Natural Language Processing
Course code: CSE3015

Module 3
Parsing Structure in Text

Prepared by

Dr. Venkata Rami Reddy Ch

SCOPE

Syllabus

- Shallow vs Deep parsing
- Context-Free Grammar (CFG) in Parsing
- Approaches in parsing
  - Top-Down Parsing
  - Bottom-up parsing
- Types of parsing-
  - Regex parser,
  - Constituency Parsing
    1. Probabilistic Context-Free Grammar (PCFG)
    2. CKY parsing algorithm
  - Dependency parser
    1. Transition-Based Parsing
    2. Graph-Based Parsing
- Meaning Representation:
  1. Logical Semantics,
  2. Semantic Role Labelling

Parsing in NLP

- Parsing is the process of analyzing the grammatical structure of a sentence to determine its syntactic or semantic meaning.
- It is a crucial component of NLP and helps machines understand human language.

The main purposes of parsing in NLP include:

1. **Understanding Sentence Structure** – It helps in breaking down a sentence into its grammatical components, such as nouns, verbs, subjects, and objects.
2. **Dependency Analysis** – It determines the relationships between words in a sentence, which is useful for applications like machine translation, question answering, and Chatbots
3. **Semantic Analysis** – Parsing provides a foundation for understanding meaning by identifying roles like subjects, predicates, and modifiers.
4. **Machine Translation** – Syntax-based translation models rely on parsing to preserve the grammatical structure in translated languages.
5. **Question Answering and Chatbots** – Understanding the structure of user queries helps generate more relevant and meaningful responses.

Parsing in NLP

- The word syntax originates from the Greek word syntaxis, meaning “arrangement”, and refers to how the words are arranged together.

- Sentence = S = Noun Phrase + Verb Phrase + Preposition Phrase

  S = NP + VP + PP

- The different word groups that exist according to English grammar rules are:–

  Noun Phrase(NP): Determiner + Nominal Nouns = DET + Nominal

  Verb Phrase (VP): Verb + range of combinations

  Prepositional Phrase (PP): Preposition + Noun Phrase = P + NP

Shallow Parsing (Chunking) in NLP

- Shallow parsing, also known as **chunking**, is used to extract phrases or chunks from sentences without analyzing their deeper syntactic structure.
- It identifies syntactic constituents (chunks) such as noun phrases (NPs), verb phrases (VPs), and prepositional phrases (PPs).

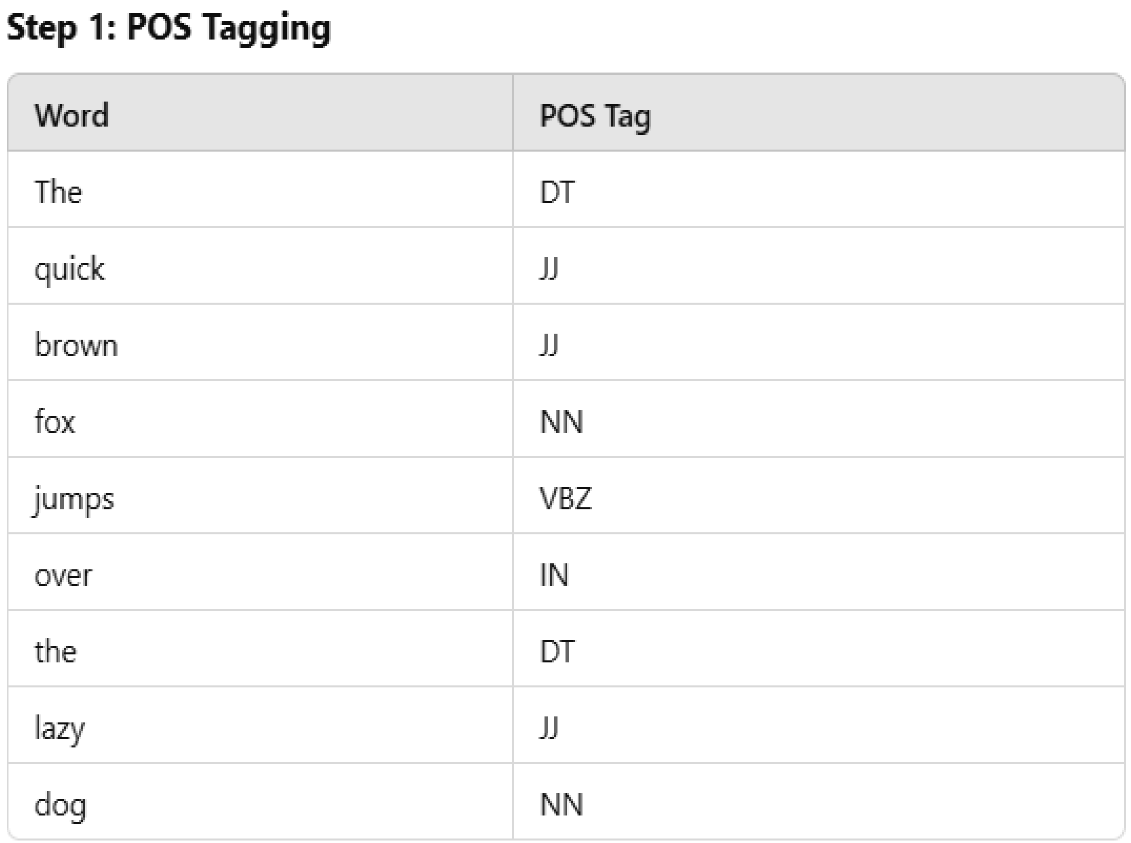
Steps in Shallow Parsing

1. **Tokenization** – Splitting text into words.
2. **POS Tagging** – Assigning part-of-speech (POS) tags to words.
3. **Chunking** – Grouping words into meaningful phrases (noun phrases, verb phrases, etc.)

Example of Shallow Parsing

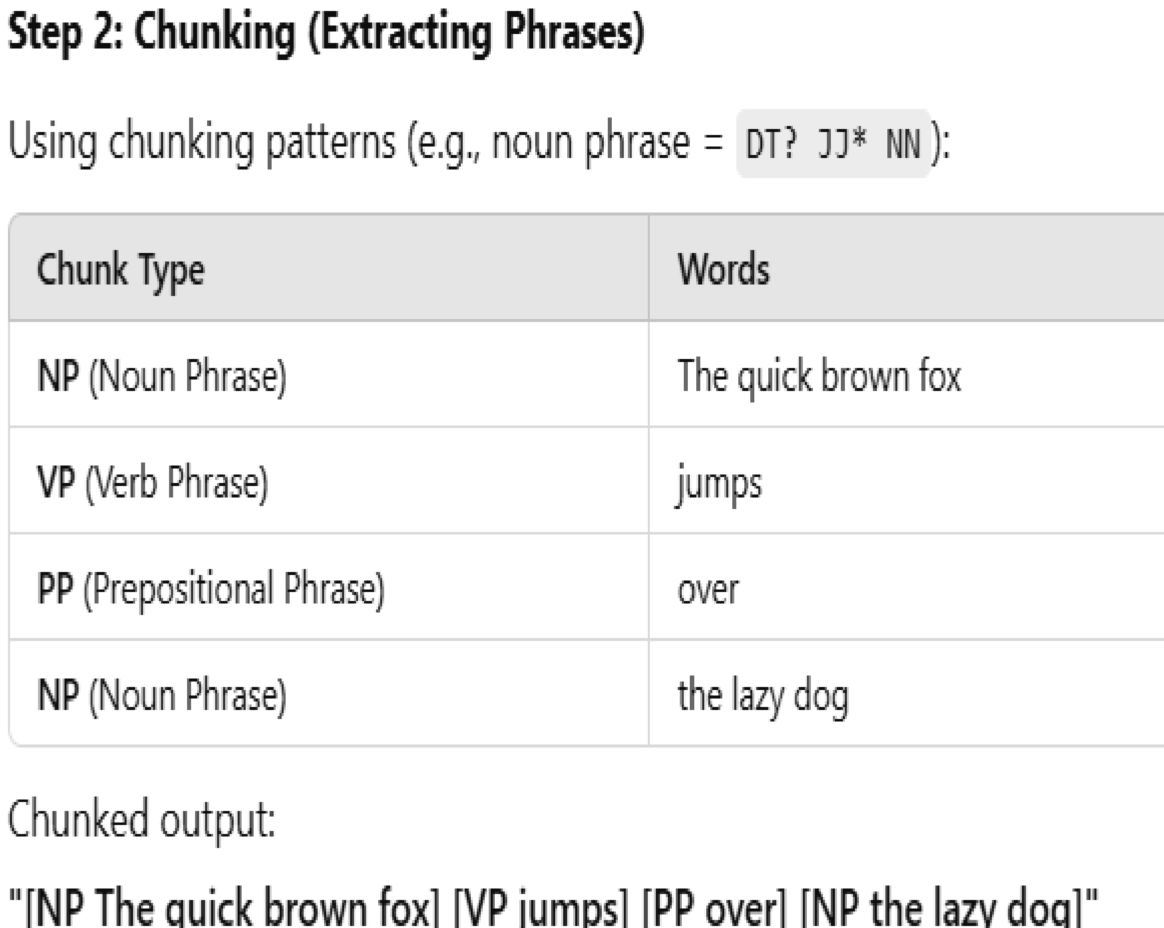
Sentence

- *"The quick brown fox jumps over the lazy dog."*



Step 1: POS Tagging

Word	POS Tag
The	DT
quick	JJ
brown	JJ
fox	NN
jumps	VBZ
over	IN
the	DT
lazy	JJ
dog	NN



Step 2: Chunking (Extracting Phrases)

Using chunking patterns (e.g., noun phrase = `DT? JJ* NN`):

Chunk Type	Words
NP (Noun Phrase)	The quick brown fox
VP (Verb Phrase)	jumps
PP (Prepositional Phrase)	over
NP (Noun Phrase)	the lazy dog

Chunked output:

"[NP The quick brown fox] [VP jumps] [PP over] [NP the lazy dog]"

Deep Parsing in NLP

- Deep parsing, also known as **full parsing** or **syntactic parsing**, is the process of analyzing the full syntactic structure of a sentence, typically producing a **parse tree** or **dependency graph**.
- Unlike shallow parsing , which only identifies phrases like noun phrases (NPs) and verb phrases (VPs), deep parsing provides a detailed hierarchical structure of the sentence, including grammatical relations.

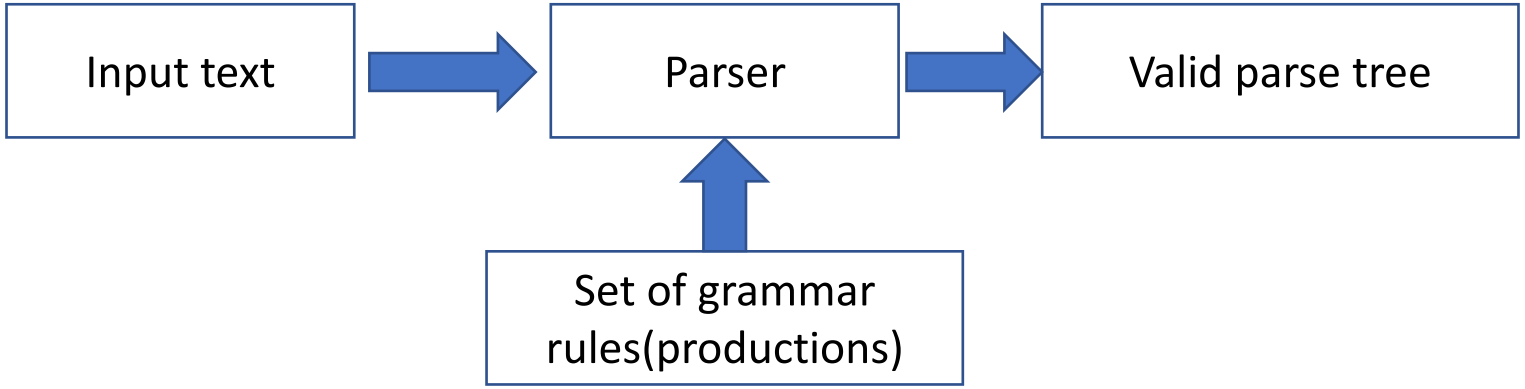
- **Key Aspects of Deep Parsing**

Full Sentence Structure

1. Identifies subject, object, verb, modifiers, etc.
2. Determines phrase structure (e.g., noun phrases, verb phrases).
3. Builds a parse tree (constituency tree) or dependency graph.

Deep Parsing in NLP

- Parser is used to implement the task of parsing.
- It may be defined as the software component designed for taking input data (text) and giving structural representation of the input after checking for correct syntax as per formal grammar.
- It builds a data structure generally in the form of parse tree or syntax tree or other hierarchical structure.
- As the parser splits the sequence of text into a bunch of words that are related in a sort of phrase.

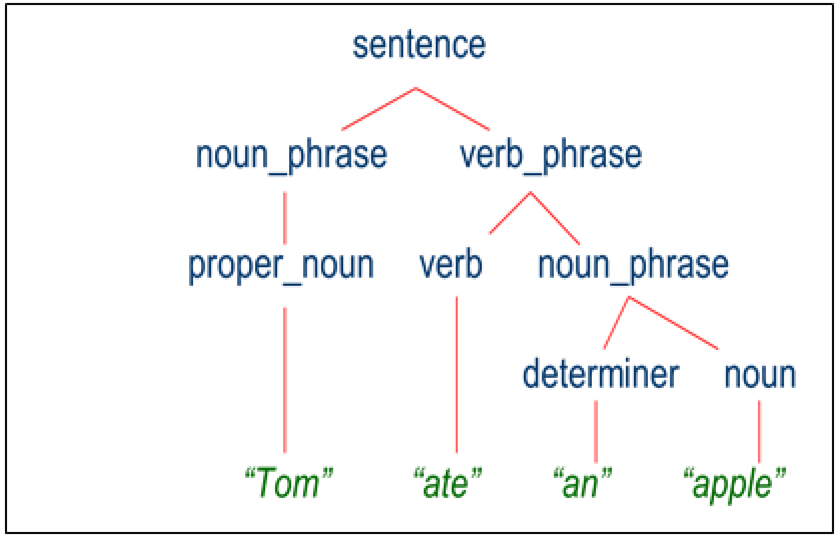


Example

Input : Tom ate an apple

```
sentence -> noun_phrase, verb_phrase
noun_phrase -> proper_noun
noun_phrase -> determiner, noun
verb_phrase -> verb, noun_phrase
proper_noun -> [Tom]
noun -> [apple]
verb -> [ate]
determiner -> [an]
```

parse tree



Shallow VS Deep Parsing

Shallow Parsing	**Deep Parsing**
This technique generates only phrases of the syntactic structure of a sentence.	This parsing technique generates the complete syntactic structure of the sentence.
It can be used for less complex NLP applications	It is suitable for complex NLP applications.
It is also called chunking.	It is also called full parsing.
Applications: NER, chunking, POS tagging	Applications: Syntax analysis, translation, QA

Context-Free Grammar (CFG) in Parsing

- A context-free grammar (CFG) is a set of production rules used to generate all the possible sentences in a given language
- CFG is widely used for syntactic parsing, where a parser constructs a **parse tree** to analyze the structure of a sentence based on the given grammar.
- These rules specify how individual words in a sentence can be grouped to form constituents such as noun phrases, verb phrases, preposition phrases, etc

Key Components of CFG

- A CFG is defined as a 4-tuple:

$$G=(N,\Sigma,P,S)$$

where:

- **N (Non-terminals)**: A set of non-terminal symbols (e.g., S, NP, VP).
- **Σ (Terminals)**: A set of terminal symbols (actual words or tokens in a language).
- **P (Production Rules)**: Rules that define how non-terminals can be replaced by other non-terminals or terminals.
- **S (Start Symbol)**: A special non-terminal from which parsing starts.

Context-Free Grammar (CFG) in Parsing

CFG:

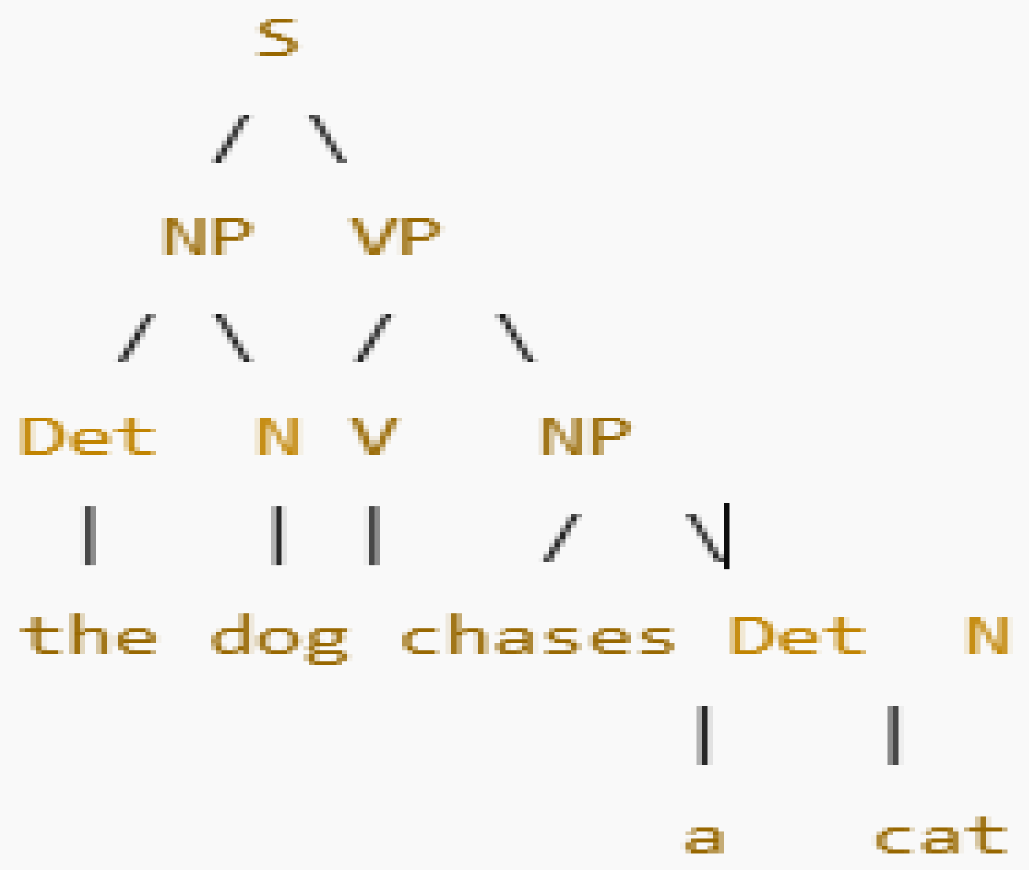
- S → NP VP
- NP → Det N
- VP → V NP
- Det → "the" | "a"
- N → "dog" | "cat"
- V → "chases" | "sees“

Derivation:

- S → NP VP
- S → Det N VP
- S → the dog VP
- S → the dog V NP
- S → the dog chases NP
- S → the dog chases Det N
- S → the dog chases a N
- S → the dog chases a cat

- A **parse tree** (or **syntax tree**) in **parsing** plays a crucial role in **NLP** by visually representing the **syntactic structure** of a sentence based on a given grammar.
- It is essential for understanding sentence structure and meaning.

parse tree



Approaches in parsing

- Top-Down Parsing
- Bottom-up parsing

Top-Down parsing

- Top-down parsing starts from the root (start symbol) of the grammar and tries to derive the input sentence by applying production rules.
- It recursively expands the non-terminals until the input is matched or parsing fails.
- Top-down, left-to-right and backtracking are prominent search strategies are used in this approach.

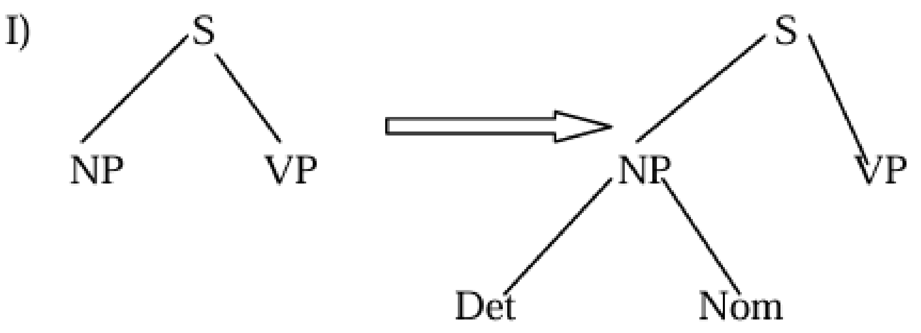
Steps of Top-Down Parsing

1. **Start with the start symbol**
   1. Begin with the root node (usually **S** in a context-free grammar).
2. **Expand using grammar rules**
   1. Apply production rules to expand non-terminals in a depth-first manner.
3. **Match the input sentence**
   1. Compare generated symbols with input tokens.
   2. If a match is found, continue expanding the next non-terminal.
4. **Backtracking (if necessary)**
   1. If a production does not match with input tokens, backtrack and try a different production.
5. **Repeat until the input is fully parsed**
   1. Continue until the entire input sentence is matched with a valid parse tree.

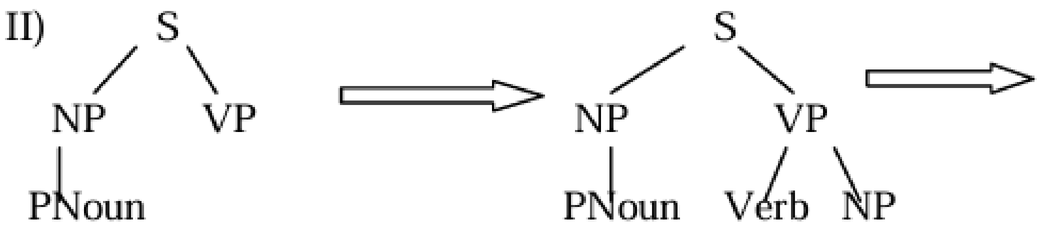
Top-Down parsing

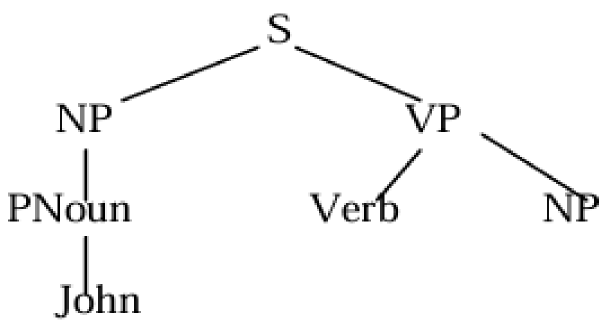
Example 1 : John is playing a game

S -> NP VP | NP AuxV VP
NP-> DET Nom | PNoun | Noun
VP->Verb NP | V NP
PNoun-> "John"
AuxV -> is
V->playing
Noun-> Game

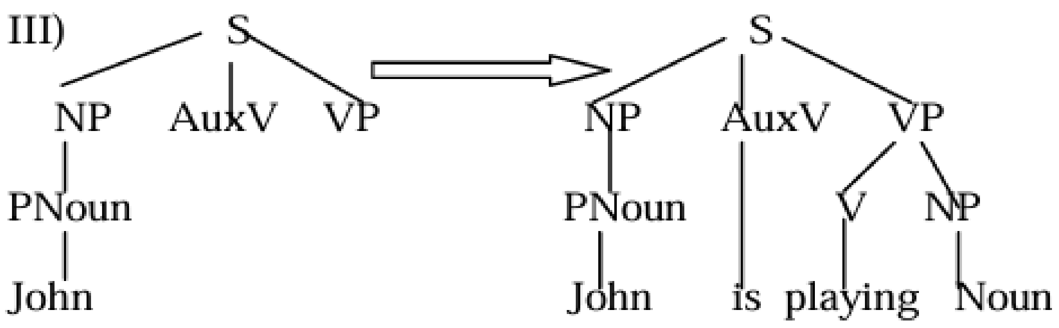


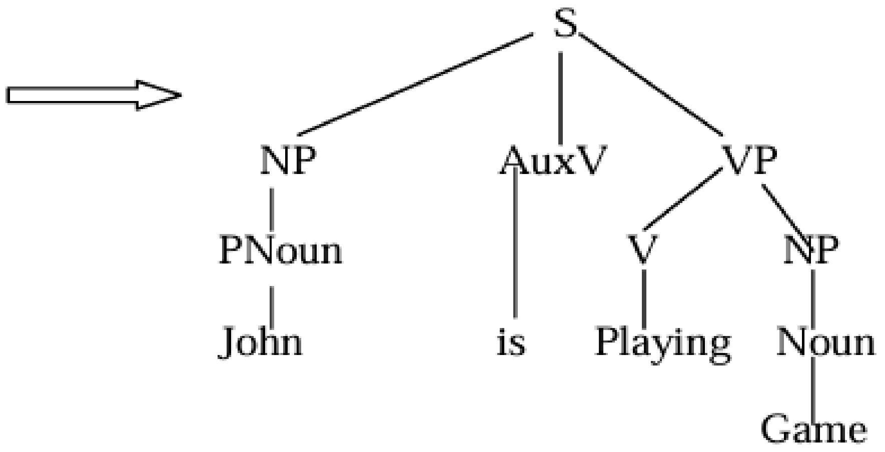
Part-of-speech does not match the input string, backtrack to the node NP





Part-of-speech Verb does not match the input string, backtrack to the node S, since PNoun is matched.

