SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES,
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
III B.TECH II SEMESTER CSE R23 REGULATION
LECTURE NOTES
NATURAL LANGUAGE PROCESSING (23CAI353T)

UNIT II

UNIT II: Word-Level and Syntactic Analysis

Introduction, Part-of-Speech (POS) Tagging: Rule-Based, Stochastic and Transformation-Based Approaches, Hidden Markov Models (HMM) and Maximum Entropy Models for POS Tagging, Context-Free Grammar (CFG) and Constituency Parsing, Treebanks and Normal Forms for Grammar, Top-Down and Bottom-Up Parsing Strategies, CYK Parsing Algorithm, Probabilistic Context-Free Grammars (PCFGs), Feature Structures and Unification.

WORD-LEVEL AND SYNTACTIC ANALYSIS

- **Word-level analysis**: Tokens, POS tags, morphological structure, and named entities.
- **Syntactic analysis**: Parses sentence structure using **constituency or dependency grammars**.
- Together, they enable machines to **understand language structure** for various NLP applications.

1. Introduction

- **Word-level analysis** deals with understanding individual words, their forms, and meanings.
- **Syntactic analysis** (or parsing) deals with **sentence structure** and grammatical relationships between words.
- Both are essential for **semantic understanding, machine translation, question answering, and text generation**.

2. Word-Level Analysis

1. **Tokenization**
   - Breaking text into **words or subwords**.
2. **Morphological Analysis**
   - Examining **word structure** (prefix, root, suffix).
   - Includes **stemming** and **lemmatization**.
3. **Part-of-Speech (POS) Tagging**
   - Assign grammatical categories to words: noun, verb, adjective, etc.
   - Example: "The cat sleeps." → The/DT cat/NN sleeps/VB

4. **Named Entity Recognition (NER)**
   - Identify **proper nouns** like person names, locations, organizations.
   - Example: "Barack Obama was born in Hawaii." → Barack Obama/PER, Hawaii/LOC

3. Syntactic Analysis

- **Goal:** Understand the **structure of sentences** and relationships between words.

1. **Constituency Parsing (Phrase Structure)**
   - Represents sentences as **nested phrases** (NP, VP, etc.)
   - Example: "The cat sleeps" → S → NP (The cat) + VP (sleeps)
2. **Dependency Parsing**
   - Represents **grammatical relationships** as directed links between words.
   - Example: "The cat sleeps" → sleeps is head, cat is subject.
3. **Grammar Formalisms**
   - **Context-Free Grammar (CFG)**: Rules like S → NP VP
   - **Dependency Grammar**: Focus on head-dependent relations

4. Applications of Word-Level and Syntactic Analysis

- **Machine Translation**: Understand structure to preserve meaning.
- **Question Answering**: Identify subjects, objects, and relations.
- **Information Extraction**: Extract structured data from unstructured text.
- **Text Summarization**: Analyze sentence structure for key points.
- **Spell Checking & Grammar Correction**: Detect structural errors.

5. Challenges

- **Ambiguity**: Words can have multiple POS tags.
  - Example: "Book a flight" vs "Read a book"
- **Complex Sentences**: Handling nested or long sentences.
- **Free Word Order Languages**: Harder parsing for languages like Hindi or Japanese.

- **Resource Scarcity**: Limited annotated corpora for many languages.

PART-OF-SPEECH (POS) TAGGING

POS tagging is a **key step in NLP pipelines** that labels each word with its grammatical category. Techniques range from **rule-based to deep learning**, and it is crucial for **parsing, translation, and semantic understanding**.

1. Introduction

- **POS Tagging** is the process of assigning **grammatical categories** (noun, verb, adjective, etc.) to each word in a sentence.
- It is a **fundamental step** in NLP for:
  - **Syntactic parsing**
  - **Information extraction**
  - **Machine translation**
  - **Question answering**

2. Common POS Tags

- **Noun (NN)**: cat, book, city
- **Proper Noun (NNP)**: John, India
- **Verb (VB)**: run, eat
- **Adjective (JJ)**: big, red
- **Adverb (RB)**: quickly, very
- **Determiner (DT)**: the, a, an
- **Pronoun (PRP)**: he, she, it
- **Preposition (IN)**: in, on, at
- **Conjunction (CC)**: and, or, but

Example:
Sentence: "The cat sleeps on the mat."
POS Tagged: The/DT cat/NN sleeps/VB on/IN the/DT mat/NN ./.

3. Techniques for POS Tagging

1. **Rule-Based Tagging**
   - Uses **hand-crafted linguistic rules** and dictionaries.
   - Example Rule: If a word ends with *-ing* → likely a verb.
2. **Stochastic / Probabilistic Tagging**
   - Uses **statistical models** like **Hidden Markov Models (HMMs)**.
   - Assigns tags based on probability of a sequence: P(tag|previous tag).
3. **Transformation-Based Tagging**
   - Uses **Brill's tagger**: learns rules from annotated corpora.
4. **Neural Network / Deep Learning Models**
   - LSTM, Bi-LSTM, Transformers (BERT-based taggers)
   - Capture long-range dependencies and context.

4. Applications of POS Tagging

- **Parsing & Grammar Checking:** Identifies structure and grammatical errors.
- **Information Extraction:** Extracts entities, relations, and facts.
- **Machine Translation:** Helps preserve syntactic structure across languages.
- **Text-to-Speech Systems:** Determines correct pronunciation (e.g., "lead" as noun vs verb).
- **Word Sense Disambiguation:** Helps infer meaning using syntactic context.

5. Challenges

- **Ambiguity:** Words with multiple possible tags.
  - Example: "Can" → verb or modal auxiliary?
- **Unknown Words:** Words not present in the training corpus.
- **Domain Variation:** POS patterns differ across text types (news, social media, medical text).
- **Free Word Order Languages:** Complexity increases for languages like Hindi or Japanese.

RULE-BASED POS TAGGING

Rule-Based POS tagging relies on a **dictionary + handcrafted linguistic rules** to assign grammatical categories. While **interpretable and useful for small-scale systems**, it struggles with ambiguity and scalability.

1. Introduction

- **Rule-Based POS Tagging** assigns a part-of-speech (POS) to each word in a sentence using:
    1. **Lexical knowledge** (dictionary of words and possible tags)
    2. **Hand-crafted linguistic rules**
- One of the **earliest POS tagging methods**, especially before statistical approaches became common.

2. Components of a Rule-Based Tagger

1. **Lexicon / Dictionary**
    - A list of words and their possible POS tags.
    - Example:
        - "book" → noun, verb
        - "run" → noun, verb
2. **Rules**
    - Linguistic rules applied to resolve ambiguity.
    - Types of rules:
        - **Contextual Rules**: Use surrounding words to decide the tag.
            - Example: If a word follows a determiner (DT), tag it as a noun (NN).
        - **Morphological Rules**: Use word suffix/prefix patterns.
            - Example: Words ending in -ing → verb (VBG).
        - **Fallback Rules**: Default to the most common tag in lexicon if no other rules apply.

3. Working of a Rule-Based Tagger

1. **Look up each word** in the dictionary for possible POS tags.
2. **Apply disambiguation rules** based on context or morphology.
3. **Assign the most appropriate tag** to each word.

Example:
Sentence: "The cat sleeps on the mat."

- Lexicon lookup:
    - "The" → DT
    - "cat" → NN
    - "sleeps" → VB, NNS
- Rule application:
    - If previous word = DT, current word = NN → "cat" tagged as NN
    - "sleeps" follows NN → likely VB → tagged as VB

Output: "The/DT cat/NN sleeps/VB on/IN the/DT mat/NN ./"

4. Advantages

- **No training data required**
- Can be **very accurate** for well-defined domains
- Easy to **interpret and debug**

5. Disadvantages

- **Labor-intensive**: Rules must be manually crafted for each language.
- **Limited coverage**: Cannot handle all lexical ambiguities or unknown words.
- **Not scalable** for large corpora or multiple languages.
- **Context limitation**: Cannot capture long-range dependencies like neural models.

6. Applications

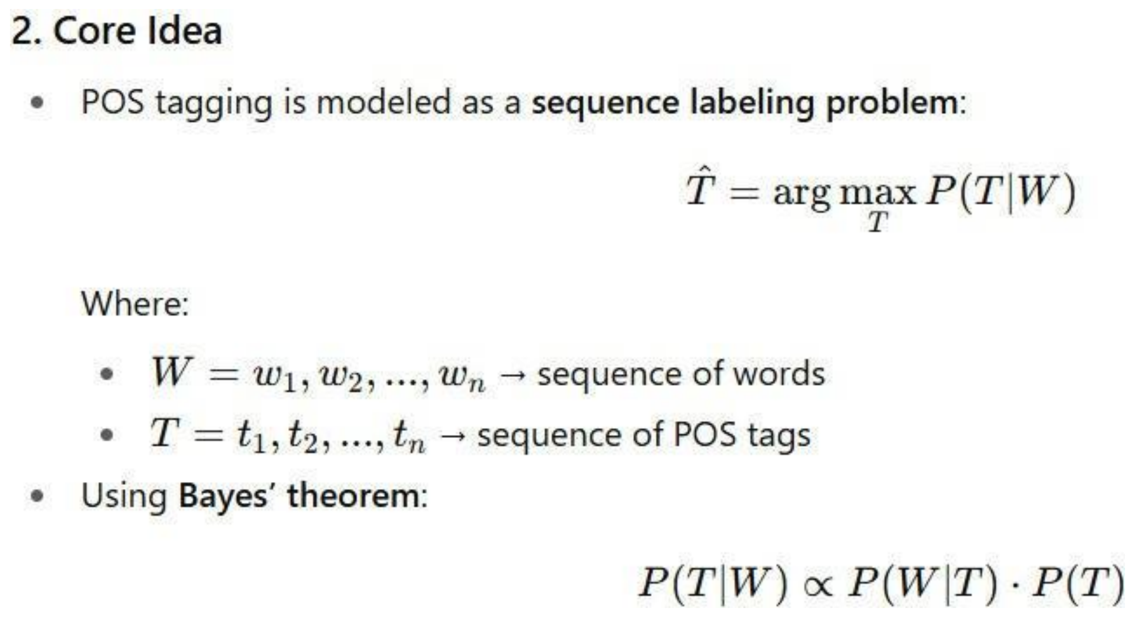
- Early **POS tagging systems** in English and other languages.
- Useful in **domain-specific NLP systems** where training data is scarce.

- Basis for **hybrid systems** combining rule-based + statistical models.

STOCHASTIC (PROBABILISTIC) POS TAGGING

1. Introduction

- **Stochastic POS Tagging** assigns part-of-speech tags based on **probabilities derived from annotated corpora**.
- It is also called **statistical POS tagging**.
- Uses **contextual information** to resolve ambiguities that rule-based methods may fail to handle.



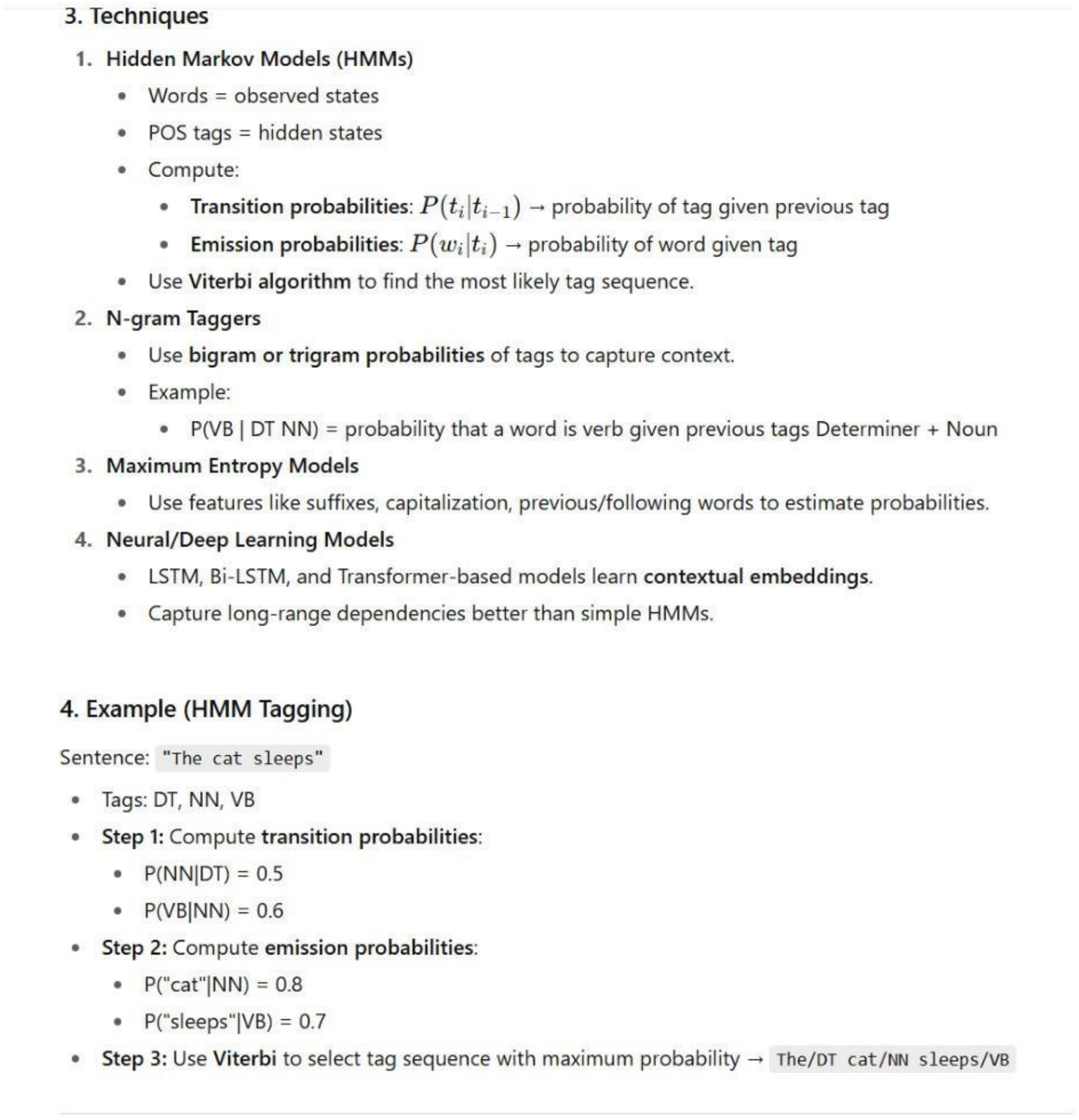
2. Core Idea

- POS tagging is modeled as a **sequence labeling problem**:

$$\hat{T} = \arg\max_{T} P(T|W)$$

  Where:
  - $W = w_1, w_2, ..., w_n$ → sequence of words
  - $T = t_1, t_2, ..., t_n$ → sequence of POS tags
- Using **Bayes' theorem**:

$$P(T|W) \propto P(W|T) \cdot P(T)$$



3. Techniques

1. **Hidden Markov Models (HMMs)**
   - Words = observed states
   - POS tags = hidden states
   - Compute:
     - **Transition probabilities**: $P(t_i|t_{i-1})$ → probability of tag given previous tag
     - **Emission probabilities**: $P(w_i|t_i)$ → probability of word given tag
   - Use **Viterbi algorithm** to find the most likely tag sequence.
2. **N-gram Taggers**
   - Use **bigram or trigram probabilities** of tags to capture context.
   - Example:
     - P(VB | DT NN) = probability that a word is verb given previous tags Determiner + Noun
3. **Maximum Entropy Models**
   - Use features like suffixes, capitalization, previous/following words to estimate probabilities.
4. **Neural/Deep Learning Models**
   - LSTM, Bi-LSTM, and Transformer-based models learn **contextual embeddings**.
   - Capture long-range dependencies better than simple HMMs.

4. Example (HMM Tagging)

Sentence: `"The cat sleeps"`

- Tags: DT, NN, VB
- **Step 1:** Compute **transition probabilities**:
  - P(NN|DT) = 0.5
  - P(VB|NN) = 0.6
- **Step 2:** Compute **emission probabilities**:
  - P("cat"|NN) = 0.8
  - P("sleeps"|VB) = 0.7
- **Step 3:** Use **Viterbi** to select tag sequence with maximum probability → `The/DT cat/NN sleeps/VB`

5. Advantages

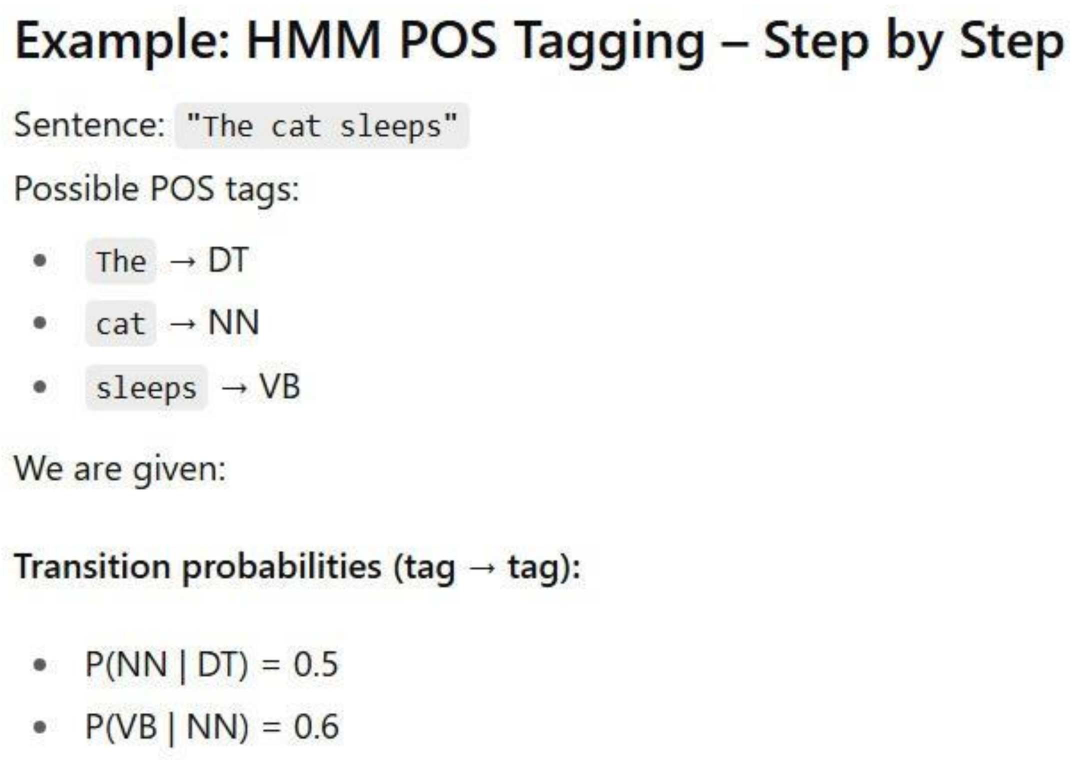
- Can handle **ambiguous words** using context.
- Learns from **data**, adaptable to new domains.
- Scalable for large corpora and multiple languages.

