M02S02 Parsing

Natural Language Processing – Syntactic Processing

Introduction

There are three broad levels of syntactic processing:

- Part-of-speech tagging
- Constituency parsing
- Dependency parsing

We studied POS tagging in detail, though not enough for understanding the complex grammatical structures and ambiguities in sentences which most human languages comprise of.

Therefore, we need to learn techniques which can help us understand the grammatical structures of complex sentences. **Constituency parsing** and **Dependency parsing** can help us achieve that.

In This Session

Parsing techniques and algorithms to understand the grammatical structure of sentences. In **constituency parsing**, parse complex sentences using **grammar rules**. An alternative parsing technique is **dependency parsing**.

Topics covered in this session include:

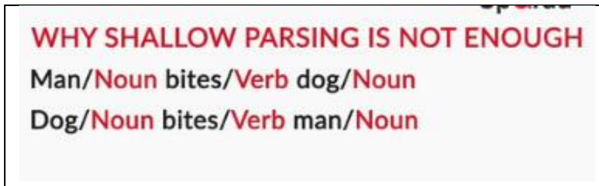
1. Constituency Parsing
   - Context Free Grammar (CFG)
   - Bottom-up and top-down parsing algorithms
   - Probabilistic CFG, or PCFG
2. Dependency Parsing
   - Fixed and free word order languages
   - Elements of dependency grammar
   - Overview of dependency parsing algorithms

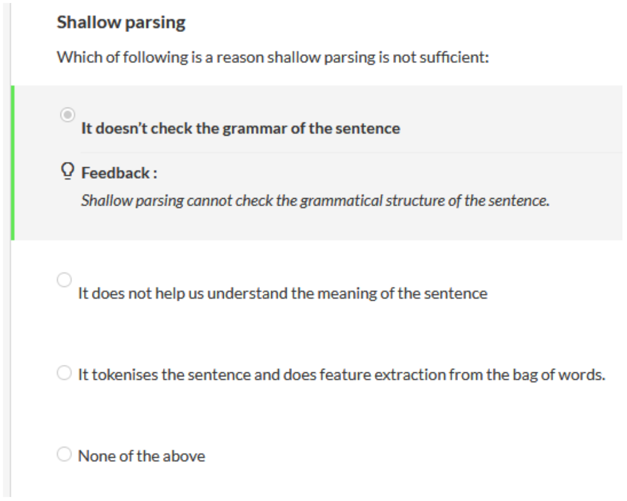
Dependency parsing is a recent topic in the field of NLP whose study involves a fairly deep understanding of the English grammar and parsing algorithms. We'll cover basics of dependency parsing techniques which should enable you to consume more sophisticated texts on the topic and introduce you to such additional resources for further study.

Why Shallow Parsing is Not Sufficient

Shallow parsing refers to fairly shallow levels of parsing such as POS tagging, chunking, etc. But such techniques cannot check grammatical structure of the sentence - whether a sentence is grammatically correct or understand the dependencies between words in a sentence.

Let's see why deeper parsing techniques are required.

 WHY SHALLOW PARSING IS NOT ENOUGH Man/Noun bites/Verb dog/Noun Dog/Noun bites/Verb man/Noun	Thus, POS tagging will although help us identify the linguistic role of the word in a sentence, it wouldn't enable us to understand how these words are related to each other in a sentence.



Shallow parsing

Which of following is a reason shallow parsing is not sufficient:

[✓ **It doesn't check the grammar of the sentence**]

Feedback :

Shallow parsing cannot check the grammatical structure of the sentence.

It does not help us understand the meaning of the sentence

It tokenises the sentence and does feature extraction from the bag of words.

None of the above

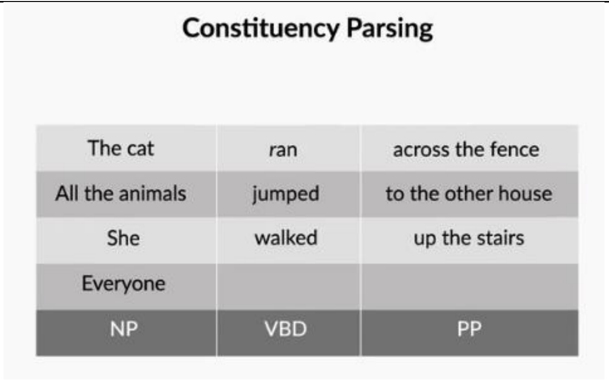
Constituency Grammars

To deal with the complexity and ambiguity of natural language, identify and define commonly observed grammatical patterns.

