Unit 2
Syntax Analysis

Unit 2 – Syntax Analysis

Context Free Grammars, Grammar Rules for English, Top-Down Parsing, Bottom-Up Parsing, Ambiguity, CKY Parsing, Dependency Parsing, Earley Parsing - Probabilistic Context-Free Grammars

Context Free Grammars

Definition 1:

A context-free grammar consists of a set of rules or productions, each of which expresses the ways that symbols of the language can be grouped and ordered together, and a lexicon of words and symbols.

Definition 2:

A context-free grammar (CFG) is a list of rules that define the set of all well-formed sentences in a language. Each rule has a left-hand side, which identifies a syntactic category, and a right-hand side, which defines its alternative component parts, reading from left to right.

Context Free Grammars

A context-free grammar (CFG) is a set of production rules used to generate all the possible sentences in a given language.

A CFG consists of a set of terminals, which are the basic units of the language, and a set of non-terminals, which are used to generate the terminals through a set of production rules.

Context Free Grammars

Context-free grammar G is a 4-tuple.

$$G = (V, T, S, P)$$

These parameters are as follows;

- **V** – ***Set of variables*** (also called as **Non-terminal symbols**)
- **T** – ***Set of terminal symbols*** (**lexicon**)
  - The symbols that refer to words in a language are called **terminal symbols.**
- **Lexicon** is a set of rules that introduce these symbols.
- **S** – ***Designated start symbol*** (one of the non-terminals, $S \in V$)

Context Free Grammars

- **S** – ***Designated start symbol*** (one of the non-terminals, $S \in V$)
- **P** – ***Set of productions*** (also called as **rules**).

Each rule in P is of the form $\mathbf{A} \rightarrow \mathbf{s}$, where

- A is a non-terminal (variable) symbol.
- Each rule can have only one non-terminal symbol on the left hand side of the rule.
- s is a sequence of terminals and non-terminals. It is from $(T \cup V)^*$, infinite set of strings.
- *A grammar G generates a language L*.

Basic Grammer

Determiner → The | a | an | this | these | that | those | many | enough
Noun → History | house | ocean | guest | artwork | map | years
Verb → Keeps | make | walk | went | see | looks | is | are
Adjective → Great | exact | unfortunate | bad | full | deep
NP → Determiner nominal
Nominal → Noun | nominal noun

Context Free Grammars

- A leftmost derivation is a sequence of strings s1,s2, …sn
  - s1 =S, the start symbol
  - sn, includes only terminal symbols

  Example

  [S]

  [S] [NP VP]

  [S] [NP VP] [DT N VP]

  …..

  [S] [NP VP]

Context Free Grammars

- ["the", "child", "ate", "the", "cake", "with", "the", "fork"]
- S -> NP VP
- NP -> DT N | NP PP
- PP -> PRP NP
- VP -> V NP | VP PP
- DT -> 'a' | 'the'
- N -> 'child' | 'cake' | 'fork'
- PRP -> 'with' | 'to'
- V -> 'saw' | 'ate'