6. Disadvantages

- Requires **annotated corpora** for training.
- Probabilities may be **sparse** for rare words or sequences.
- Cannot capture **long-range dependencies** without advanced models.

7. Applications

- **POS tagging** in large-scale corpora
- **Syntactic parsing**
- **Machine translation**
- **Speech recognition**



Example: HMM POS Tagging – Step by Step

Sentence: `"The cat sleeps"`

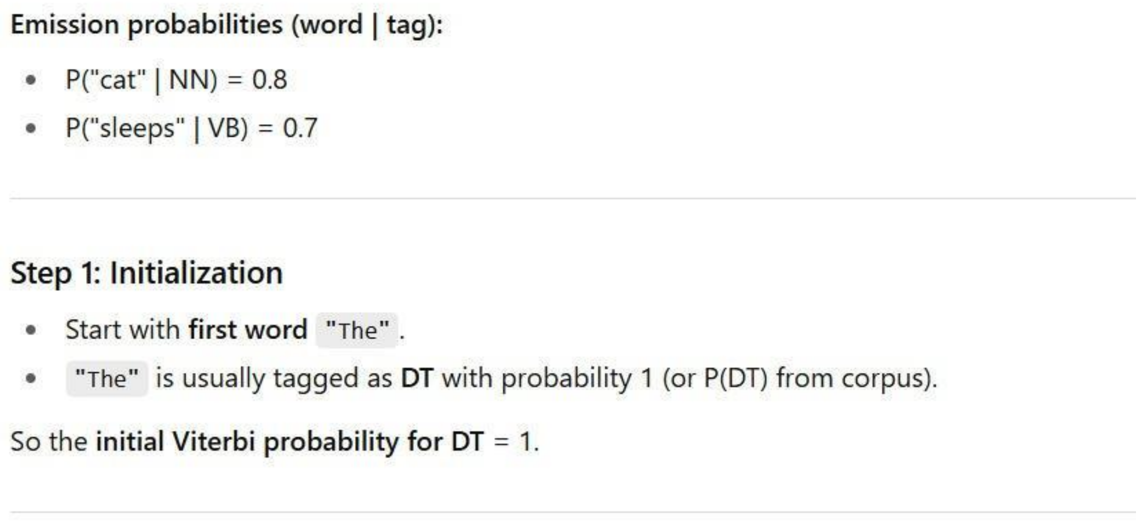
Possible POS tags:

- `The` → DT
- `cat` → NN
- `sleeps` → VB

We are given:

Transition probabilities (tag → tag):

- P(NN | DT) = 0.5
- P(VB | NN) = 0.6



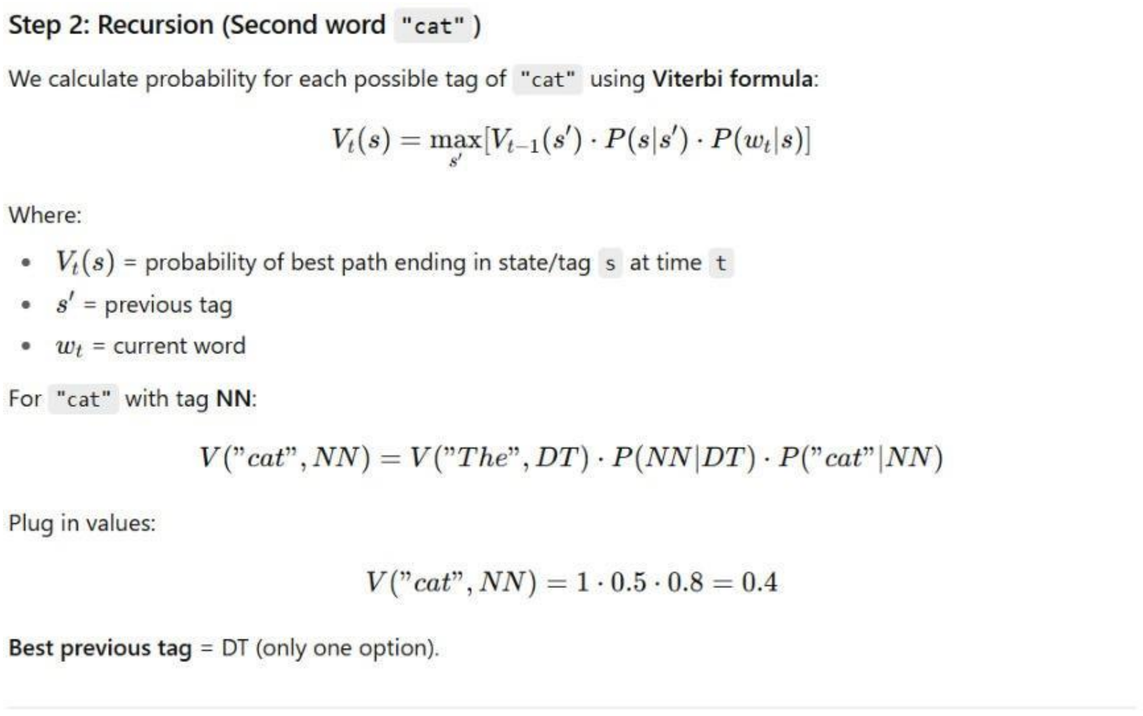
Emission probabilities (word | tag):

- P("cat" | NN) = 0.8
- P("sleeps" | VB) = 0.7

Step 1: Initialization

- Start with **first word** `"The"`.
- `"The"` is usually tagged as **DT** with probability 1 (or P(DT) from corpus).

So the **initial Viterbi probability for DT** = 1.



Step 2: Recursion (Second word `"cat"`)

We calculate probability for each possible tag of `"cat"` using **Viterbi formula**:

$$V_t(s) = \max_{s'}[V_{t-1}(s') \cdot P(s|s') \cdot P(w_t|s)]$$

Where:

- $V_t(s)$ = probability of best path ending in state/tag `s` at time `t`
- s' = previous tag
- w_t = current word

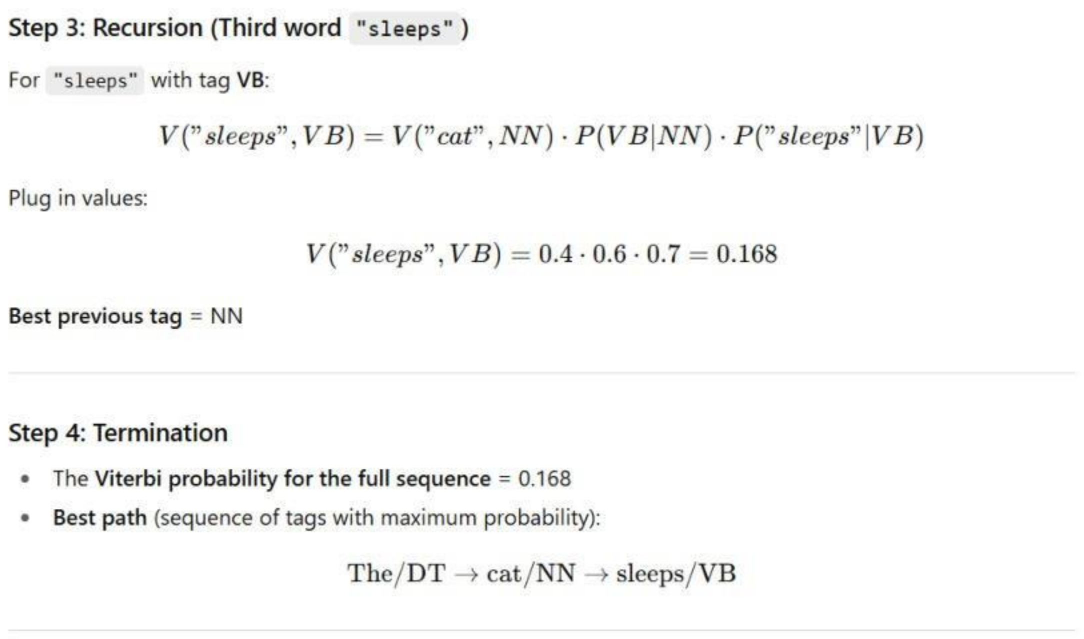
For `"cat"` with tag **NN**:

$$V(\text{"}cat\text{"}, NN) = V(\text{"}The\text{"}, DT) \cdot P(NN|DT) \cdot P(\text{"}cat\text{"}|NN)$$

Plug in values:

$$V(\text{"}cat\text{"}, NN) = 1 \cdot 0.5 \cdot 0.8 = 0.4$$

Best previous tag = DT (only one option).



Step 3: Recursion (Third word `"sleeps"`)

For `"sleeps"` with tag **VB**:

$$V(\text{"}sleeps\text{"}, VB) = V(\text{"}cat\text{"}, NN) \cdot P(VB|NN) \cdot P(\text{"}sleeps\text{"}|VB)$$

Plug in values:

$$V(\text{"}sleeps\text{"}, VB) = 0.4 \cdot 0.6 \cdot 0.7 = 0.168$$

Best previous tag = NN

Step 4: Termination

- The **Viterbi probability for the full sequence** = 0.168
- **Best path** (sequence of tags with maximum probability):

$$\text{The/DT} \rightarrow \text{cat/NN} \rightarrow \text{sleeps/VB}$$

Step 5: Interpretation

- The Viterbi algorithm **chooses the most probable tag sequence** based on:
    1. **Transition probabilities** (grammar context)
    2. **Emission probabilities** (likelihood of word given tag)
- Final tagging respects both **word meaning** and **syntactic context**.

Word	Tag	Viterbi Probability	Previous Tag
The	DT	1	–
cat	NN	0.4	DT
sleeps	VB	0.168	NN

TRANSFORMATION-BASED POS TAGGING (BRILL TAGGER)

- **Transformation-Based POS Tagging** starts with an **initial tagging**, then **iteratively corrects errors** using learned rules from training data.

- It is **interpretable, accurate**, and widely used in hybrid NLP systems.

1. Introduction

- **Transformation-Based Learning (TBL)** is a **hybrid approach** to POS tagging.
- Introduced by **Eric Brill (1992)**.
- Combines the **accuracy of rule-based tagging** with **learning from annotated corpora**.
- Tags are initially assigned using a **simple method**, then **iteratively improved** by applying learned transformations.

2. Core Idea

1. Start with **initial tags** (e.g., from a lexicon or default most frequent tag).
2. **Learn rules** that correct errors in the tagged text using a **training corpus**.
3. Apply the **learned transformation rules** to new text.

- Each **transformation rule** has the form:

“Change tag **X** to **Y** when the word’s context satisfies condition **C**.”

3. Example of Transformation Rules

- **Initial tagging:** Most frequent tag per word
- **Transformation rules learned from data:**
    1. Change NN → VB if the previous word is "to"
        - Example: "to run" → run/VB
    2. Change NN → JJ if the next word is "car"
        - Example: "fast car" → fast/JJ car/NN
    3. Change VB → NN if the word ends with -ing and previous word is the

4. Advantages

- **High accuracy**: Can approach state-of-the-art POS tagging performance.
- **Interpretable rules**: Each rule is readable and understandable.
- **Requires less data** than fully statistical methods.
- **Combines advantages** of rule-based and statistical approaches.

5. Disadvantages

- **Training can be time-consuming** for large corpora.
- May **not generalize well** to highly different domains.
- Complexity increases if too many rules are learned.

6. Applications

- **POS Tagging** in English and other languages.
- Basis for **hybrid NLP systems** that combine rules and statistical models.
- Useful for **error correction in tagging** ambiguous or rare words.

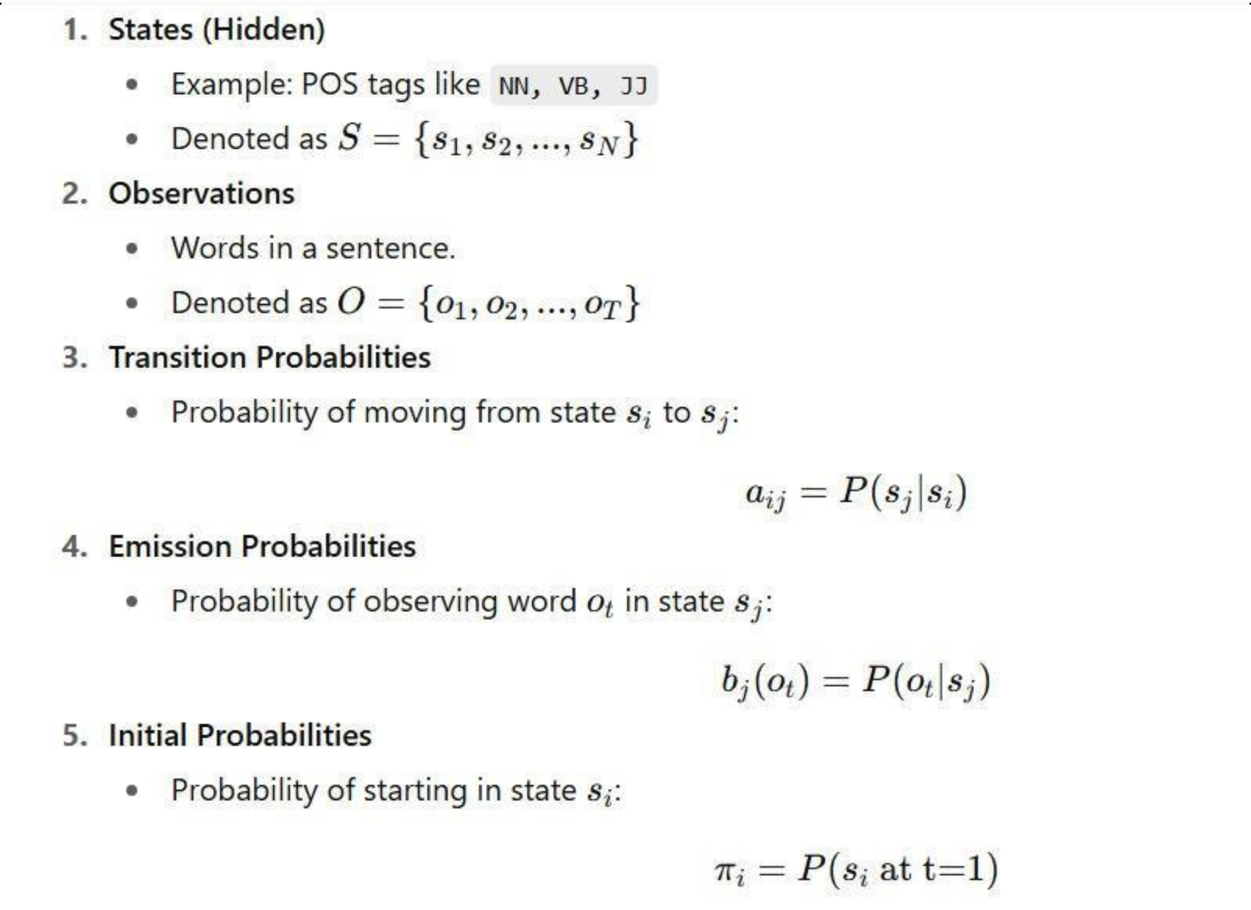
HIDDEN MARKOV MODELS (HMM)

- **HMMs model sequential data in NLP where the observations (words) are visible but states (tags) are hidden.**
- **Core components: transition, emission, and initial probabilities.**
- **The Viterbi algorithm decodes the best sequence of hidden states.**

1. Introduction

- **HMMs** are **statistical models** for sequences where the system is assumed to be a **Markov process with hidden states**.
- Widely used in NLP tasks such as:
  - **POS tagging**
  - **Speech recognition**
  - **Named Entity Recognition (NER)**
- Key idea: **We observe outputs (words), but the underlying states (tags) are hidden.**

2. Components of an HMM



1. **States (Hidden)**
   - Example: POS tags like `NN, VB, JJ`
   - Denoted as $S = \{s_1, s_2, ..., s_N\}$
2. **Observations**
   - Words in a sentence.
   - Denoted as $O = \{o_1, o_2, ..., o_T\}$
3. **Transition Probabilities**
   - Probability of moving from state s_i to s_j:

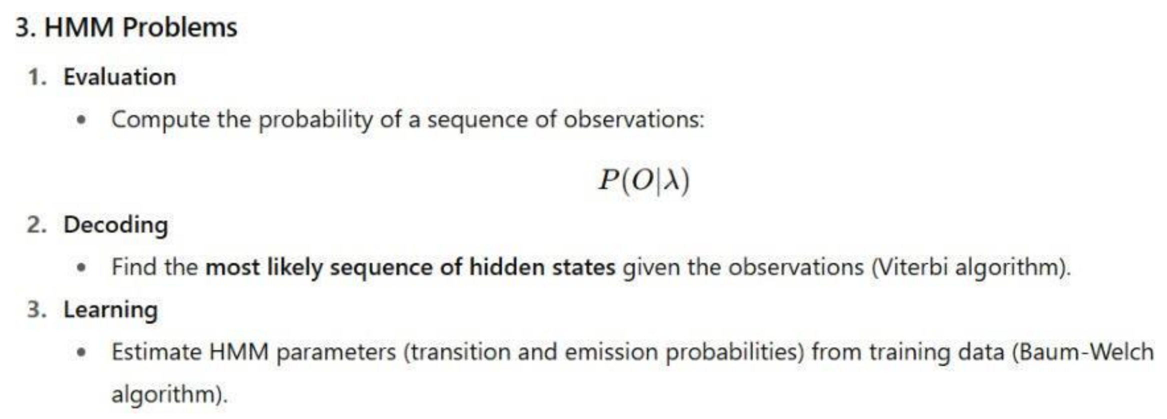
$$a_{ij} = P(s_j|s_i)$$

4. **Emission Probabilities**
   - Probability of observing word o_t in state s_j:

$$b_j(o_t) = P(o_t|s_j)$$

5. **Initial Probabilities**
   - Probability of starting in state s_i:

$$\pi_i = P(s_i \text{ at t=1})$$

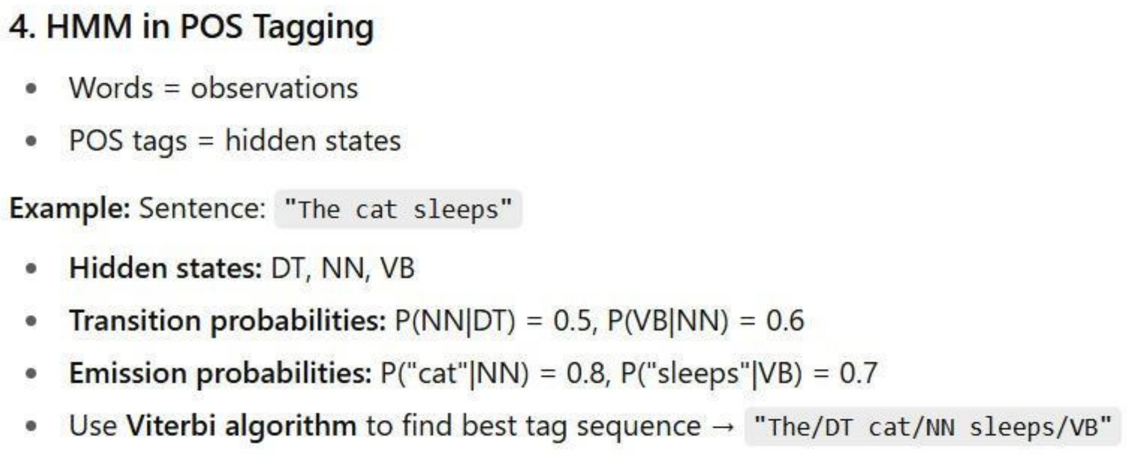


3. HMM Problems

1. **Evaluation**
   - Compute the probability of a sequence of observations:

$$P(O|\lambda)$$

2. **Decoding**
   - Find the **most likely sequence of hidden states** given the observations (Viterbi algorithm).
3. **Learning**
   - Estimate HMM parameters (transition and emission probabilities) from training data (Baum-Welch algorithm).



