoun→they	0.25



$$P(t_1) = 0.2 * 0.5 * 0.2 * 0.2 * 0.35 * 0.25 * 1.0 * 0.25 * 0.4 * 0.35 * 0.25 = 0.0000030625$$

$$P(t_2) = 0.2 * 0.2 * 0.5 * 0.2 * 0.4 * 0.35 * 0.25 * 1 * 0.25 * 0.4 * 0.35 * 0.25 = 0.000001225$$

The first tree has a higher probability leading to correct interpretation.

We can calculate probability to a sentence s by summing up probabilities of all possible parses associated with it.

$$P(s) = \sum_{t \in \tau(s)} P(t, s)$$

$$= \sum_{t \in \tau(s)} P(t)$$

The sentence will have the probability=

$$P(t_1) + P(t_2) = 0.0000030625 + 0.000001225$$

$$= 0.0000042875$$

5.2 Parsing PCFGs

Given a PCFG, a probabilistic parsing algorithm assigns *the most likely parse* $\mathcal{Q}$ to a sentence s .

$$\mathcal{Q} = \operatorname{argmax}_{T \in \tau(s)} P(T | s)$$

where $\tau(S)$ is the set of all possible parse trees of s .

Probabilistic CYK $w = W_1 W_2 W_3 W_j \dots W_j \dots W_n$ represents a sentence consisting of n words.

Let $\mathcal{Q}[i, j, A]$ represent the maximum probability parse for a constituent with non-terminal A spanning words $i, i+1$, up to $i+j-1$. This means it is a sub-tree rooted at “ A ” that derives sequence of “ j ” words beginning at position “ i ” and has a probability greater than all other possible sub-trees deriving the same word sequence.

Initialization:

```

for  $i := 1$  to  $n$  do
  for all rules  $A \rightarrow w_i$  do
     $\phi[i, 1, A] = P(A \rightarrow w_i)$ 

Recursive Step:
for  $j = 2$  to  $n$  do
  for  $i = 1$  to  $n-j+1$  do
    begin
       $\phi[i, j, A] = \phi$ 
      for  $k = 1$  to  $j-1$  do
         $\phi'[i, j, A] = \max_k \phi'[i, k, B] \times \phi'[k, j, C] \times P(A \rightarrow BC)$ ,
        such that  $A \rightarrow BC$  is a production rule in grammar
       $BP[i, j, A] = \{ k, A, B \}$ 
    end
  end

```

- An array named BP is used to store back pointers. These pointers allow us to recover the best parse.

- Initialize the maximum probable parse trees deriving a string of length 1, with the probabilities of the terminal derivation rules used to derive them.

$$\phi[i, 1, A] = P(A \rightarrow w_i)$$

- Recursive step involves breaking a string into all possible ways and identifying the maximum probable parse.

$$\phi'[i, j, A] = \max_k \phi'[i, k, B] \times \phi'[k, j, C] \times P(A \rightarrow BC)$$

- The rest of the steps follow those of basic CYK parsing algorithm.

5.3 Problems with PCFG

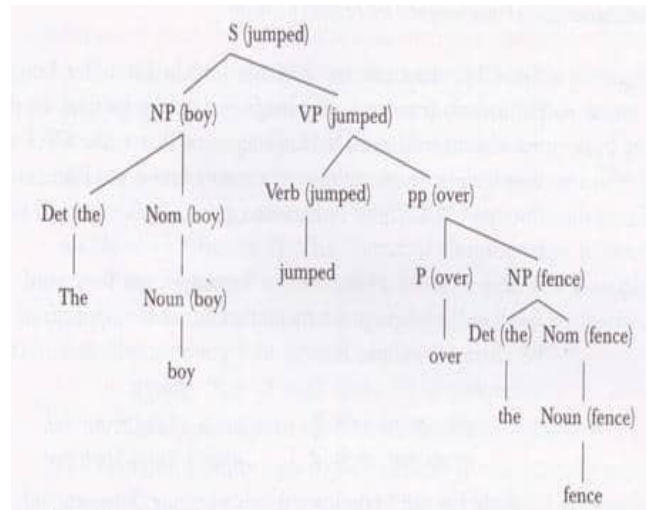
- The probability of a parse tree assumes that the rules are independent of each other.