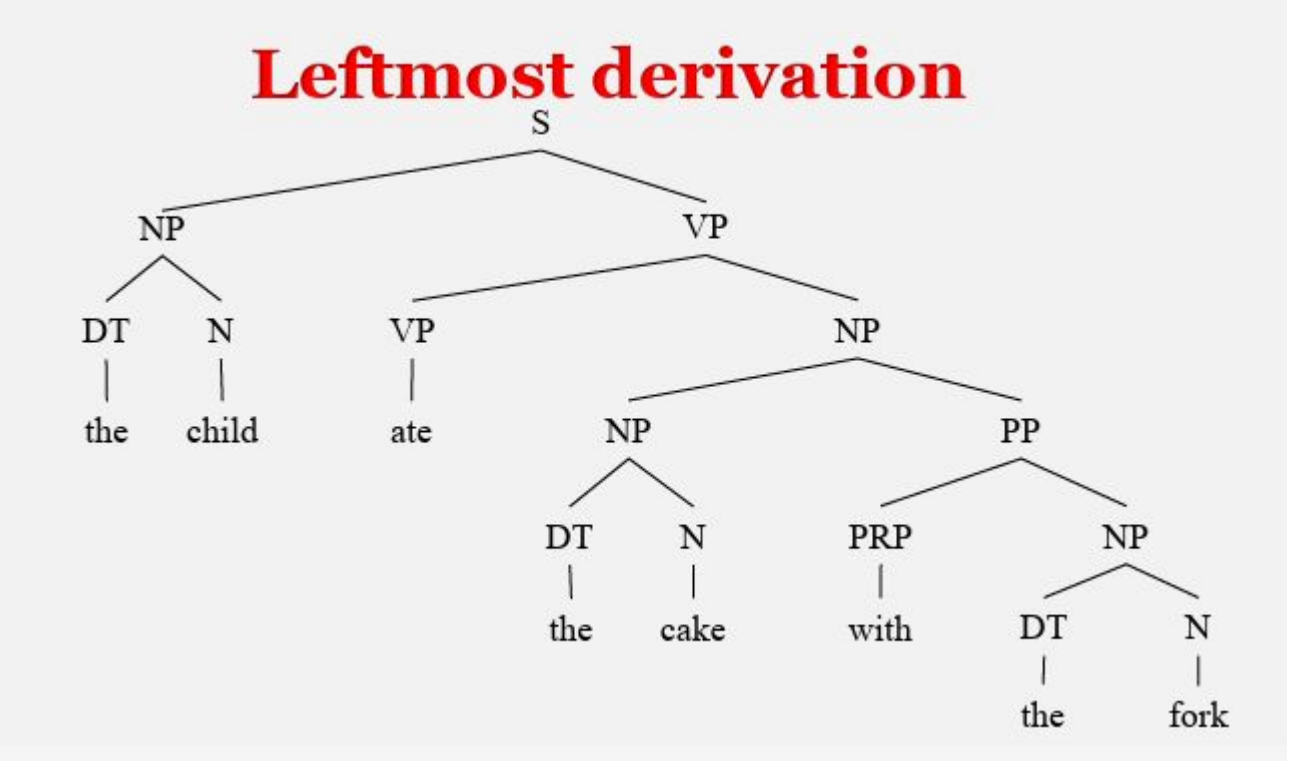
1. $S \rightarrow NP\ VP$
2. $NP \rightarrow Det\ N$
3. $VP \rightarrow V\ NP\ PP$
4. $VP \rightarrow V\ NP$
5. $PP \rightarrow P\ NP$
6. $Det \rightarrow$ "The"
7. $N \rightarrow$ "child"
8. $N \rightarrow$ "cake"
9. $N \rightarrow$ "fork"
10. $V \rightarrow$ "ate"
11. $P \rightarrow$ "with"

1. $S \Rightarrow NP\ VP$ Production Rule1
2. $NP \Rightarrow Det\ N$ Production Rule2
3. $Det \Rightarrow$ "The"
4. $N \Rightarrow$ "child"
5. $VP \Rightarrow V\ NP\ PP$ Production Rule 4 and 3
5. $V \Rightarrow$ "ate"
7. $NP \Rightarrow Det\ N$ Production Rule2
3. $Det \Rightarrow$ "the"
9. $N \Rightarrow$ "cake"
0. $PP \Rightarrow P\ NP$ Production Rule 5
1. $P \Rightarrow$ "with"
2. $NP \Rightarrow Det\ N$
3. $Det \Rightarrow$ "the"
4. $N \Rightarrow$ "fork"