4. HMM in POS Tagging

- Words = observations
- POS tags = hidden states

Example: Sentence: `"The cat sleeps"`

- **Hidden states:** DT, NN, VB
- **Transition probabilities:** P(NN|DT) = 0.5, P(VB|NN) = 0.6
- **Emission probabilities:** P("cat"|NN) = 0.8, P("sleeps"|VB) = 0.7
- Use **Viterbi algorithm** to find best tag sequence → `"The/DT cat/NN sleeps/VB"`

5. Advantages

- Captures sequential dependencies between tags.
- Probabilistic approach handles ambiguity naturally.
- Can be trained from annotated corpora.

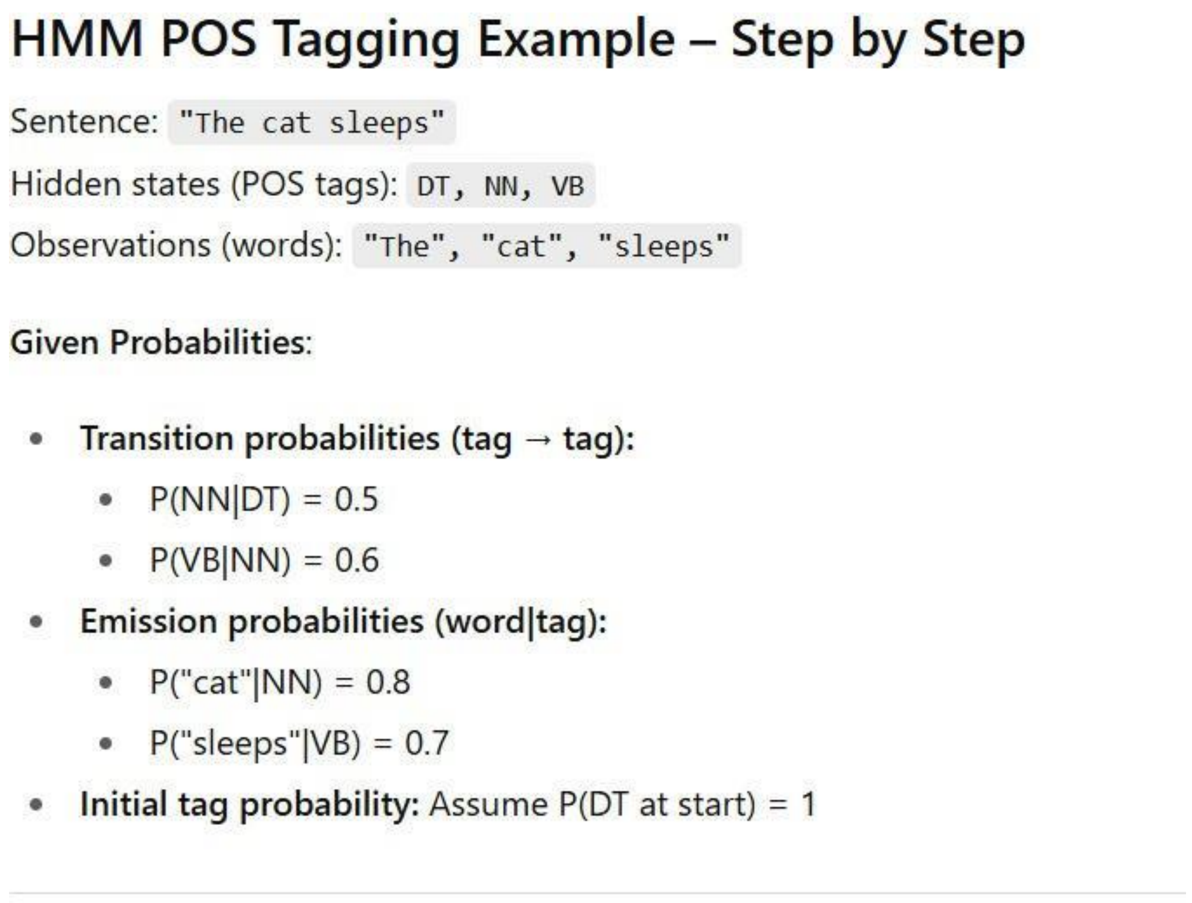
6. Disadvantages

- Assumes Markov property (current state depends only on previous state), ignoring long-range dependencies.
- Emission probabilities may be sparse for rare words.
- Requires large annotated corpora for accurate parameter estimation.

7. Applications

- POS Tagging
- Speech Recognition
- Named Entity Recognition (NER)
- Machine Translation (alignment modeling)
- Bioinformatics (gene sequence modeling)

- **Each cell in the Viterbi lattice = probability of best path ending in that tag.**
- **Transition probabilities = likelihood of one tag following another.**
- **Emission probabilities = likelihood of a word being generated by a tag.**
- **The Viterbi algorithm combines these to find the most probable sequence.**



HMM POS Tagging Example – Step by Step

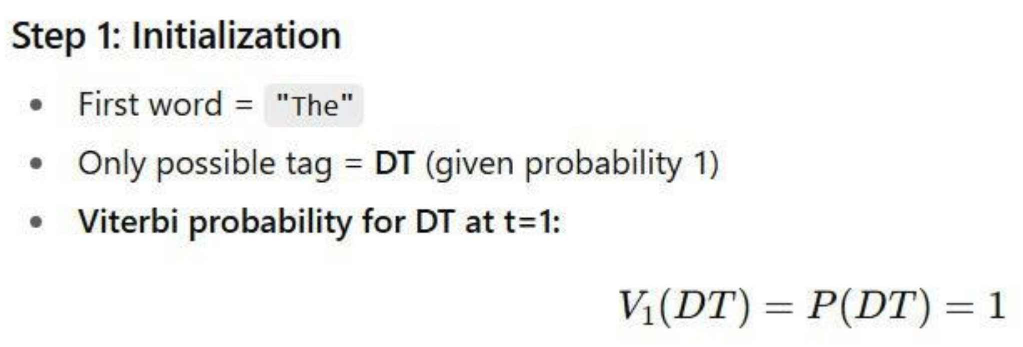
Sentence: `"The cat sleeps"`

Hidden states (POS tags): `DT, NN, VB`

Observations (words): `"The", "cat", "sleeps"`

Given Probabilities:

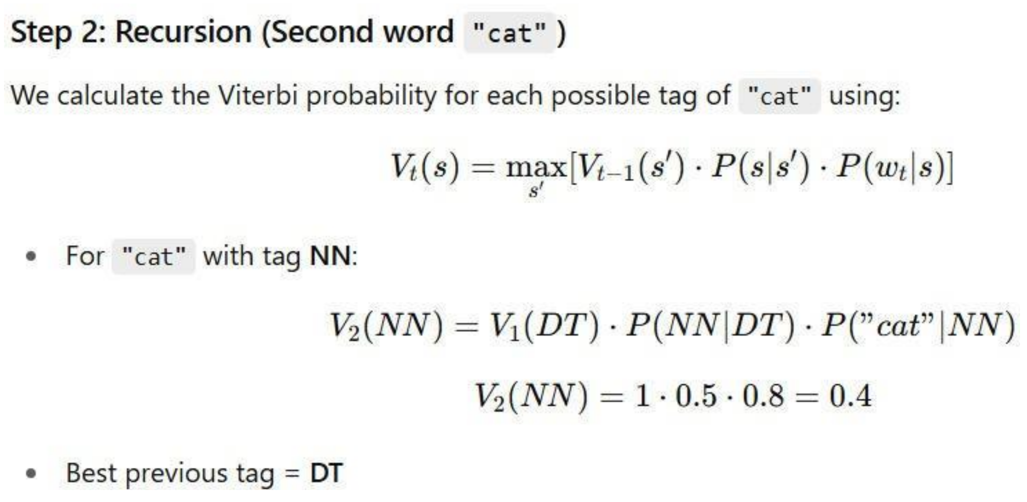
- **Transition probabilities (tag → tag):**
  - P(NN|DT) = 0.5
  - P(VB|NN) = 0.6
- **Emission probabilities (word|tag):**
  - P("cat"|NN) = 0.8
  - P("sleeps"|VB) = 0.7
- **Initial tag probability:** Assume P(DT at start) = 1



Step 1: Initialization

- First word = `"The"`
- Only possible tag = **DT** (given probability 1)
- **Viterbi probability for DT at t=1:**

$$V_1(DT) = P(DT) = 1$$



Step 2: Recursion (Second word `"cat"`)

We calculate the Viterbi probability for each possible tag of `"cat"` using:

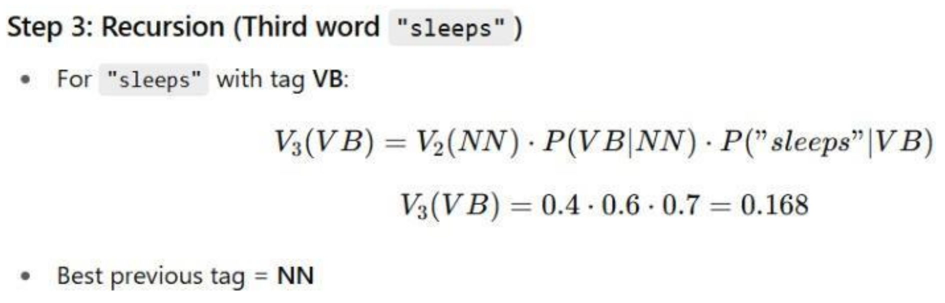
$$V_t(s) = \max_{s'}[V_{t-1}(s') \cdot P(s|s') \cdot P(w_t|s)]$$

- For `"cat"` with tag **NN**:

$$V_2(NN) = V_1(DT) \cdot P(NN|DT) \cdot P(''cat''|NN)$$

$$V_2(NN) = 1 \cdot 0.5 \cdot 0.8 = 0.4$$

- Best previous tag = **DT**



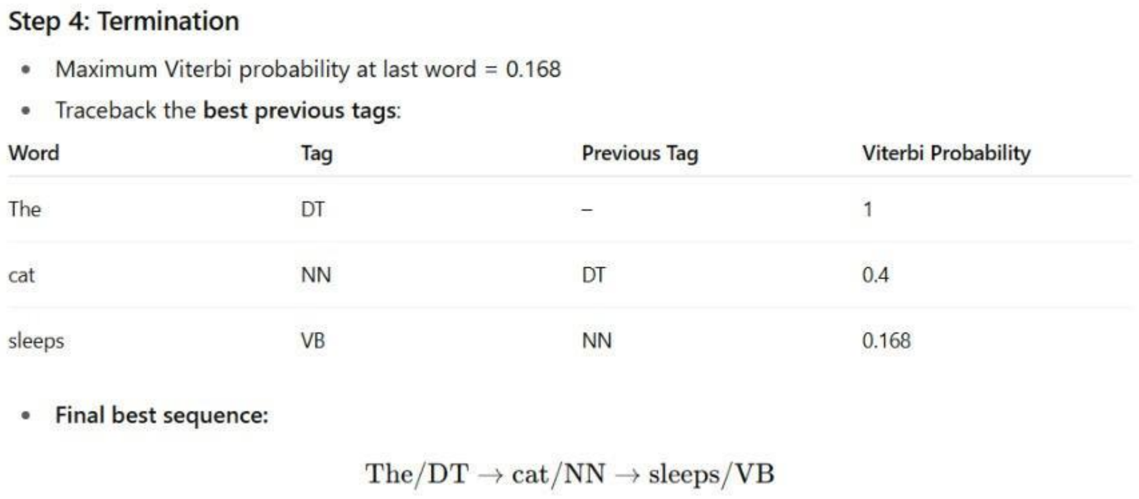
Step 3: Recursion (Third word `"sleeps"`)

- For `"sleeps"` with tag **VB**:

$$V_3(VB) = V_2(NN) \cdot P(VB|NN) \cdot P(''sleeps''|VB)$$

$$V_3(VB) = 0.4 \cdot 0.6 \cdot 0.7 = 0.168$$

- Best previous tag = **NN**



Step 4: Termination

- Maximum Viterbi probability at last word = 0.168
- Traceback the **best previous tags**:

Word	Tag	Previous Tag	Viterbi Probability
The	DT	–	1
cat	NN	DT	0.4
sleeps	VB	NN	0.168

- **Final best sequence:**

$$\text{The/DT} \rightarrow \text{cat/NN} \rightarrow \text{sleeps/VB}$$

Step 5: Interpretation

1. Start with initial probability of first tag.
2. Multiply previous Viterbi probability × transition probability × emission probability at each step.
3. Choose the maximum probability path at each step.
4. Trace back to get the most likely sequence of POS tags.

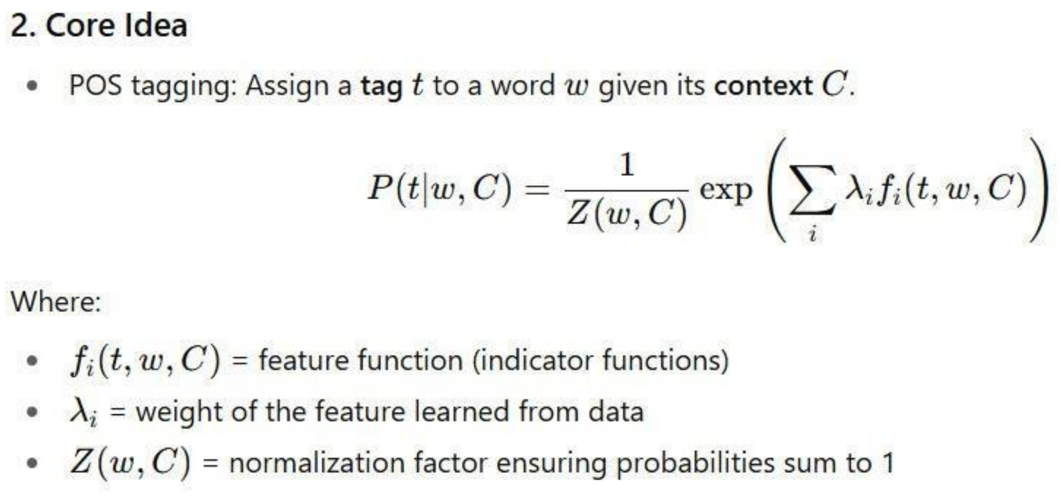
MAXIMUM ENTROPY (MAXENT) MODELS FOR POS TAGGING

- MaxEnt models compute **probabilities of tags using rich features** from words and context.
- Tag with **maximum probability** is assigned.
- Advantages over HMM: **flexible, feature-rich, no strong independence assumptions**.

1. Introduction

- Maximum Entropy Models are probabilistic models used in NLP for sequence labeling tasks like POS tagging.
- Based on the principle of maximum entropy: among all probability distributions satisfying given constraints, choose the one with highest entropy (most uniform / least biased).

- Advantages: Can incorporate diverse features, not limited to sequential dependencies like HMMs.



2. Core Idea

- POS tagging: Assign a **tag** t to a word w given its **context** C.

$$P(t|w,C) = \frac{1}{Z(w,C)} \exp\left(\sum_i \lambda_i f_i(t,w,C)\right)$$

Where:

- $f_i(t,w,C)$ = feature function (indicator functions)
- λ_i = weight of the feature learned from data
- $Z(w,C)$ = normalization factor ensuring probabilities sum to 1

3. Features Used in POS Tagging

MaxEnt models allow **rich features**, such as:

1. **Lexical Features**
   - Current word, suffixes, prefixes, capitalization
   - Example: If word ends in -ing, likely VB
2. **Contextual Features**
   - Previous and next words or tags
   - Example: If previous word = to, current word → VB
3. **Orthographic Features**
   - Numbers, hyphens, punctuation
   - Example: If word contains digits → NN (numeric)
4. **Combined Features**
   - Previous tag + current word, word shape, etc.