The first step in understanding grammar is to divide a sentence into **groups of words** called **constituents** based on their grammatical role in the sentence.

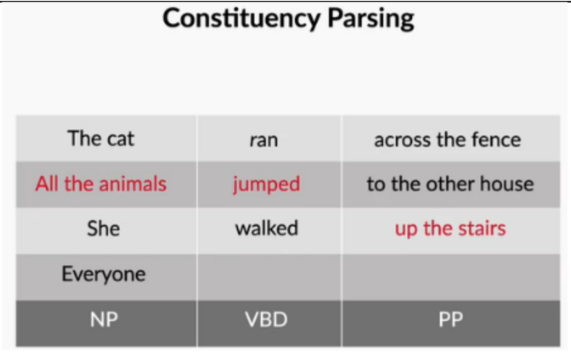
In sentence: “The fox ate the squirrel.”, Each underlined group of words represents a grammatical unit or a constituent - "The fox" represents a *noun phrase*, "ate" represents a *verb phrase*, "the squirrel" is another *noun phrase*.

Upcoming few lectures covers how **constituency parsers** can 'parse' the grammatical structure of sentences. Let's first understand the concept of **constituents**.



Constituency Parsing

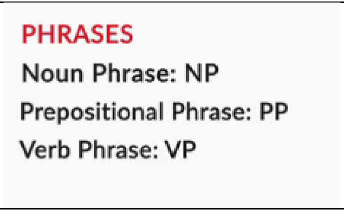
The cat	ran	across the fence
All the animals	jumped	to the other house
She	walked	up the stairs
Everyone		
NP	VBD	PP



Constituency Parsing

The cat	ran	across the fence
All the animals	jumped	to the other house
She	walked	up the stairs
Everyone		
NP	VBD	PP

Syntactically valid sentence although may not be semantically valid.



PHRASES
Noun Phrase: NP
Prepositional Phrase: PP
Verb Phrase: VP

To understand concept of constituencies, consider following two sentences:

- 'Ram   read   an article on data science'
- 'Shruti   ate   dinner'

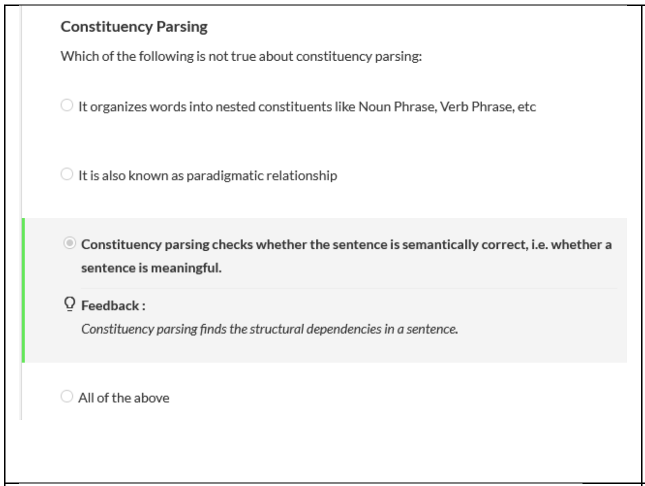
The underlined groups of words form a constituent (or a phrase). The rationale for clubbing these words in a single unit is provided by the notion of **substitutability,** i.e., a constituent can be replaced with another equivalent constituent while keeping the sentence **syntactically valid**.

For example, replacing the constituency 'an article on data science' (a noun phrase) with 'dinner' (another noun phrase) doesn't affect the syntax of the sentence, though the resultant sentence "Ram read dinner" is semantically meaningless.