CFG Rule & Leftmost derivation

THE CHILD ATE THE CAKE WITH THE FORK

Left most derivation

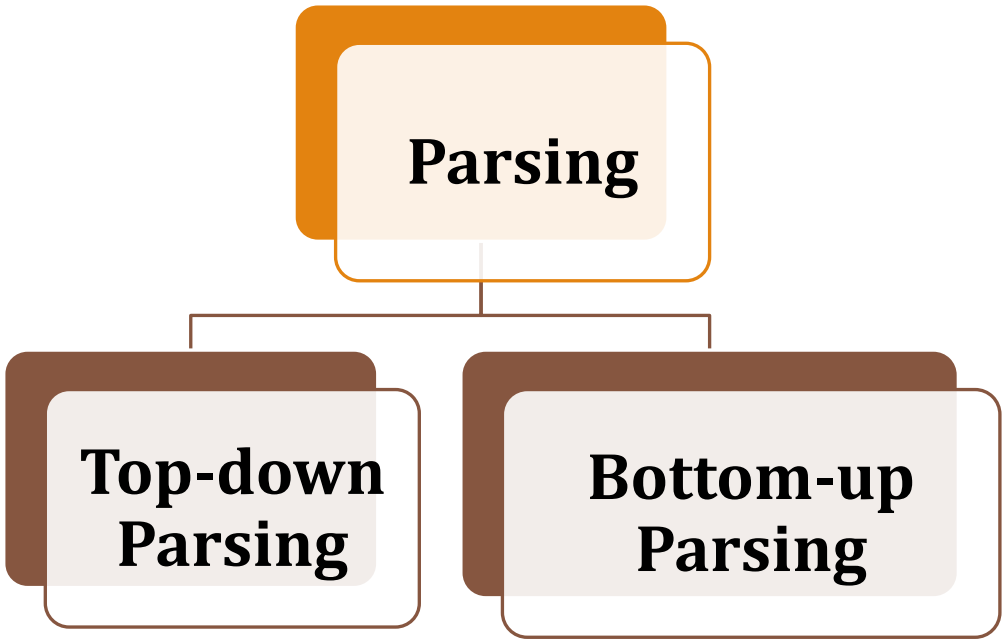


Grammar Rules for English

- **Sentence structure:** A sentence must have a subject and a predicate.
- **Parts of speech:** Words are categorized into nouns, verbs, adjectives, adverbs, pronouns, prepositions, conjunctions, and interjections.
- **Agreement:** Subject and verb must agree in number.
- **Tense consistency:** Verbs must be used consistently in the same tense.
- **Pronoun agreement**: Pronouns must agree with the antecedent in gender and number.
- **Modifiers:** Adjectives and adverbs must be placed correctly in sentences.
- **Parallelism:** Ideas in a list or comparison must be expressed in parallel form.
- **Capitalization:** The first word of a sentence and proper nouns must be capitalized.
- **Punctuation:** Proper use of punctuation is crucial for clear and concise writing.
- **Spelling:** Correct spelling is important for clarity and credibility.
- **Word choice:** Precise word choice is essential for effective communication.
- **Tone:** The tone of a piece of writing should match the purpose and audience.

Parsing

Parsing is the technique of examining a text containing a string of tokens, to find its grammatical structure according to the given grammar.



Parsing

- The parser receives a string of token from the lexical analyser.
- When the string of tokens can be generated by the grammar of the source language, then the parse tree can be constructed.
- Additionally, it reports the syntax errors present in the course thing. After this, the generated parse tree is transferred to the next stage of the compiler.

Top-Down Parsing

- **Top-down parsing is a parsing strategy used in Natural Language Processing (NLP) to analyze sentence structure based on a predefined grammar (such as Context-Free Grammar - CFG).**
- **It starts from the root (start symbol) and expands downward by applying grammar rules, attempting to match the input sentence.**

How Top-Down Parsing works?

Start with the Start Symbol

- **The parser begins with the root of the grammar (e.g., S for sentence).**

Expand Using Grammar Rules

- **Apply production rules (from a grammar) to break the sentence into smaller components.**

Match Input Sentence

- **Compare expanded rules with the actual input string.**

Backtracking (if needed)

- **If a rule fails, the parser backtracks and tries an alternative rule.**

Top-Down Parsing

S → NP VP

NP → Det N | N

VP → V NP

Det → 'the' | 'a'

N → 'cat' | 'dog'

V → 'chased' | 'saw'

Sentence to Parse

"The cat chased a dog"

Parsing Process (Top-Down)

Start with S

Expand S → NP VP

Expand NP → Det N → "The cat"

Expand VP → V NP → "chased a dog"

The sentence matches the grammar!

Top-down Parsing

```
import nltk
from nltk import CFG
# Define the grammar
grammar = CFG.fromstring("""
    S -> NP VP
    NP -> Det N | N
    VP -> V NP
    Det -> 'the' | 'a'
    N -> 'cat' | 'dog'
    V -> 'chased' | 'saw'
""")
```

```
# Create a recursive descent parser
(Top-Down)
parser =
nltk.RecursiveDescentParser(grammar)
# Sentence to parse
sentence = "the cat chased a
dog".split()
# Parsing the sentence
for tree in parser.parse(sentence):
    print(tree)  # Print the parse tree
    tree.pretty_print()  # Display tree
structure
```

Top-down Parsing

Advantages

✔Easy to implement using recursive functions.

✔ Uses grammar rules directly for parsing.

✔ Good for small grammars in NLP.

Bottom-up Parsing

Bottom-up parsing is a parsing strategy used in Natural Language Processing (NLP) where the parser starts from the input words (tokens) and builds up towards the start symbol (S) using grammar rules.

How Bottom-up Parsing works?