- Example: Pronouns occur more frequently as subjects rather than objects.
- These dependencies are not captured by a PCFG. ○ Expanding an NP as a pronoun versus a lexical NP ○ NP appears as a subject or an object.
- Lack of sensitivity to lexical information.
 - Two structurally different parses that use the same rules will have the same probability under a PCFG.

Solution: This however, **requires a model which captures lexical dependency statistics** for different words. **7 Lexicalization**

Lexicalization

- Words do affect the choice of the rule.
- **Involvement of actual words** in the sentences, **to decide the structure of the parse tree.**
- Lexicalization is also helpful in choosing phrasal attachment positions.
- One way to achieve lexicalization is to mark each phrasal node in a parse tree by its head word.
- This **lexicalized version** keeps track of **headwords** (e.g., "jumped" in VP) and improves parsing accuracy.
- A **lexicalized PCFG** assigns **specific words** to rules, making parsing **more accurate** by capturing relationships between words.
 - The **verb (jumped)** affects parsing probability. ○ **Dependencies** between words like "jumped" and "boy" are captured.
 - A sentence like **"The boy jumped over the fence"** is parsed more accurately.



6. Indian Languages

- Some of the characteristics of Indian languages that make CFG unsuitable.
- Paninian grammar can be used to model Indian languages.
 1. Indian languages are **free word order**.

○ सब खाना खाती है। Saba khana khati hai. ○ खाना सब
खाती है। Khana Saba khati hai.

The CFG we used for parsing English is basically positional, but it fails to model free word order languages.

2. **Complex predicates (CPs)** is another property that most Indian languages have in common. • A complex predicate combines a light verb with a verb, noun, or adjective, to produce a new verb.

- For example:

(a) सबा आयी। ७ (Saba Ayi.) ७ Saba came.

(b) सबा आ गयी। ७ (Saba a gayi.) ७ Saba come went. ७ Saba arrived.

(c) सबा आ पड़ी। ७ Saba a pari. ७ Saba come fell. ७ Saba came (suddenly).

The use of post-position case markers and the auxiliary verbs in this sequence provide information about tense, aspect, and modality.

Paninian grammar provides a framework to model Indian languages. It focuses on the extraction of Karak relations from a sentence.

Bharti and Sangal (1990) described an approach for parsing of Indian languages based on Paninian grammar formalism. Their parser works in two stages.

1st stage: Identifying word groups.

2nd stage: Assigning a parse structure to the input sentence.

Example:

लड़कियाँ मैदान में हार्की खेल रही हैं।

Ladkiyan maidaan mein hockey khel rahi hein.

1st stage:

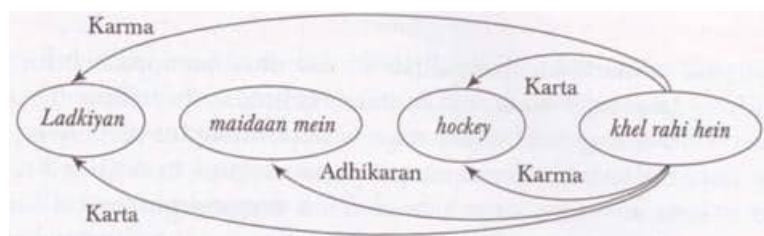
- Word *ladkiyan* forms one unit, the words *maidaan* and *mein* are grouped together to form a noun group, and the word sequence *khel rahi hein* forms a verb group.

2nd stage:

- The parser takes the word groups formed during first stage and identifies (i) Karaka relations among them, and (ii) senses of words.
- Karaka chart** is created to store additional information like Karaka-Vibhakti mapping.