Most common constituencies in English are **Noun Phrases (NP)**, **Verb Phrases (VP)**, and **Prepositional Phrases (PP)**. The following table summarises these phrases:

Type of Phrases	**Definition**	**Examples**
Noun Phrase	Has a primary noun and other words that modify it	A crazy white cat, the morning flight, a large elephant
Verb Phrase	Starts with a verb and other words that syntactically depend on it	saw an elephant, made a cake, killed the squirrel
Prepositional Phrase	Starts with a preposition and other words (usually a Noun Phrase) that syntactically depend on it	on the table, into the solar system, down the road, by the river

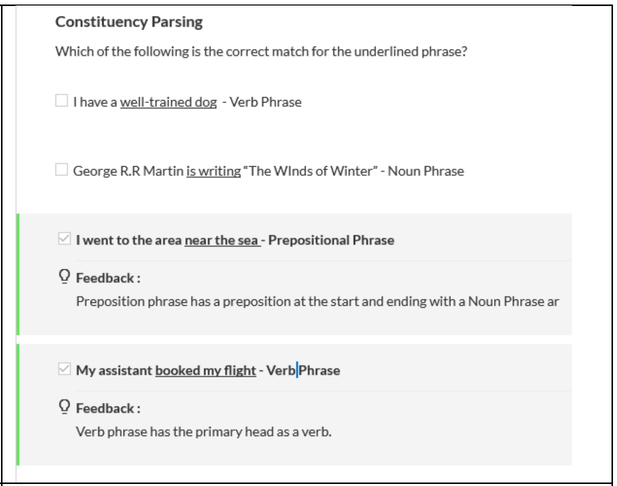
There are various other types of phrases, such as an **adverbial phrase**, a **nominal** (N), etc., though in most cases, we need to work with only the above three phrases along with the nominal.


Constituency Parsing

Which of the following is not true about constituency parsing:

- It organizes words into nested constituents like Noun Phrase, Verb Phrase, etc
- It is also known as paradigmatic relationship
- [✓ **Constituency parsing checks whether the sentence is semantically correct, i.e. whether a sentence is meaningful.**]

  Feedback :
  Constituency parsing finds the structural dependencies in a sentence.
- All of the above

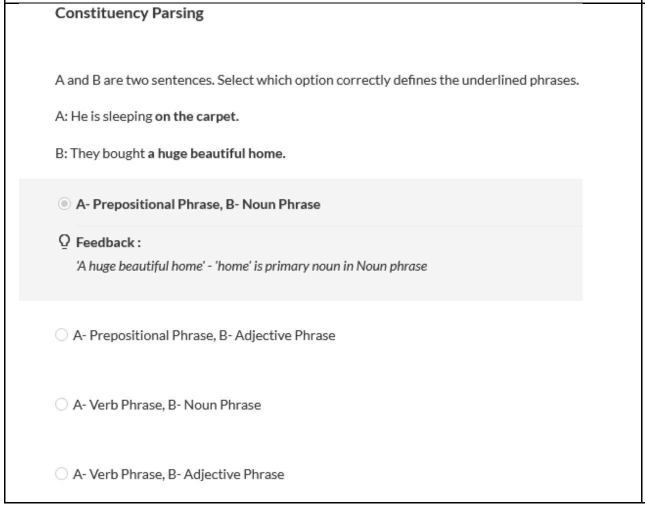

Constituency Parsing

Which of the following is the correct match for the underlined phrase?

- I have a well-trained dog - Verb Phrase
- George R.R Martin is writing "The WInds of Winter" - Noun Phrase
- [✓ **I went to the area near the sea - Prepositional Phrase**]

  Feedback :
  Preposition phrase has a preposition at the start and ending with a Noun Phrase ar
- [✓ **My assistant booked my flight - Verb Phrase**]

  Feedback :
  Verb phrase has the primary head as a verb.


Constituency Parsing

A and B are two sentences. Select which option correctly defines the underlined phrases.