Start from the Input Words

- **Begin with individual words (tokens) in the sentence.**

Apply Grammar Rules in Reverse

- **Try to form higher-level constructs (phrases, clauses) by combining tokens.**

Reduce to the Start Symbol (S)

- **Continue merging rules until the full sentence is recognized as S.**

Shift-Reduce Mechanism (Used in Bottom-Up Parsers)

- **Shift: Read the next input token.**
- **Reduce: Apply a grammar rule if possible.**
- **Repeat until the full sentence is parsed.**

Bottom-up Parsing

S → NP VP

NP → Det N | N

VP → V NP

Det → 'the' | 'a'

N → 'cat' | 'dog'

V → 'chased' | 'saw'

Sentence to Parse

"The cat chased a dog"

Parsing Process (Bottom-Up)

Start with words: [the, cat, chased, a, dog]

Apply Det → the, N → cat → Reduce to NP

Apply V → chased

Apply Det → a, N → dog → Reduce to NP

Apply VP → V NP

Apply S → NP VP

✅ **Successfully parsed!**

Bottom-up Parsing

```
import nltk
from nltk import CFG

# Define the grammar
grammar = CFG.fromstring("""
    S -> NP VP
    NP -> Det N | N
    VP -> V NP
    Det -> 'the' | 'a'
    N -> 'cat' | 'dog'
    V -> 'chased' | 'saw'
""")

# Create a shift-reduce parser (Bottom-Up)
parser = nltk.ShiftReduceParser(grammar)

# Sentence to parse
sentence = "the cat chased a dog".split()

# Parsing the sentence
for tree in parser.parse(sentence):
    print(tree)  # Print the parse tree
    tree.pretty_print()  # Display tree structure
```

Bottom-up Parsing

Advantages of Bottom-Up Parsing

✔ Efficient for large grammars since it avoids unnecessary expansions.
✔ Handles left-recursive grammars naturally.
✔ No need to predict next rules like in top-down parsing.

Top-down vs Bottom-up

BASIS FOR COMPARISON	TOP-DOWN PARSING	BOTTOM-UP PARSING
Initiates from	Root	Leaves
Working	Production is used to derive and check the similarity in the string.	Starts from the token and then go to the start symbol.
Uses	Backtracking (sometimes)	Handling
Strength	Moderate	More powerful
Producing a parser	Simple	Hard
Type of derivation	Leftmost derivation	Rightmost derivation

CYK Parsing

The Cocke-Younger-Kasami (CYK) algorithm is a bottom-up parsing technique used to determine whether a given string belongs to a context-free grammar (CFG) and, if so, construct a parse tree.

Features

- ✔ **Efficient for Context-Free Grammars (CFGs)**
- ✔ **Uses Dynamic Programming (DP) to parse a string in $O(n^3)$ time**
- ✔ **Requires the grammar to be in Chomsky Normal Form (CNF)**

CYK Parsing

Before using CYK, the grammar must be in CNF, meaning:

Every production rule is either:

$A \rightarrow BC$ (two non-terminals)

$A \rightarrow a$ (a single terminal)

No empty (ε) or unit productions ($A \rightarrow B$) allowed.

CYK Parsing

S → NP VP

NP → Det N | N

VP → V NP

Det → 'the' | 'a'

N → 'cat' | 'dog'

V → 'chased' | 'saw'

CYK Parsing Algorithm

1)Create a 2D Table (Parse Table)

- Rows: **Substrings of input sentence**
- Columns: **Grammar rules applied**

2) Fill the Table Using Dynamic Programming

- Assign **terminal rules** in the bottom row.
- Combine non-terminals based on **grammar rules.**