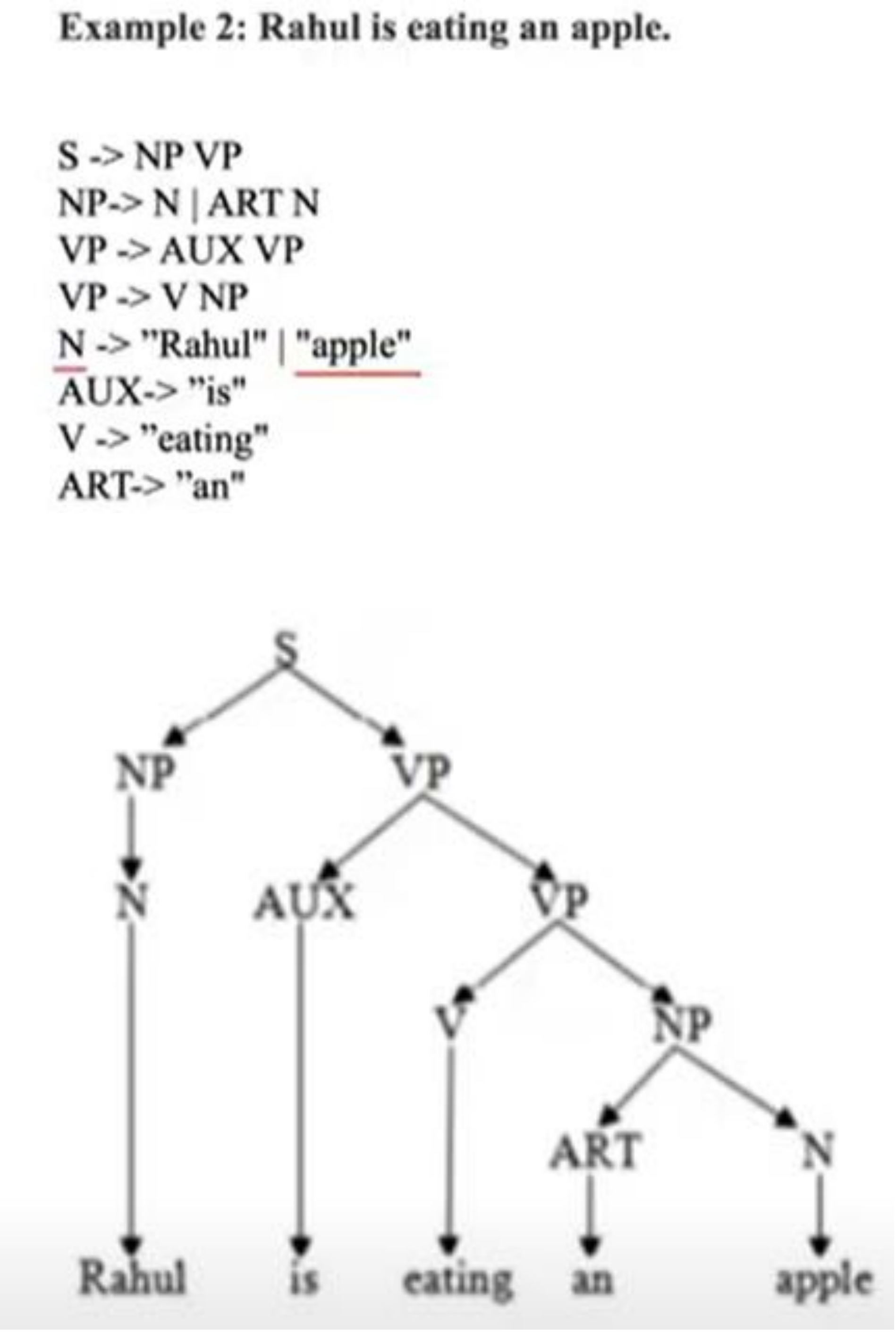




Final parse tree for the sentence

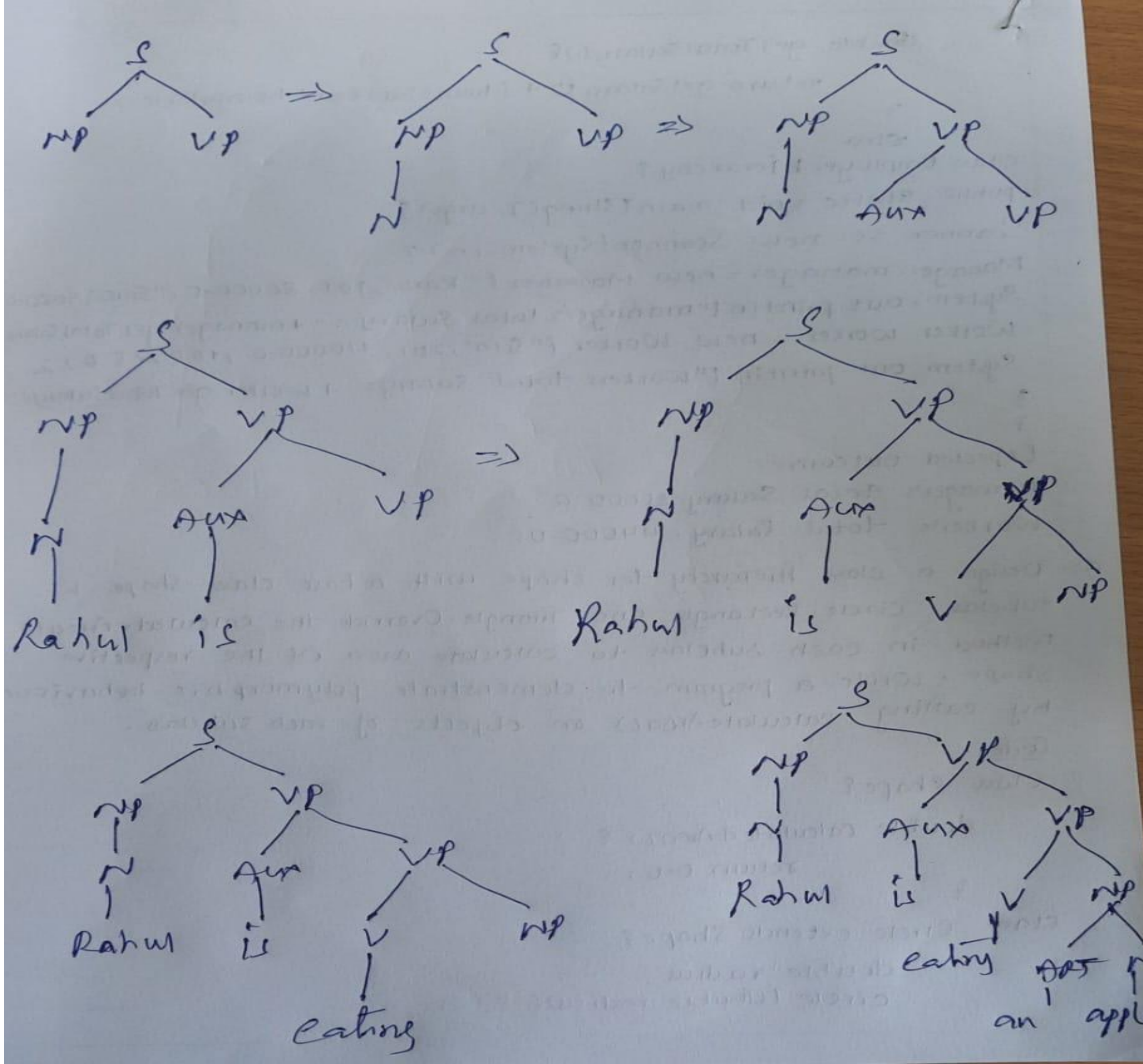
Example 2: Rahul is eating an apple.

S -> NP VP
NP-> N | ART N
VP -> AUX VP
VP -> V NP
N -> "Rahul" | "apple"
AUX-> "is"
V -> "eating"
ART-> "an"



Example 2: Rahul is eating an apple.

S -> NP VP
NP-> N | ART N
VP -> AUX VP
VP -> V NP
N -> "Rahul" | "apple"
AUX-> "is"
V -> "eating"
ART-> "an"



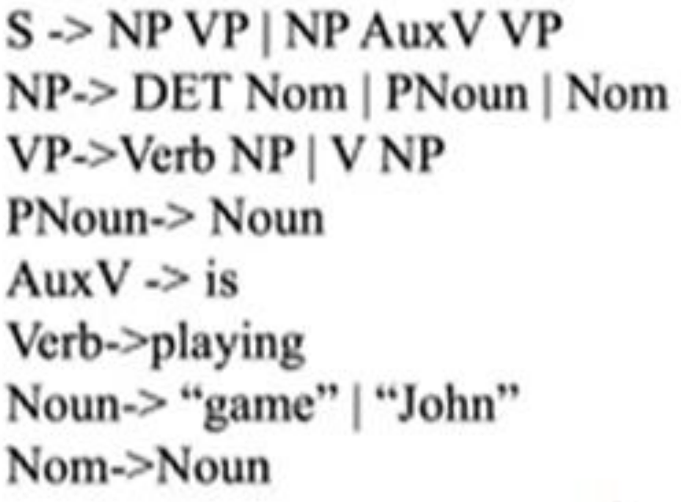
Bottom-up parsing

- Bottom-up parsing starts with the input words (tokens) and applies grammar rules in reverse to construct the parse tree, moving from the leaves to the root.
- The goal of reaching the starting symbol S is accomplished through a series of reductions; when the right-hand side of some rule matches the substring of the input string, the substring is replaced with the left-hand side of the matched production, and the process is repeated until the starting symbol is reached.

How It Works:

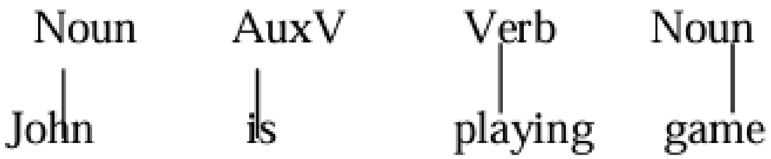
1. Start with the input sentence (sequence of words).
2. Identify phrases (subtrees) by matching grammar rules.
3. Continue merging phrases until reaching the root (start symbol of the grammar).

Sentence: John is playing a game

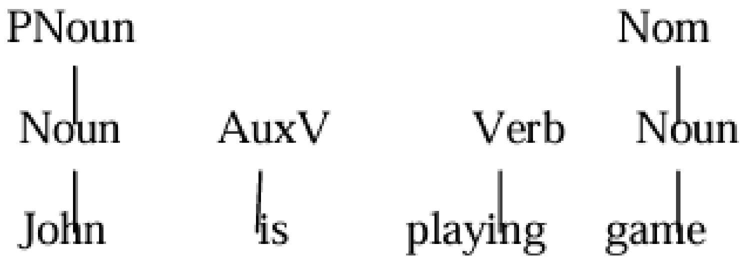


S -> NP VP | NP AuxV VP
NP-> DET Nom | PNoun | Nom
VP->Verb NP | V NP
PNoun-> Noun
AuxV -> is
Verb->playing
Noun-> “game” | “John”
Nom->Noun

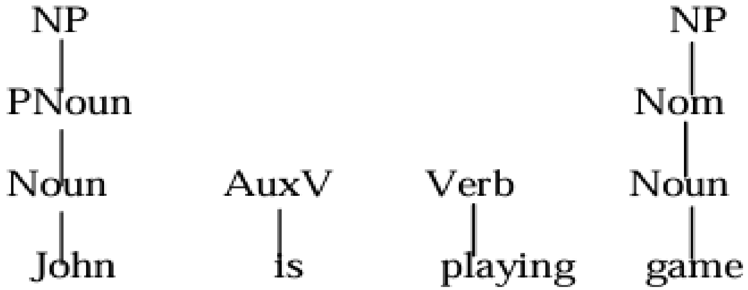
I)



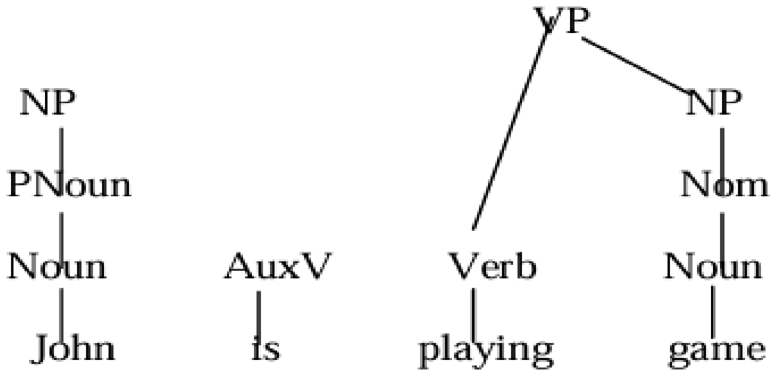
II)



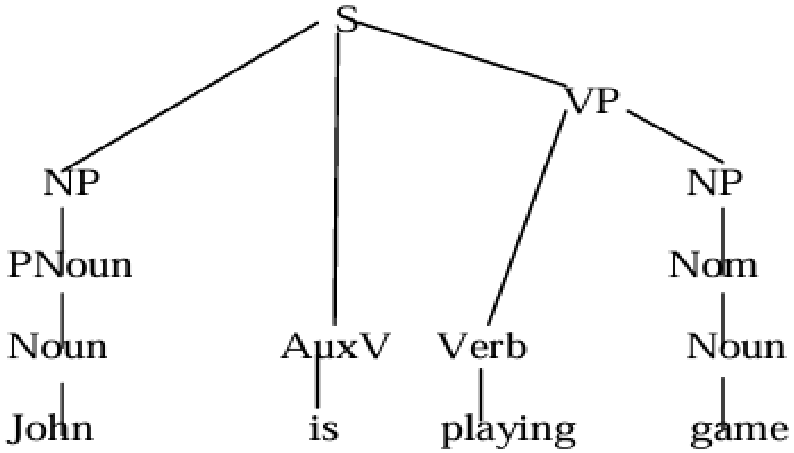
III)



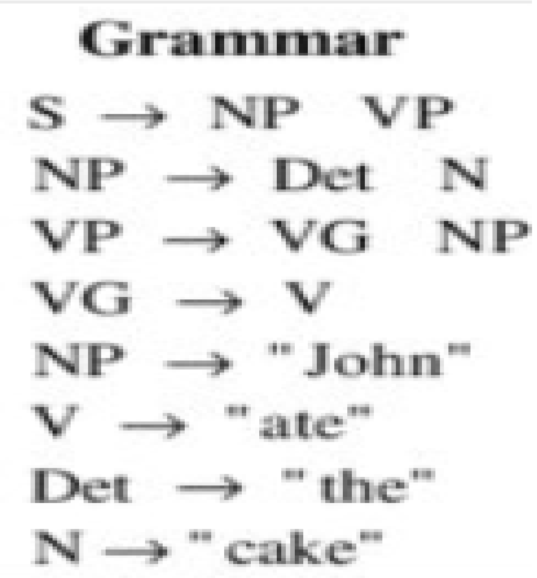
IV)



V)



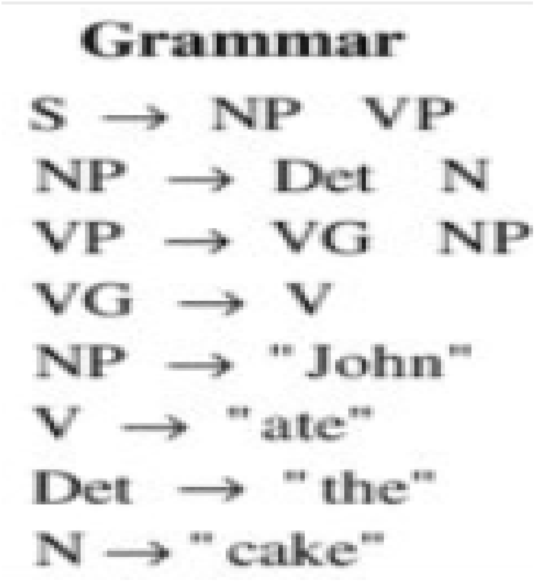
Sentence: John ate the cake



Grammar

S → NP VP
NP → Det N
VP → VG NP
VG → V
NP → " John"
V → " ate"
Det → " the"
N → " cake"

Sentence: John ate the cake



Grammar

S → NP VP

NP → Det N

VP → VG NP

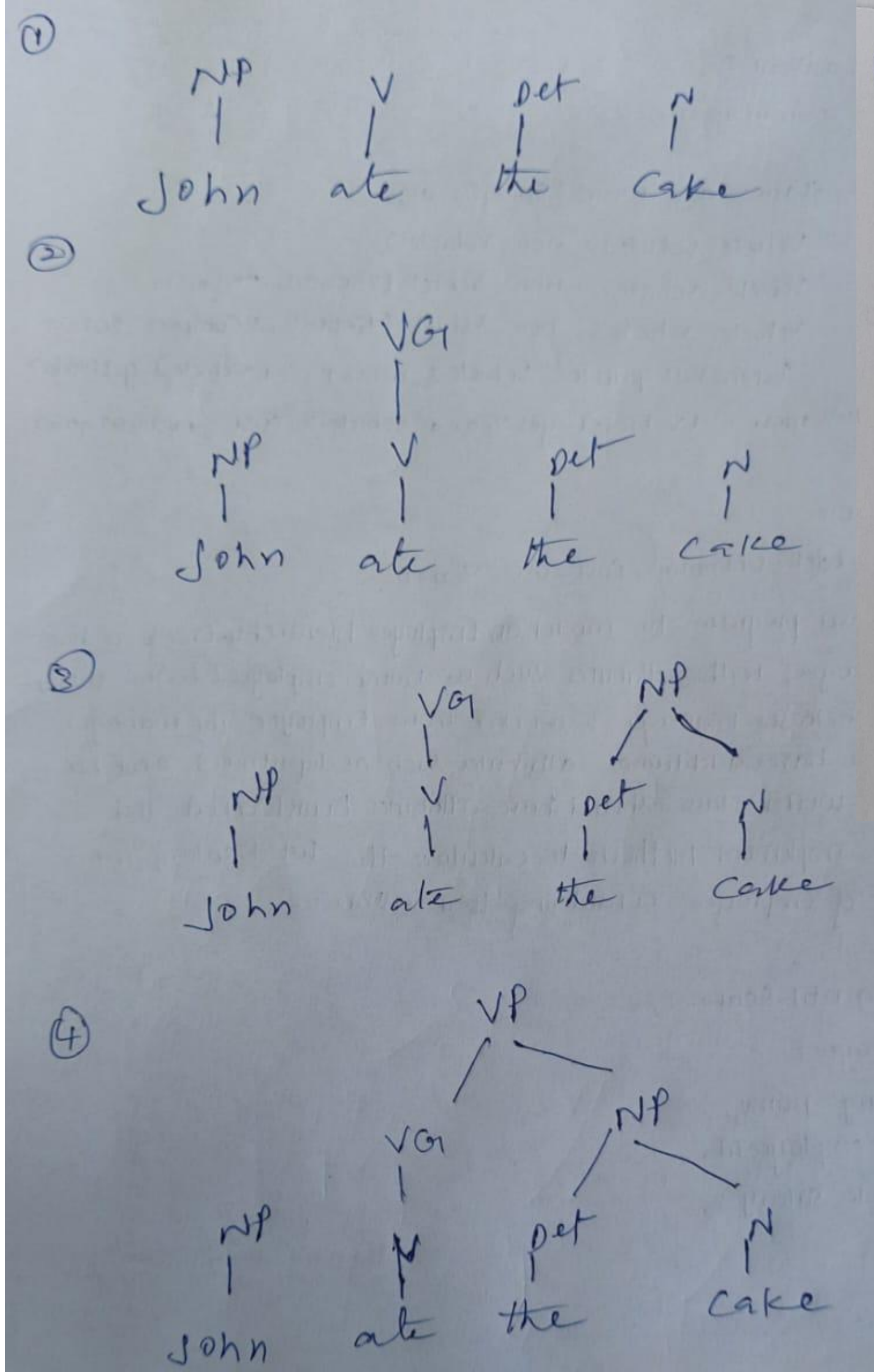
VG → V

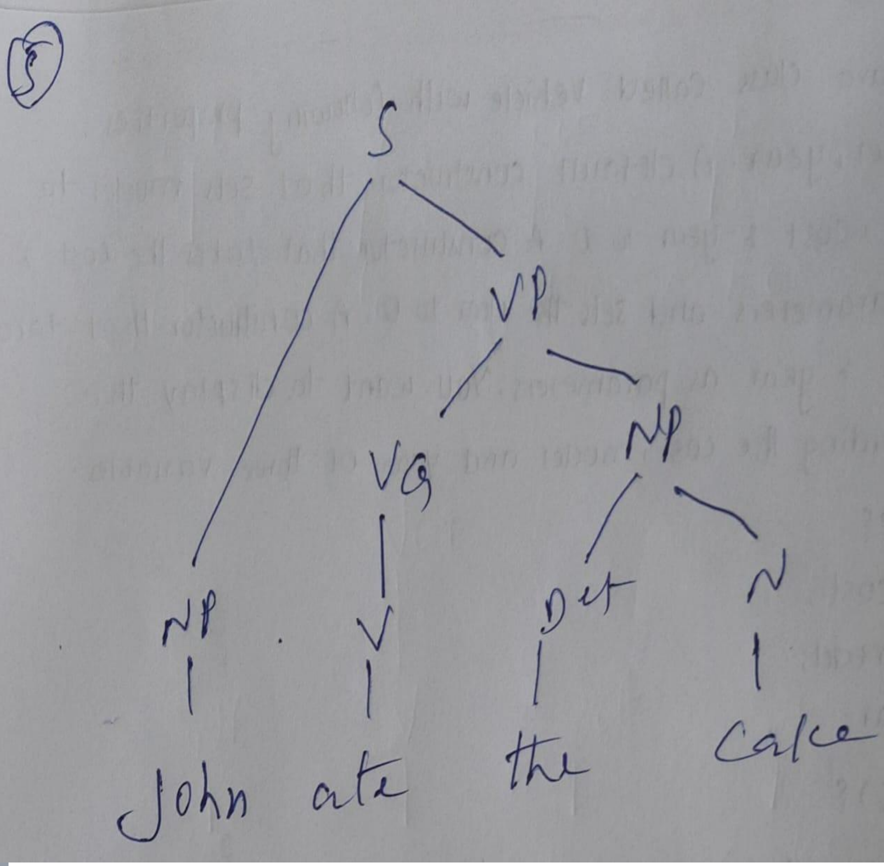
NP → "John"

V → "ate"

Det → "the"

N → "cake"





Cons of Top-Down Parsing

1. **Backtracking Overhead** – Can suffer from excessive backtracking in ambiguous grammars, making parsing inefficient.
2. **Limited Context Sensitivity** – Struggles with complex languages where parsing depends on deeper contextual information.

Cons of Bottom-Up Parsing

1. **Complex Implementation** – More difficult to implement compared to top-down approaches.
2. **Higher Memory Usage** – Stores intermediate parse states, leading to increased memory consumption.
3. **Slower for Simple Grammars** – For straightforward languages, top-down parsers (like recursive descent) may be faster.
4. **Less Readable Parse Trees** – The parse tree is built in a non-intuitive order (from leaves to root), making debugging harder.

Types of parsing/parsers

- Regex parser,
- Dependency parser,
- Constituency Parsing

Regexp Parser

- A **Regex Parser** in **NLP** is a rule-based method for chunking and parsing text using **regular expressions** applied to **Part-of-Speech (POS)-tagged** words.
- **Regexp Parser** uses regular expressions defined in the form of grammar and applied on POS-tagged string to generate a parse tree

Regexp Parser

Define the Grammar Rules

Create regex-based rules to **identify phrases** like **Noun Phrases (NP), Verb Phrases (VP), and Prepositional Phrases (PP)**

```
grammar = r"""
NP: {<DT>? <JJ>* <NN>*} # Noun Phrase (Determiner + Adjective(s) + Noun(s))
 P: {<IN>} # Preposition (e.g., in, on, at)
V: {<V.*>} # Verb (any verb form)
PP: {<P> <NP>} # Prepositional Phrase (P + NP)
 VP: {<V> <NP|PP>*} # Verb Phrase (V + NP/PP) ""“
```

Create a Regex Parser

```
reg_parser = RegexpParser(grammar)
```

Tokenize and POS Tag a Sentence

```
sentence = "The quick brown fox jumps over the lazy dog“
pos_tags = pos_tag(word_tokenize(sentence) )
```

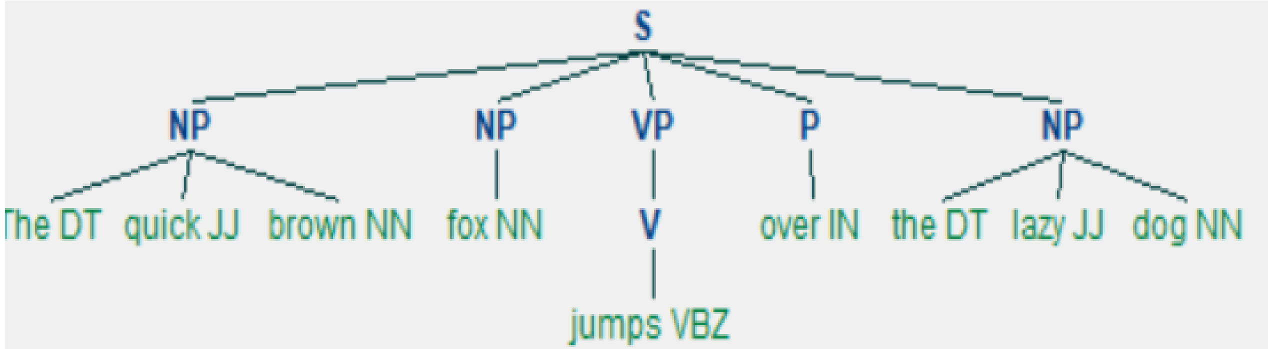
Parse the Sentence Using the Regex Parser

```
parsed_sentence = reg_parser.parse(pos_tags)
```

Regexp Parser

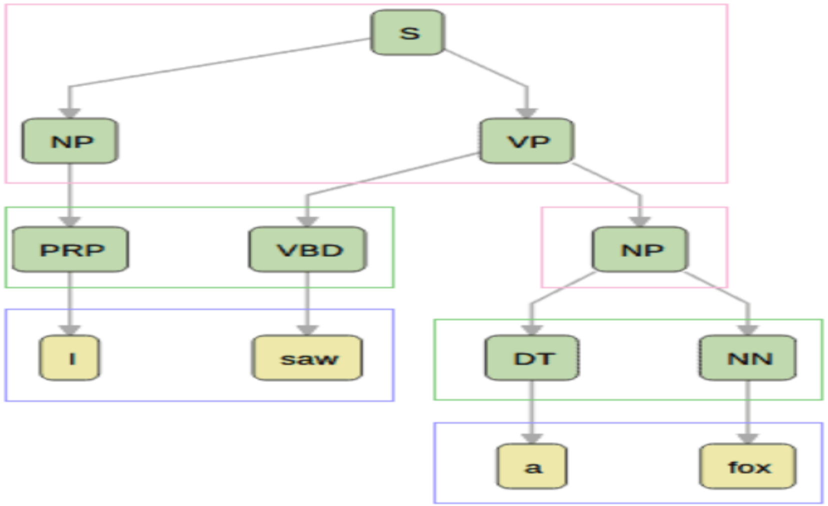
```
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
from nltk import pos_tag, word_tokenize, RegexpParser

text = "The quick brown fox jumps over the lazy dog"
tokens = word_tokenize(text)
tags = pos_tag(tokens)

reg_parser = RegexpParser("""
NP: {<DT>?<JJ>*<NN>} # To extract Noun Phrases
P: {<IN>} # To extract Prepositions
V: {<V.*>} # To extract Verbs
PP: {<IN><NP>} # To extract Prepositional Phrases
VP: {<V.*><NP|PP>*} # To extract Verb Phrases
""")
result = reg_parser .parse(tags)
print('Parse Tree:', result)
result.draw()
```

```
Parse Tree: (S
  (NP The/DT quick/JJ brown/NN)
  (NP fox/NN)
  (VP (V jumps/VBZ))
  (P over/IN)
  (NP the/DT lazy/JJ dog/NN))
```



Constituency Parsing

- Constituency Parsing in NLP is a syntactic analysis technique that breaks down a sentence into its constituent components (phrases) based on a context-free grammar (CFG).
- Constituency parsing represents the syntactic structure of a sentence as a **parse tree** according to grammar rules.
- The process involves identifying noun phrases, verb phrases, and other constituents, and then determining the relationships between them.
- It helps in understanding the grammatical structure of sentences, which is crucial for various NLP tasks such as text summarization, machine translation, question answering, and text classification.