4. How MaxEnt POS Tagging Works

1. **Training Phase**

- Input: Annotated corpus (words + correct tags)
- Learn weights (λi\lambda_iλi) for each feature to maximize likelihood

2. **Tagging Phase**
    - For each word:
        - Extract features from word and context
        - Compute probabilities of all possible tags
        - Assign tag with **highest probability**

5. Example

Sentence: "The cat sleeps"

Features for "sleeps":

- Current word = "sleeps"
- Previous word = "cat"
- Previous tag = "NN"
- Word suffix = "ps"

Compute probability for each candidate tag:

- P(VB | features) = 0.75
- P(NN | features) = 0.10
- P(JJ | features) = 0.05

→ Assign **VB** as tag for "sleeps" because it has **highest probability**.

6. Advantages

- Can incorporate **arbitrary, overlapping features**.
- Does **not require independence assumptions** like HMMs.
- Often achieves **higher accuracy** in POS tagging than HMMs.

7. Disadvantages

- Computationally **more expensive** than HMMs for large feature sets.

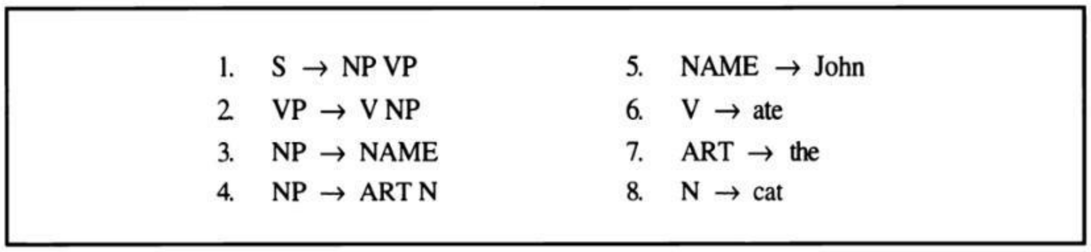
- Requires **good feature engineering** (though deep learning reduces this need).
- Needs a **large annotated corpus** for robust performance.

8. Applications

- **POS Tagging** (main application)
- **Named Entity Recognition (NER)**
- **Chunking / Shallow Parsing**
- **Information Extraction**

<u>CONTEXT-FREE GRAMMAR (CFG)</u>

- CFG is a **formal grammar** with rules defining how sentences are structured.
- Consists of **non-terminals, terminals, production rules, and a start symbol**.
- Widely used in **parsing and syntactic analysis** in NLP.



1.	$S \rightarrow NP\ VP$	5.	$NAME \rightarrow John$
2.	$VP \rightarrow V\ NP$	6.	$V \rightarrow ate$
3.	$NP \rightarrow NAME$	7.	$ART \rightarrow the$
4.	$NP \rightarrow ART\ N$	8.	$N \rightarrow cat$

Grammar 3.2 A simple grammar

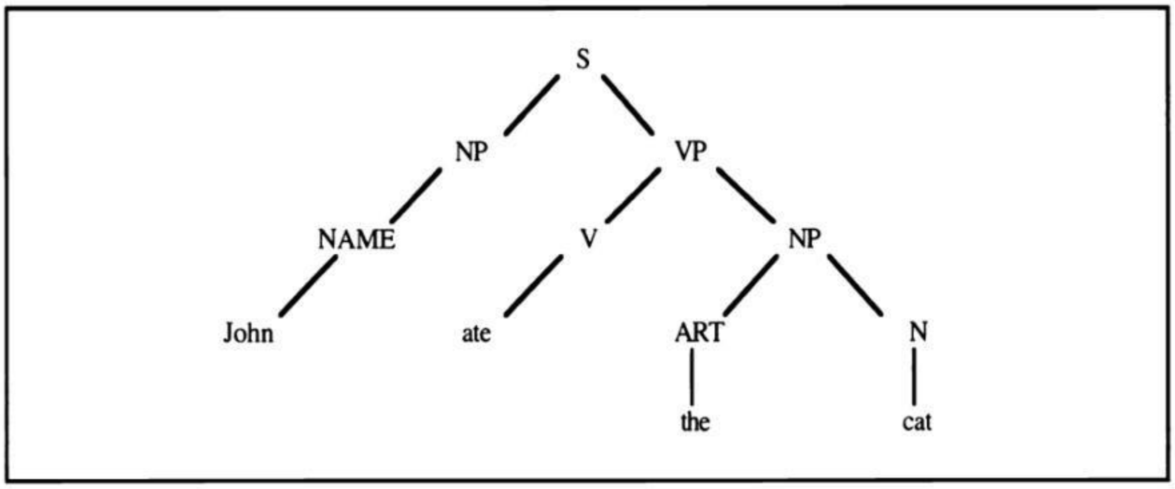
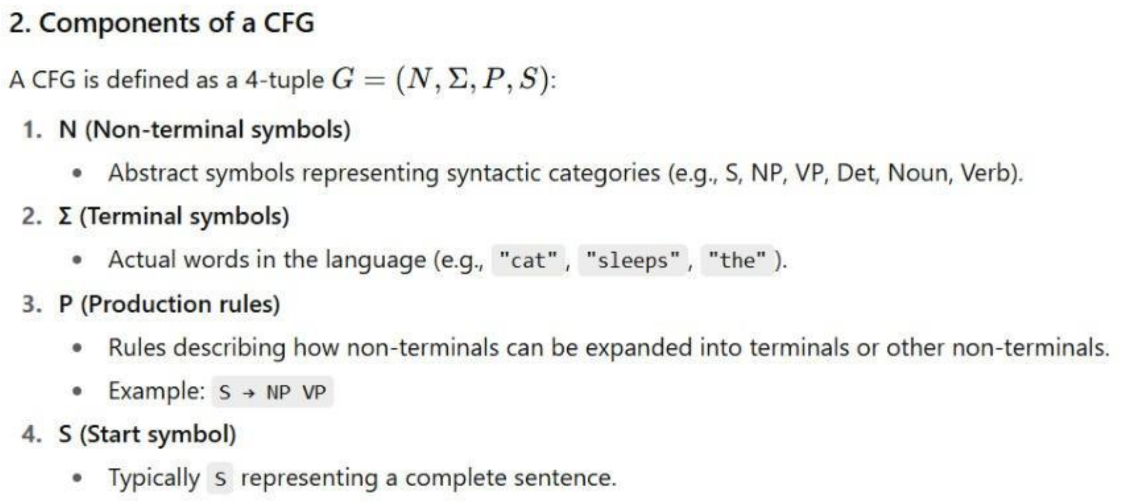


Figure 3.1 A tree representation of *John ate the cat*

1. Introduction

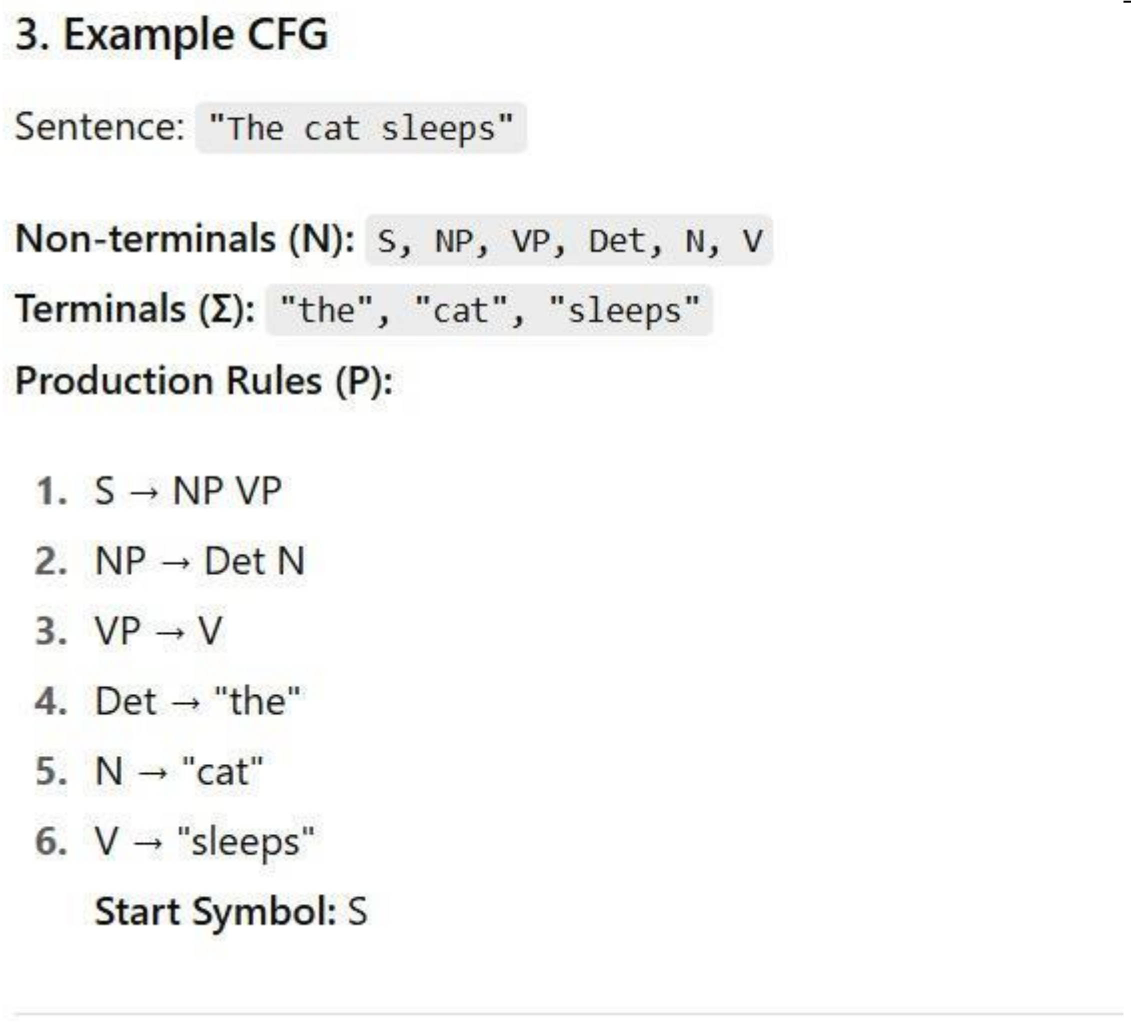
- **CFG** is a formal grammar used to describe **syntactic structures of languages**.
- Widely used in **parsing sentences**, **syntactic analysis**, and **compiler design**.
- A CFG consists of **rules that describe how sentences can be generated from a set of symbols**.



2. Components of a CFG

A CFG is defined as a 4-tuple $G = (N, \Sigma, P, S)$:

1. **N (Non-terminal symbols)**
   - Abstract symbols representing syntactic categories (e.g., S, NP, VP, Det, Noun, Verb).
2. **Σ (Terminal symbols)**
   - Actual words in the language (e.g., `"cat"`, `"sleeps"`, `"the"`).
3. **P (Production rules)**
   - Rules describing how non-terminals can be expanded into terminals or other non-terminals.
   - Example: `S → NP VP`
4. **S (Start symbol)**
   - Typically `S` representing a complete sentence.



3. Example CFG

Sentence: `"The cat sleeps"`

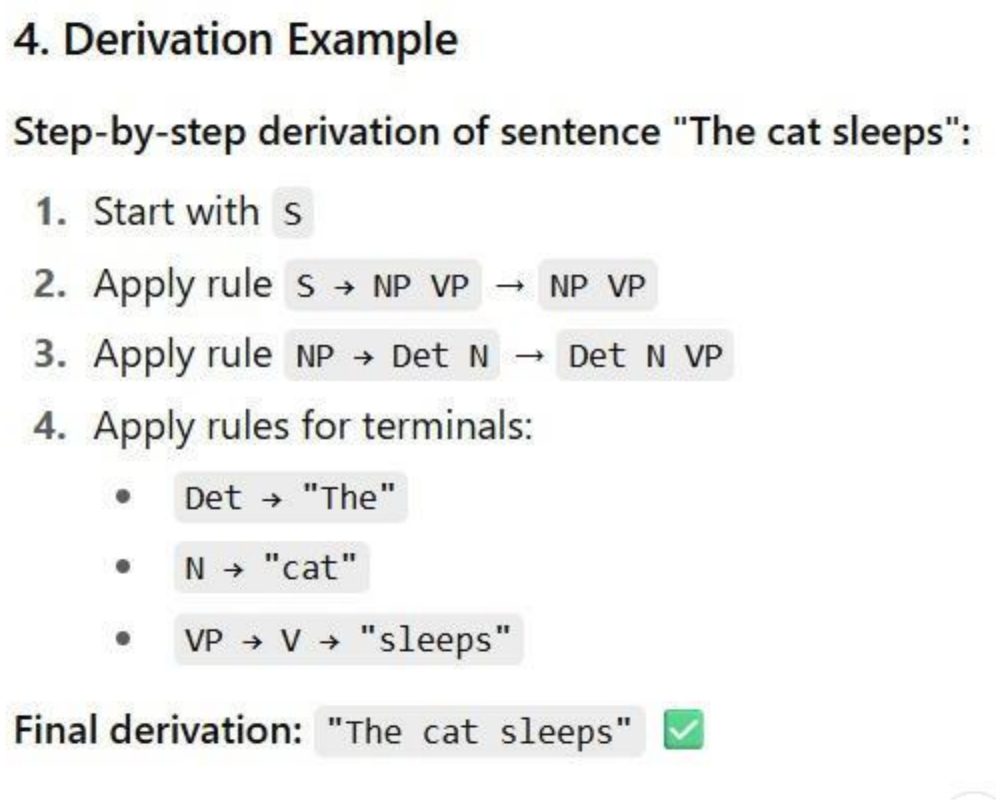
Non-terminals (N): `S, NP, VP, Det, N, V`

Terminals (Σ): `"the", "cat", "sleeps"`

Production Rules (P):

1. S → NP VP
2. NP → Det N
3. VP → V
4. Det → "the"
5. N → "cat"
6. V → "sleeps"

   Start Symbol: S

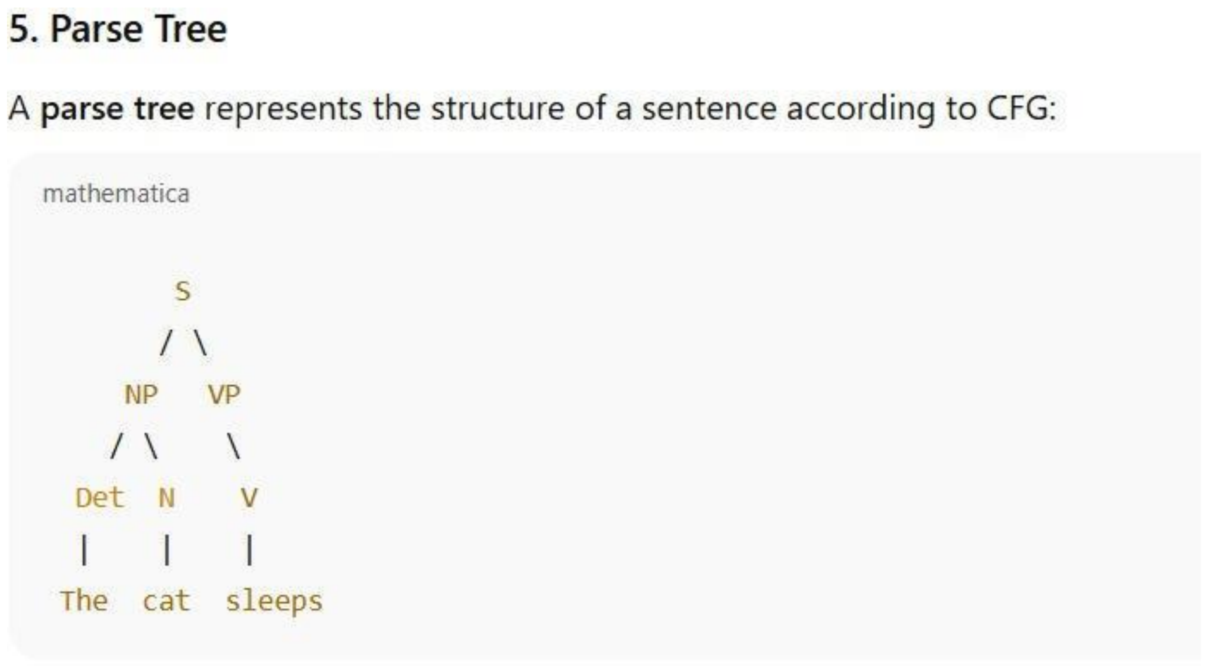


4. Derivation Example

Step-by-step derivation of sentence "The cat sleeps":

1. Start with `S`
2. Apply rule `S → NP VP` → `NP VP`
3. Apply rule `NP → Det N` → `Det N VP`
4. Apply rules for terminals:
   - `Det → "The"`
   - `N → "cat"`
   - `VP → V → "sleeps"`

Final derivation: `"The cat sleeps"` ✅



5. Parse Tree

A **parse tree** represents the structure of a sentence according to CFG:

```mathematica
        S
       / \
     NP   VP
    / \     \
  Det  N     V
  |    |     |
 The  cat  sleeps
```

6. Advantages of CFG

- Can model **hierarchical structure** of sentences.
- Supports **automated parsing**.
- Useful in **NLP tasks** like POS tagging, syntax checking, and machine translation.