3) Check if the Start Symbol (S) Appears at the Top

- If S is found in the last row, the sentence **is valid.**

CYK Parsing

Example Parse Table for "the cat chased a dog“

Word	the	cat	Chased	a	dog
Row 1	Det	N	V	Det	N
Row 2	NP	-	VP	NP	-
Row 3	-	S	-	-	-

CYK Parsing

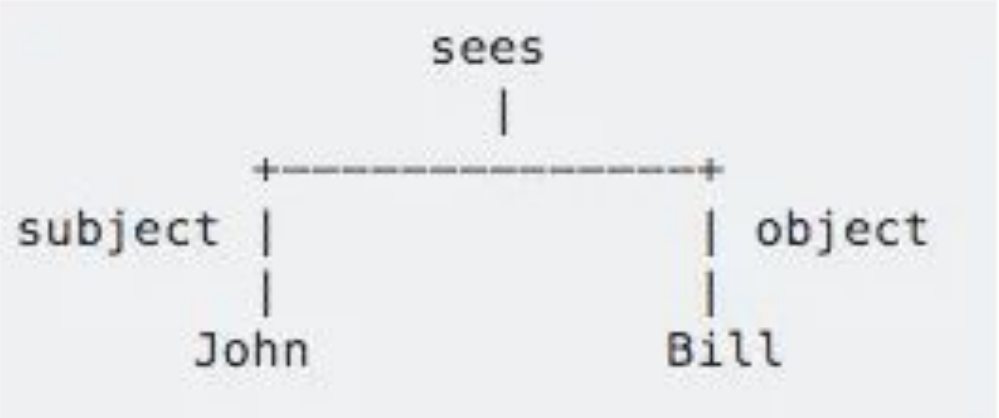
```
import nltk

# Define a CFG in Chomsky Normal Form
(CNF)
cfg = nltk.CFG.fromstring("""
    S -> NP VP
    NP -> Det N | N
    VP -> V NP
    Det -> 'the' | 'a'
    N -> 'cat' | 'dog'
    V -> 'chased' | 'saw'
""")

# Convert to CNF (for CYK Parsing)
cnf_grammar = cfg.chomsky_normal_form()
```

```
# CYK Parser in NLTK
from nltk.parse.chart import
ChartParser

# Create a CYK parser
parser = ChartParser(cfg)

# Sentence to parse
sentence = "the cat chased a
dog".split()

# Generate parse trees
for tree in parser.parse(sentence):
    print(tree)
    tree.pretty_print()
```

Dependency Parsing

It aims at breaking a sentence **depending on the relationship** between the words rather than any predefined ruleset.

The same sentence '**John sees Bill**' has the below dependency parsing:



Dependency Parsing

- **Dependency parsing is a technique used in Natural Language Processing (NLP) to analyze the grammatical structure of a sentence by establishing relationships (dependencies) between words.**
- **It helps determine the syntactic structure of a sentence by identifying the head (governing) words and their dependent words.**

Dependency Parsing

Head and Dependent

Every word (except the root) is dependent on another word (its head).

Example: In "She eats an apple", eats is the head, and She, an, and apple depend on it.

Dependency Tree

A tree-like structure that represents word relationships.

Example dependency tree for "She eats an apple":

eats

/ | \

Dependency Structure

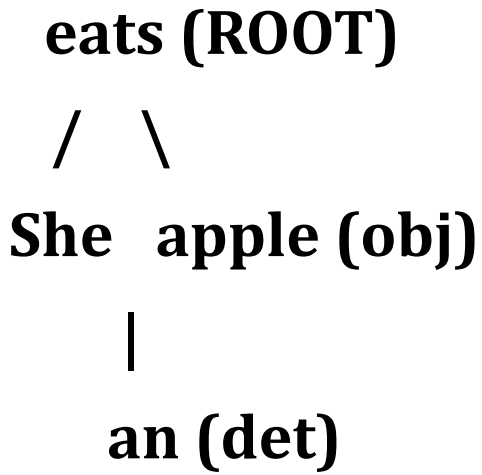
Root (Head Verb): "eats" (main verb)

Subject (nsubj - nominal subject): "She" (subject of "eats")

Object (obj - direct object): "apple" (direct object of "eats")

Determiner (det - determiner): "an" (modifier of "apple")

Dependency Tree:



Dependency Parsing

Dependency Relations

Words have specific grammatical relationships with their heads.

Common relations:

- **nsubj (Nominal Subject) → "She" depends on "eats"**
- **dobj (Direct Object) → "apple" depends on "eats"**
- **det (Determiner) → "an" depends on "apple"**

Dependency Parsing

```
import spacy

# Load English language model
nlp = spacy.load("en_core_web_sm")

# Sample sentence
sentence = "She eats an apple"

# Process the sentence
doc = nlp(sentence)
# Print dependencies
for token in doc:
    print(f"{token.text} -->
{token.dep_} --> {token.head.text}")
```

Output

```
She --> nsubj --> eats

eats --> ROOT --> eats

an --> det --> apple

apple --> dobj --> eats
```

Applications of Dependency Parsing

✅ **Machine Translation** – Understanding sentence structure improves accuracy.

✅ **Question Answering Systems** – Extracts relationships between words.