Constituency Parsing

Steps in Constituency Parsing:

1.Defining a Context-Free Grammar (CFG)

2. Constructing a Parse Tree

- A parse tree can be constructed from the CFG using either a Top-Down or Bottom-Up parsing approach.

Example:

CFG:

S → NP VP

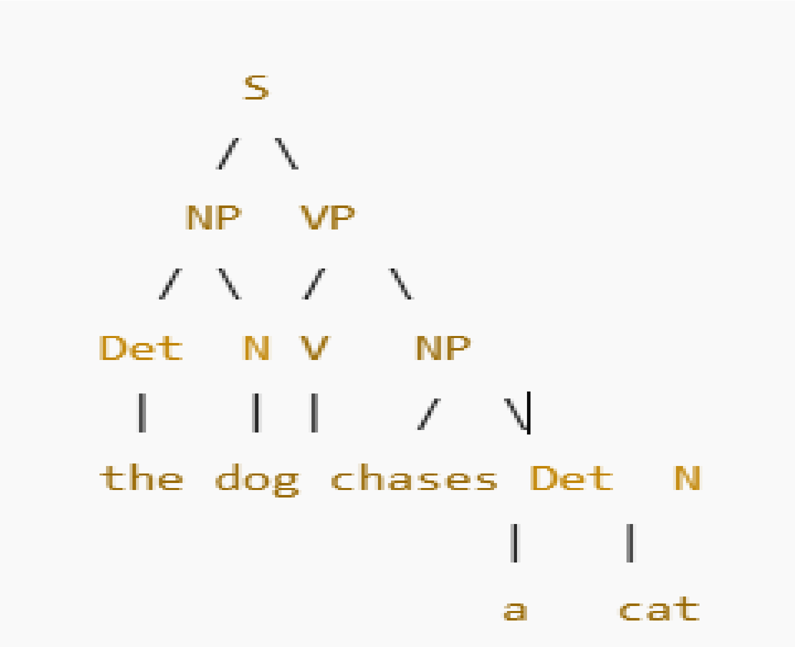
NP → Det N

VP → V NP

Det → "the" | "a"

N → "dog" | "cat"

V → "chases" | "sees“



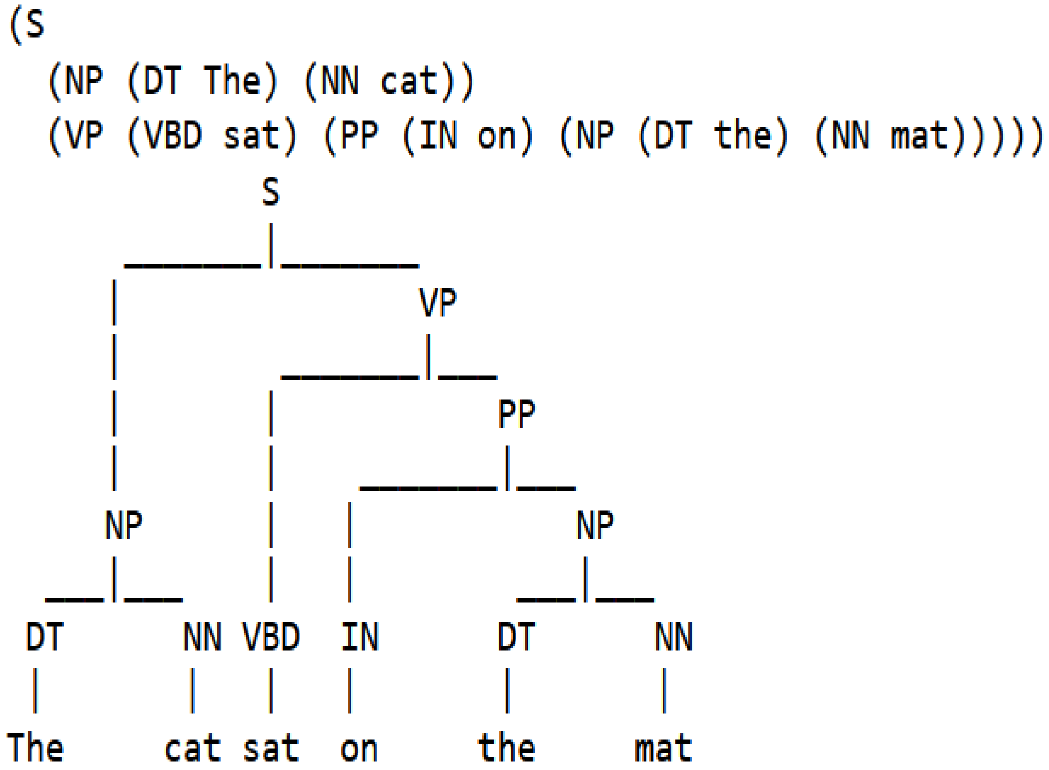
```
     S
    / \
   NP  VP
  / \  /  \
Det  N V   NP
 |   | |   /  \|
the dog chases Det  N
              |   |
              a   cat
```

Constituency Parsing

```python
import nltk
from nltk import CFG
# Define a Context-Free Grammar (CFG)
grammar = CFG.fromstring("""
    S -> NP VP
    NP -> DT NN | DT NN
    VP -> VBD PP
    PP -> IN NP
    DT -> 'The' | 'the'
    NN -> 'cat' | 'mat'
    VBD -> 'sat'
    IN -> 'on'
""")

# Create a ChartParser(Botton-up parser)
parser = nltk.ChartParser(grammar)

sentence = "The cat sat on the mat".split()
# Generate parse tree
for tree in parser.parse(sentence):
        print(tree)
        tree.pretty_print()
```

Note: The **ChartParser** is a **bottom-up dynamic programming** parsing algorithm that efficiently constructs a parse tree by storing intermediate results in a **chart table**.

Output:



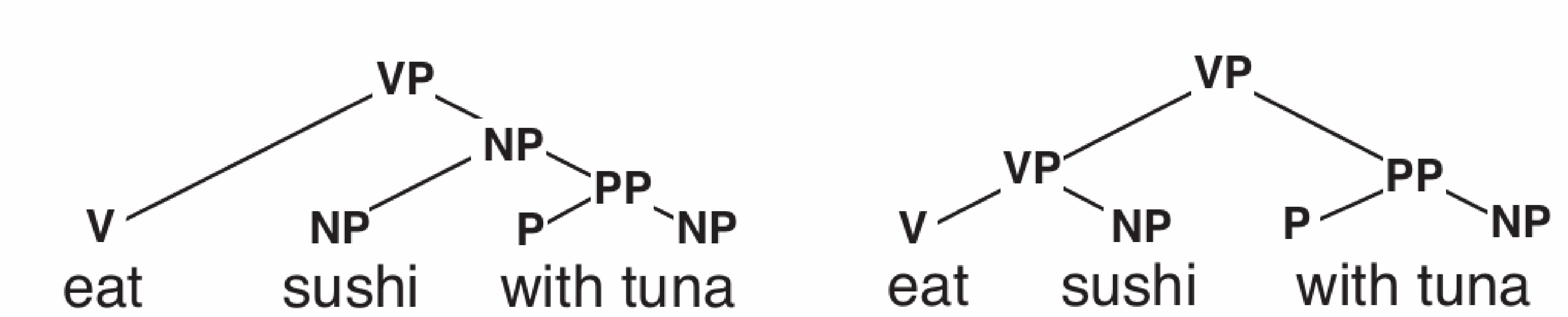
```
(S
  (NP (DT The) (NN cat))
  (VP (VBD sat) (PP (IN on) (NP (DT the) (NN mat)))))
             S
      _______|_______
     |               VP
     |        _______|___
     |       |           PP
     |       |    _______|___
     NP      |   |           NP
  ___|___    |   |        ___|___
 DT      NN VBD  IN      DT      NN
 |       |   |   |       |       |
The     cat sat  on     the     mat
```

Application of Constituency Parsing

1. Machine Translation
2. Information Retrieval
3. Question Answering
4. Text Summarization
5. Sentiment analysis
6. Grammar Checking

Ambiguity in parsing

- **Ambiguity in parsing** occurs when a sentence can have multiple valid syntactic structures (parse trees).
- we can have more than one parse tree for the sentence as per the CFG due to ambiguity, making it difficult for a parser to determine the correct interpretation.



Ambiguity produces two different parse trees

What’s the most likely parse τ for sentence S ?

Probabilistic Context-Free Grammar (PCFG)

- A **Probabilistic Context-Free Grammar (PCFG)** is an extension of **Context-Free Grammar (CFG)** that assigns probabilities to production rules.
- This helps in resolving ambiguity in parsing by selecting the most probable parse tree for a given sentence.
- In PCFG, each rule in a CFG is assigned a probability, which represents the likelihood of that rule being used.

How to calculate the probability of a parse tree in PCFG

- Given a parse tree t, with the production rules $\alpha_1 \rightarrow \beta_1$, $\alpha_2 \rightarrow \beta_2$, … , $\alpha_n \rightarrow \beta_n$ from R (ie., $\alpha_i \rightarrow \beta_i \in R$), we can find the probability of tree t using PCFG as follows;

$$P(t) = \prod_{i=1}^{n} P(\alpha_i \rightarrow \beta_i)$$

As per the equation, the probability *P(t)* of parse tree is the product of probabilities of production rules in the tree t.

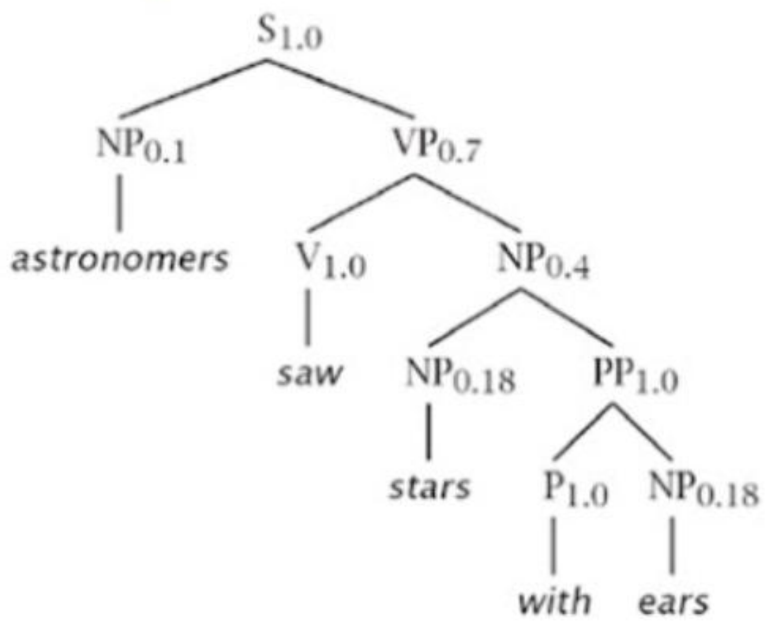
Probabilistic Context-Free Grammar (PCFG)

S→NP VP	1.0
VP→V NP	0.7
VP→ VP PP	0.3
NP→NP PP	0.4
PP→P NP	1.0
NP→astronomers	0.1
NP→stars	0.18
V→saw	1.0
P→with	1.0
NP→ears	0.18

Sentence:

Astronomers saw stars with ears

tree t_1



$$P(t) = \prod_{i=1}^{n} P(\alpha_i \rightarrow \beta_i)$$

$$= P(\mathrm{S} \rightarrow \mathrm{NP\ VP}) * P(\mathrm{NP} \rightarrow \mathrm{astronomers}) * P(\mathrm{VP} \rightarrow \mathrm{V\ NP})$$

$$* P(\mathrm{V} \rightarrow \mathrm{saw}) * P(\mathrm{NP} \rightarrow \mathrm{NP\ PP}) * P(\mathrm{NP} \rightarrow \mathrm{stars})$$

$$* P(\mathrm{PP} \rightarrow \mathrm{P\ NP}) * P(\mathrm{P} \rightarrow \mathrm{with}) * P(\mathrm{NP} \rightarrow \mathrm{ears})$$

$$= 1.0 * 0.1 * 0.7 * 1.0 * 0.4 * 0.18 * 1.0 * 1.0 * 0.18$$

$$= 0.0009072$$

The probability of the parse tree t is 0.0009072.

Dealing Ambiguity with PCFG

1. Define the PCFG Rules
   - Assign probabilities to each rule based on their likelihood.

2. Construct the Parse Tree

3. Compute the Probability of a Parse Tree

- Multiply the probabilities of all applied production rules in a parse tree

4. Choose the Most Likely Parse Tree(Viterbi algorithm)

- If multiple parse trees exist, select the one with the highest probability:

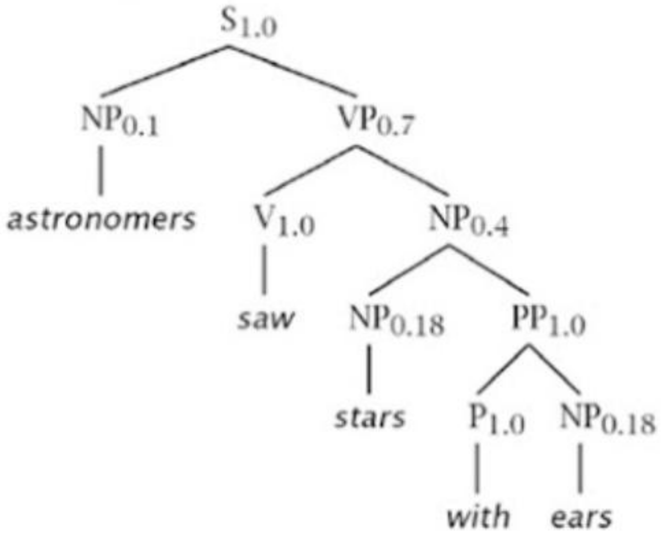
$$T^* = \arg\max_{T} P(T)$$

Problem-1

S→NP VP	1.0
VP→V NP	0.7
VP→ VP PP	0.3
NP→NP PP	0.4
PP→P NP	1.0
NP→astronomers	0.1
NP→stars	0.18
V→saw	1.0
P→with	1.0
NP→ears	0.18

Sentence:
Astronomers saw stars with ears

tree t_1



Probability of tree t_1

$$P(t_1) = \prod_{i=1}^{n} P(\alpha_i \rightarrow \beta_i)$$

$$= P(S \rightarrow NP\ VP) * P(NP \rightarrow astronomers) * P(VP \rightarrow V\ NP)$$

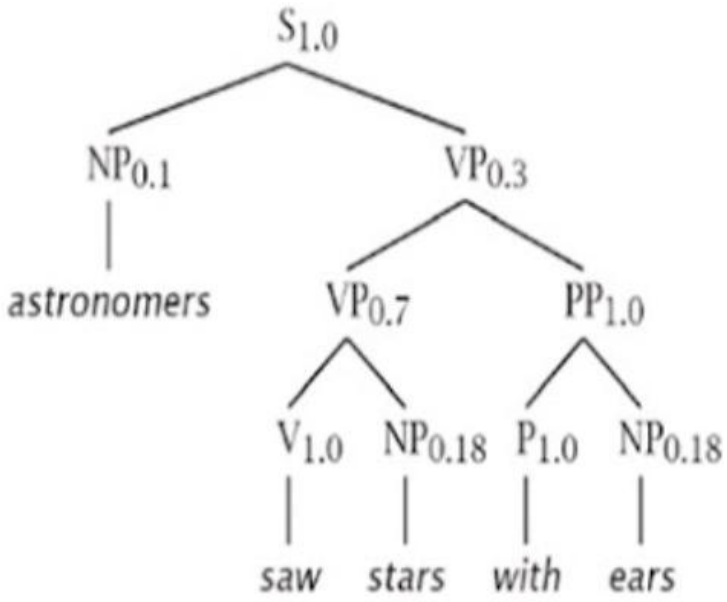
$$* P(V \rightarrow saw) * P(NP \rightarrow NP\ PP) * P(NP \rightarrow stars)$$

$$* P(PP \rightarrow P\ NP) * P(P \rightarrow with) * P(NP \rightarrow ears)$$

$$= 1.0 * 0.1 * 0.7 * 1.0 * 0.4 * 0.18 * 1.0 * 1.0 * 0.18$$

$$= 0.0009072$$

tree t_2



Probability of tree t_2

$$P(t_2) = 1.0 * 0.1 * 0.3 * 0.7 * 1.0 * 0.18 * 1.0 * 1.0 * 0.18$$

$$= 0.0006804$$

Which is the most probable tree?

The probability of the parse tree t_1 is greater than the probability of parse tree t_2. Hence, t_1 is the more probable of the two parses.

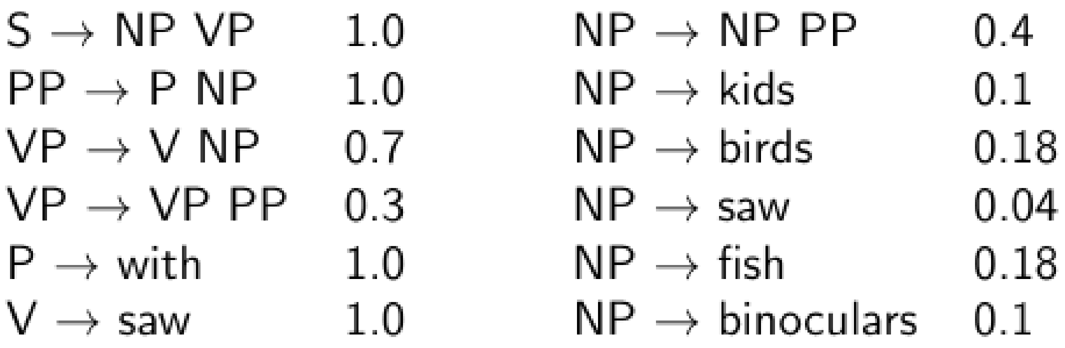
Probability of a sentence:

- Probability of a sentence is the sum of probabilities of all parse trees that can be derived from the sentence under PCFG;

Probability of the sentence “astronomers saw the stars with ears”;

$$\sum_{i=1}^{n} P(t_i) = P(t_1) + P(t_2) = 0.0009072 + 0.0006804 = 0.001588$$

Problem-2



S → NP VP	1.0	NP → NP PP	0.4
PP → P NP	1.0	NP → kids	0.1
VP → V NP	0.7	NP → birds	0.18
VP → VP PP	0.3	NP → saw	0.04
P → with	1.0	NP → fish	0.18
V → saw	1.0	NP → binoculars	0.1

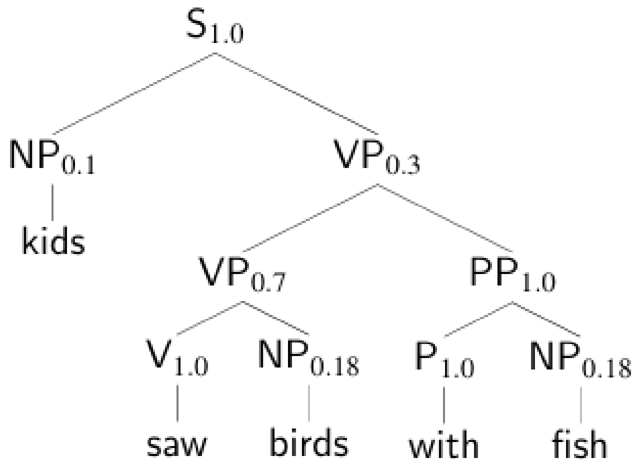
Sentence: kids saw birds with fish

Problem-2

S → NP VP	1.0	NP → NP PP	0.4
PP → P NP	1.0	NP → kids	0.1
VP → V NP	0.7	NP → birds	0.18
VP → VP PP	0.3	NP → saw	0.04
P → with	1.0	NP → fish	0.18
V → saw	1.0	NP → binoculars	0.1

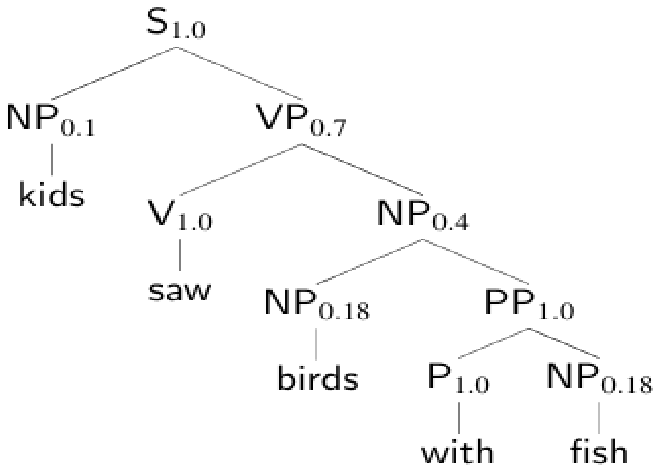
Sentence: kids saw birds with fish

Probability of parse 2



- $P(t_2) = 1.0 \cdot 0.1 \cdot 0.3 \cdot 0.7 \cdot 1.0 \cdot 0.18 \cdot 1.0 \cdot 1.0 \cdot 0.18 = 0.0006804$
- which is less than $P(t_1) = 0.0009072$, so t_1 is preferred. Yay!

Probability of parse 1



- $P(t_1) = 1.0 \cdot 0.1 \cdot 0.7 \cdot 1.0 \cdot 0.4 \cdot 0.18 \cdot 1.0 \cdot 1.0 \cdot 0.18 = 0.0009072$

The probability of a sentence

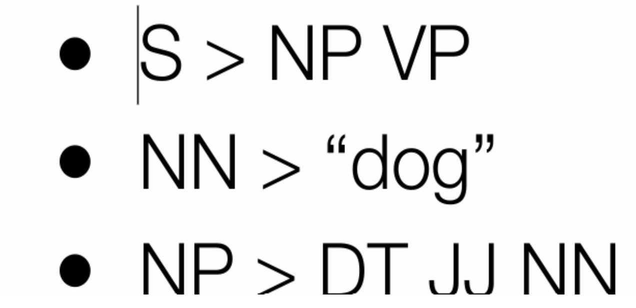
$$P(\text{kids saw birds with fish}) = 0.0006804 + 0.0009072.$$

Chomsky Normal Form (CNF)

- A CFG in Chomsky Normal Form (CNF) allows only two kinds of right-hand sides and no empty productions:
  - – Two nonterminals: VP → ADV VP
  - -One terminal: VP → eat

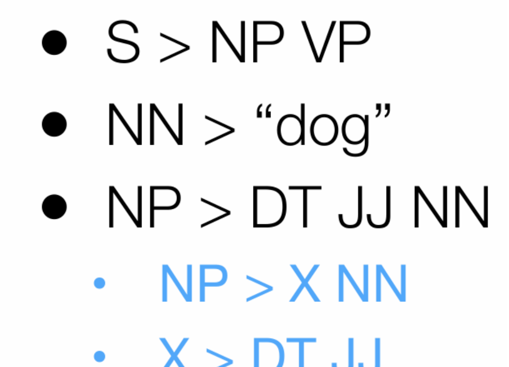
converting the simple CFG:

CFG:

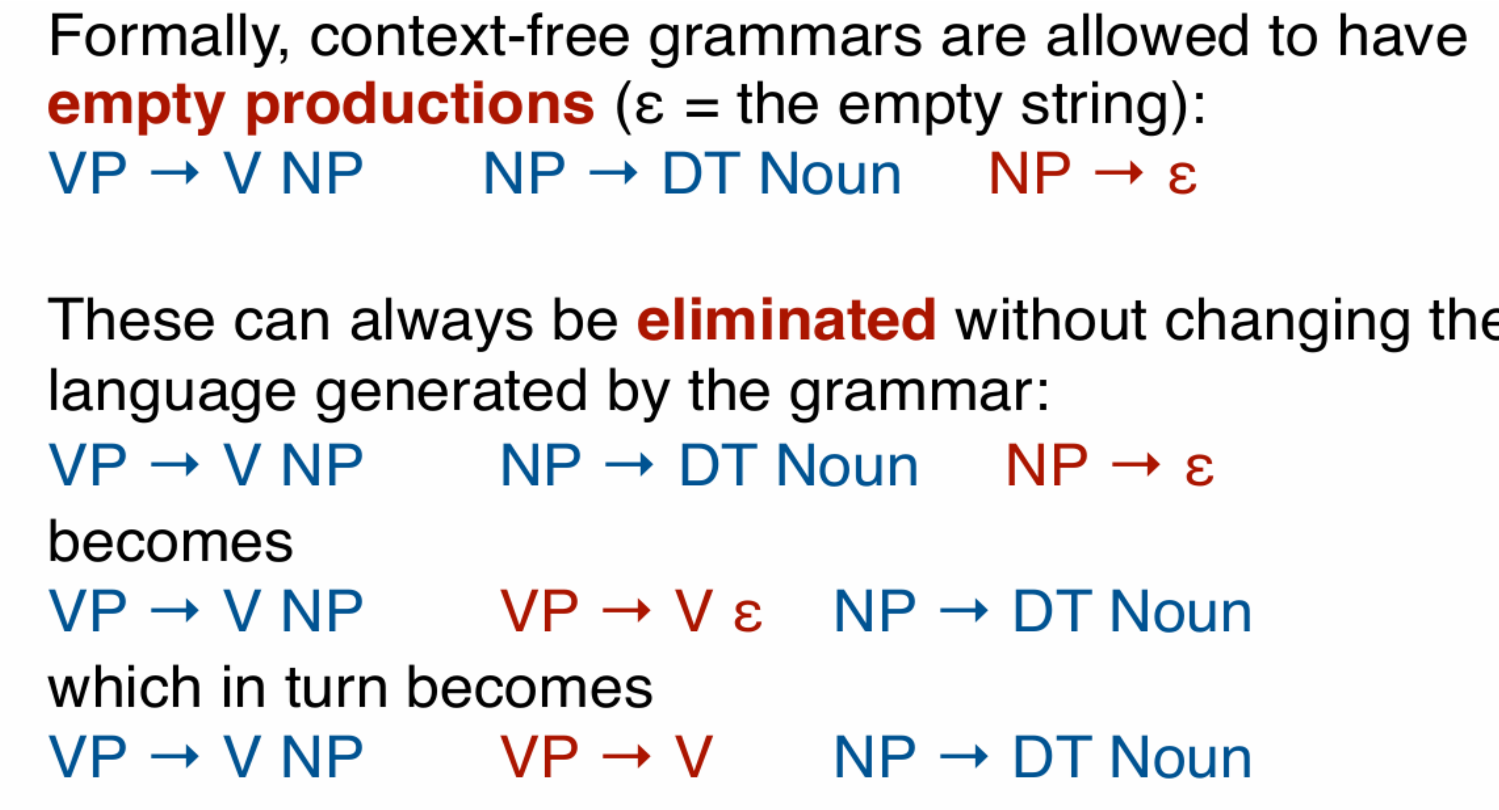


- S > NP VP
- NN > “dog”
- NP > DT JJ NN

CNF



- S > NP VP
- NN > “dog”
- NP > DT JJ NN
  - NP > X NN
  - X > DT JJ



Formally, context-free grammars are allowed to have **empty productions** (ε = the empty string):