7. Limitations of CFG

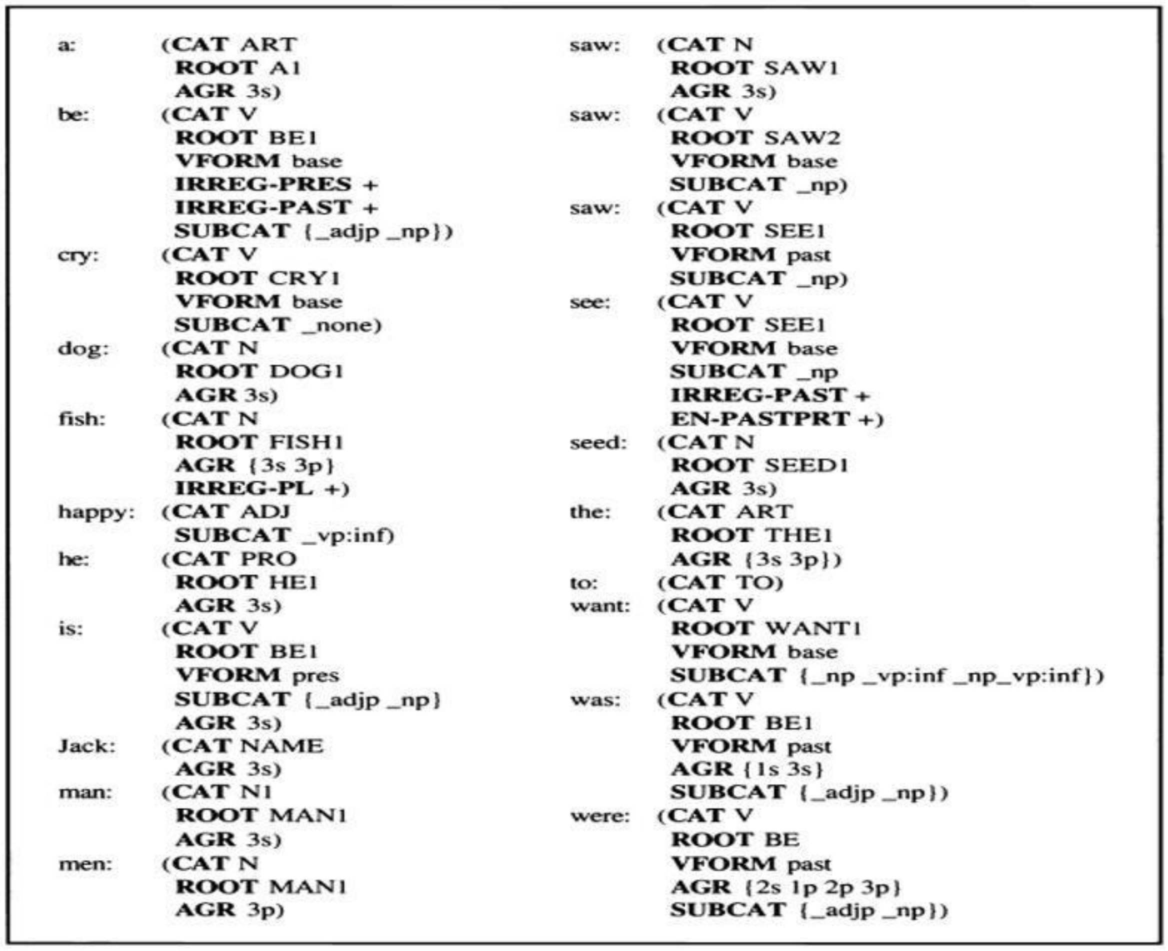
- Cannot easily handle **long-distance dependencies** (e.g., subject-verb agreement across clauses).
- May not capture **all linguistic nuances**, especially in free-word-order languages.
- Ambiguity can lead to **multiple parse trees** for the same sentence.

8. Applications in NLP

- **Syntactic parsing**
- **Grammar checking**
- **Machine Translation**
- **Question Answering**
- **Information Extraction**

CONSTITUENCY PARSING

- Constituency Parsing identifies **phrases and their hierarchical relationships** in a sentence.
- Uses **CFG or probabilistic methods** to generate a **parse tree**.
- Provides **structured syntactic information** for various NLP tasks.

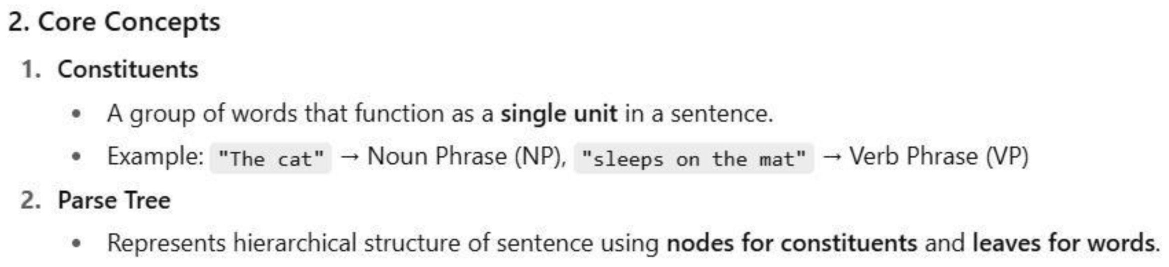


a:	(CAT ART ROOT A1 AGR 3s)
be:	(CAT V ROOT BE1 VFORM base IRREG-PRES + IRREG-PAST + SUBCAT {_adjp _np})
cry:	(CAT V ROOT CRY1 VFORM base SUBCAT _none)
dog:	(CAT N ROOT DOG1 AGR 3s)
fish:	(CAT N ROOT FISH1 AGR {3s 3p} IRREG-PL +)
happy:	(CAT ADJ SUBCAT _vp:inf)
he:	(CAT PRO ROOT HE1 AGR 3s)
is:	(CAT V ROOT BE1 VFORM pres SUBCAT {_adjp _np} AGR 3s)
Jack:	(CAT NAME AGR 3s)
man:	(CAT N1 ROOT MAN1 AGR 3s)
men:	(CAT N ROOT MAN1 AGR 3p)
saw:	(CAT N ROOT SAW1 AGR 3s)
saw:	(CAT V ROOT SAW2 VFORM base SUBCAT _np)
saw:	(CAT V ROOT SEE1 VFORM past SUBCAT _np)
see:	(CAT V ROOT SEE1 VFORM base SUBCAT _np IRREG-PAST + EN-PASTPRT +)
seed:	(CAT N ROOT SEED1 AGR 3s)
the:	(CAT ART ROOT THE1 AGR {3s 3p})
to:	(CAT TO)
want:	(CAT V ROOT WANT1 VFORM base SUBCAT {_np _vp:inf _np_vp:inf})
was:	(CAT V ROOT BE1 VFORM past AGR {1s 3s} SUBCAT {_adjp _np})
were:	(CAT V ROOT BE VFORM past AGR {2s 1p 2p 3p} SUBCAT {_adjp _np})

Figure 4.6 A lexicon

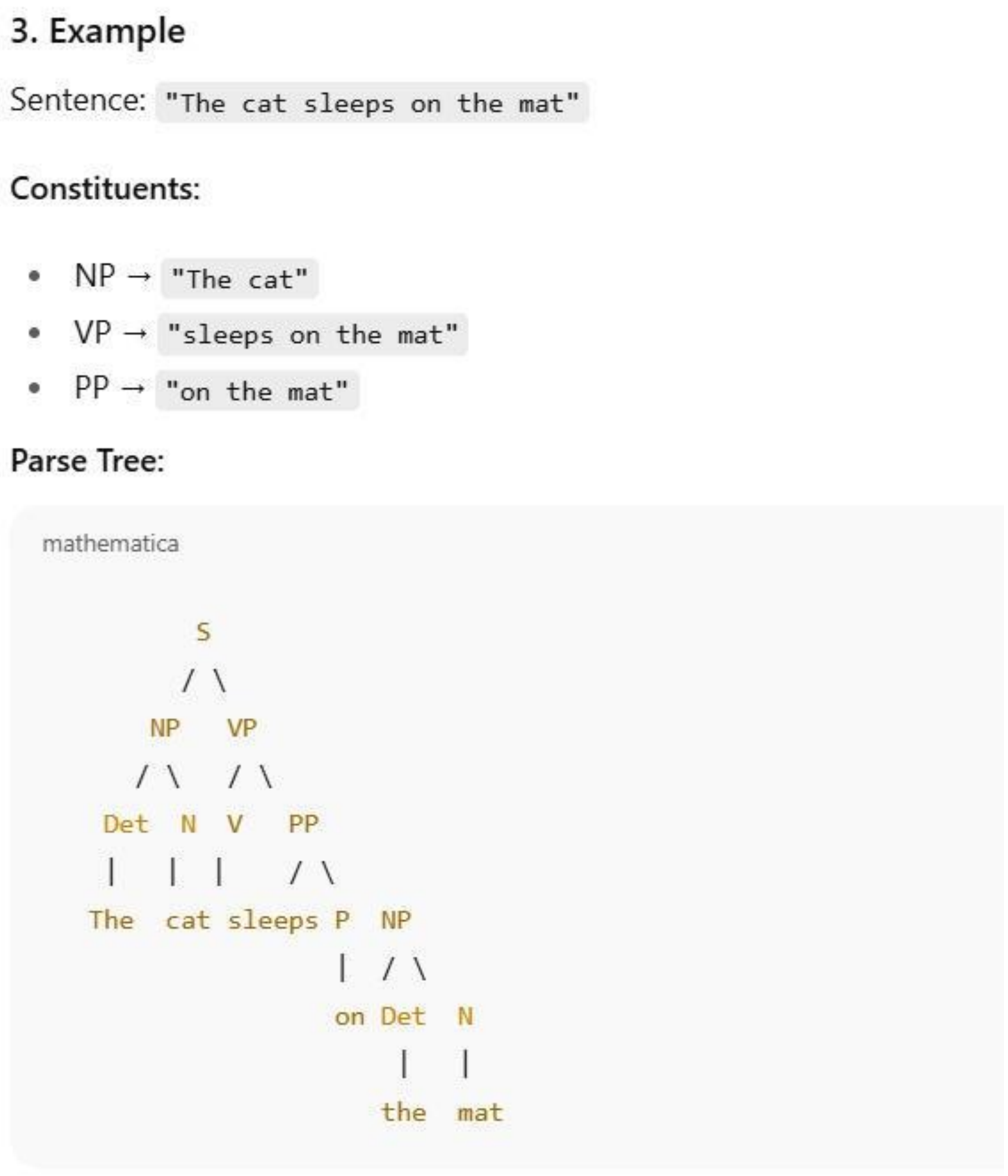
1. Introduction

- **Constituency Parsing** (or **Phrase Structure Parsing**) analyzes a sentence to identify its **constituent parts**, such as **noun phrases (NP), verb phrases (VP), and prepositional phrases (PP)**.
- Based on **Context-Free Grammar (CFG)**.
- Helps machines understand **hierarchical structure** of sentences.



2. Core Concepts

1. **Constituents**
   - A group of words that function as a **single unit** in a sentence.
   - Example: `"The cat"` → Noun Phrase (NP), `"sleeps on the mat"` → Verb Phrase (VP)
2. **Parse Tree**
   - Represents hierarchical structure of sentence using **nodes for constituents** and **leaves for words**.



3. Example

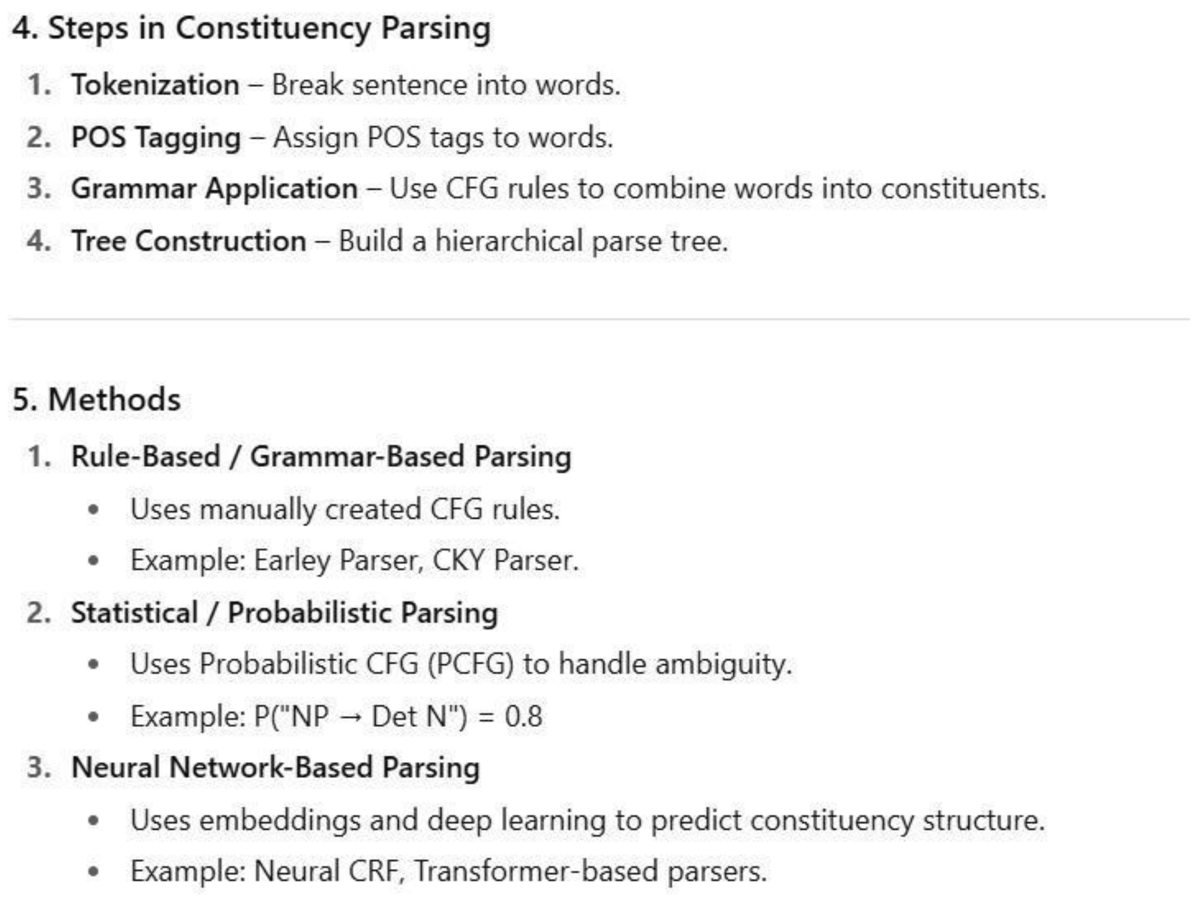
Sentence: `"The cat sleeps on the mat"`

Constituents:

- NP → `"The cat"`
- VP → `"sleeps on the mat"`
- PP → `"on the mat"`

Parse Tree:

```mathematica
       S
      / \
    NP   VP
   / \   / \
 Det  N  V   PP
 |   |  |    / \
The  cat sleeps P  NP
                |  / \
                on Det  N
                     |   |
                   the  mat
```



4. Steps in Constituency Parsing

1. **Tokenization** – Break sentence into words.
2. **POS Tagging** – Assign POS tags to words.
3. **Grammar Application** – Use CFG rules to combine words into constituents.
4. **Tree Construction** – Build a hierarchical parse tree.

5. Methods

1. **Rule-Based / Grammar-Based Parsing**
    - Uses manually created CFG rules.
    - Example: Earley Parser, CKY Parser.
2. **Statistical / Probabilistic Parsing**
    - Uses Probabilistic CFG (PCFG) to handle ambiguity.
    - Example: P("NP → Det N") = 0.8
3. **Neural Network-Based Parsing**
    - Uses embeddings and deep learning to predict constituency structure.
    - Example: Neural CRF, Transformer-based parsers.

6. Advantages

- Captures **hierarchical and phrasal structure**.
- Useful for **machine translation, summarization, and question answering**.
- Provides **interpretable syntactic analysis**.

7. Limitations

- **Ambiguity:** Sentences may have multiple valid parse trees.
- **Complexity:** Parsing long sentences can be computationally expensive.
- Less effective for **free-word-order languages** unless probabilistic or neural methods are used.

8. Applications

- **Syntactic analysis** for NLP pipelines.

- **Machine Translation:** Helps maintain sentence structure.
- **Question Answering & Information Extraction:** Identify subject, object, and phrases.
- **Grammar Checking & Correction**

TREEBANKS AND NORMAL FORMS FOR GRAMMAR

- **Treebanks:** Annotated corpora with POS tags and parse trees for training and evaluation.
- **Normal forms:** Standardized representations of grammar rules (CNF, GNF) to simplify parsing.
- Both are **essential for building and evaluating NLP parsing systems**.

1. Treebanks

Definition:

- A **treebank** is a **corpus of sentences annotated with syntactic or semantic parse trees**.
- Provides **gold-standard examples** for training and evaluating parsers.

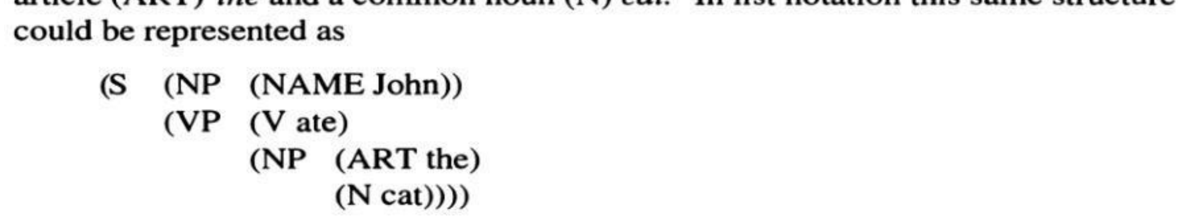
Purpose in NLP:

- Helps **train statistical parsers**.
- Used in **POS tagging, syntactic parsing, and grammar evaluation**.
- Enables **comparative evaluation** of parsing algorithms.

Examples of Treebanks:

1. **Penn Treebank (PTB)** – English, widely used in NLP research.
2. **Universal Dependencies (UD) Treebanks** – multilingual, dependency-based annotation.
3. **NEGRA Treebank** – German, constituency-based.

Structure:



article (ART) *the* and a common noun (N) *cat*. In list notation this same structure could be represented as

```
(S   (NP   (NAME John))
     (VP   (V ate)
           (NP   (ART the)
                 (N cat))))
```

- Each sentence is annotated with:
  - POS tags for each word
  - Phrase structure or dependency structure
- Stored as **bracketed trees** or **dependency graphs**

Example (Bracketed):
Sentence: "The cat sleeps"

```
(S
  (NP (DT The) (NN cat))
  (VP (VB sleeps))
)
```

2. Normal Forms for Grammar

Definition:

- **Normal forms** are standardized ways to represent grammar rules.
- Simplify parsing and grammar analysis.