✅ **Sentiment Analysis** – Identifies dependencies for sentiment-related words.

✅ **Chatbots & Virtual Assistants** – Helps in semantic understanding.

Earley Parsing

Earley Parsing is a chart-based, top-down parsing algorithm used for context-free grammars (CFGs), including ambiguous and left-recursive grammars.

It was introduced by Jay Earley in 1970 and is efficient for parsing natural language and programming languages.

Features of Earley Parsing

- ✔ **Handles all CFGs, including left-recursive and ambiguous grammars.**
- ✔ **Works efficiently for both recognition and parsing.**
- ✔ **Best case: $O(n)$, Average case: $O(n^2)$, Worst case: $O(n^3)$.**
- ✔ **Uses dynamic programming to store intermediate parsing results.**

How Earley Parsing works?

Earley parsing uses a chart-based approach where parsing progresses through three operations:

1. **Prediction – Expands non-terminals using grammar rules.**
2. **Scanning – Matches input words with grammar rules.**
3. **Completion – Resolves dependencies and completes parsing.**

It maintains a chart (table) where each entry stores parsing states.

Earley Parsing

Given Grammar:

S → NP VP

NP → Det N | N

VP → V NP

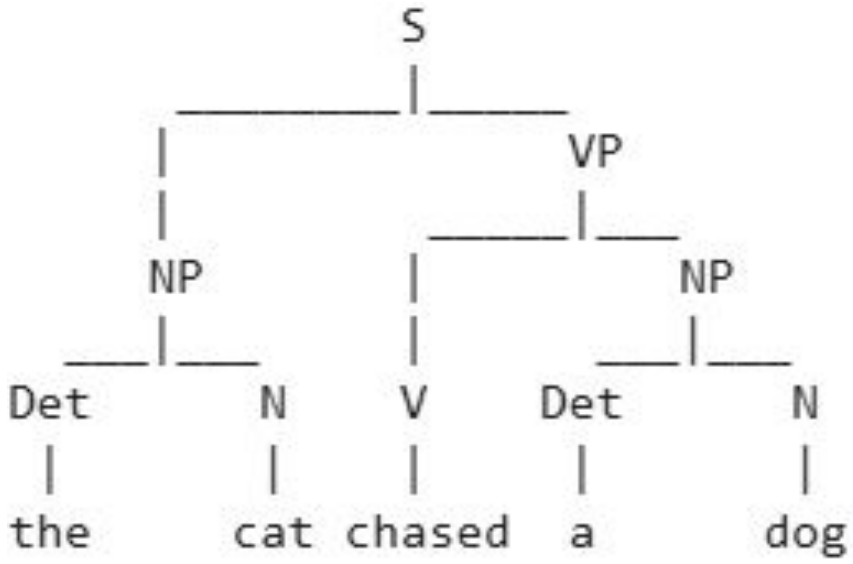
Det → 'the' | 'a'

N → 'cat' | 'dog'

V → 'chased' | 'saw'

Sentence to Parse

"The cat chased a dog“



Earley vs CYK vs Dependency Parsing

Feature	Earley Parsing	CYK Parsing	Dependency Parsing
Type	Top-down, dynamic	Bottom-up, dynamic	Word-to-word relations
Grammar Support	All CFGs	CNF-only	Non-CFGs possible
Handles Ambiguity?	Yes	No	Yes
Use Case	General NLP, Speech	Formal NLP, Parsing	Dependency-based NLP

Probabilistic CFG

A Probabilistic Context-Free Grammar (PCFG) is an extension of a Context-Free Grammar (CFG) where each production rule is assigned a probability. This probability represents the likelihood of a rule being used in generating a sentence.

Features of PCFG

✔ **Handles Ambiguity:** Helps choose the most probable parse among multiple possible parses.
✔ **Improves NLP Applications:** Used in speech recognition, machine translation, and syntax parsing.
✔ **Enables Probabilistic Parsing:** Rather than just checking grammatical correctness, PCFG predicts the most likely structure.