VP → V NP NP → DT Noun NP → ε

These can always be **eliminated** without changing the language generated by the grammar:

VP → V NP NP → DT Noun NP → ε

becomes

VP → V NP VP → V ε NP → DT Noun

which in turn becomes

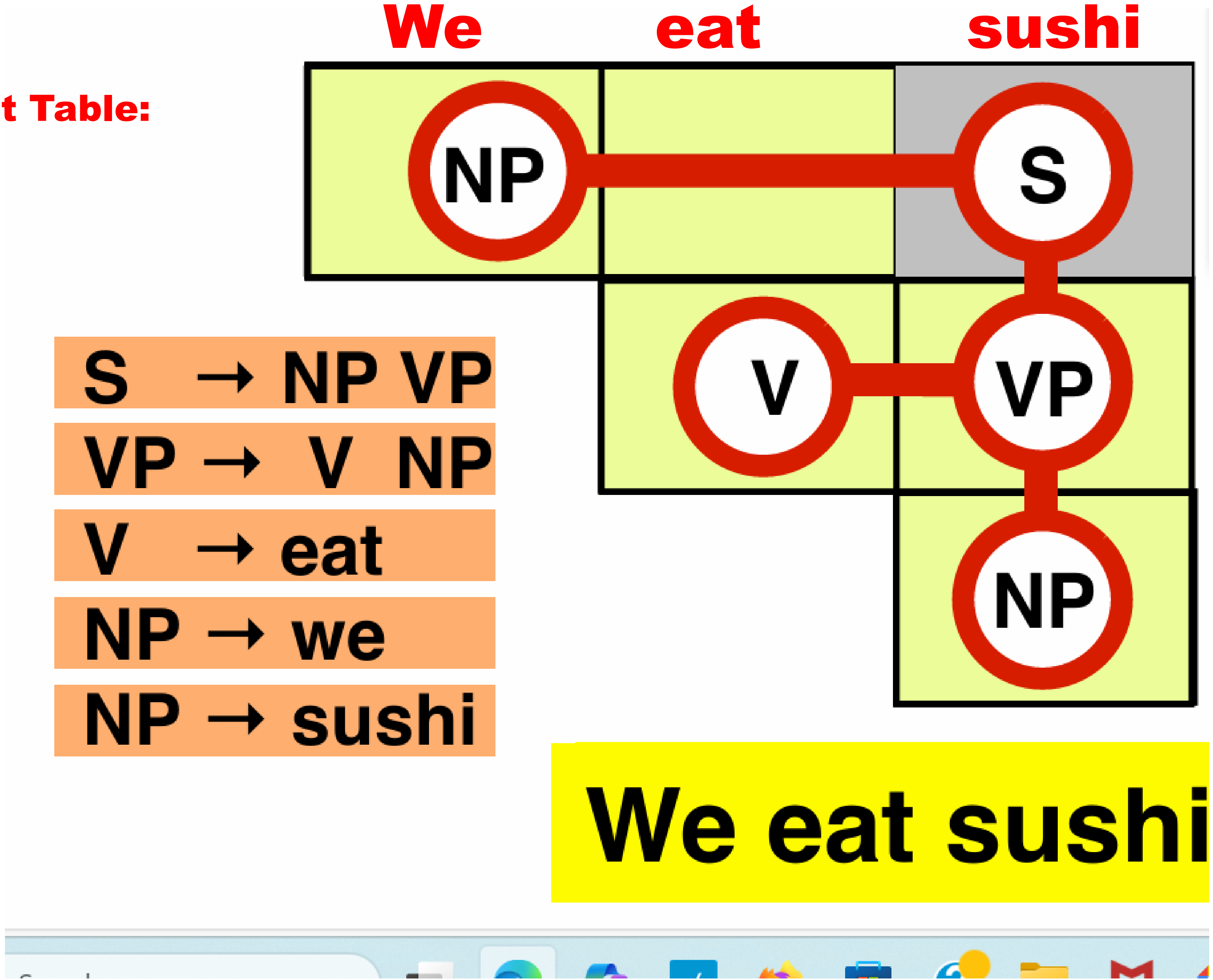
VP → V NP VP → V NP → DT Noun

CKY parsing algorithm for Constituency Parsing

- The **Cocke-Kasami-Younger (CKY) algorithm** is a **chart parsing algorithm** used to construct **parse trees** for syntactic analysis.
- It assumes grammar in Chomsky Normal Form.
- Bottom-up parsing:
  - start with the words
- Dynamic programming:
  - It saves the intermediate results in a table/chart
  - re-use these results in finding larger constituents

CKY parsing algorithm

Chart Table:



S → NP VP

VP → V NP

V → eat

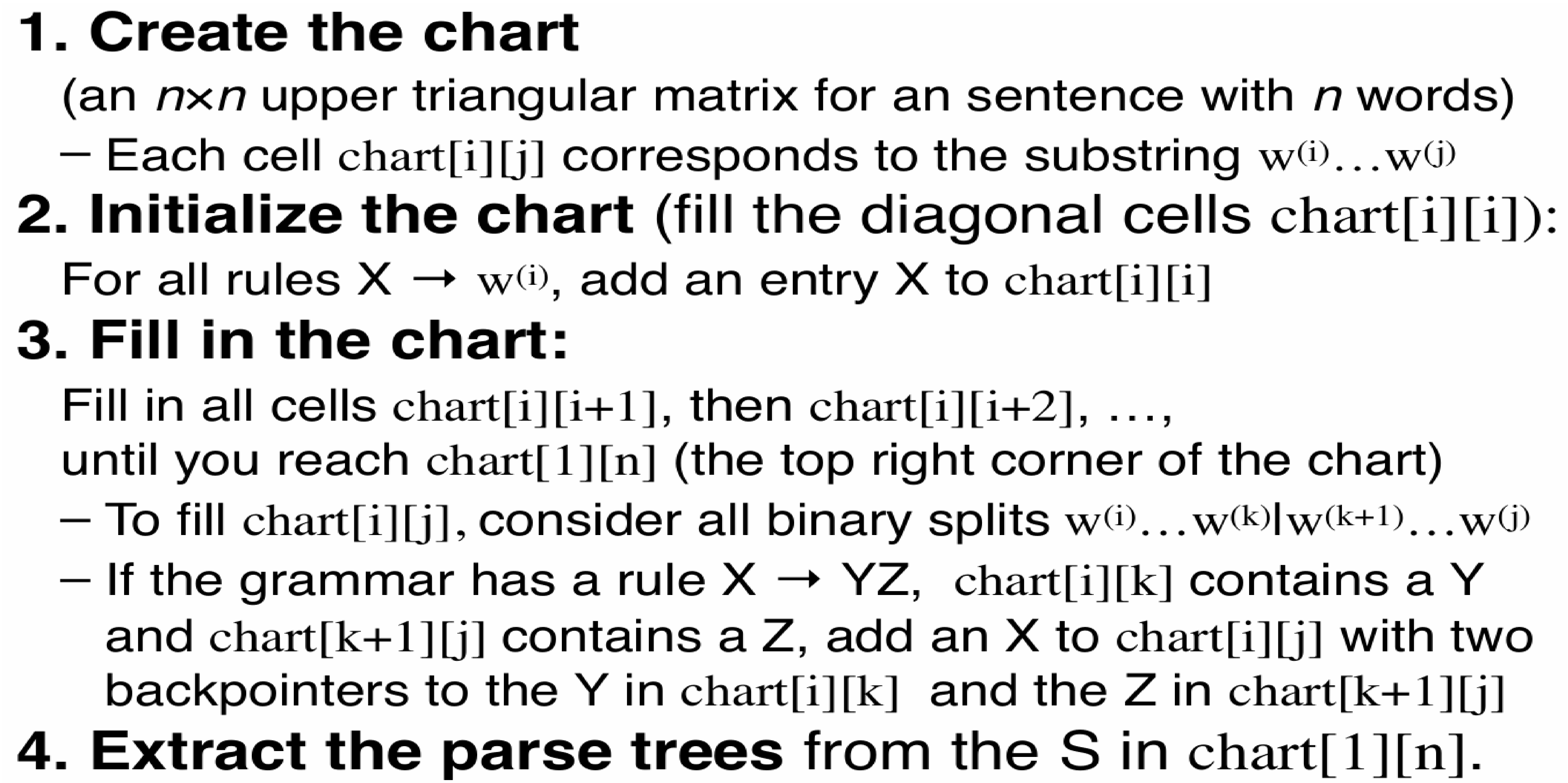
NP → we

NP → sushi

We eat sushi

To recover the parse tree, each entry needs **pairs** of backpointers.

CKY algorithm



1. Create the chart

(an *n*×*n* upper triangular matrix for an sentence with *n* words)

– Each cell chart[i][j] corresponds to the substring $w^{(i)} \ldots w^{(j)}$

2. Initialize the chart (fill the diagonal cells chart[i][i]):

For all rules X → $w^{(i)}$, add an entry X to chart[i][i]

3. Fill in the chart:

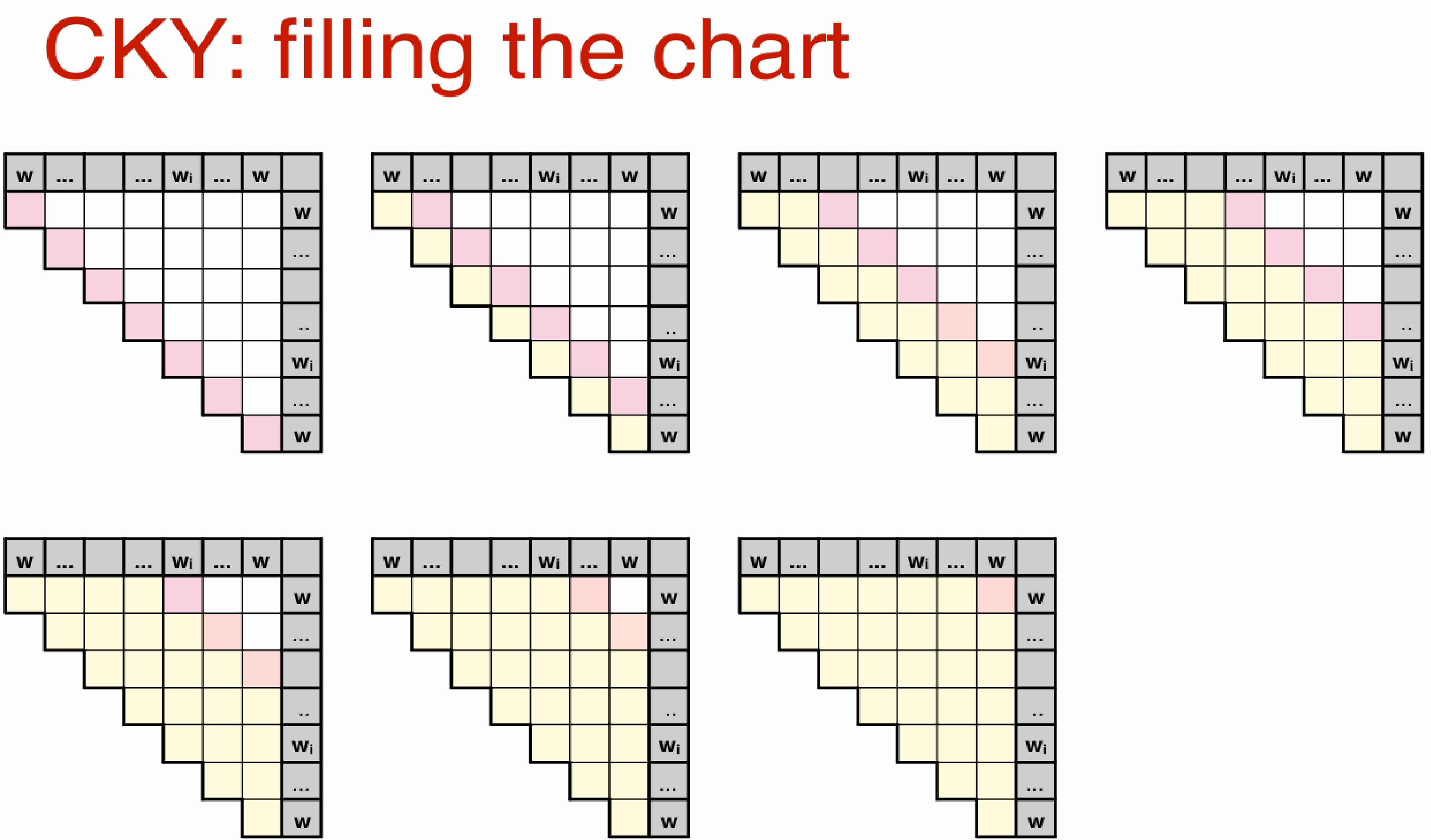
Fill in all cells chart[i][i+1], then chart[i][i+2], …, until you reach chart[1][n] (the top right corner of the chart)

– To fill chart[i][j], consider all binary splits $w^{(i)} \ldots w^{(k)} | w^{(k+1)} \ldots w^{(j)}$

– If the grammar has a rule X → YZ, chart[i][k] contains a Y and chart[k+1][j] contains a Z, add an X to chart[i][j] with two backpointers to the Y in chart[i][k] and the Z in chart[k+1][j]

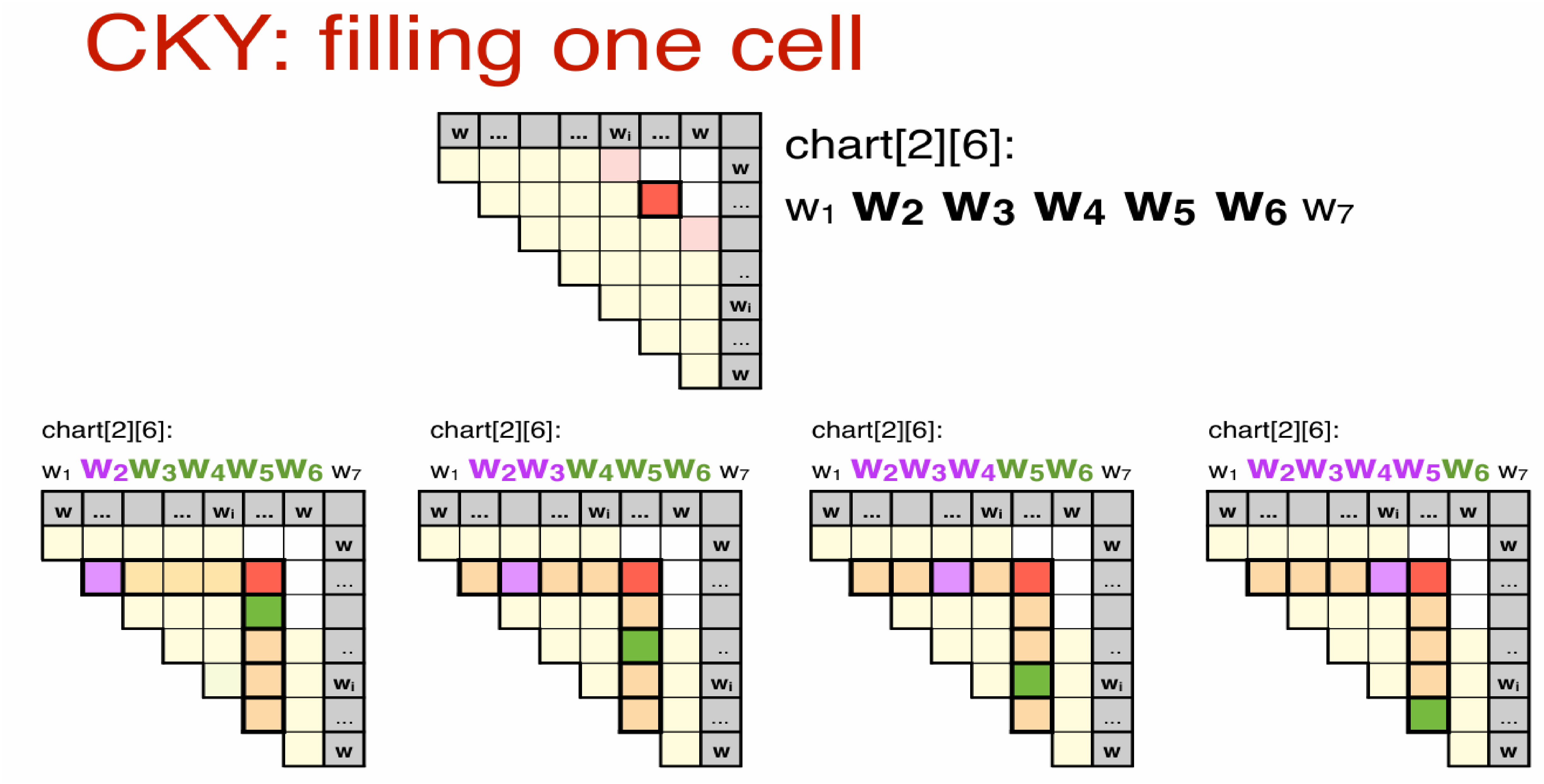
4. Extract the parse trees from the S in chart[1][n].

CKY algorithm



CKY: filling the chart

CKY algorithm



CKY: filling one cell

chart[2][6]:

w_1 $\mathbf{W_2}$ $\mathbf{W_3}$ $\mathbf{W_4}$ $\mathbf{W_5}$ $\mathbf{W_6}$ w_7

chart[2][6]:

w_1 $W_2W_3W_4W_5W_6$ w_7

chart[2][6]:

w_1 $W_2W_3W_4W_5W_6$ w_7

chart[2][6]:

w_1 $W_2W_3W_4W_5W_6$ w_7

chart[2][6]:

w_1 $W_2W_3W_4W_5W_6$ w_7

CKY algorithm

S → NP VP

VP → V NP

VP → VP PP

V → eat

NP → NP PP

NP → we

NP → sushi

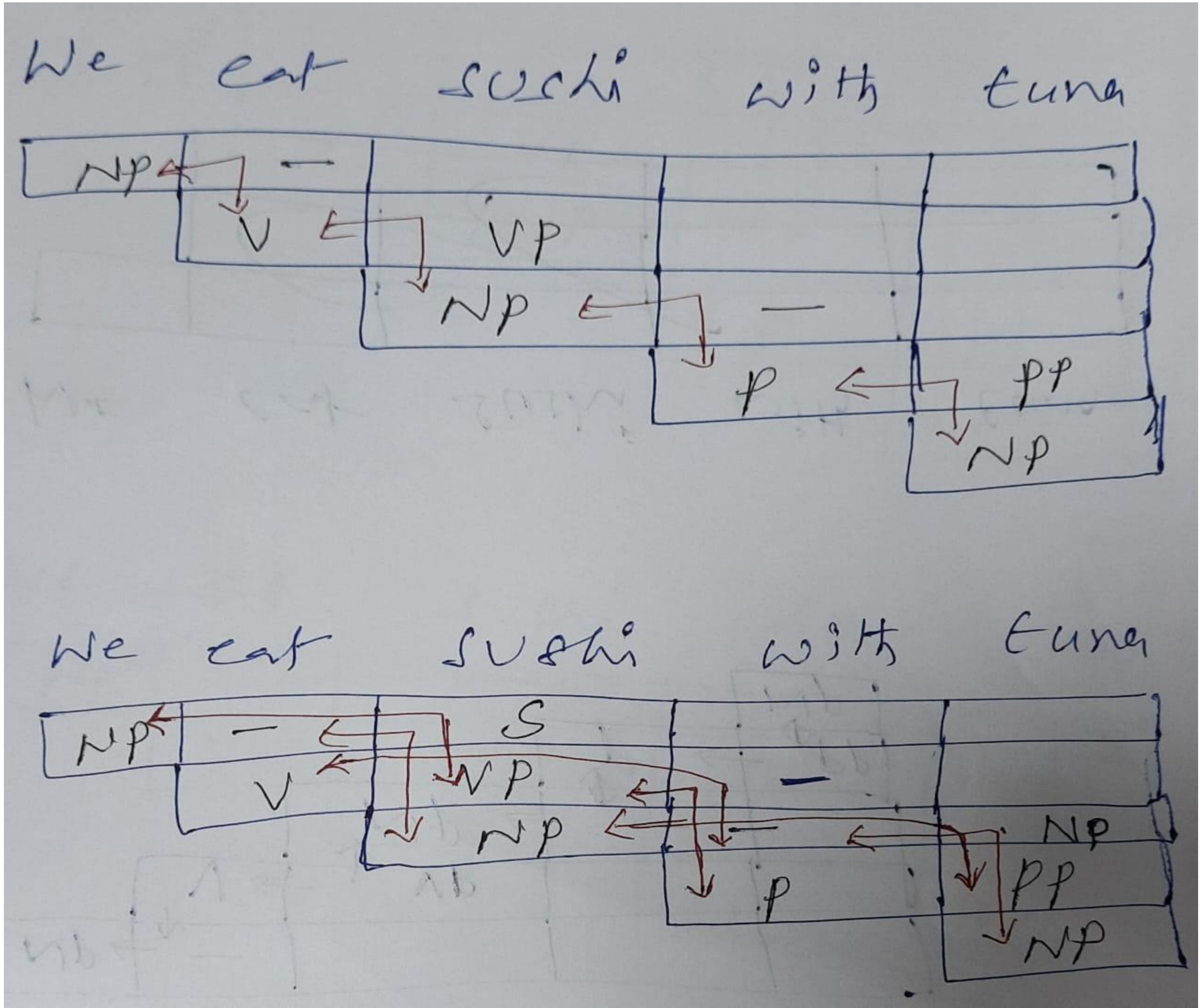
NP → tuna

PP → P NP

P → with

Sentence:

We eat sushi with tuna



We eat sushi with tuna

CKY algorithm

S → NP VP

VP → V NP

VP → VP PP

V → eat

NP → NP PP

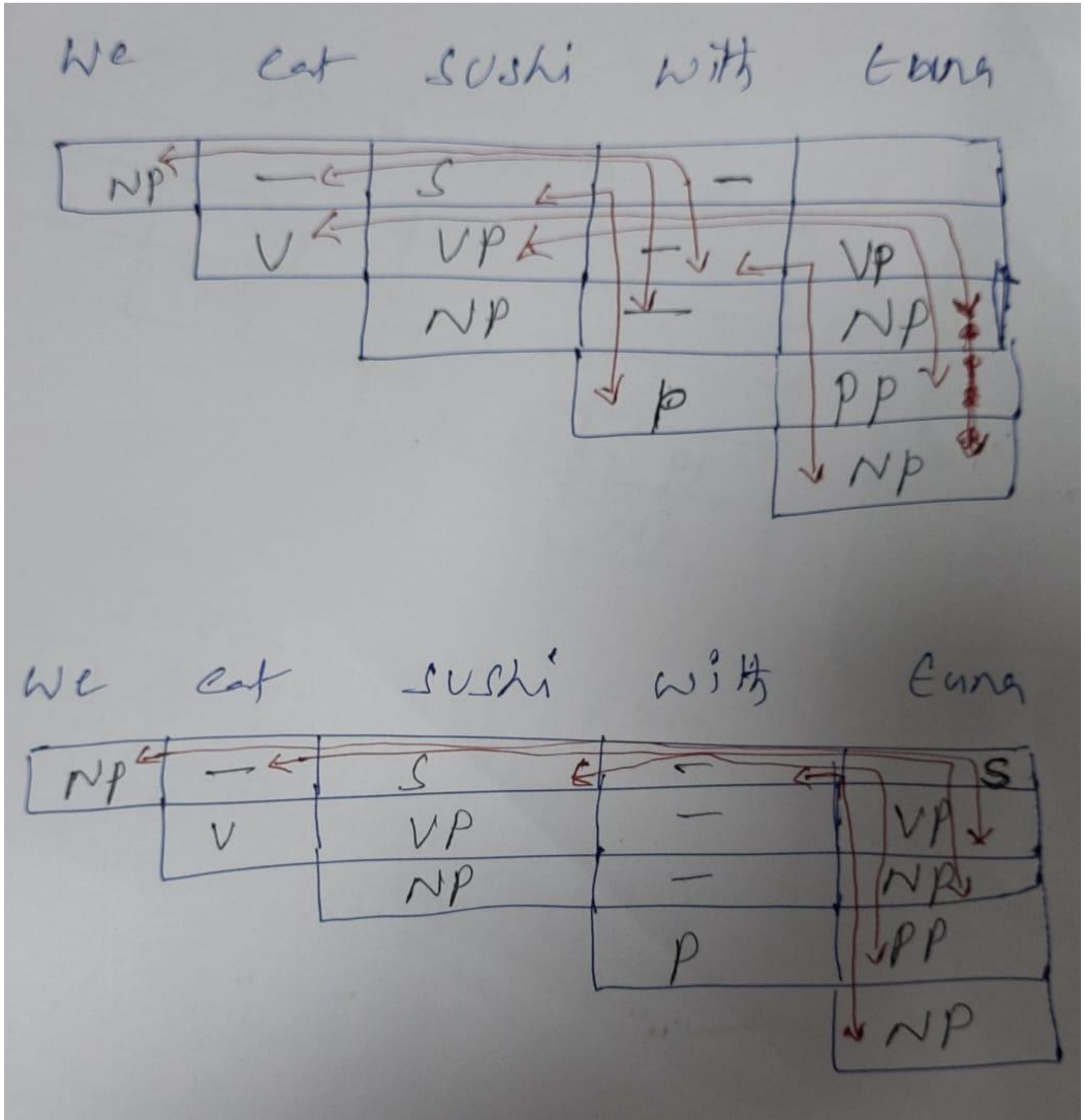
NP → we

NP → sushi

NP → tuna

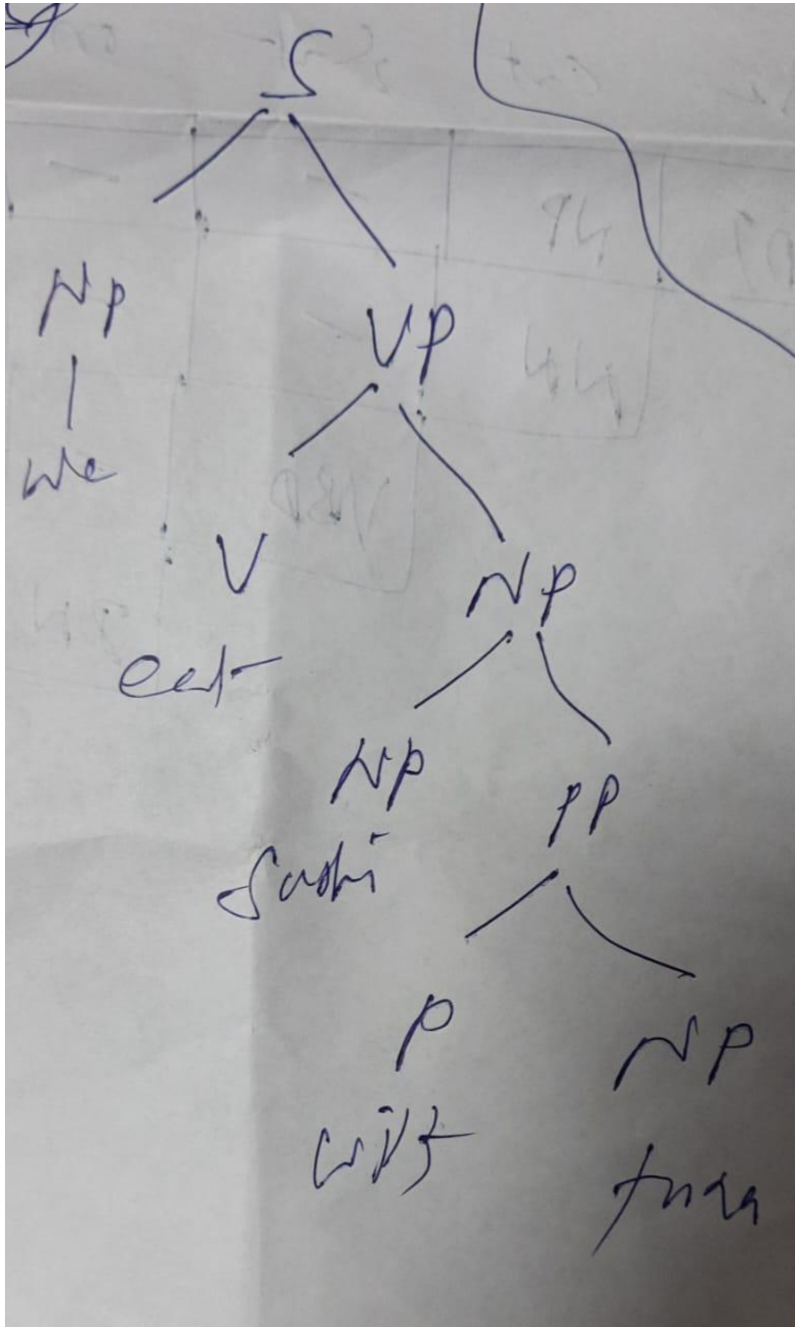
PP → P NP

P → with



We	eat	sushi	with	tuna
NP	—	S	—	
	V	VP	—	VP
		NP	—	NP
			P	PP
				NP

We	eat	sushi	with	tuna
NP	—	S	—	S
	V	VP	—	VP
		NP	—	NP
			P	PP
				NP



We eat sushi with tuna

CKY algorithm

S → NP VP

VP → V NP

VP → VP PP

V → eat

NP → NP PP

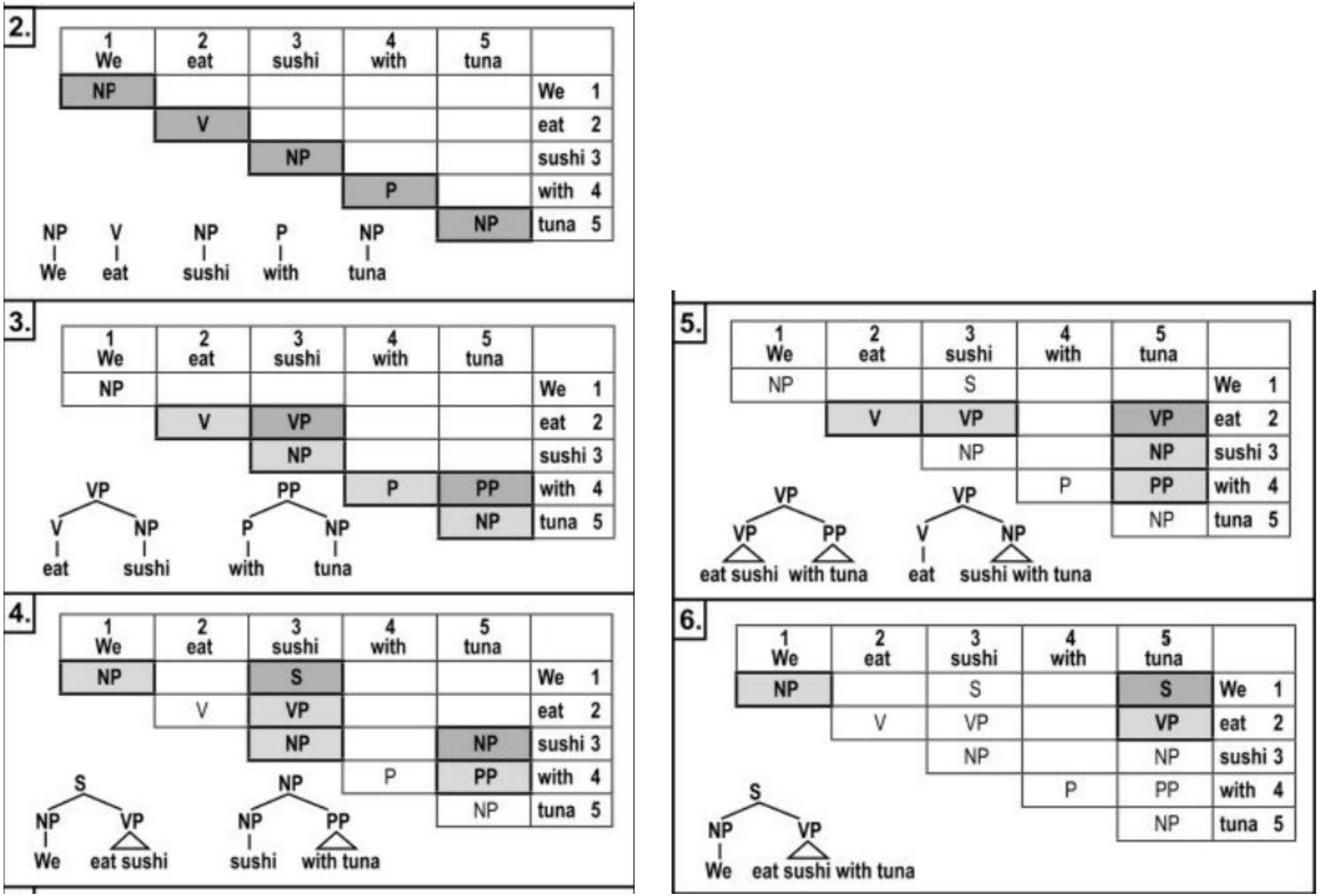
NP → we

NP → sushi

NP → tuna

PP → P NP

P → with



2.

1 We	2 eat	3 sushi	4 with	5 tuna	
NP					We 1
	V				eat 2
		NP			sushi 3
			P		with 4
				NP	tuna 5

3.

1 We	2 eat	3 sushi	4 with	5 tuna	
NP					We 1
	V	VP			eat 2
		NP			sushi 3
			P	PP	with 4
				NP	tuna 5

4.

1 We	2 eat	3 sushi	4 with	5 tuna	
NP		S			We 1
	V	VP			eat 2
		NP		NP	sushi 3
			P	PP	with 4
				NP	tuna 5

5.

1 We	2 eat	3 sushi	4 with	5 tuna	
NP		S			We 1
	V	VP		VP	eat 2
		NP		NP	sushi 3
			P	PP	with 4
				NP	tuna 5

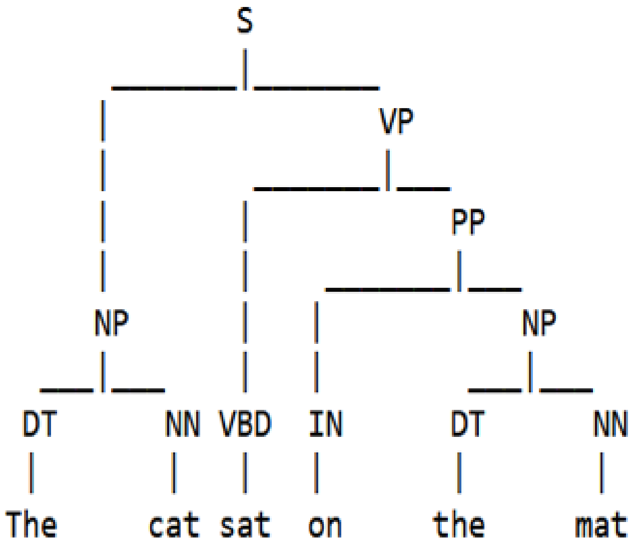
6.

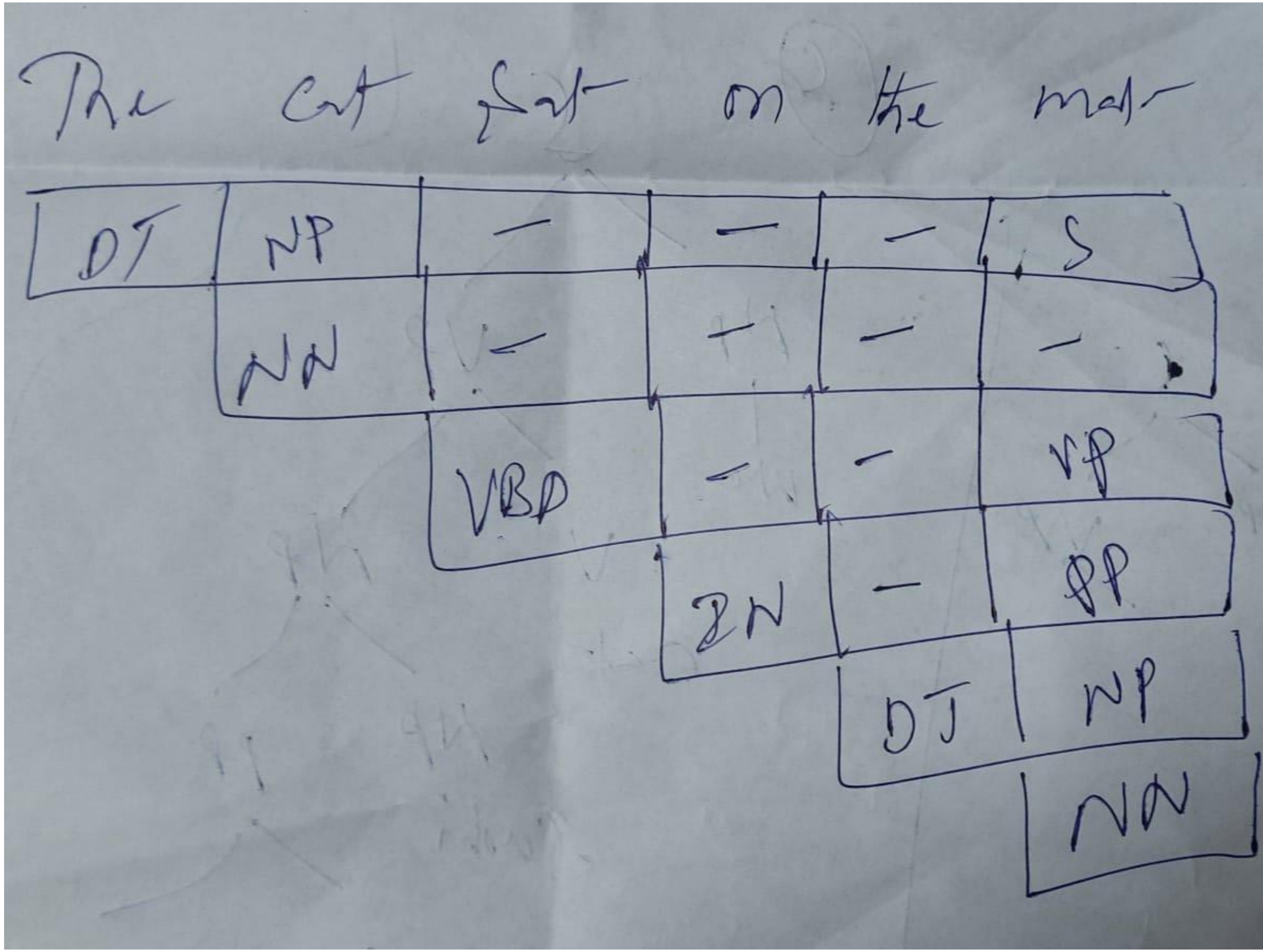
1 We	2 eat	3 sushi	4 with	5 tuna	
NP		S		S	We 1
	V	VP		VP	eat 2
		NP		NP	sushi 3
			P	PP	with 4
				NP	tuna 5

The cat sat on the mat

Chart Table

```
S -> NP VP
     NP -> DT NN | DT NN
     VP -> VBD PP
     PP -> IN NP
     DT -> 'The' | 'the'
     NN -> 'cat' | 'mat'
     VBD -> 'sat'
     IN -> 'on'
```

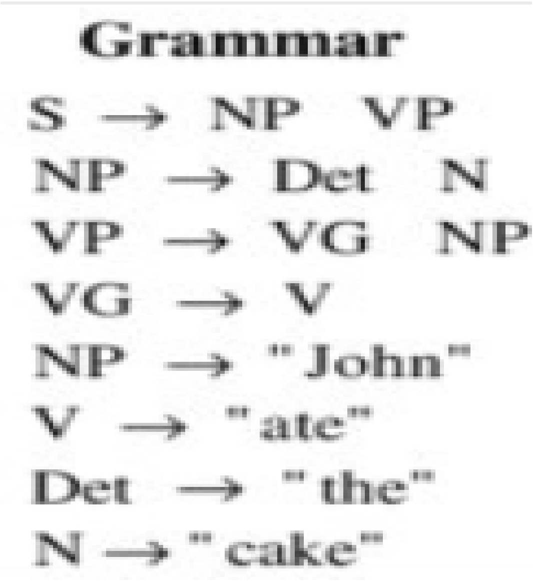




The	cat	sat	on	the	mat
DT	NP	–	–	–	S
	NN	–	–	–	–
		VBD	–	–	VP
			IN	–	PP
				DT	NP
					NN

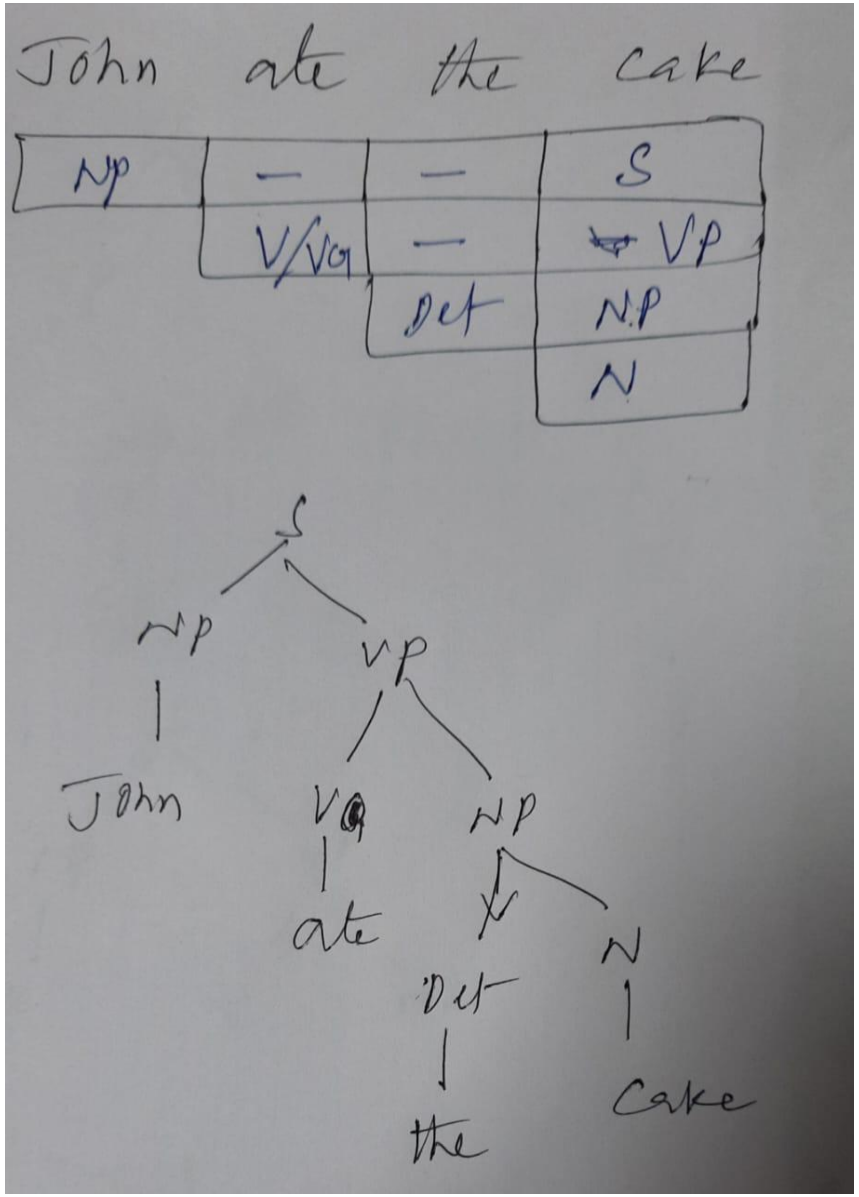
Sentence: John ate the cake

Chart Table



Grammar

S → NP VP
NP → Det N
VP → VG NP
VG → V
NP → "John"
V → "ate"
Det → "the"
N → "cake"



John	ate	the	cake
NP	—	—	S
	V/VG	—	VP
		Det	NP
			N

CKY Parsing Algorithm Code

```python
import nltk
from nltk import CFG
# Define a Context-Free Grammar (CFG)
grammar = CFG.fromstring("""
    S -> NP VP
    NP -> DT NN | DT NN
    VP -> VBD PP
    PP -> IN NP
    DT -> 'The' | 'the'
    NN -> 'cat' | 'mat'
    VBD -> 'sat'
    IN -> 'on'
""")

# Create a ChartParser(Botton-up parser)
parser = nltk.ChartParser(grammar)

sentence = "The cat sat on the mat".split()
# Generate parse tree
for tree in parser.parse(sentence):
        print(tree)
        tree.pretty_print()
```

Note: The **ChartParser** is a **bottom-up dynamic programming** parsing algorithm that efficiently constructs a parse tree by storing intermediate results in a **chart table**.

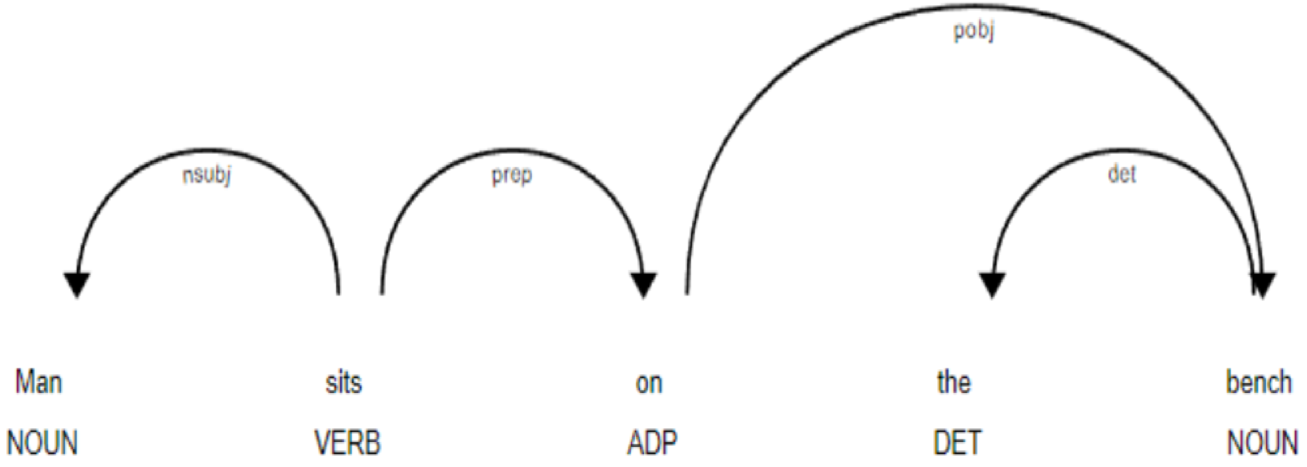
Output:

```
(S
  (NP (DT The) (NN cat))
  (VP (VBD sat) (PP (IN on) (NP (DT the) (NN mat)))))
             S
      _______|_______
     |               VP
     |        _______|___
     |       |           PP
     |       |    _______|___
     NP      |   |           NP
  ___|___    |   |        ___|___
 DT      NN VBD  IN      DT      NN
 |       |   |   |       |       |
The     cat sat  on     the     mat
```

Dependency Parsing

- Dependency parsing is used to analyze the grammatical structure of a sentence by identifying the dependency relationships between words
- It helps identify the dependency relations between words based on grammatical rules
- Dependency parsing represents the syntactic structure of a sentence as a **dependency tree**, where:
    - **Nodes** represent words.
    - **Edges** represent grammatical relationships (dependencies) between words.
    - The **root** of the tree is usually the main verb of the sentence.

Example 1: “**Man sits on the bench**“



Key Concepts of Dependency Parsing

Head: It is the central word that governs other words in the sentence.

•For instance, in the sentence *"The dog sat on the mat"*, “sat” is the head because it is the main verb that governs the sentence's structure.

Dependent: This word depends on another (the head) to express its full meaning, relying on the head to establish context.

dependency tree consists of components such as root, node, and edges.

Nodes: These represent individual words in the sentence.

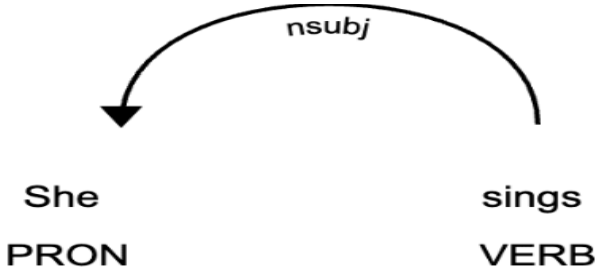
Edges: These are directed links between words, showing which word governs another. For instance, an edge from “sat” to “dog” indicates that “sat” is the head of the subject “dog.”

Root: The root is the topmost word in the tree, often representing the main verb or the core of the sentence.

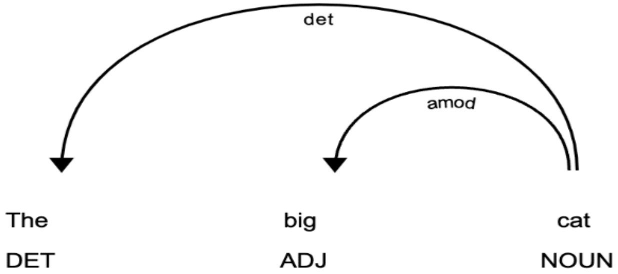
Grammatical Relationships

- The edges in a dependency tree represent grammatical relationships.
- These relationships define words’ roles in a sentence, such as subject, object, modifier, or adverbial.

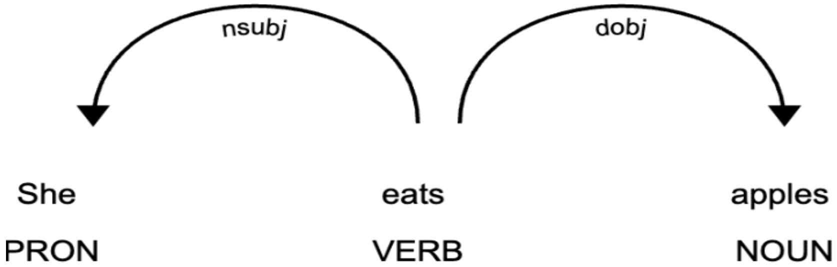
Subject-Verb Relationship: In a sentence like “She sings,” the word “She” depends on “sings” as the subject of the verb.



Modifier-Head Relationship: In the sentence “The big cat,” “big” modifies “cat,” creating a modifier-head relationship.