Common Normal Forms:

1. **Chomsky Normal Form (CNF)**
   - Each production rule is either:
     1. A → BC (two non-terminals)
     2. A → a (a single terminal)
   - Example:

- Original: S → NP VP
- CNF: S → X VP, X → NP

2. **Greibach Normal Form (GNF)**
    - Each production starts with a **terminal** followed by **zero or more non-terminals**.
    - Example: S → aA B

Why Normal Forms Are Useful:

- **Simplify parsing algorithms** (like CYK parser uses CNF).
- Reduce **ambiguity and complexity** in automated parsing.
- Facilitate **formal proofs and computational efficiency**.

3. Applications in NLP

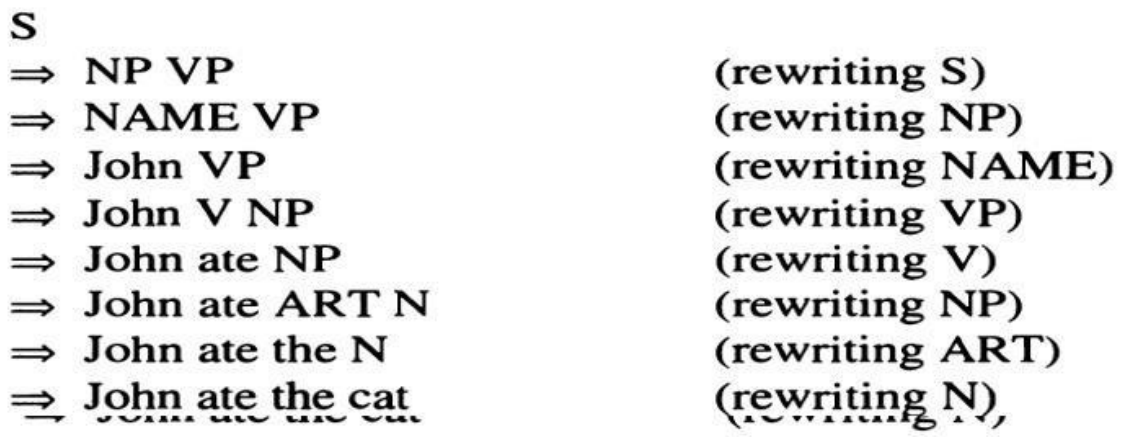
- **Treebanks**
    - Train and evaluate **statistical and neural parsers**.
    - Serve as **gold-standard datasets**.
- **Normal Forms**
    - Simplify **syntactic parsing algorithms**.
    - Used in **compiler design, CFG parsing, and NLP education**.

TOP-DOWN AND BOTTOM-UP PARSING STRATEGIES

Parsing strategies determine **how a parser constructs a parse tree** for a sentence based on a grammar. The two main strategies are **Top-Down** and **Bottom-Up Parsing**.

- **Top-Down Parsing:** Start from root, expand non-terminals, match input.
- **Bottom-Up Parsing:** Start from input tokens, combine into constituents, reach root.
- Both produce the **same parse tree** but follow opposite directions.

1. Top-Down Parsing



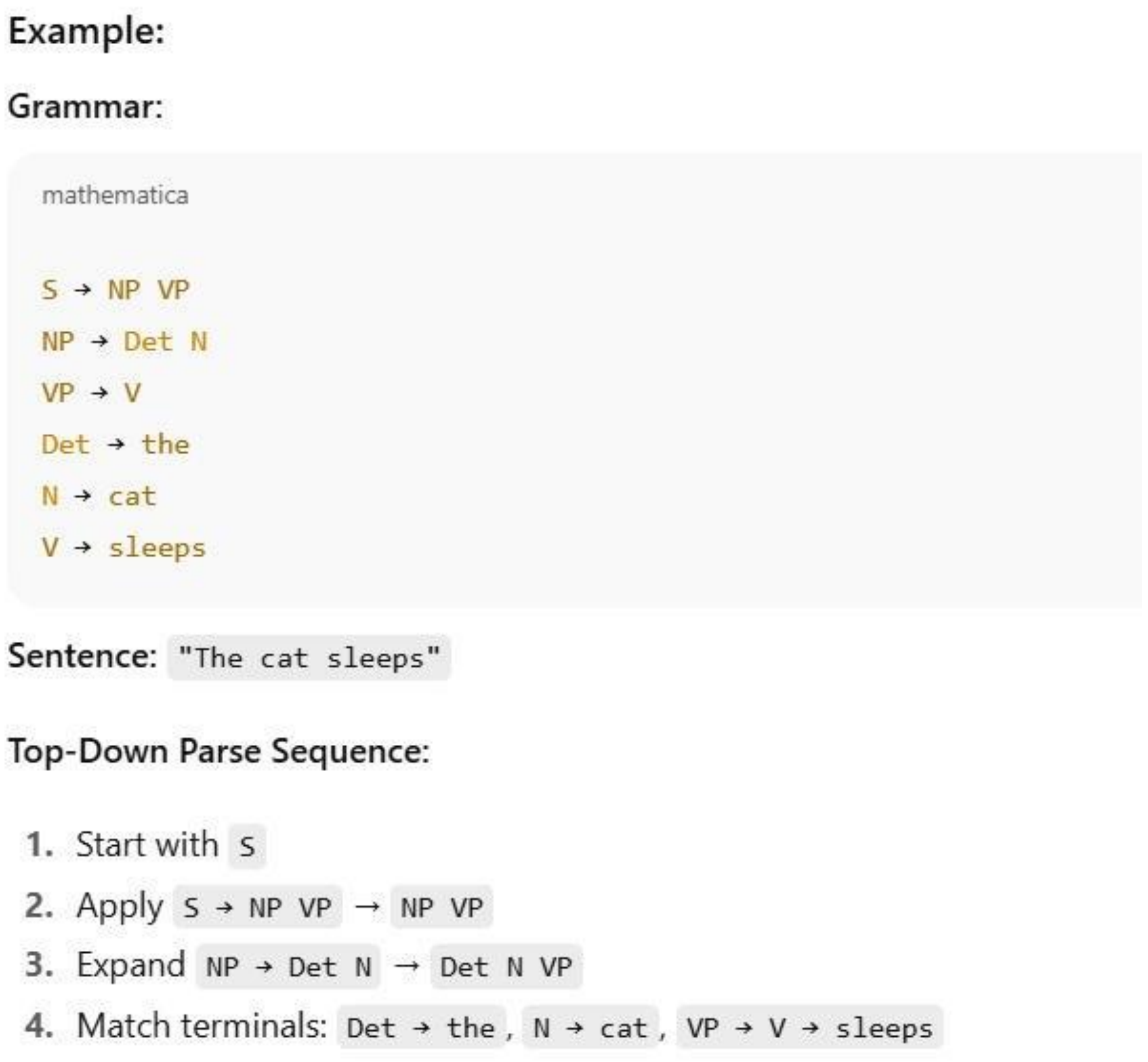
S	
⇒ NP VP	(rewriting S)
⇒ NAME VP	(rewriting NP)
⇒ John VP	(rewriting NAME)
⇒ John V NP	(rewriting VP)
⇒ John ate NP	(rewriting V)
⇒ John ate ART N	(rewriting NP)
⇒ John ate the N	(rewriting ART)
⇒ John ate the cat	(rewriting N)

Definition:

- **Top-Down Parsing** starts from the **start symbol (S)** and tries to derive the **input sentence** by recursively expanding non-terminals using grammar rules.
- Works **from root to leaves** of the parse tree.

Characteristics:

- Predictive approach: tries to match input tokens by applying rules.
- Can use **lookahead** to reduce backtracking (e.g., in **LL parsers**).
- Can suffer from **left recursion** and may require grammar modification.



Example:

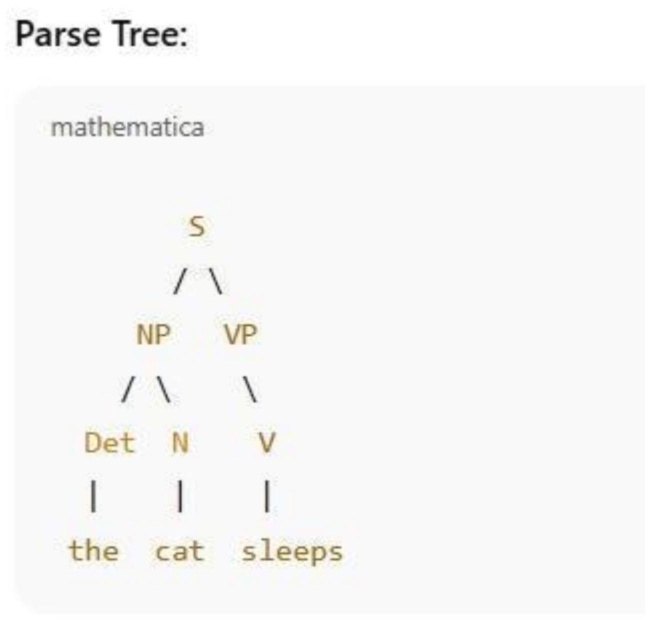
Grammar:

```mathematica
S → NP VP
NP → Det N
VP → V
Det → the
N → cat
V → sleeps
```

Sentence: `"The cat sleeps"`

Top-Down Parse Sequence:

1. Start with `S`
2. Apply `S → NP VP` → `NP VP`
3. Expand `NP → Det N` → `Det N VP`
4. Match terminals: `Det → the`, `N → cat`, `VP → V → sleeps`

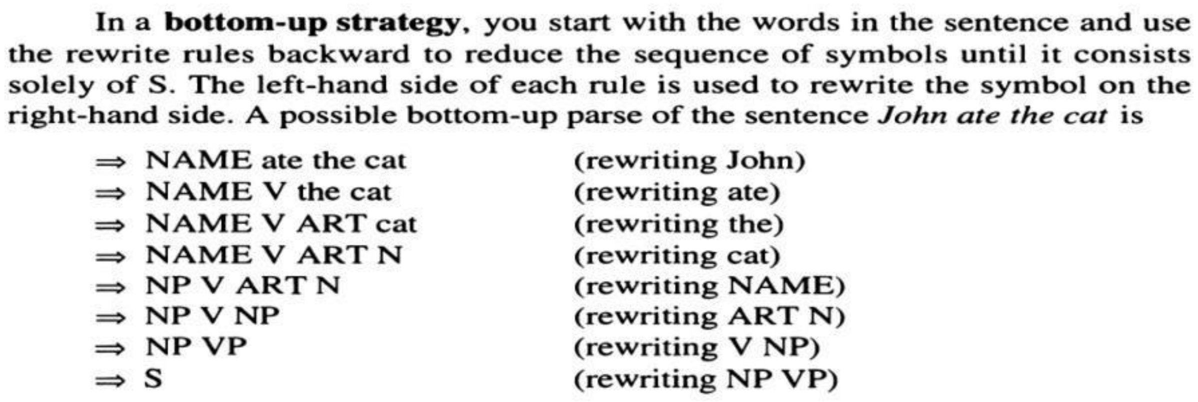


Parse Tree:

mathematica

```
        S
       / \
     NP   VP
    / \    \
  Det  N    V
   |   |    |
  the  cat  sleeps
```

3. **Bottom-Up Parsing**



In a **bottom-up strategy**, you start with the words in the sentence and use the rewrite rules backward to reduce the sequence of symbols until it consists solely of S. The left-hand side of each rule is used to rewrite the symbol on the right-hand side. A possible bottom-up parse of the sentence *John ate the cat* is

⇒ NAME ate the cat	(rewriting John)
⇒ NAME V the cat	(rewriting ate)
⇒ NAME V ART cat	(rewriting the)
⇒ NAME V ART N	(rewriting cat)
⇒ NP V ART N	(rewriting NAME)
⇒ NP V NP	(rewriting ART N)
⇒ NP VP	(rewriting V NP)
⇒ S	(rewriting NP VP)

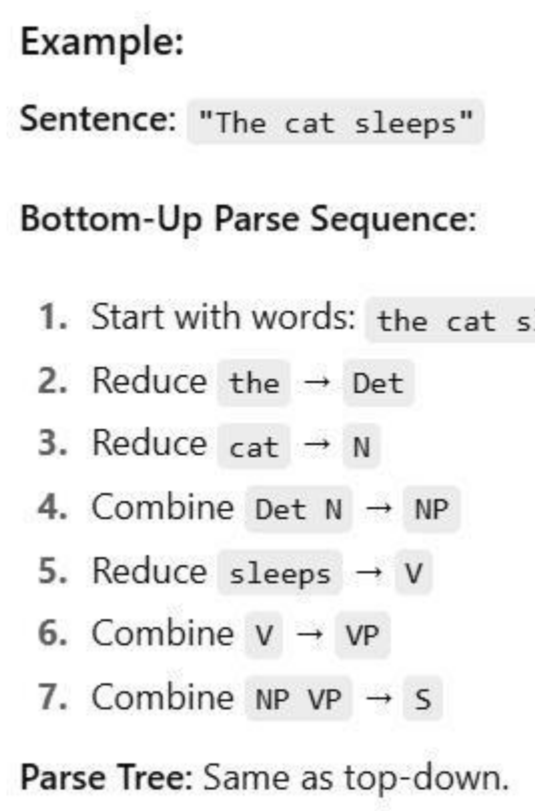
Definition:

- **Bottom-Up Parsing** starts from the **input sentence (leaves)** and tries to construct the **parse tree up to the start symbol**.
- Works **from leaves to root**.

Characteristics:

- Data-driven approach: combines words into phrases, then phrases into sentences.

- Avoids left-recursion problems.
- Can be implemented using **shift-reduce parsers**.



Example:

Sentence: `"The cat sleeps"`

Bottom-Up Parse Sequence:

1. Start with words: `the cat sleeps`
2. Reduce `the` → `Det`
3. Reduce `cat` → `N`
4. Combine `Det N` → `NP`
5. Reduce `sleeps` → `V`
6. Combine `V` → `VP`
7. Combine `NP VP` → `S`

Parse Tree: Same as top-down.

Feature	**Top-Down Parsing**	**Bottom-Up Parsing**
Approach	Root → Leaves (Start → Input)	Leaves → Root (Input → Start)
Predictive	Yes	No
Handles Left Recursion	Poor	Good
Backtracking Needed	Often	Sometimes
Example Implementation	Recursive Descent, LL Parser	Shift-Reduce, LR Parser

4. Applications

- **Top-Down Parsing:** Suitable for **predictive parsers** and simple CFGs.

- **Bottom-Up Parsing:** Used in **LR parsers**, **shift-reduce parsers**, and large-scale NLP parsing systems.
- Both strategies are **foundational for syntactic analysis, machine translation, and grammar checking**.

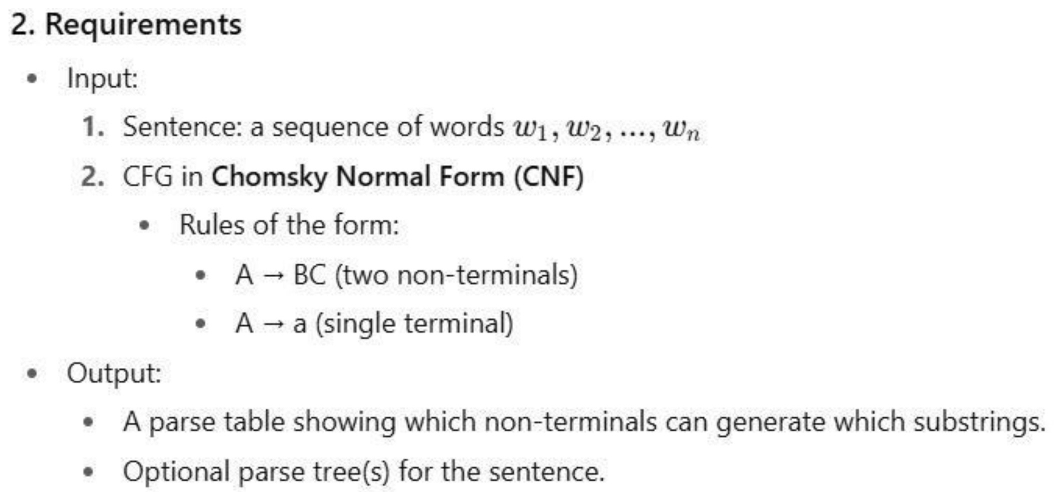
CYK PARSING ALGORITHM

CYK (Cocke–Younger–Kasami) Parsing Algorithm

- **CYK Algorithm** is a **bottom-up parser** using **CNF CFGs**.
- Fills a **triangular table** with non-terminals that generate substrings.
- Efficiently checks **sentence validity** and can construct parse trees.

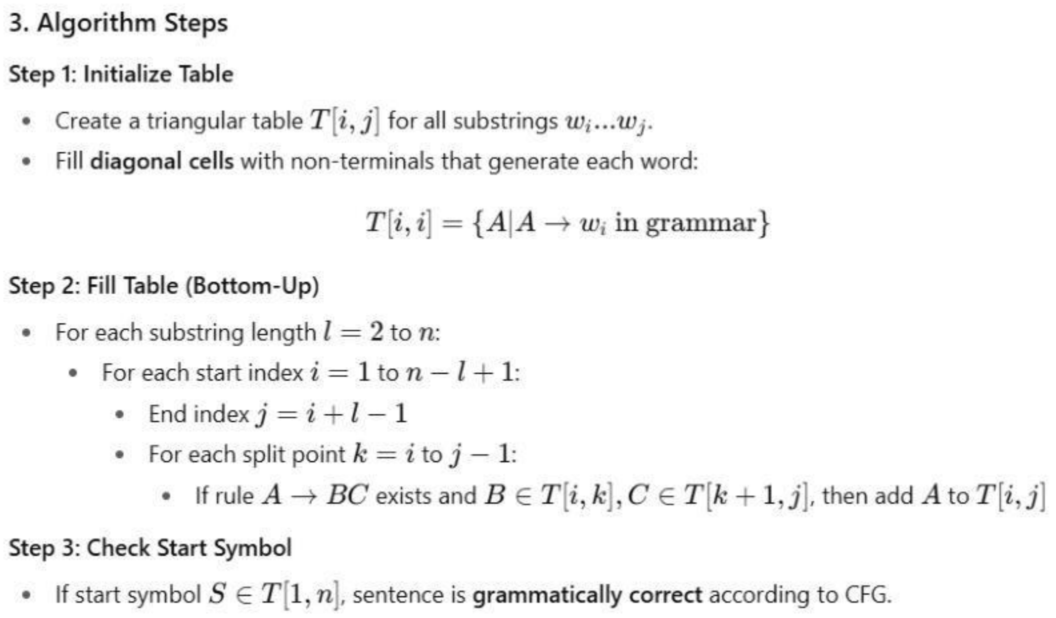
1. Introduction

- **CYK Algorithm** is a **bottom-up parsing algorithm** for **Context-Free Grammars (CFG)**.
- Works only with grammars in **Chomsky Normal Form (CNF)**.
- Efficiently determines **if a sentence can be generated by a CFG** and produces a **parse tree**.
- Widely used in **NLP for syntactic parsing**.