Karaka (case relations)	Vibhakti (case markers or post-positions)	Necessity
Karta	φ	Mandatory
Karma	Ko or φ	Mandatory
Adhikaran	Mein or par	Optional
Sampradan	Ko or ke liye	Optional

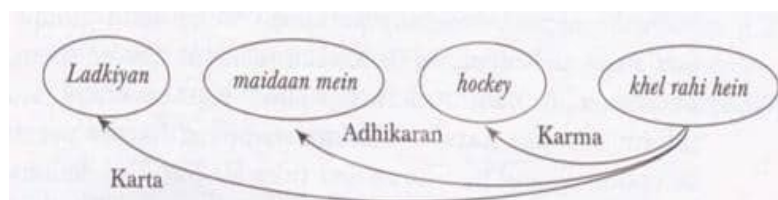
- Constraint graph for sentence:** The Karaka relation between a verb group and a noun group can be depicted using a constraint graph.



• **A parse of the sentence:**

Each sub-graph of the constraint graph that satisfies the following constraints yields a parse of the sentence.

1. It contains all the nodes of the graph.
2. It contains exactly one outgoing edge from a verb group for each of its mandatory Karakas. These edges are labelled by the corresponding Karaka.
3. For each of the optional Karaka in Karaka chart, the sub-graph can have at most one outgoing edge labelled by the Karaka from the verb group.
4. For each noun group, the sub-graph should have exactly one incoming edge.



Question Bank

1. Define a finite automaton that accepts the following language: (aa)(bb).
2. A typical URL is of the form:

http	://	<u>www.abc.com</u>	/nlppaper/public	/xxx.html
1	2	3	4	5

In this table, 1 is a protocol, 2 is name of a server, 3 is the directory, and 4 is the name of a document. Suppose you have to write a program that takes a URL and returns the protocol used, the DNS name of the server, the directory and the document name. Develop a regular expression that will help you in writing this program.

3. Distinguish between non-word and real-word error.
4. Compute the minimum edit distance between paecflu and peaceful.
5. Comment on the validity of the following statements:

- (a) Rule-based taggers are non-deterministic.
- (b) Stochastic taggers are language independent.
- (c) Brill's tagger is a rule-based tagger.

6. How can unknown words be handled in the tagging process?
7. Give two possible parse trees for the sentence, Stolen painting found by tree.
8. Identify the noun and verb phrases in the sentence, My soul answers in music.\
9. Give the correct parse of sentence.
10. Discuss the disadvantages of the basic top-down parser with the help of an appropriate example.
11. Tabulate the sequence of states created by CYK algorithm while parsing, The sun rises in the east. Augment the grammar in section 4.4.5 with appropriate rules of lexicon.
12. Discuss the disadvantages of probabilistic context free grammar.
13. What does lexicalized grammar mean? How can lexicalization be achieved? Explain with the help of suitable examples.
14. List the characteristics of a garden path sentence. Give an example of a garden path sentence and show its correct parse.
15. What is the need of lexicalization?
16. Use the following grammar:

$S \rightarrow NP VP$ $S \rightarrow VP$ $NP \rightarrow Det \quad Noun$
 $NP \rightarrow Noun$ $NP \rightarrow NP PP$ $VP \rightarrow VP PP$
 $VP \rightarrow Verb$ $VP \rightarrow VP NP$ $PP \rightarrow Preposition \quad NP$

Give two possible parse of the sentence: 'Pluck the flower with the stick. Introduce lexicon rules for words appearing in the sentence. Using these parse trees obtain maximum likelihood estimates for the grammar rules used in the tree. Calculate probability of any one parse tree using these estimates.

Lab Exercises

1. Write a program to find minimum edit distance between two input strings.
2. Use any tagger available in your lab to tag a text file. Now write a program to find the most likely tag in the tagged text.
3. Write a program to find the probability of a tag given previous two tags, i.e., $P(t_3/t_2 t_1)$.
4. Write a program to extract all the noun phrases from a text file. Use the phrase structure rule given in this chapter.

5. Write a program to check whether a given grammar is context free grammar or not.
6. Write a program to convert a given CFG grammar in CNF.
7. Write a program to implement a basic top-down parser.
8. Implement Earley parsing algorithm.