PCFG Definition

A PCFG is defined as (N, Σ, R, S, P):

- N = Non-terminals (e.g., S, NP, VP)
- Σ = Terminals (words like 'the', 'cat', 'chased')
- R = Set of production rules
- S = Start symbol
- P = Probability assigned to each rule

PCFG Grammar

S → NP VP (1.0)

NP → Det N (0.6) | N (0.4)

VP → V NP (1.0)

Det → 'the' (0.8) | 'a' (0.2)

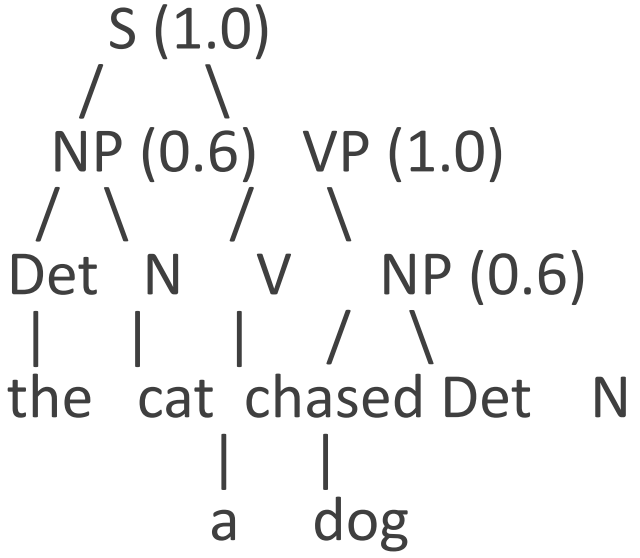
N → 'cat' (0.5) | 'dog' (0.5)

V → 'chased' (0.7) | 'saw' (0.3)

Note:Each rule's probabilities sum up to 1 for a given non-terminal.

Given:
"The cat chased a dog"

Possible Parse Tree:



PCFG Grammar

Probability of this parse:

P(S → NP VP) = 1.0

P(NP → Det N) = 0.6

P(Det → the) = 0.8

P(N → cat) = 0.5

P(VP → V NP) = 1.0

P(V → chased) = 0.7

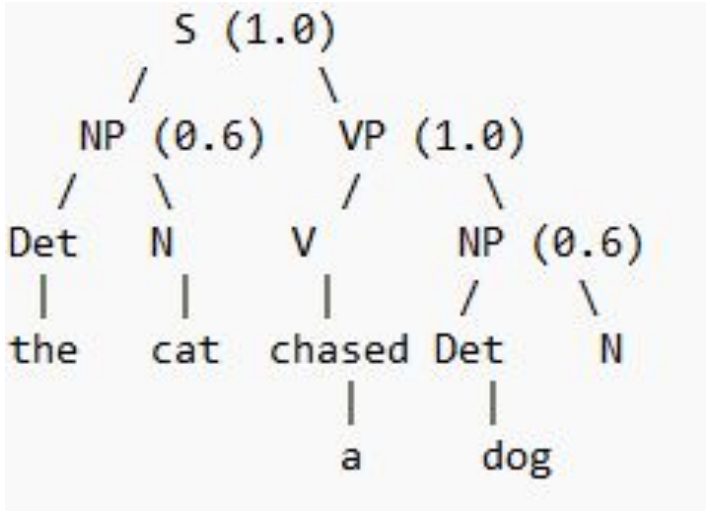
P(NP → Det N) = 0.6

P(Det → a) = 0.2

P(N → dog) = 0.5

Given:
"The cat chased a dog"

Possible Parse Tree:



Final Probability: 1.0 × 0.6 × 0.8 × 0.5 × 1.0 × 0.7 × 0.6 × 0.2 × 0.5 = 0.0168

PCFG vs CFG vs Dependency Parsing

Features	PCFG	Regular CFG	Dependency Parsing
Type	Probabilistic	Rule-based	Word-to-word relations
Handles Ambiguity?	✅ Yes	❌ No	✅ Yes
Focus	Syntax + Probability	Syntax only	Dependencies
Used In	Speech, AI, NLP	Formal NLP	Chatbots, AI models

Ambiguity

Ambiguity in parsing occurs when a sentence has multiple valid parse trees or interpretations. This makes it difficult for a parser to determine the intended meaning of a sentence.

Ambiguity is a major challenge in Natural Language Processing (NLP), compilers, and AI-driven text analysis.

Types of Ambiguity

- **Syntactic Ambiguity**
- **Lexical Ambiguity**
- **Attachment Ambiguity**
- **Coordination Ambiguity**
- **Scope Ambiguity**

Syntactic Ambiguity (Structural Ambiguity)

It occurs when a sentence has more than one grammatical structure, leading to multiple valid parse trees.