A: He is sleeping **on the carpet.**

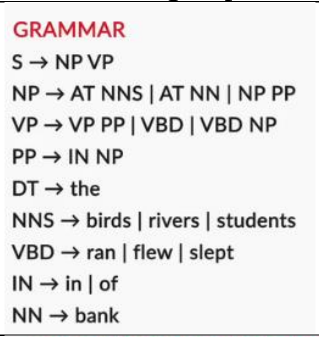
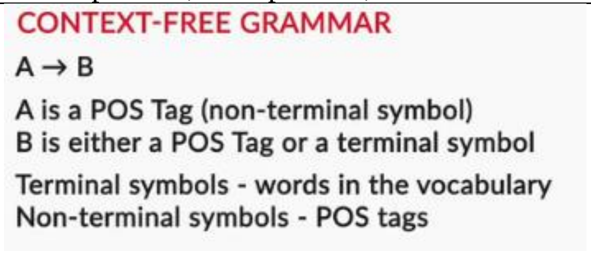
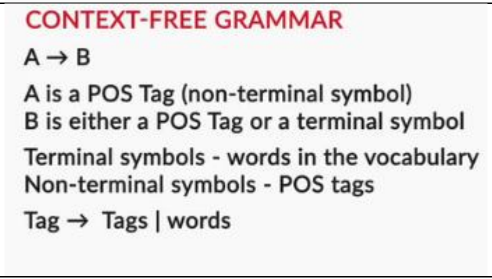
B: They bought **a huge beautiful home.**

- [✓ **A- Prepositional Phrase, B- Noun Phrase**]

  Feedback :
  'A huge beautiful home' - 'home' is primary noun in Noun phrase
- A- Prepositional Phrase, B- Adjective Phrase
- A- Verb Phrase, B- Noun Phrase
- A- Verb Phrase, B- Adjective Phrase

Context-Free Grammars

The most commonly used technique to organize sentences into constituencies is **Context-Free Grammars or CFGs.** CFGs define a set of **grammar rules** (or productions) which specify how words can be grouped to form constituents such as noun phrases, verb phrases, etc.

 GRAMMAR S → NP VP NP → AT NNS | AT NN | NP PP VP → VP PP | VBD | VBD NP PP → IN NP DT → the NNS → birds | rivers | students VBD → ran | flew | slept IN → in | of NN → bank	 CONTEXT-FREE GRAMMAR A → B A is a POS Tag (non-terminal symbol) B is either a POS Tag or a terminal symbol Terminal symbols - words in the vocabulary Non-terminal symbols - POS tags
 CONTEXT-FREE GRAMMAR A → B A is a POS Tag (non-terminal symbol) B is either a POS Tag or a terminal symbol Terminal symbols - words in the vocabulary Non-terminal symbols - POS tags Tag → Tags | words	

A context-free grammar is a series of production rules.

Let's understand production rules. Below production rule says that a noun phrase can be formed using either a determiner (DT) followed by a noun (N) or a noun phrase (NP) followed by a prepositional phrase (PP).:

NP -> DT N | NP PP

Some example phrases that follow this production rule are:

- The/**DT** man/**N**
- The/**DT** man/**N** over/**P** the/**DT** bridge/**N**

Both above are noun phrases NP. *The man* is a noun phrase that follows the first rule: NP -> DT N.

The second phrase (*The man over the bridge)* follows the second rule: NP -> NP PP

It has a noun phrase (*The man)* and a prepositional phrase (*over the bridge).*

In this way, using grammar rules, parse sentences into different constituents. In general, any production rule can be written as A -> B C, where A is a **non-terminal** symbol (NP, VP, N etc.) and B and C are either non-terminals or **terminal** symbols (i.e. words in vocabulary such as flight, man etc.).

Some other examples of commonly observed production rules in English grammar are provided in the table below. Note that a **nominal** (Nom) refers to an entity such as morning, flight etc. which commonly follows the rule Nominal > Nominal Noun. There is a subtle difference and a significant overlap between a nominal (Nom) and a noun (NN); read more about it here, though need not worry much about these nuances in this course.