2. Requirements

- Input:
  1. Sentence: a sequence of words $w_1, w_2, ..., w_n$
  2. CFG in **Chomsky Normal Form (CNF)**
     - Rules of the form:
       - A → BC (two non-terminals)
       - A → a (single terminal)
- Output:
  - A parse table showing which non-terminals can generate which substrings.
  - Optional parse tree(s) for the sentence.



3. Algorithm Steps

Step 1: Initialize Table

- Create a triangular table $T[i, j]$ for all substrings $w_i \dots w_j$.
- Fill **diagonal cells** with non-terminals that generate each word:

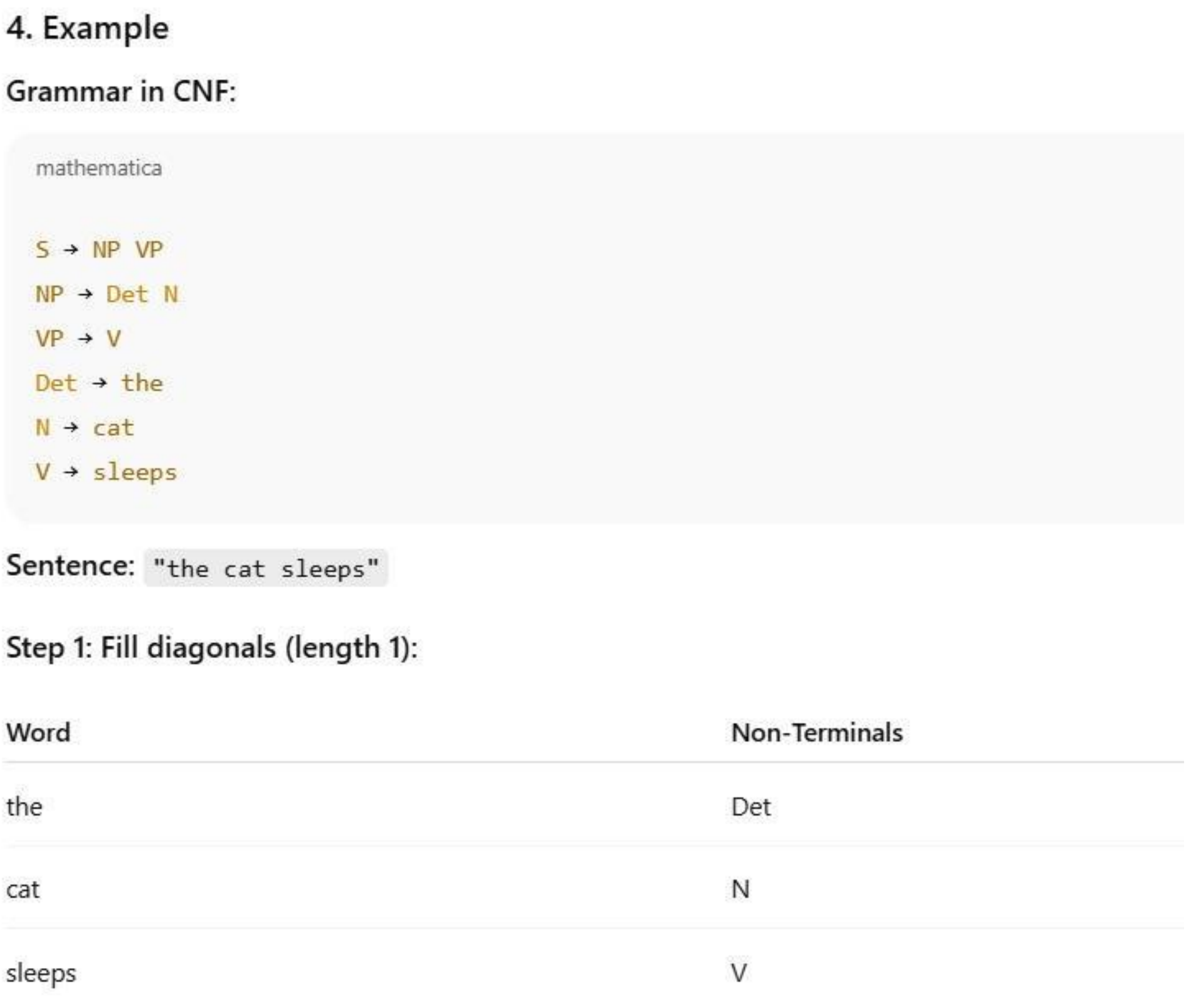
$$T[i, i] = \{A | A \rightarrow w_i \text{ in grammar}\}$$

Step 2: Fill Table (Bottom-Up)

- For each substring length $l = 2$ to n:
  - For each start index $i = 1$ to $n - l + 1$:
    - End index $j = i + l - 1$
    - For each split point $k = i$ to $j - 1$:
      - If rule $A \rightarrow BC$ exists and $B \in T[i, k], C \in T[k + 1, j]$, then add A to $T[i, j]$

Step 3: Check Start Symbol

- If start symbol $S \in T[1, n]$, sentence is **grammatically correct** according to CFG.



4. Example

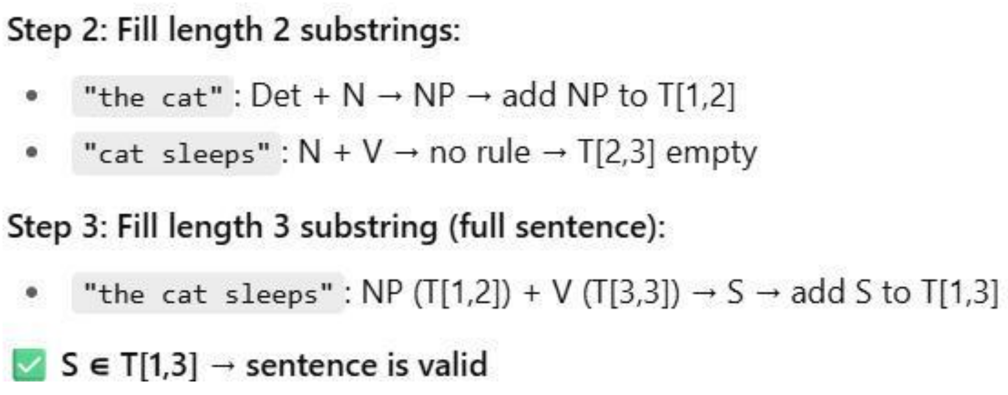
Grammar in CNF:

```mathematica
S → NP VP
NP → Det N
VP → V
Det → the
N → cat
V → sleeps
```

Sentence: `"the cat sleeps"`

Step 1: Fill diagonals (length 1):

Word	Non-Terminals
the	Det
cat	N
sleeps	V



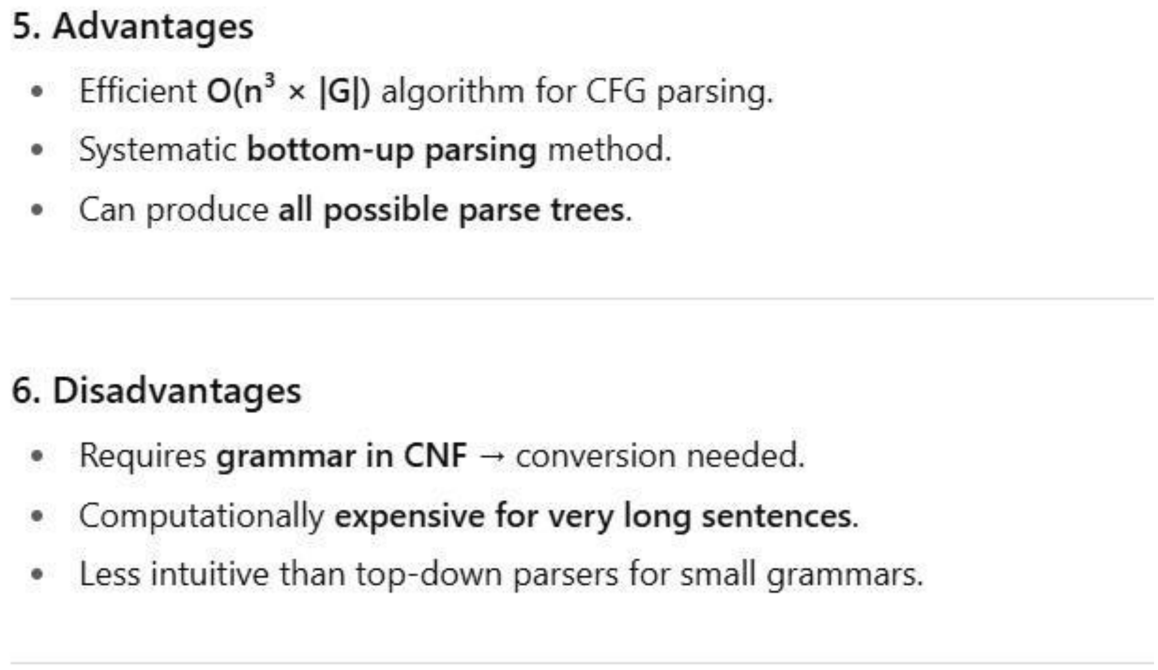
Step 2: Fill length 2 substrings:

- `"the cat"` : Det + N → NP → add NP to T[1,2]
- `"cat sleeps"` : N + V → no rule → T[2,3] empty

Step 3: Fill length 3 substring (full sentence):

- `"the cat sleeps"` : NP (T[1,2]) + V (T[3,3]) → S → add S to T[1,3]

✅ **S ∈ T[1,3] → sentence is valid**



5. Advantages

- Efficient **O(n^3 × |G|)** algorithm for CFG parsing.
- Systematic **bottom-up parsing** method.
- Can produce **all possible parse trees**.

6. Disadvantages

- Requires **grammar in CNF** → conversion needed.
- Computationally **expensive for very long sentences**.
- Less intuitive than top-down parsers for small grammars.

7. Applications

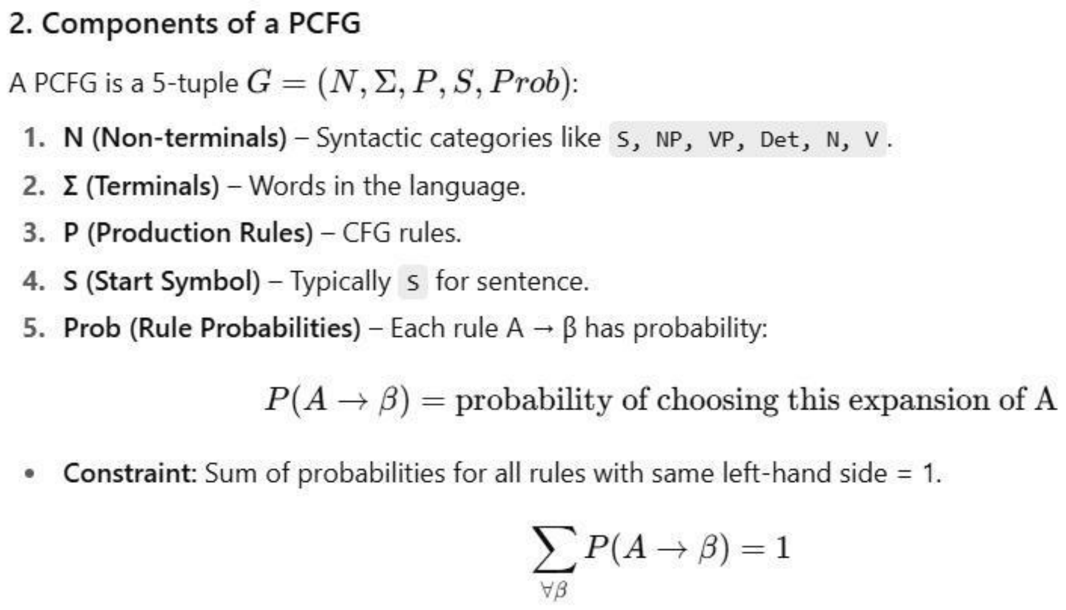
- **Syntactic parsing** in NLP
- **Grammar checking**
- **Machine translation**
- **Bioinformatics (sequence parsing)**

PROBABILISTIC CONTEXT-FREE GRAMMARS (PCFGS)

- **PCFGs** = CFG + probabilities.
- **Probabilities** allow choosing the **most likely parse** among multiple possibilities.
- Essential in **statistical NLP** for disambiguation and robust parsing.

1. Introduction

- **PCFGs** are an extension of **Context-Free Grammars (CFGs)** that assign **probabilities to production rules**.
- Used to handle **ambiguity in natural language**, e.g., multiple parse trees for a sentence.
- Widely used in **statistical parsing, machine translation, and speech recognition**.



2. Components of a PCFG

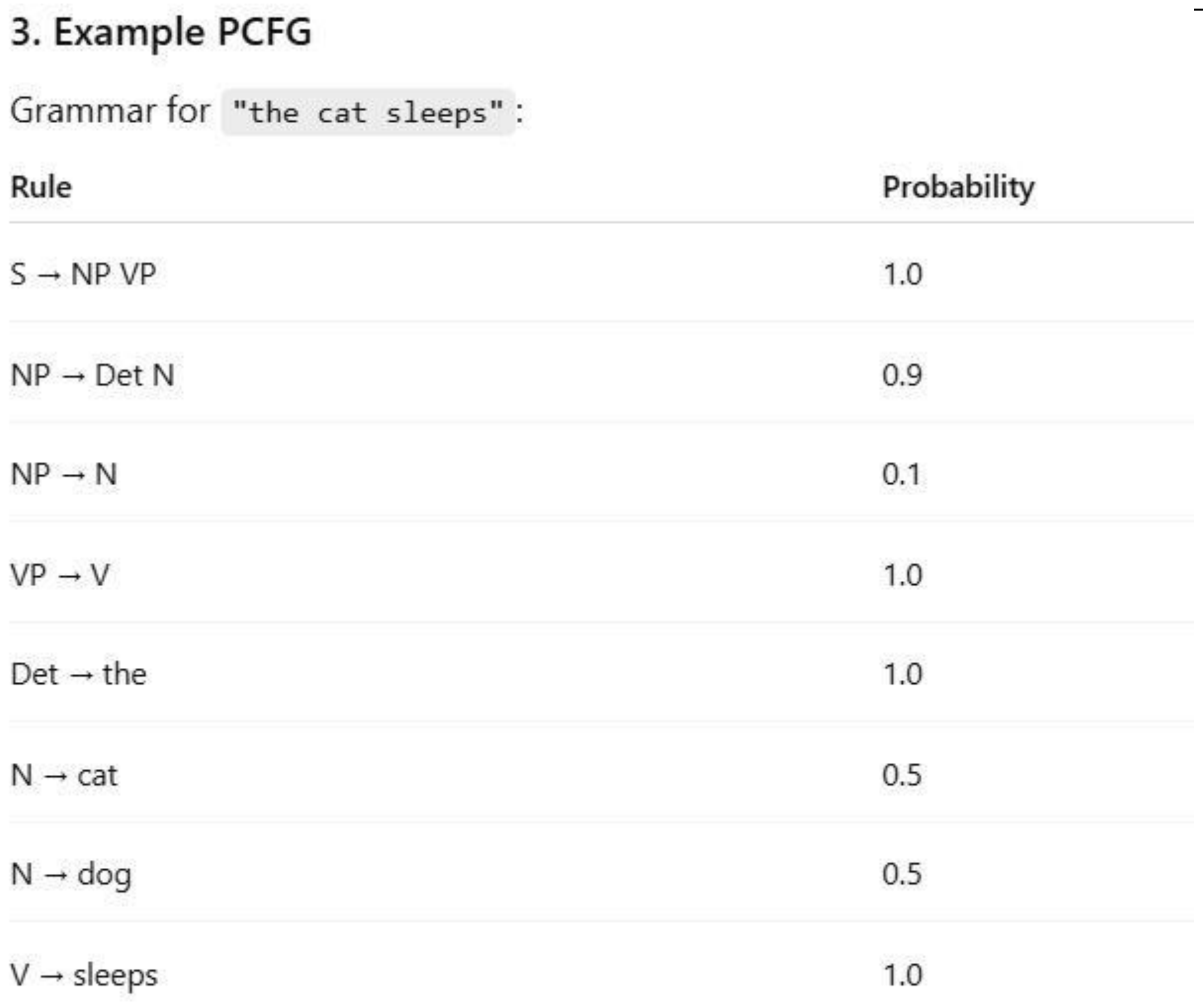
A PCFG is a 5-tuple $G = (N, \Sigma, P, S, Prob)$:

1. **N (Non-terminals)** – Syntactic categories like `S, NP, VP, Det, N, V`.
2. **Σ (Terminals)** – Words in the language.
3. **P (Production Rules)** – CFG rules.
4. **S (Start Symbol)** – Typically `S` for sentence.
5. **Prob (Rule Probabilities)** – Each rule A → β has probability:

$$P(A \to \beta) = \text{probability of choosing this expansion of A}$$

- **Constraint**: Sum of probabilities for all rules with same left-hand side = 1.