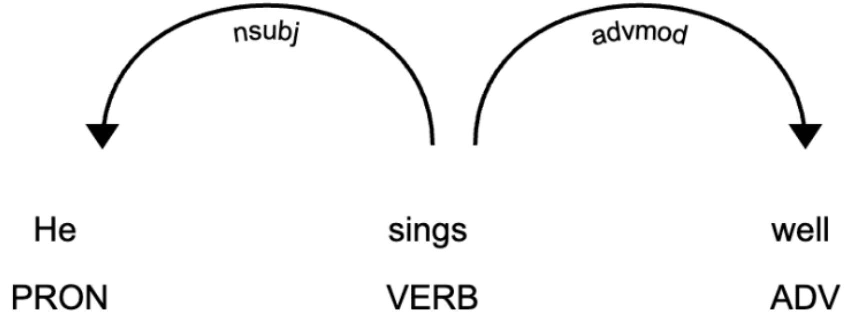


Direct Object-Verb Relationship: In “She eats apples,” “apples” is the direct object that depends on the verb “eats



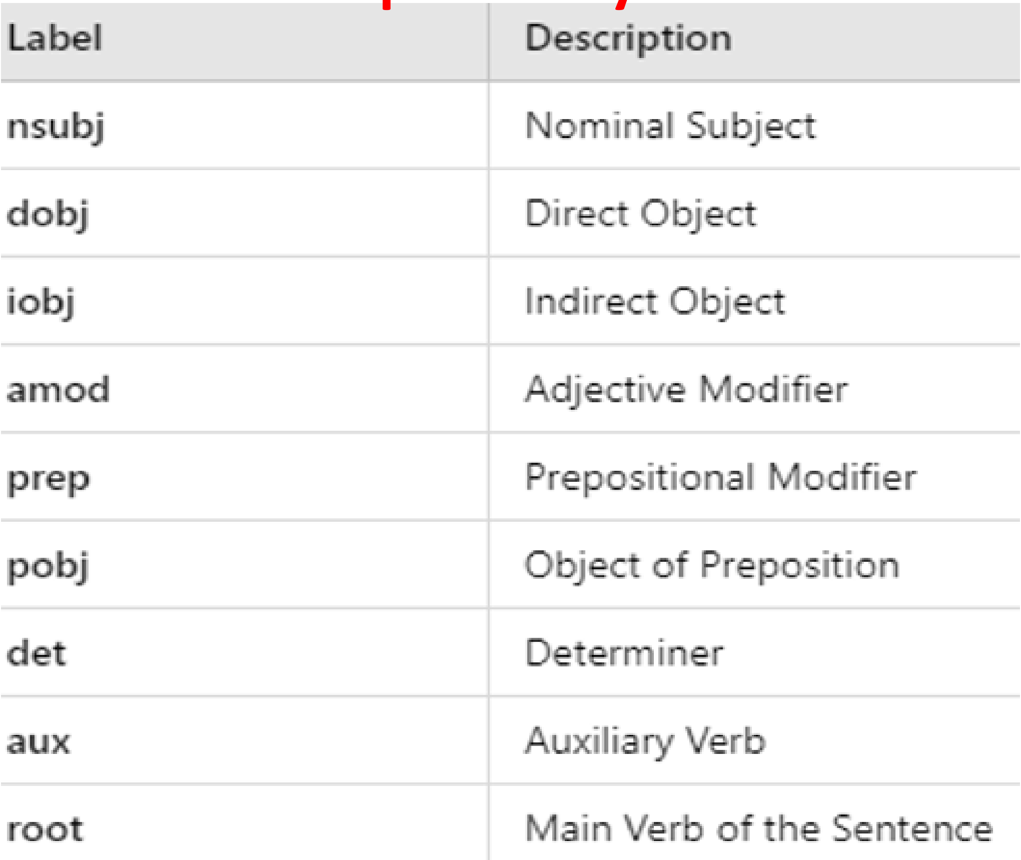
Grammatical Relationships

- **Adverbial-Verb Relationship:** In “He sings well,” “well” modifies the verb “sings” and forms an adverbial-verb relationship.



- **Preposition-Object (pobj):** In a prepositional phrase, the preposition governs the object it introduces.
- For instance, consider the sentence, "The cat sat on the mat." “On” is the preposition, and “mat” is its object.
- **Auxiliary-Verb (Auxiliary-Head):** An auxiliary verb helps to form different tenses, moods, or voices and depends on the main verb.
- For example, in *"She is running", “is” is the auxiliary verb modifying the main verb “running.”*

Dependency Labels



Label	Description
nsubj	Nominal Subject
dobj	Direct Object
iobj	Indirect Object
amod	Adjective Modifier
prep	Prepositional Modifier
pobj	Object of Preposition
det	Determiner
aux	Auxiliary Verb
root	Main Verb of the Sentence

How to construct Dependency Tree

Preprocess the Sentence

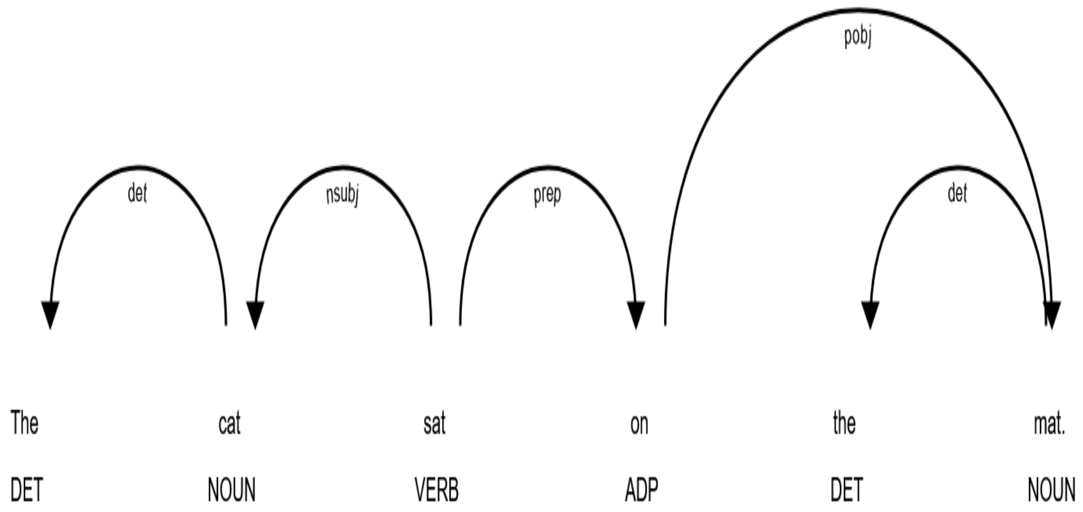
- **Tokenization**: Split the sentence into individual words (tokens).
- **POS Tagging**: Assign a part of speech (POS) tag to each word (e.g., noun, verb, adjective).

Identify the Root Verb

- Find the **main verb** (predicate) of the sentence, as it acts as the root of the tree.
- In *"The cat sat on the mat"*, the verb **"sat"** is the root.

Determine Dependencies

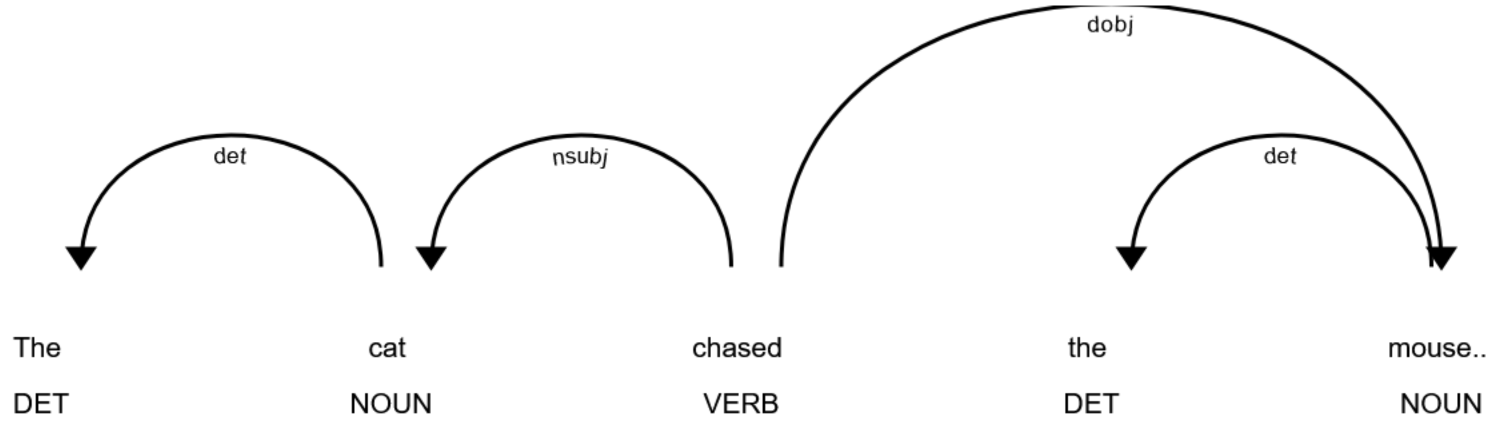
- Identify the grammatical relationships between words: **Subject** (who/what performs the action),**Object** (if any),**Modifiers** (adjectives, adverbs, etc.) and **Prepositional phrases** (relationships through prepositions)
- Example:
  - **"cat"** (subject) depends on **"sat"**.
  - **"on"** (preposition) depends on **"sat"**.
  - **"mat"** (object of "on") depends on **"on"**.
  - **"The"** (determiner) depends on **"cat"** and **"mat"**.



Problems

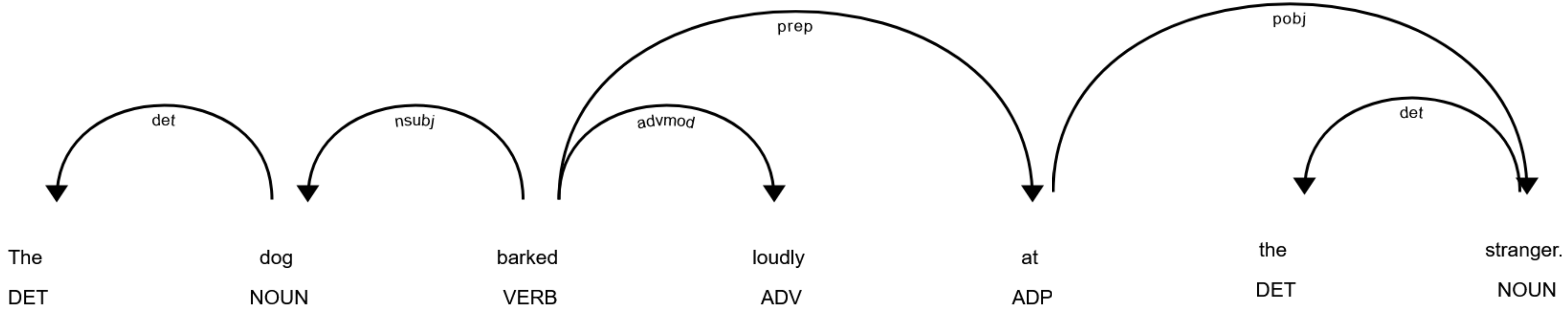
Example-1:

The cat chased the mouse.

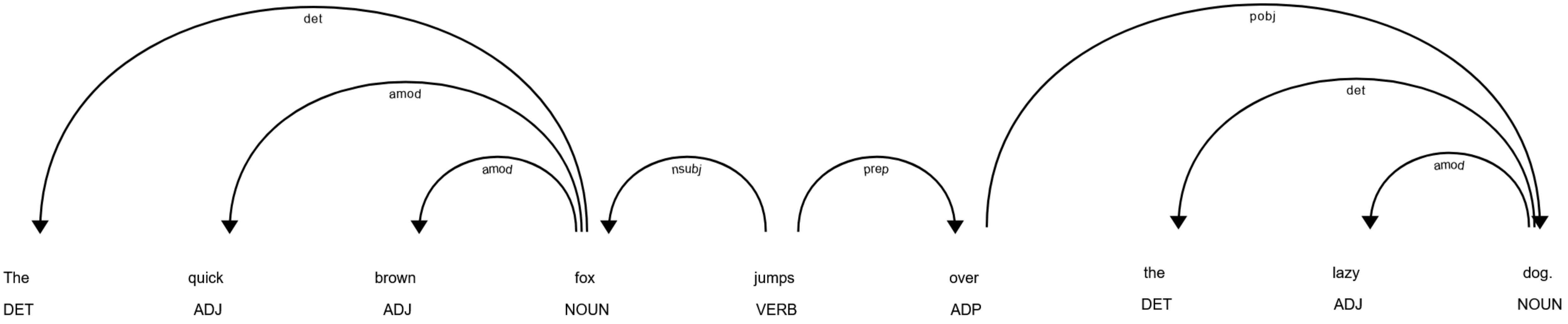


Example-2:

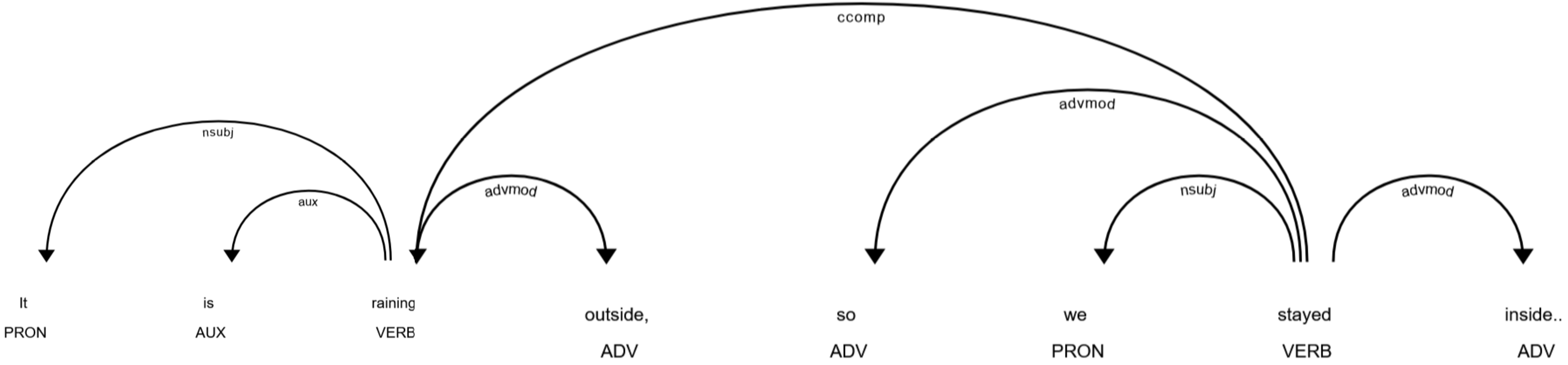
the dog barked loudly at the stranger



The quick brown fox jumps over the lazy dog



It is raining outside, so we stayed inside



Program to construct a Dependency Tree

```python
import spacy
from spacy import displacy

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

# Parse sentence
sentence = "The quick brown fox jumps over the lazy dog"
doc = nlp(sentence)

displacy.render(doc, style="dep", jupyter=True)  # If using Jupyter Notebook
```

Applications of Dependency Parsing

Machine Translation

- Improves the accuracy of translations by preserving syntactic relationships between words.

Question Answering (QA) Systems

- Extracts relevant answers by analyzing the syntactic structure of the question and matching it with the text.

Sentiment Analysis

- Determines the sentiment of a sentence by analyzing how words depend on each other (e.g., "not happy" vs. "very happy").

Text Summarization

- Identifies key elements in a sentence by recognizing dependencies, which helps in generating concise summaries.

Grammar Checking & Correction

- Detects grammatical errors by analyzing incorrect dependencies in sentence structure.

Chatbots & Conversational AI

- Improves chatbot responses by analyzing user input and understanding the syntactic structure.

Dependency parsing techniques/Dependency Parsers

1. Transition-Based Parsing
2. Graph-Based Parsing

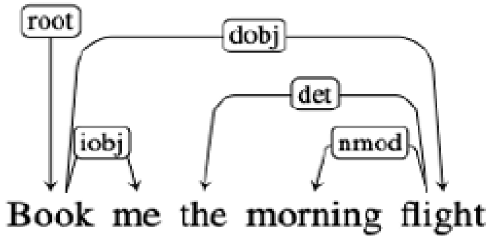
Transition-Based Parsing

- Transition-based parsing is a dependency parsing approach that constructs a dependency tree for a given sentence using a sequence of transitions
- Transition-based parsers(shift-reduce parser) process sentences from left to right and return a single tree.
- The algorithm we will consider can only produce projective dependency trees (no crossing edges)

Transition-Based Parsing

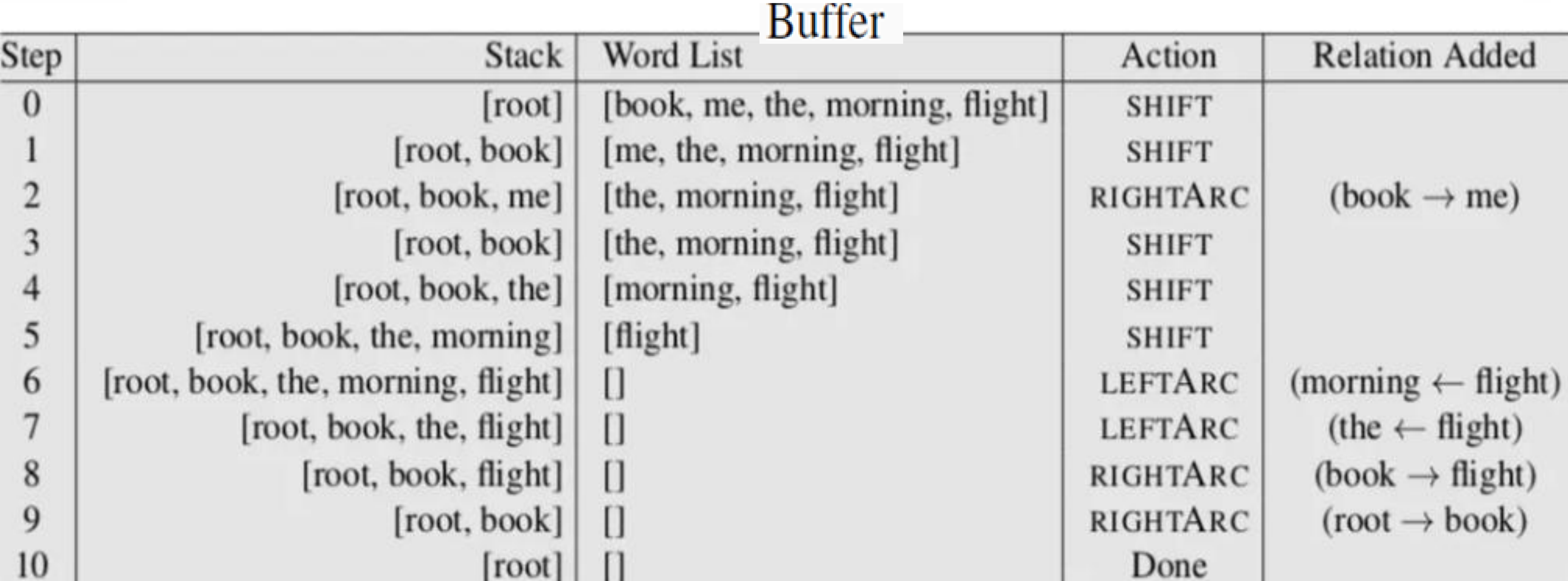
- The parsing process is modeled as a sequence of transitions
- A configuration consists of a stack s, a buffer b and a set of dependency arcs A: $c = (s, b, A)$
- Initially, $s = [\text{ROOT}]$, $b = [w_1, w_2, \ldots, w_n]$, $A = \emptyset$
- Three types of transitions (s_1, s_2: the top 2 words on the stack; b_1: the first word in the buffer)
  - LEFT-ARC (r): add an arc $s_1 \longleftarrow s_2$ to A, remove s_1 from the stack
  - RIGHT-ARC (r): add an arc $s_1 \longrightarrow s_2$ to A, remove s_2 from the stack
  - SHIFT: move b_1 from the buffer to the stack
- A configuration is terminal if $s = [\text{ROOT}]$ and $b = \emptyset$

Transition-based dependency parsing: Example 1



"Book me the morning flight"

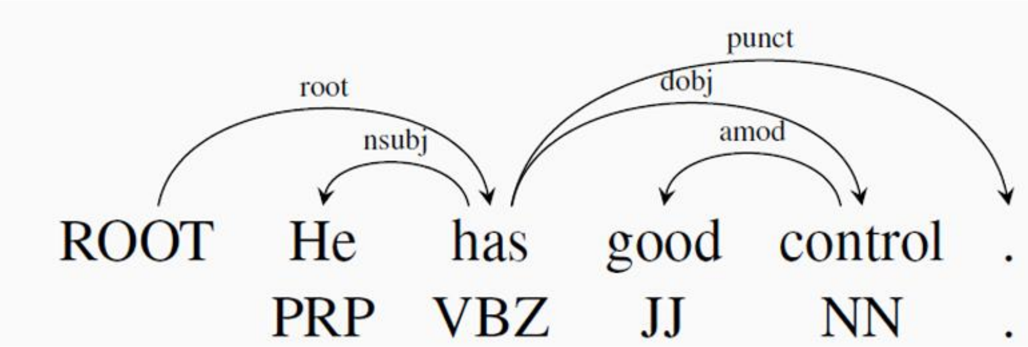
A running example

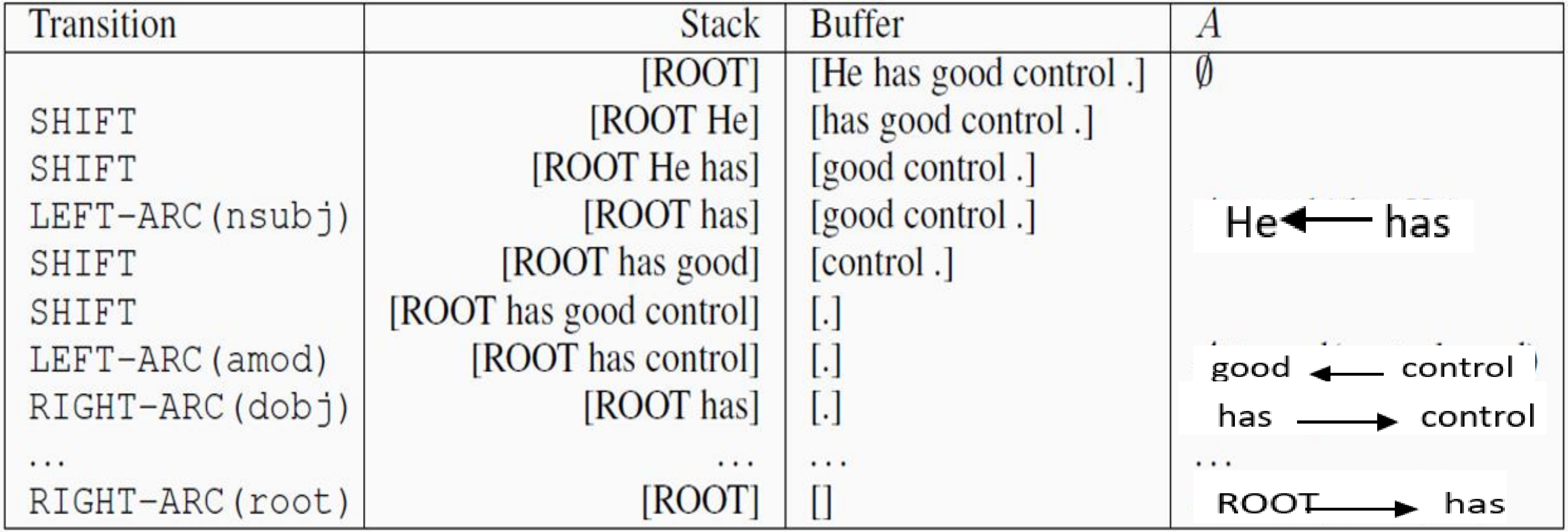


Buffer

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book → me)
3	[root, book]	[the, morning, flight]	SHIFT	
4	[root, book, the]	[morning, flight]	SHIFT	
5	[root, book, the, moming]	[flight]	SHIFT	
6	[root, book, the, morning, flight]	[]	LEFTARC	(morning ← flight)
7	[root, book, the, flight]	[]	LEFTARC	(the ← flight)
8	[root, book, flight]	[]	RIGHTARC	(book → flight)
9	[root, book]	[]	RIGHTARC	(root → book)
10	[root]	[]	Done	

Transition-based dependency parsing: Example 2

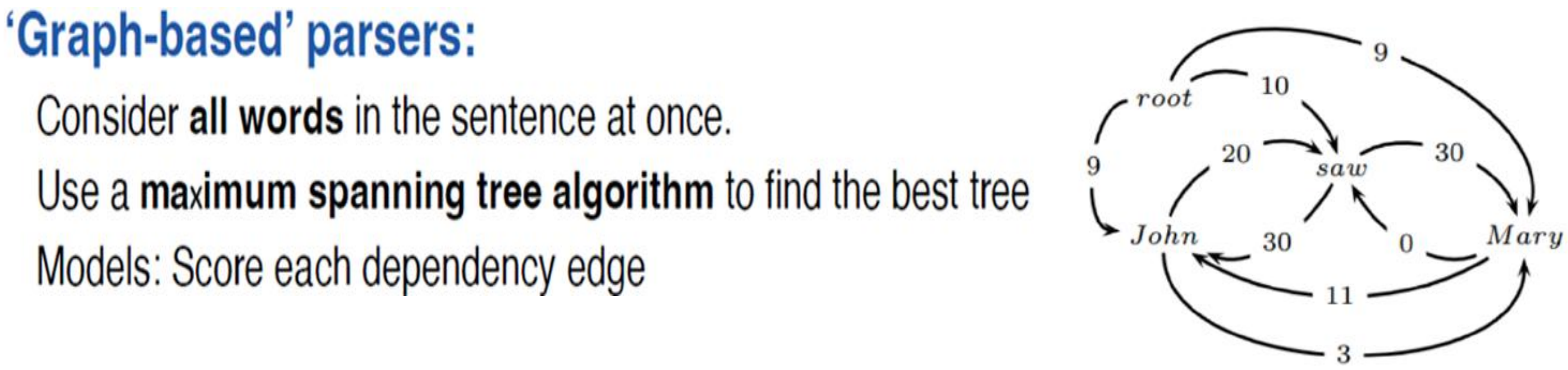




Transition	Stack	Buffer	*A*
	[ROOT]	[He has good control .]	∅
SHIFT	[ROOT He]	[has good control .]	
SHIFT	[ROOT He has]	[good control .]	
LEFT-ARC(nsubj)	[ROOT has]	[good control .]	He ← has
SHIFT	[ROOT has good]	[control .]	
SHIFT	[ROOT has good control]	[.]	
LEFT-ARC(amod)	[ROOT has control]	[.]	good ← control
RIGHT-ARC(dobj)	[ROOT has]	[.]	has → control
...	...	...	...
RIGHT-ARC(root)	[ROOT]	[]	ROOT → has

Graph-based dependency parsing

- Graph-Based Parsing is a syntactic parsing approach that represents a sentence as a **directed graph**, where words are **nodes**, and possible syntactic relations are **edges**.
- The goal is to find the **best dependency tree** that represents the syntactic structure of the sentence.
- Graph-based parsing is primarily used in **Dependency Parsing**, where we find the best **Maximum Spanning Tree (MST)** using algorithms like **Chu-Liu-Edmonds'(CLE) Algorithm**.