Example:

💬 **"I saw the man with the telescope."**

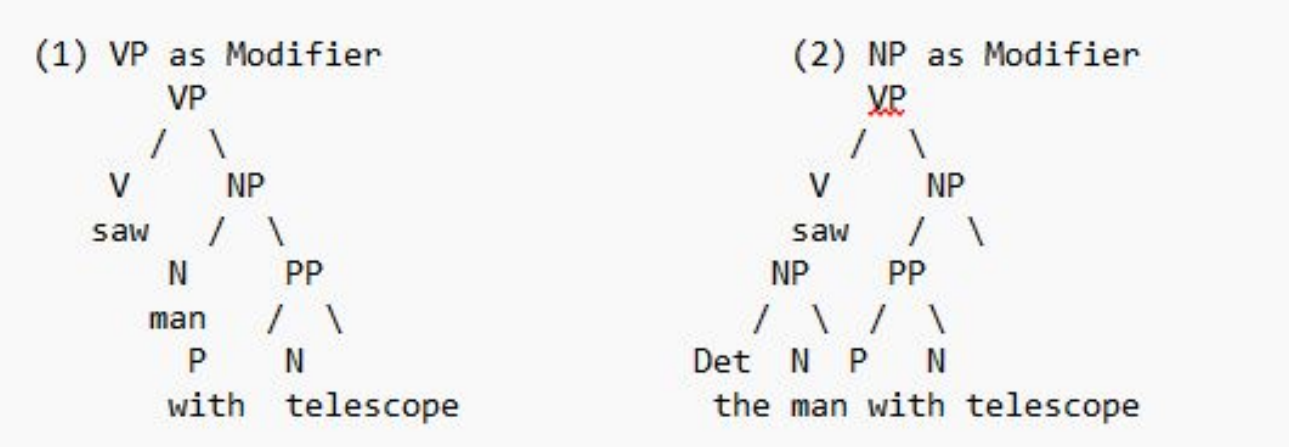
👉 **Possible Interpretations:**

- **I used a telescope to see the man.**
- **The man I saw had a telescope.**

Syntactic Ambiguity (Structural Ambiguity)

(1) VP as Modifier

(2) NP as Modifier



✅ **PCFG (Probabilistic Context-Free Grammar) can help by choosing the more likely structure.**

(1) VP as Modifier

- In this structure, "with a telescope" modifies the verb phrase (VP).
- This means that "with a telescope" describes how the action "saw" was performed.
- Interpretation: The subject (unseen in the diagram) "saw the man" using a telescope.
- Tree structure:
  - The VP consists of the verb "saw" and the NP "man with a telescope."
  - The NP "man" stands alone, and the PP "with a telescope" is attached separately to the VP.

(2) NP as Modifier

- In this structure, "with a telescope" modifies the noun phrase (NP).
- This means that "with a telescope" describes the man rather than the action.
- Interpretation: The observer "saw the man who had a telescope."

Tree structure:

  - The NP consists of "the man with a telescope" as a single unit.
  - The PP "with a telescope" is attached directly to "man," indicating possession or an attribute of the man.

Lexical Ambiguity (Word Sense Ambiguity)

Occurs when a word has multiple meanings in different contexts.

Example:

💬 **"The bank is closed."**

👉 **Possible Interpretations:**

Bank = Financial institution 🏦

Bank = Riverbank 🌊

- **Solution:**

Word sense disambiguation (WSD) techniques (e.g., BERT, WordNet) can help determine the most likely meaning based on context.

Attachment Ambiguity

Occurs when it's unclear which phrase a modifier (prepositional phrase or relative clause) should attach to.

Example:

💬 **"She hit the boy with the book."**

👉 **Possible Interpretations:**

- **She used a book to hit the boy.**
- **The boy who had a book was hit.**

  - **Solution:**

Probabilistic or dependency parsing can determine the most probable attachment.

Coordination Ambiguity

Occurs when a conjunction (and, or) connects phrases in a way that allows multiple interpretations.

Example:

💬 **"She saw the dog and the cat on the roof."**

👉 **Possible Interpretations:**

- **Both the dog and the cat were on the roof.**
- **She saw the dog, and the cat was on the roof.**

🔹 **Solution:**

Dependency parsing can clarify the correct attachment.

Scope Ambiguity

Occurs when it's unclear how far the scope of a quantifier or negation extends.

Example:

💬 **"All students didn't pass the exam."**

👉 **Possible Interpretations:**

- **No student passed.**
- **Some students failed.**

  - **Solution:**

Contextual NLP models like Transformer-based models (BERT, GPT) help resolve scope ambiguity.