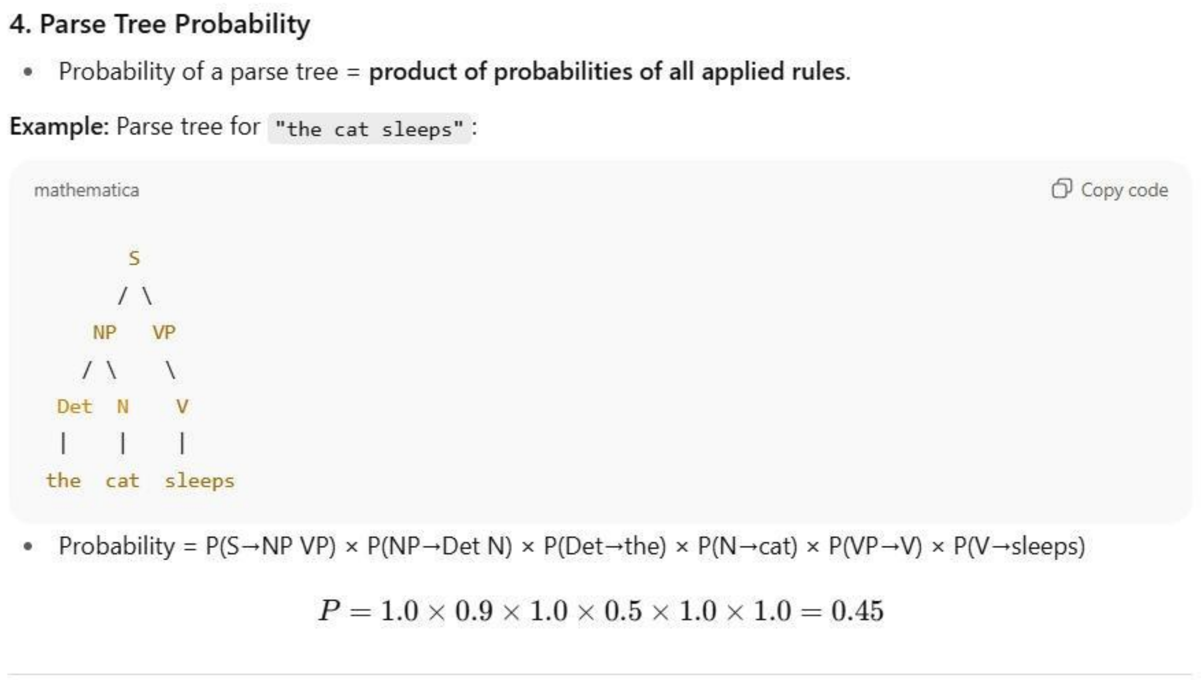
$$\sum_{\forall \beta} P(A \to \beta) = 1$$



3. Example PCFG

Grammar for `"the cat sleeps"`:

Rule	Probability
S → NP VP	1.0
NP → Det N	0.9
NP → N	0.1
VP → V	1.0
Det → the	1.0
N → cat	0.5
N → dog	0.5
V → sleeps	1.0



4. Parse Tree Probability

- Probability of a parse tree = **product of probabilities of all applied rules.**

Example: Parse tree for `"the cat sleeps"`:

```
        S
       / \
     NP   VP
    / \    \
  Det  N    V
   |   |    |
  the  cat  sleeps
```

- Probability = P(S→NP VP) × P(NP→Det N) × P(Det→the) × P(N→cat) × P(VP→V) × P(V→sleeps)

$$P = 1.0 \times 0.9 \times 1.0 \times 0.5 \times 1.0 \times 1.0 = 0.45$$

5. Advantages

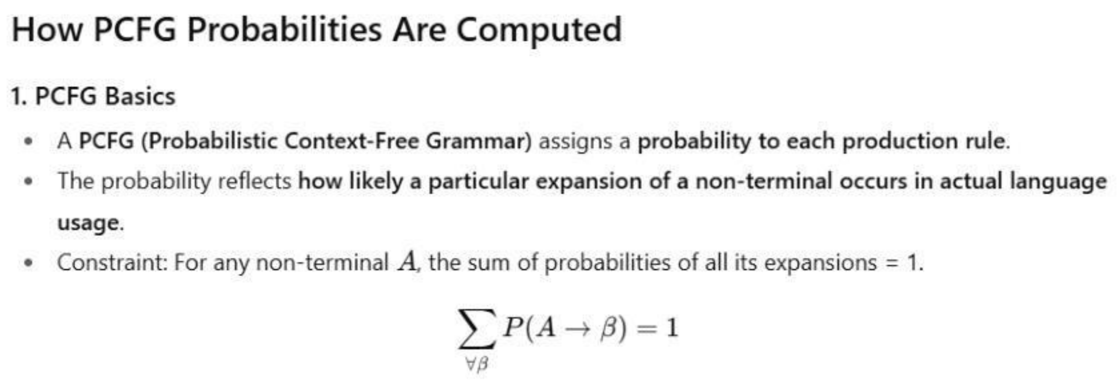
- **Handles ambiguity** by selecting the most probable parse.
- Can be trained from **treebanks** to capture real-language usage.
- Provides a **probabilistic ranking of parse trees**.

6. Disadvantages

- Accuracy depends on **quality and size of annotated corpus**.
- Assumes **independence of rules**, which may not capture long-range dependencies.
- Computationally more expensive than CFG parsing.

7. Applications

- **Statistical Syntactic Parsing** (disambiguation of multiple parse trees)
- **Machine Translation** (syntax-based translation)
- **Speech Recognition** (probable syntactic structure for word sequences)
- **Information Extraction** (finding structured information from text)

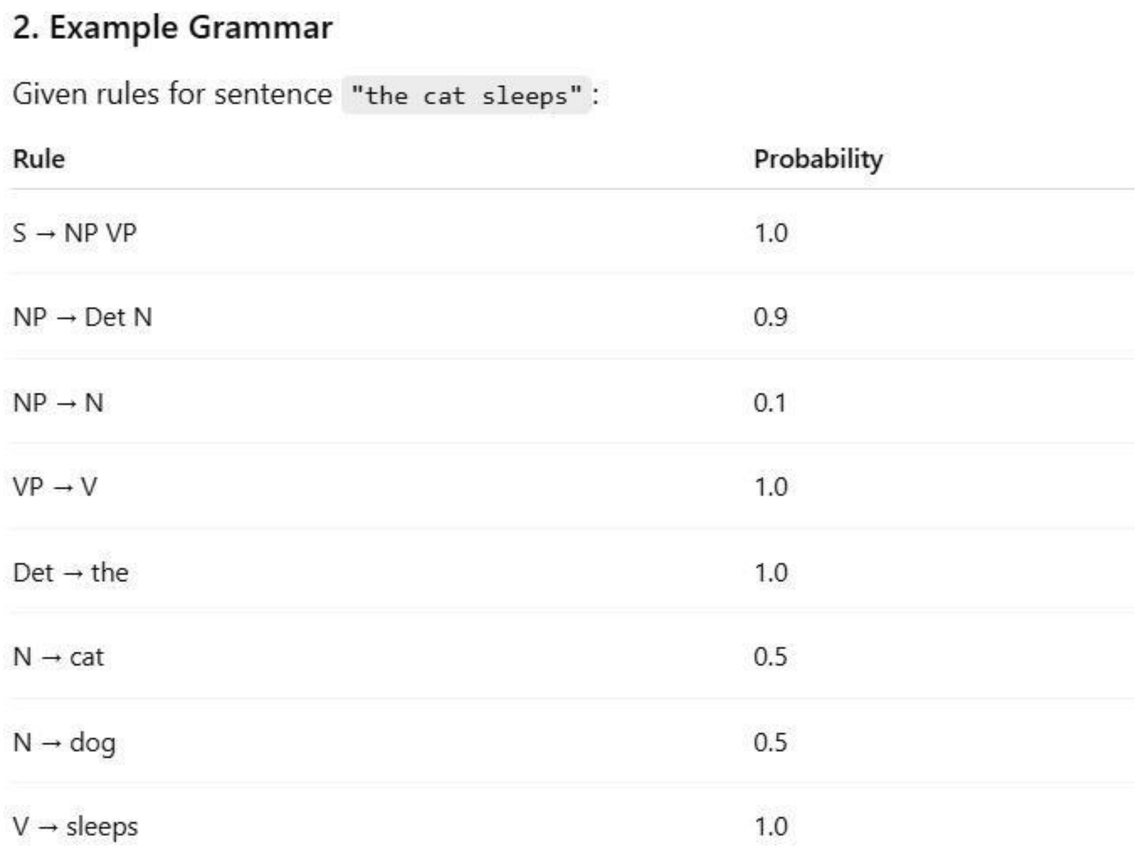


How PCFG Probabilities Are Computed

1. PCFG Basics

- A **PCFG (Probabilistic Context-Free Grammar)** assigns a **probability to each production rule**.
- The probability reflects **how likely a particular expansion of a non-terminal occurs in actual language usage**.
- Constraint: For any non-terminal A, the sum of probabilities of all its expansions = 1.

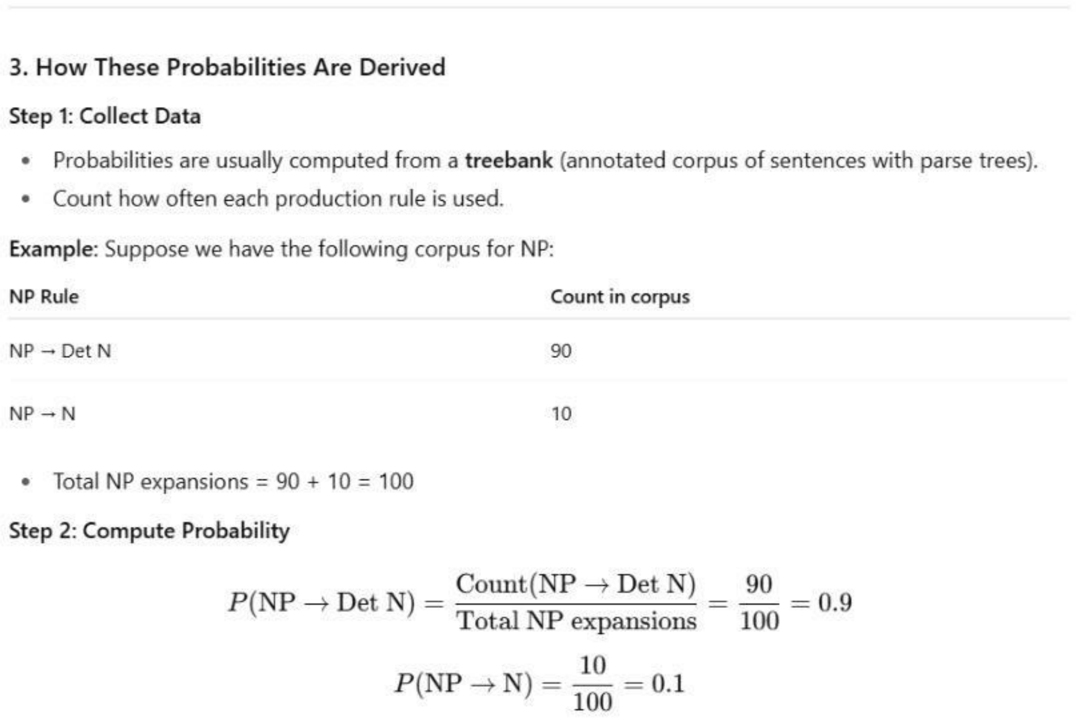
$$\sum_{\forall \beta} P(A \rightarrow \beta) = 1$$



2. Example Grammar

Given rules for sentence `"the cat sleeps"`:

Rule	Probability
S → NP VP	1.0
NP → Det N	0.9
NP → N	0.1
VP → V	1.0
Det → the	1.0
N → cat	0.5
N → dog	0.5
V → sleeps	1.0



3. How These Probabilities Are Derived

Step 1: Collect Data

- Probabilities are usually computed from a **treebank** (annotated corpus of sentences with parse trees).
- Count how often each production rule is used.

Example: Suppose we have the following corpus for NP:

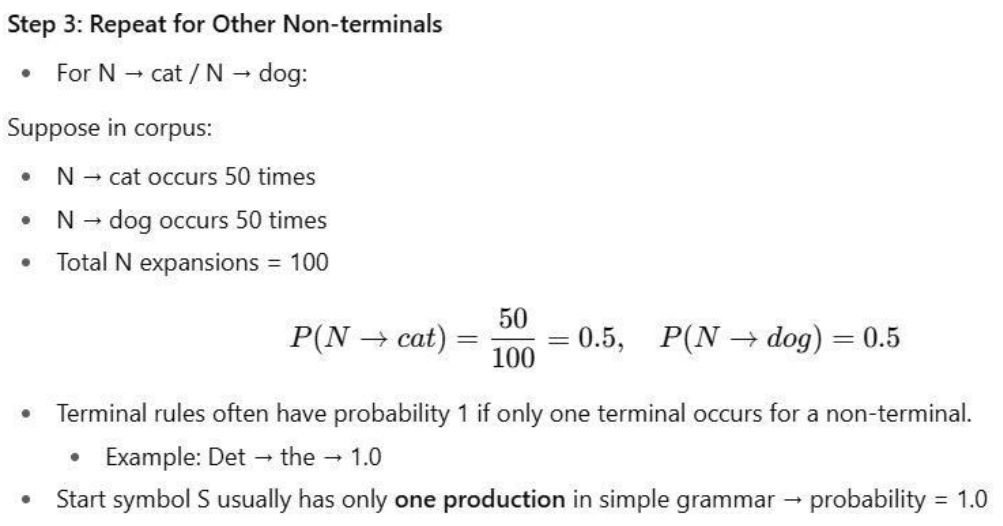
NP Rule	Count in corpus
NP → Det N	90
NP → N	10

- Total NP expansions = 90 + 10 = 100

Step 2: Compute Probability

$$P(\text{NP} \to \text{Det N}) = \frac{\text{Count(NP} \to \text{Det N)}}{\text{Total NP expansions}} = \frac{90}{100} = 0.9$$

$$P(\text{NP} \to \text{N}) = \frac{10}{100} = 0.1$$



Step 3: Repeat for Other Non-terminals

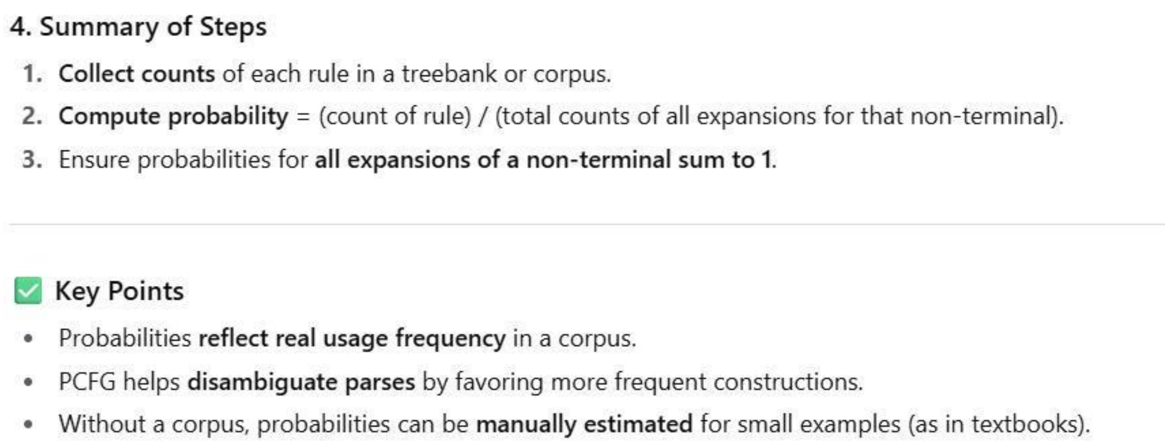
- For N → cat / N → dog:

Suppose in corpus:

- N → cat occurs 50 times
- N → dog occurs 50 times
- Total N expansions = 100

$$P(N \to cat) = \frac{50}{100} = 0.5, \quad P(N \to dog) = 0.5$$

- Terminal rules often have probability 1 if only one terminal occurs for a non-terminal.
  - Example: Det → the → 1.0
- Start symbol S usually has only **one production** in simple grammar → probability = 1.0



4. Summary of Steps

1. **Collect counts** of each rule in a treebank or corpus.
2. **Compute probability** = (count of rule) / (total counts of all expansions for that non-terminal).
3. Ensure probabilities for **all expansions of a non-terminal sum to 1**.

✅ Key Points

- Probabilities **reflect real usage frequency** in a corpus.
- PCFG helps **disambiguate parses** by favoring more frequent constructions.
- Without a corpus, probabilities can be **manually estimated** for small examples (as in textbooks).

FEATURE STRUCTURES AND UNIFICATION

- **Feature structures:** Attribute-value pairs representing linguistic info.
- **Unification:** Merging FS consistently; fails if conflicts exist.
- Essential for **agreement checking, parsing, and constraint-based grammar systems**.

1. Introduction

- **Feature Structures (FS)** are a way to represent **rich linguistic information** about words, phrases, or syntactic categories.
- Common in **unification-based grammars** like **Head-Driven Phrase Structure Grammar (HPSG)**.
- They help encode **syntactic, semantic, morphological, and agreement information** compactly.

2. What is a Feature Structure?

- A **feature structure** is essentially a **set of attribute-value pairs**.
- Example: For the word "dogs":

[

CATEGORY: Noun

NUMBER: Plural

PERSON: 3rd

  GENDER: Neutral

]

- Attributes = CATEGORY, NUMBER, PERSON, GENDER
- Values = Noun, Plural, 3rd, Neutral
- Can also include **nested feature structures**, e.g., for a verb phrase:

[

  CATEGORY: VP

  HEAD: [CATEGORY: V, TENSE: Present, NUMBER: Singular]

  SUBJ: [CATEGORY: NP, NUMBER: Singular]

]

3. Unification

- **Unification** is the process of **merging two feature structures** consistently.
- If two structures **agree on shared features**, they are **unified**.
- If they **conflict**, unification **fails**.

4. Example of Unification

FS1 (Subject NP):

[CATEGORY: NP, NUMBER: Singular]

FS2 (Verb VP Head):

[CATEGORY: V, NUMBER: Singular]

Unifying NP and VP features for agreement:

- Both have NUMBER = Singular → compatible → unification succeeds.
- If VP had NUMBER = Plural → conflict → unification fails.

5. Advantages of Feature Structures

1. **Expressive** – Can capture multiple linguistic properties in one structure.
2. **Handles agreement constraints** – E.g., subject-verb agreement.

3. **Supports modular grammars** – Features can be added or reused.
4. **Basis for unification-based parsers** – Widely used in HPSG, LFG, and TAG.

6. Applications in NLP

- **Syntactic parsing** – unification ensures features match across constituents.
- **Morphological analysis** – number, gender, tense, person.
- **Semantic interpretation** – features can include semantic roles.
- **Constraint-based grammars** – HPSG, Lexical Functional Grammar (LFG).