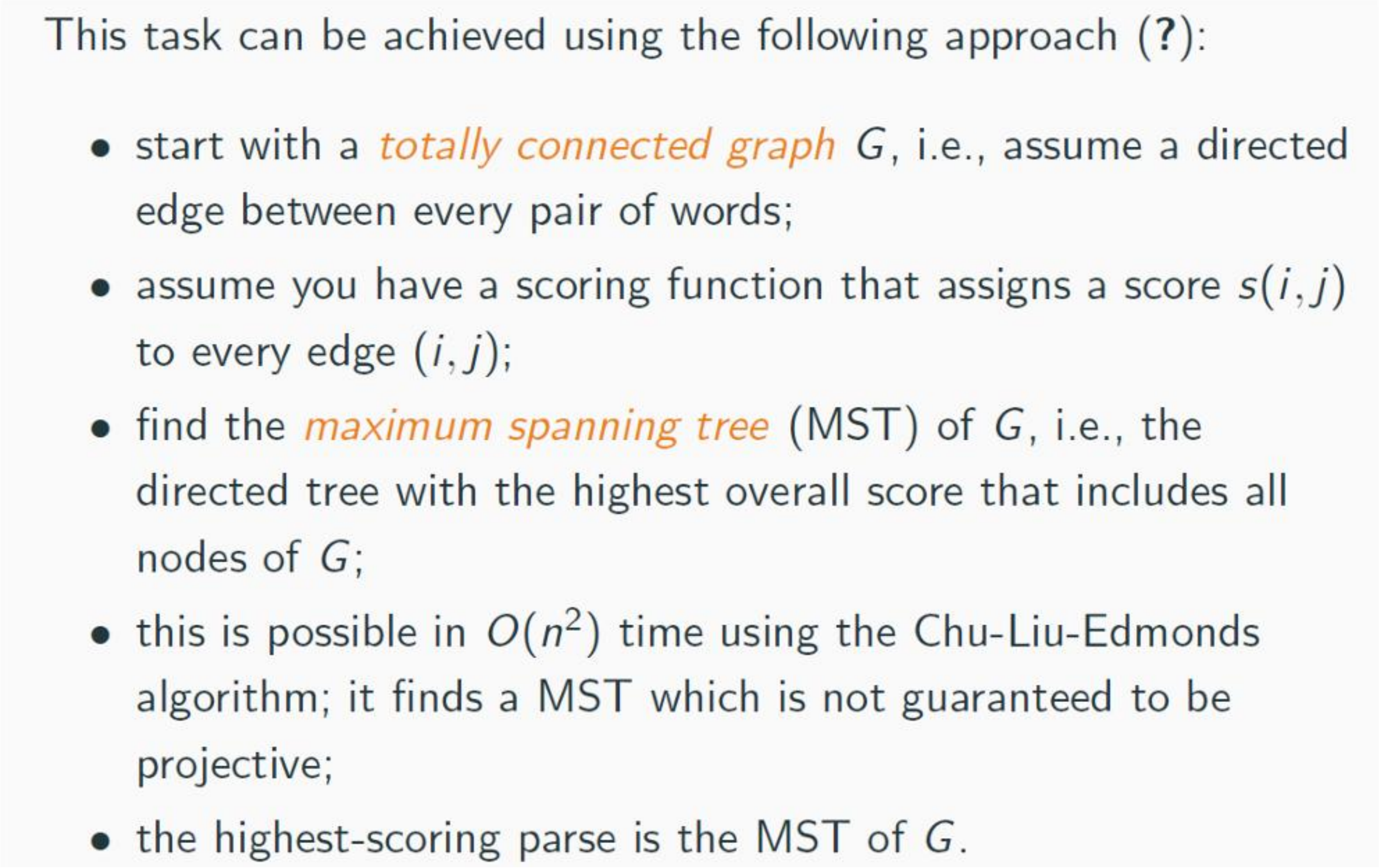
'Graph-based' parsers:

Consider **all words** in the sentence at once.

Use a **maximum spanning tree algorithm** to find the best tree

Models: Score each dependency edge

The best dependency parse is the maximum spanning tree



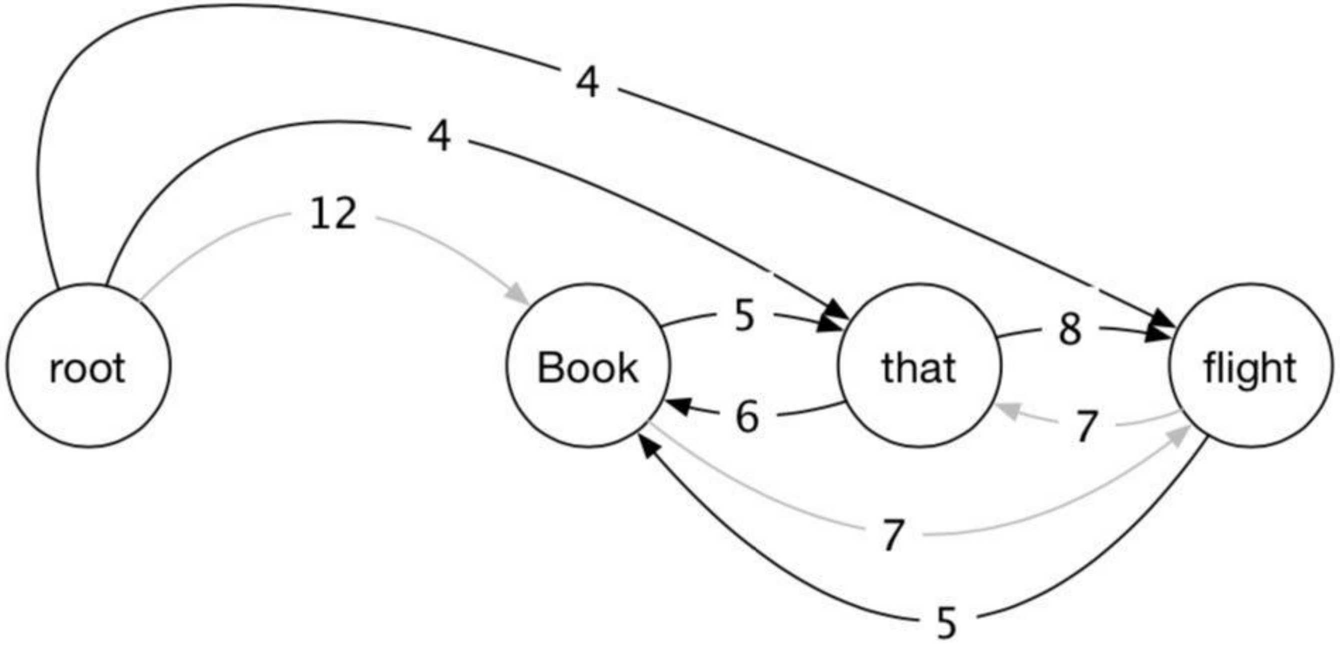
This task can be achieved using the following approach (**?**):

- start with a *totally connected graph* G, i.e., assume a directed edge between every pair of words;
- assume you have a scoring function that assigns a score $s(i,j)$ to every edge (i,j);
- find the *maximum spanning tree* (MST) of G, i.e., the directed tree with the highest overall score that includes all nodes of G;
- this is possible in $O(n^2)$ time using the Chu-Liu-Edmonds algorithm; it finds a MST which is not guaranteed to be projective;
- the highest-scoring parse is the MST of G.

Chu-Liu-Edmonds (CLE) Algorithm Example-1

Let us pick up the sentence
‘book that flight’

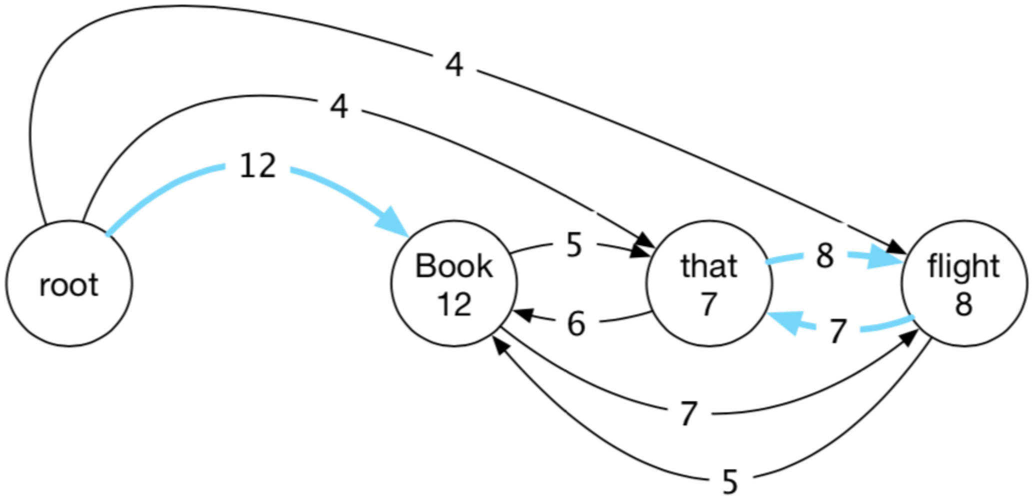
1.Create a directed graph with an additional vertice ‘Root’ having outgoing edges to all words of the sentence.
It can be observed that all words have an outgoing edge to all other words (except Root) in the Graph G. Also, each word receives an incoming edge from all the vertices (including root).



Chu-Liu-Edmonds (CLE) Algorithm Example-1

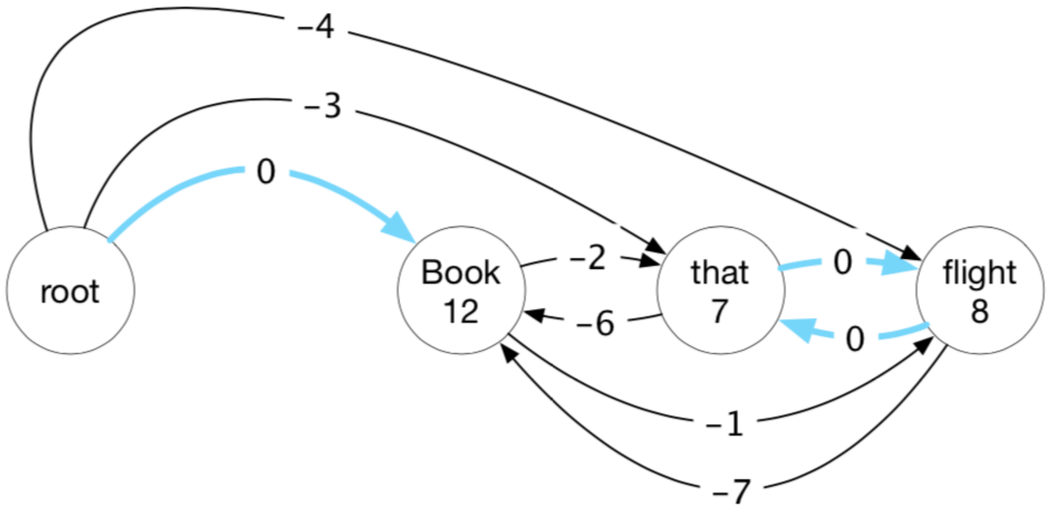
2. Greedily, choose incoming edges for each vertex with the highest weight

Here, it can be observed that the **blue lines depict the greedy selection** for incoming edge for each vertex (word). if we get an MST in the very first iteration, GREAT!!! you can stop here only. But as you see, we got a graph with cycle(that →flight, flight →that)



3. **Subtract weight chosen greedily** from weights of all other incoming edges for that vertex.

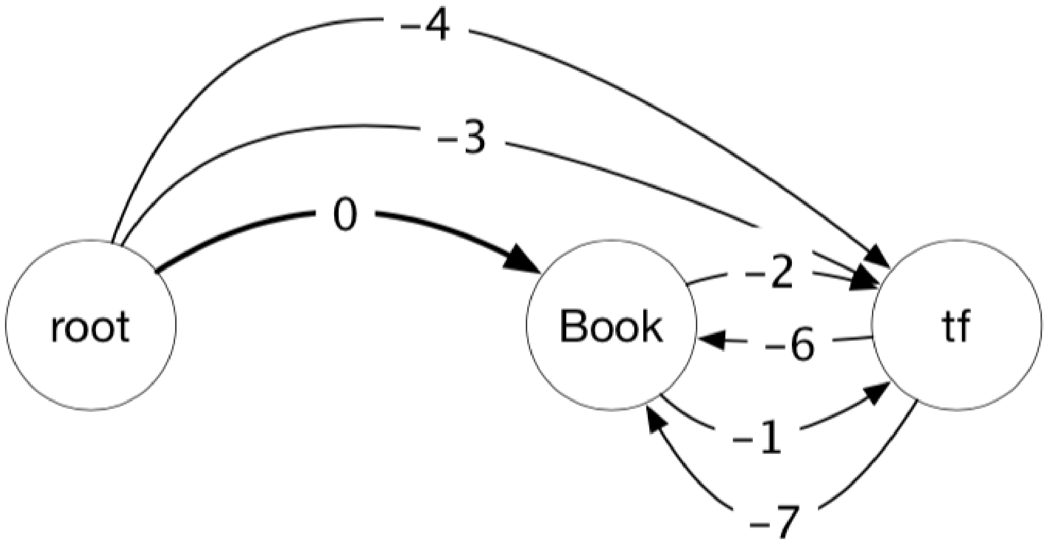
*It can be observed that for incoming edges to 'Book', 12 (weight of greedily choosen edge for 'book'. This value is 8 for 'flight' & 7 for 'that') has been subtracted leading to **(5–12=-7) for flight →Book, that →Book(6–12=-6).** Similarly, all other values are updated throughout the graph G.*



Chu-Liu-Edmonds (CLE) Algorithm Example-1

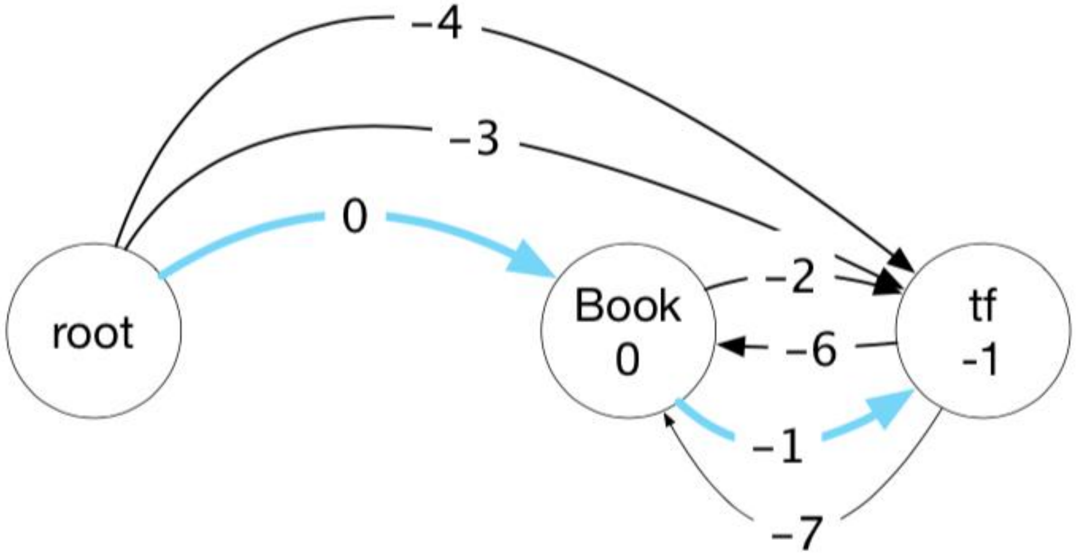
3. **Collapse the Graph**. Pick the cycle existing (that →flight, flight →that) & merge them to form a single node. Note that this can be detected if a vertice has an outgoing & incoming edge to the same vertex with weight 0 as can be observed here between ‘that’ & ‘flight’. **We get a new graph after this step.**

Here, ‘tf’ represent the merged vertice formed by ‘that’ & ‘flight’



4. Repeat from step 2 i.e Greedy choice of incoming edges for all vertices.

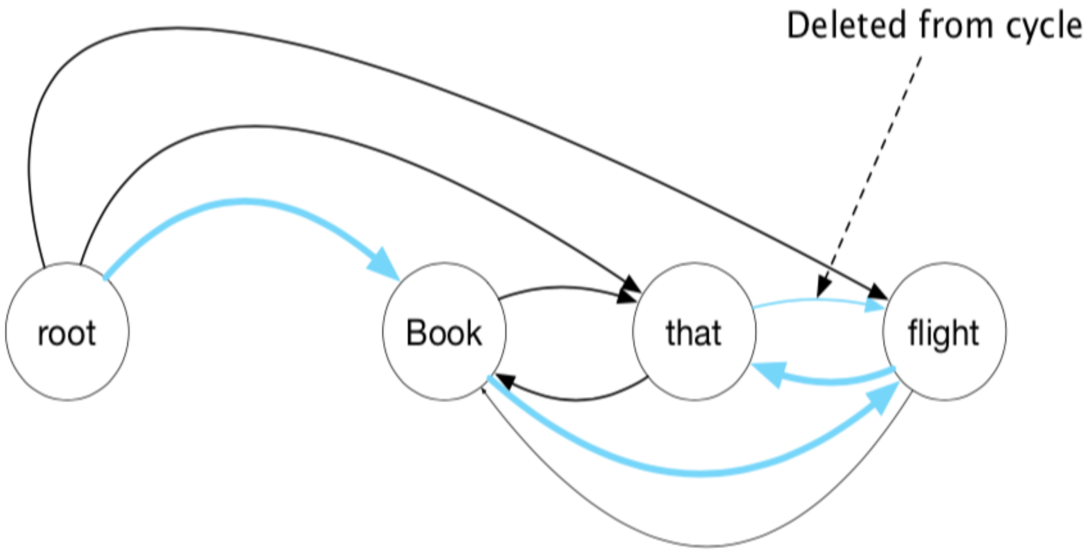
5. As we can see, we don't encounter any cycle anymore i.e **we have got an MST.**



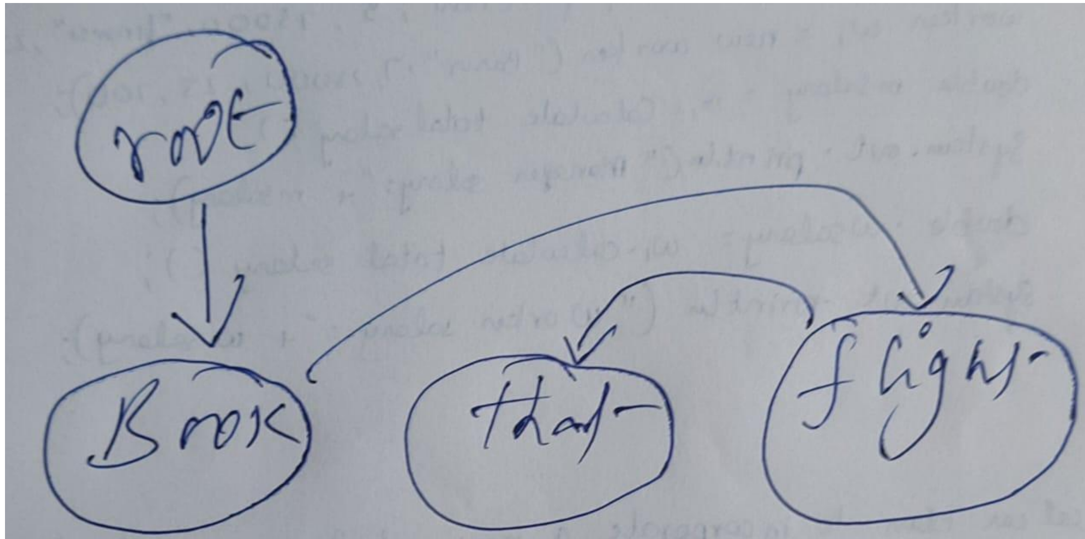
Chu-Liu-Edmonds (CLE) Algorithm Example-1

6. Once we get an MST, **we have to trace back to split all the merged vertices**. As we trace back recursively, we will be eliminating one of the edges that were forming the cycle.

we already found an incoming edge for ‘flight’ & as we can’t have multiple incoming edges for a vertex in MST, we will be deleting that →flight.



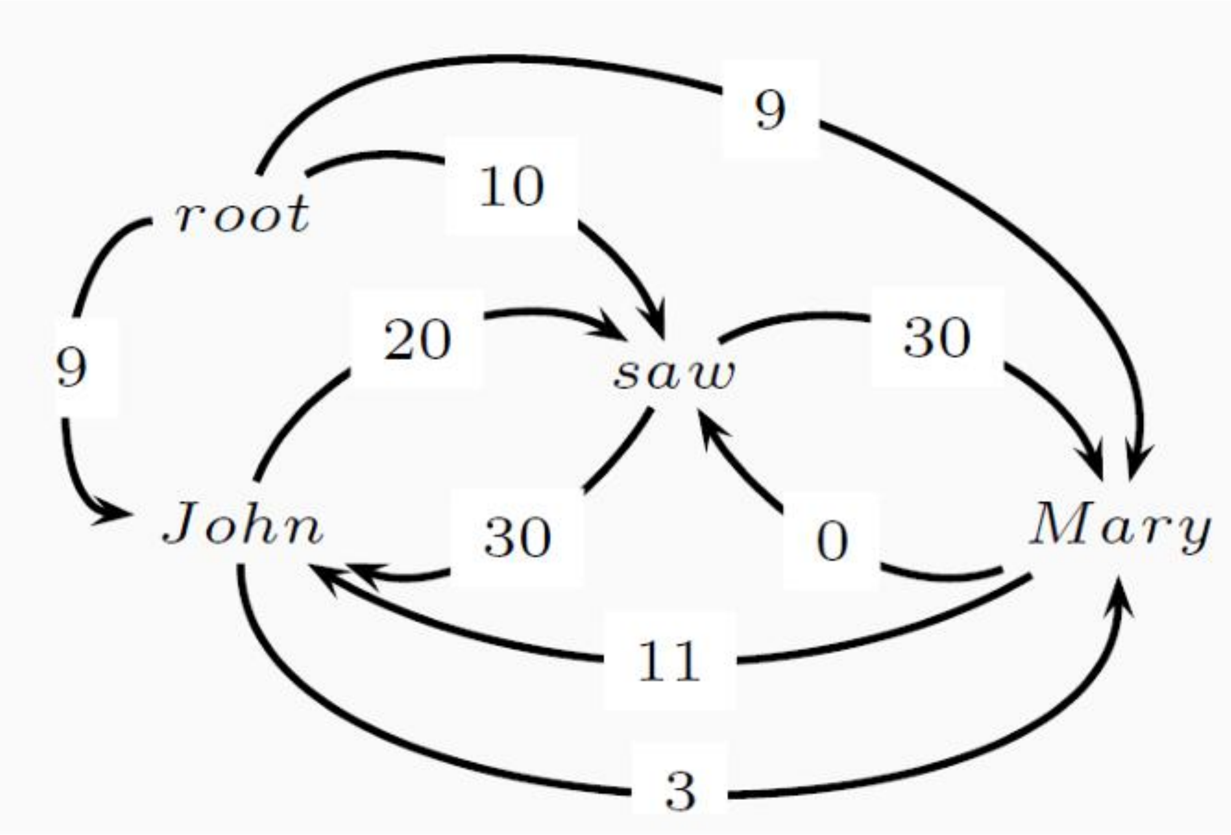
best dependency tree



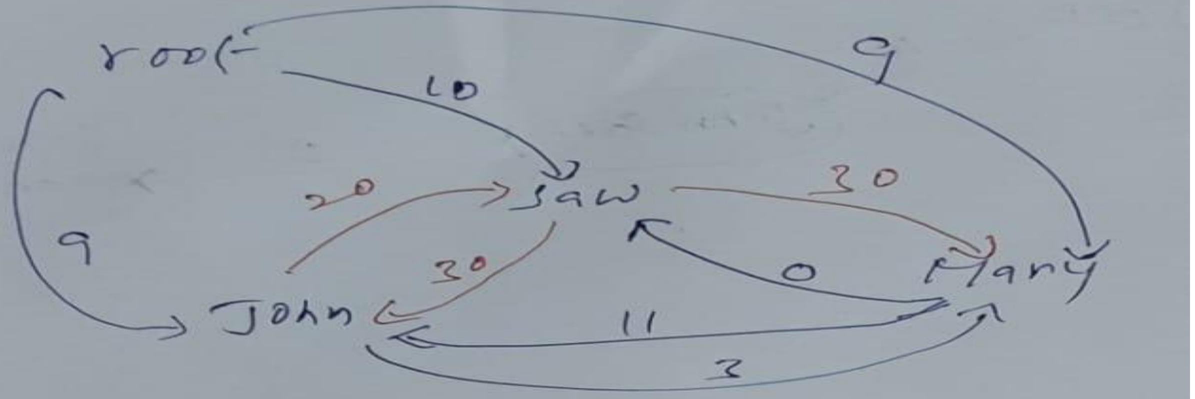
Chu-Liu-Edmonds (CLE) Algorithm Example-1

Let us pick up the sentence
‘John saw Mary’

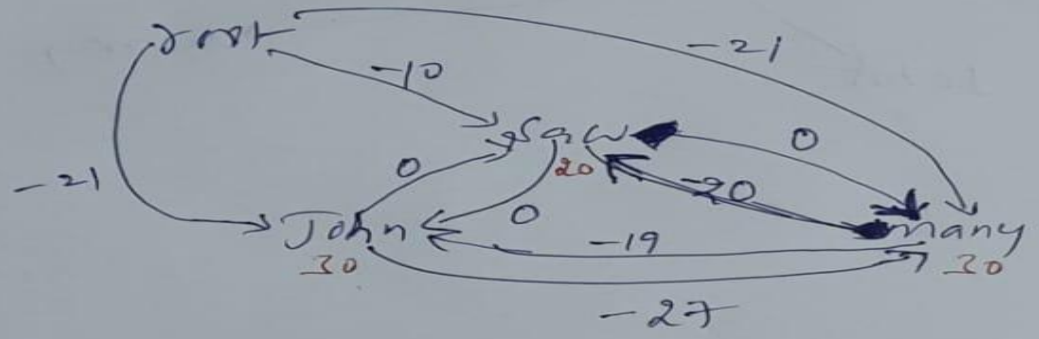
1.



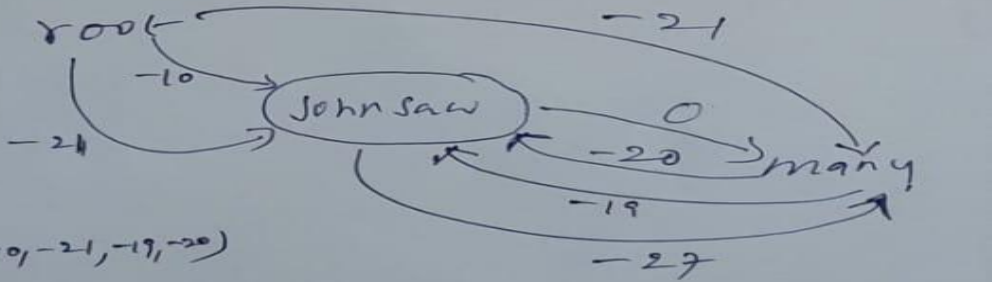
2) choose incoming edges for each vertex with the **highest weight**



3) **Subtract weight**

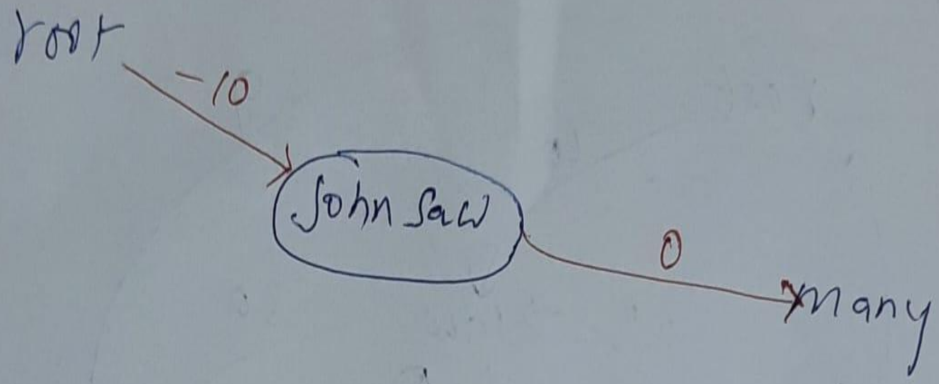


4) **Collapse the Graph**

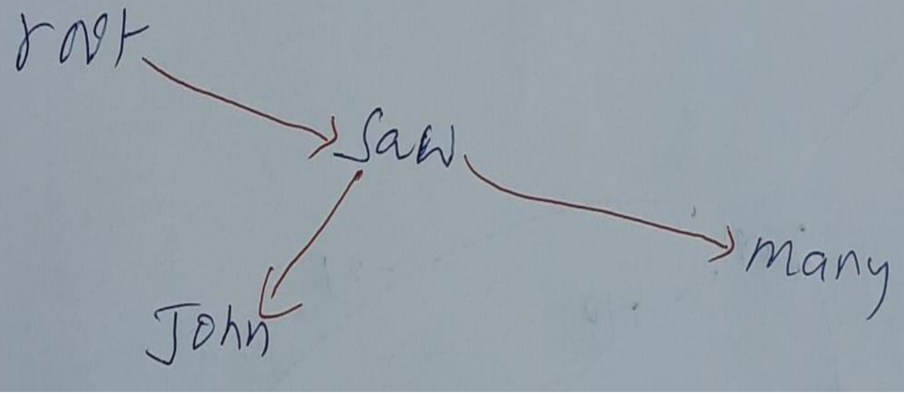


at
John saw → Max(-10,-21,-19,-20)
mary → Max(-21,0,-27)

5) No cycle anymore i.e **we have got an MST.**



6) **split all the merged vertices**



Semantic analysis

- Semantic analysis is a critical area in NLP that focuses on understanding the meaning and interpretation of words, sentences, and texts.
- Understanding Natural Language might seem a straightforward process to us as humans.
- However, due to the vast complexity and subjectivity involved in human language, interpreting it is quite a complicated task for machines.
- Semantic Analysis of Natural Language captures the meaning of the given text while taking into account context, logical structuring of sentences and grammar roles.

Meaning Representation

- While, as humans, it is pretty simple for us to understand the meaning of textual information, it is not so in the case of machines.
- Thus, machines tend to represent the text in specific formats in order to interpret its meaning.
- This formal structure that is used to understand the meaning of a text is called meaning representation.

Basic Units of Semantic System:

1.Entity: An entity refers to a particular unit or individual in specific such as a person or a location.

2.Concept: A Concept may be understood as a generalization of entities. It refers to a broad class of individual units. For example, Learning Portals, City, Students.

3.Relations: Relations help establish relationships between various entities and concepts.

4.Predicate: Predicates represent the verb structures of the sentences.

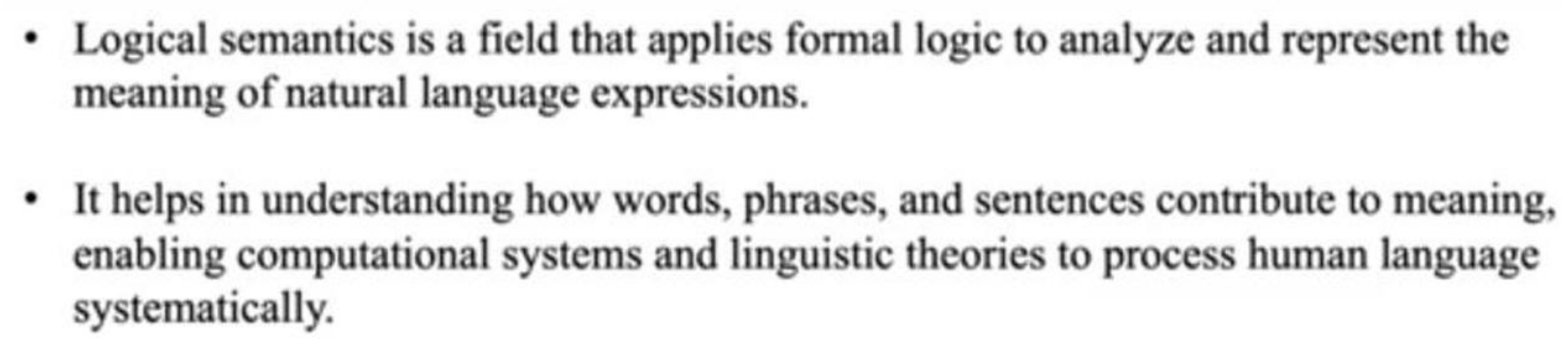
In Meaning Representation, we employ these basic units to represent textual information.

Meaning Representation

Approaches to Meaning Representations

1. First-order predicate logic (FOPL)
2. Semantic Nets
3. Frames
4. Conceptual dependency (CD)
5. Rule-based architecture
6. Case Grammar
7. Conceptual Graphs

Logical Semantics

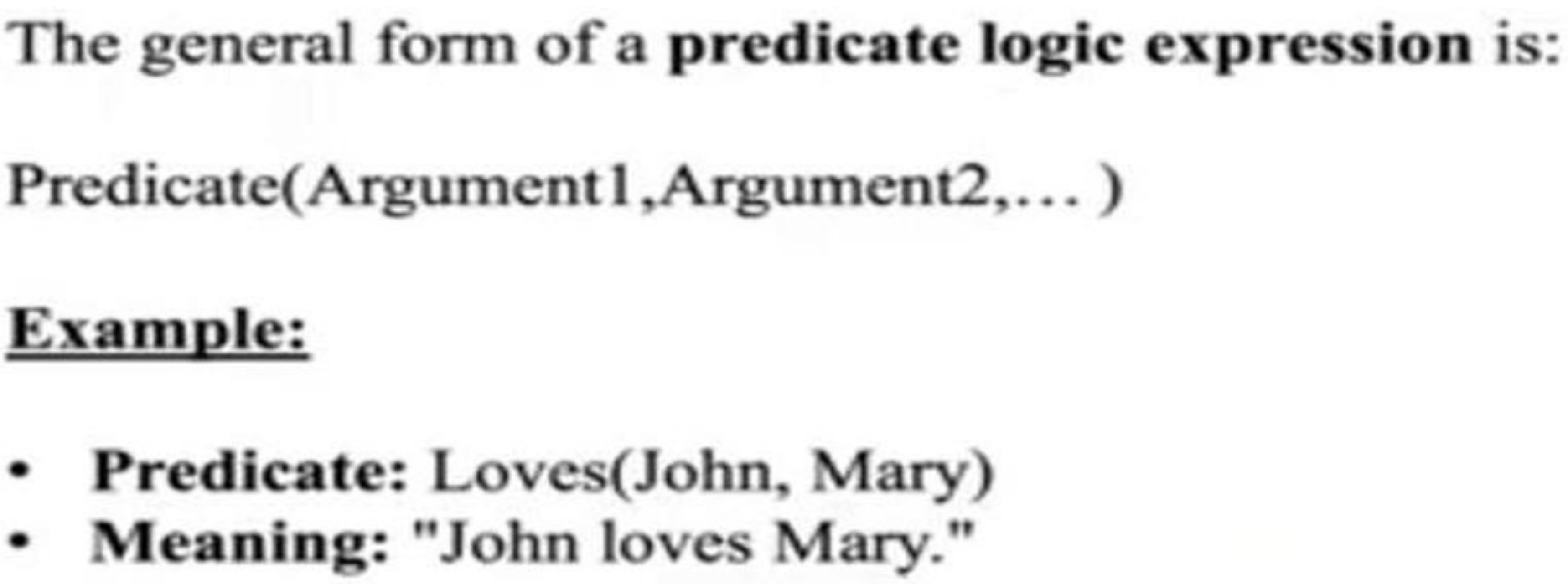


- Logical semantics is a field that applies formal logic to analyze and represent the meaning of natural language expressions.

- It helps in understanding how words, phrases, and sentences contribute to meaning, enabling computational systems and linguistic theories to process human language systematically.

First-order predicate logic (FOPL)

- First-Order Predicate Logic (FOPL) is a formal system used in logical semantics to represent the meaning of natural language sentences.



The general form of a **predicate logic expression** is:

Predicate(Argument1,Argument2,…)

Example:

- **Predicate:** Loves(John, Mary)
- **Meaning:** "John loves Mary."

Key Concepts in FOPL

Predicates:

- Predicates represent properties or relationships between entities.
- For example, Loves(John, Mary) indicates that John loves Mary.

Quantifiers:

- **Universal Quantifier (∀)**: Represents "For ALL".

  For example, ∀x (Human(x) → Mortal(x)) means "All humans are mortal."

- **Existential Quantifier (∃)**: Represents "There exists "

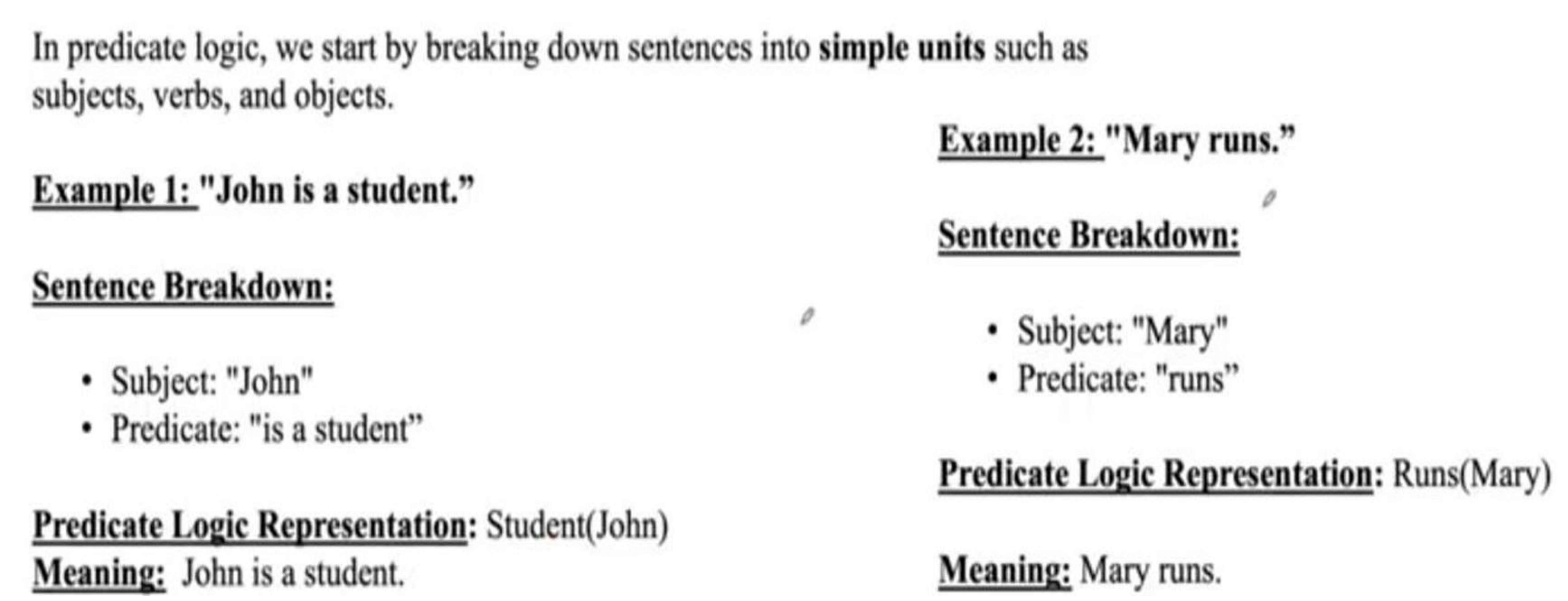
  For example, ∃x (Bird(x) ∧ CanFly(x)) means "There exists a bird that can fly."

Logical Connectives:

- Logical connectives include AND (∧), OR (∨), NOT (¬), IMPLIES (→), and EQUIVALENT (↔) to form complex logical expressions.

Representing Sentences in FOPL

Simple Sentences:



In predicate logic, we start by breaking down sentences into **simple units** such as subjects, verbs, and objects.

Example 1: "John is a student.”

Sentence Breakdown:

- Subject: "John"
- Predicate: "is a student”

Predicate Logic Representation: Student(John)

Meaning: John is a student.

Example 2: "Mary runs.”

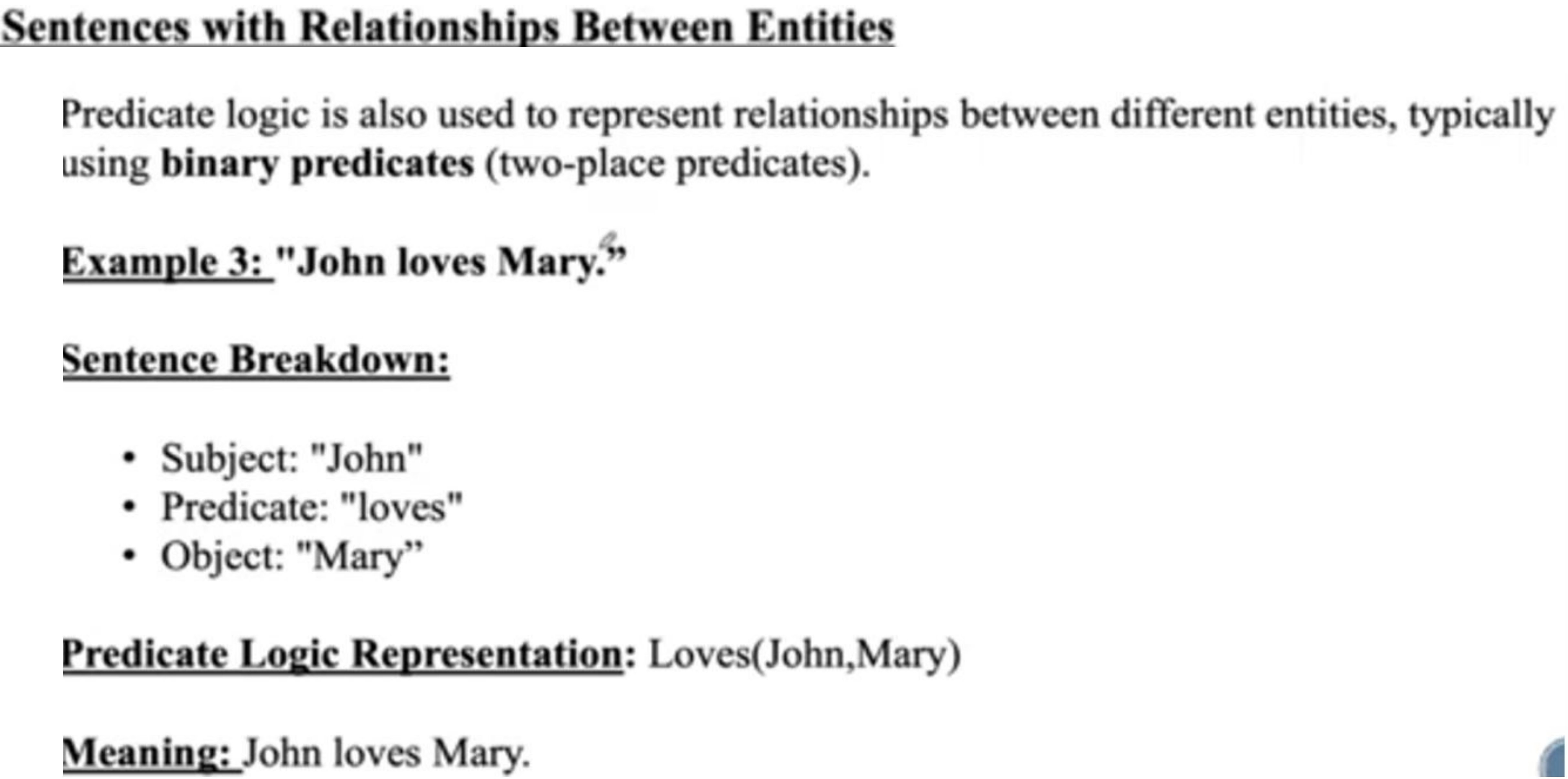
Sentence Breakdown:

- Subject: "Mary"
- Predicate: "runs”

Predicate Logic Representation: Runs(Mary)

Meaning: Mary runs.

Representing Sentences in FOPL



Sentences with Relationships Between Entities

Predicate logic is also used to represent relationships between different entities, typically using **binary predicates** (two-place predicates).

Example 3: "John loves Mary."

Sentence Breakdown:

- Subject: "John"
- Predicate: "loves"
- Object: "Mary"

Predicate Logic Representation: Loves(John,Mary)

Meaning: John loves Mary.

Representing Sentences in FOPL

Sentence with Quantifiers:

- "All humans are mortal."

FOPL: $\forall x\,(Human(x) \rightarrow Mortal(x))$

Existential Sentence:

- "There exists a student who loves mathematics."

FOPL: $\exists x\,(Student(x) \land Loves(x, Mathematics))$

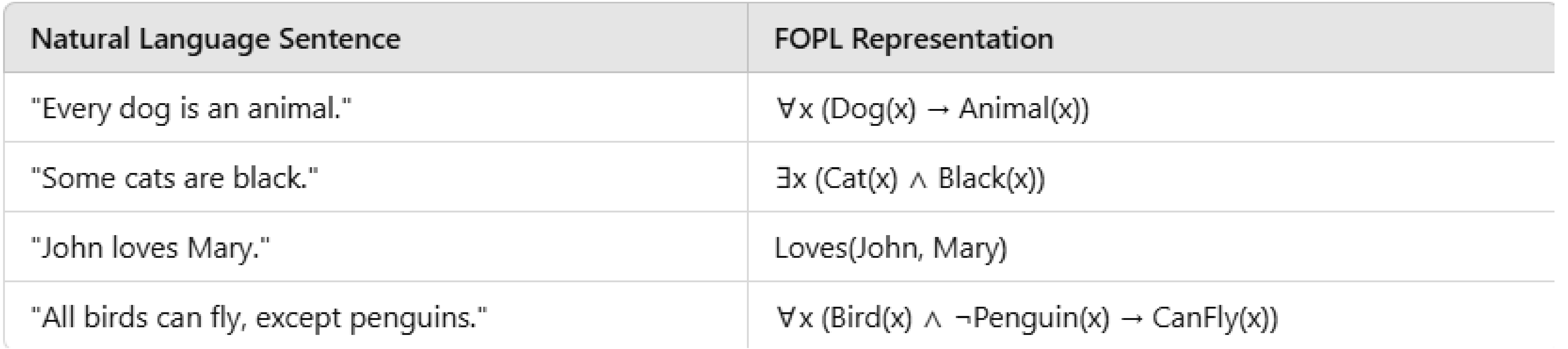
- "Every student studies a subject."

FOPL: $\forall x\,(Student(x) \rightarrow \exists y\,(Subject(y) \land Studies(x, y)))$

- "If a person likes ice cream, they are happy."

FOPL: $\forall x\,(Likes(x, IceCream) \rightarrow Happy(x))$

Representing Sentences in FOPL



Natural Language Sentence	FOPL Representation
"Every dog is an animal."	∀x (Dog(x) → Animal(x))
"Some cats are black."	∃x (Cat(x) ∧ Black(x))
"John loves Mary."	Loves(John, Mary)
"All birds can fly, except penguins."	∀x (Bird(x) ∧ ¬Penguin(x) → CanFly(x))

Logical Semantics applications in NLP

Question Answering

- By representing questions and knowledge bases in logical forms, systems can perform precise reasoning to find accurate answers.

Example: Given the knowledge base Capital(France, Paris), the question "What is the capital of France?" can be answered using logical inference.

Machine Translation

- Logical semantics can improve machine translation by providing a deeper understanding of the source language's meaning, leading to more accurate translations.

Example: Translating the sentence "He is running a marathon" into another language while preserving the logical structure: Running(He, Marathon)

Information Extraction

- By representing text in logical forms, systems can identify and extract relevant information for various applications.

Logical Semantics applications in NLP

Dialogue Systems

- Logical semantics enhances dialogue systems by enabling them to understand and generate meaningful responses based on the logical representation of user queries and context.

Text Summarization

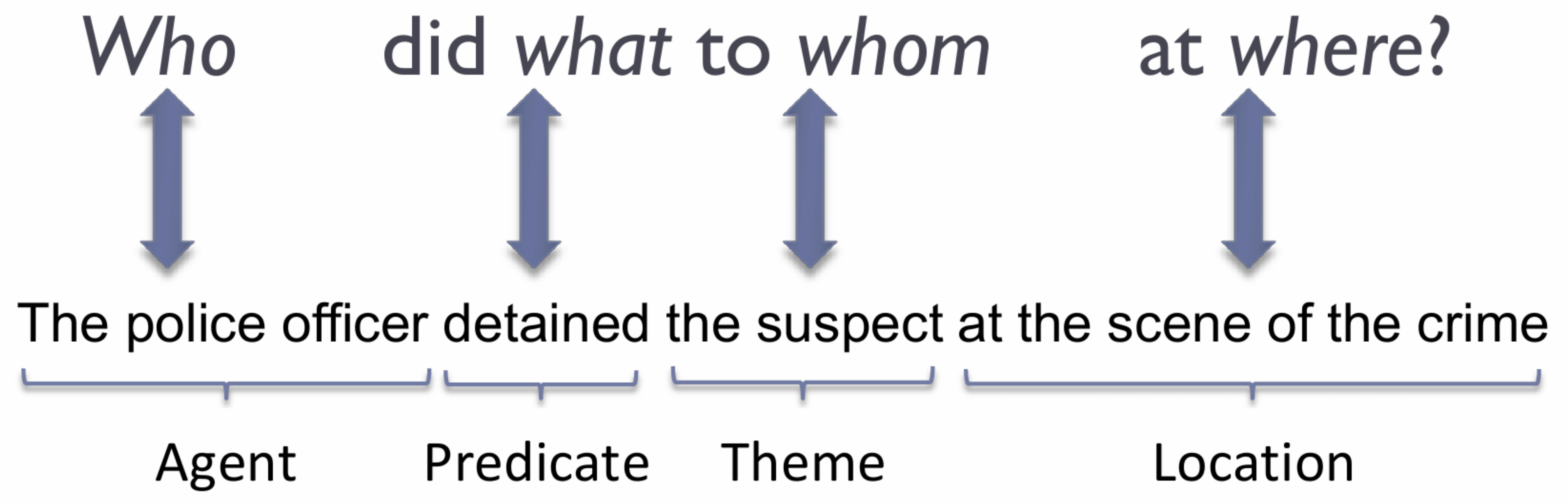
- Logical semantics can improve text summarization by identifying the key propositions and relationships in a text and generating concise summaries that capture the main points.

Sentiment Analysis

- Logical semantics can be used to represent and analyze the sentiments expressed in text.
- By understanding the logical structure of sentences, systems can identify positive, negative, or neutral sentiments more accurately.

Semantic Role Labelling

- Semantic Role Labeling (SRL), also known as shallow semantic parsing is a process in NLP that assign labels(roles) to words or phrases in a sentence to indicate their semantic role.
- Semantic Role Labeling (SRL) is a key task in NLP that assign roles to words or phrases in a sentence to determine "who did what to whom, when, where, and how."


Who
did *what* to *whom*
at *where?*
The police officer detained the suspect at the scene of the crime
Agent
Predicate
Theme
Location

Semantic Roles in NLP

Agent: The entity that performs the action.
Example: *John* (agent) kicked the ball.

Patient: The entity that is affected by the action.
Example: John kicked the *ball* (patient).

Instrument: The entity used to perform the action.
Example: She cut the bread with a *knife* (instrument).

Experiencer: The entity that experiences or perceives something.
Example: *Mary* (experiencer) heard a strange noise.

Theme: The entity that is moved or the topic of the action.
Example: She gave the *book* (theme) to him.

Location: The place where the action occurs.
Example: He stayed in the *house* (location).

Source: The starting point of the action.
Example: She came from the *village* (source).

Goal: The endpoint of the action.
Example: He walked to the *park* (goal).

Semantic Roles in NLP

Example 1:

Sentence: John sold the car to Mary.

- **Predicate**: sold
- **Agent** : John
- **Theme (object)**: the car
- **Recipient (receiver)**: Mary

Example 2:

Sentence: The cat sat on the mat.

- **Predicate**: sat
- **Agent (doer)**: The cat
- **Location**: on the mat

Semantic Roles in NLP

Example 3:

Sentence: The chef chopped the vegetables with a knife.

- **Predicate**: chopped
- **Agent (doer)**: The chef
- **Theme (object)**: the vegetables
- **Instrument**: with a knife

Sentence:

"The doctor prescribed medicine to the patient at the hospital."

Roles Identified:

- **The doctor** → *Agent* (Who performed the action?)
- **Prescribed** → *Predicate* (What is the action?)
- **Medicine** → *Theme* (What was prescribed?)
- **To the patient** → *Recipient* (Who received it?)
- **At the hospital** → *Location* (Where did it happen?)