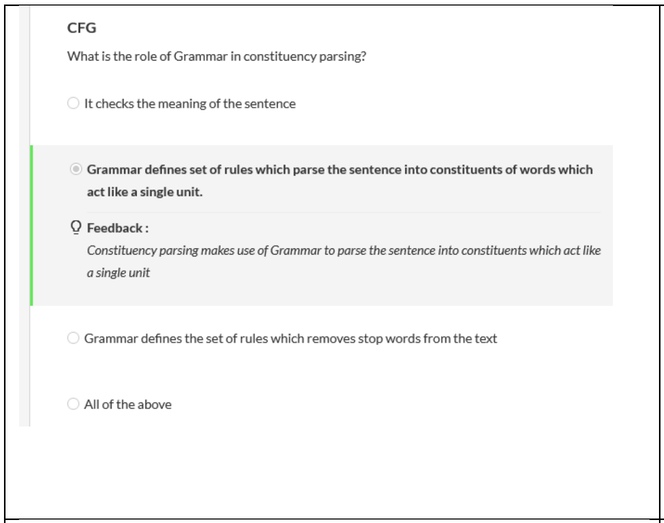
The symbol S represents an entire sentence.

Production Rules

Production Rule	**Example**
S > NP VP	he + swam
NP > Pronoun | NP PP | DT Nom	she | a man + across the river | a + river
VP > VP PP | VBD | VP NP	swam + across the river | enjoyed | ate + the squirrel

Two broad approaches for parsing sentences using CFGs:

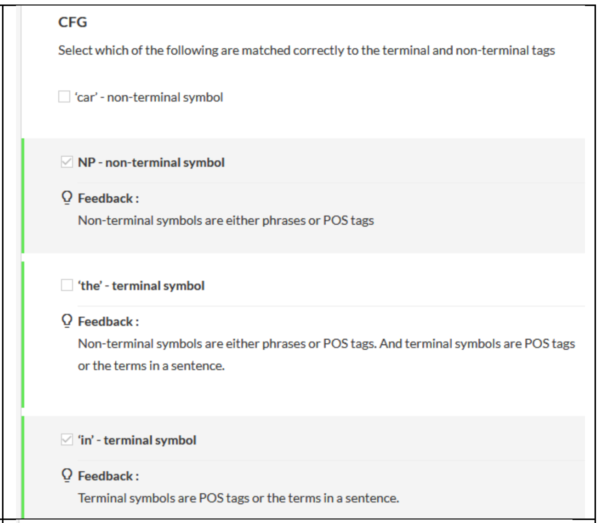
1. **Top-down**: Start from the starting symbol S and produce each word in the sentence
2. **Bottom-up:** Start from the individual words and reduce them to the sentence S


CFG

What is the role of Grammar in constituency parsing?

- It checks the meaning of the sentence
- [✓ **Grammar defines set of rules which parse the sentence into constituents of words which act like a single unit.**]

  Feedback : *Constituency parsing makes use of Grammar to parse the sentence into constituents which act like a single unit*
- Grammar defines the set of rules which removes stop words from the text
- All of the above


CFG

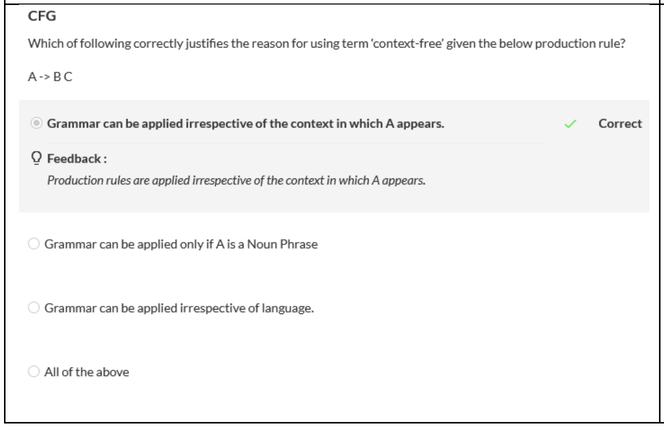
Select which of the following are matched correctly to the terminal and non-terminal tags

- 'car' - non-terminal symbol
- [✓ **NP - non-terminal symbol**]

  Feedback : Non-terminal symbols are either phrases or POS tags
- **'the' - terminal symbol**

  Feedback : Non-terminal symbols are either phrases or POS tags. And terminal symbols are POS tags or the terms in a sentence.
- [✓ **'in' - terminal symbol**]

  Feedback : Terminal symbols are POS tags or the terms in a sentence.

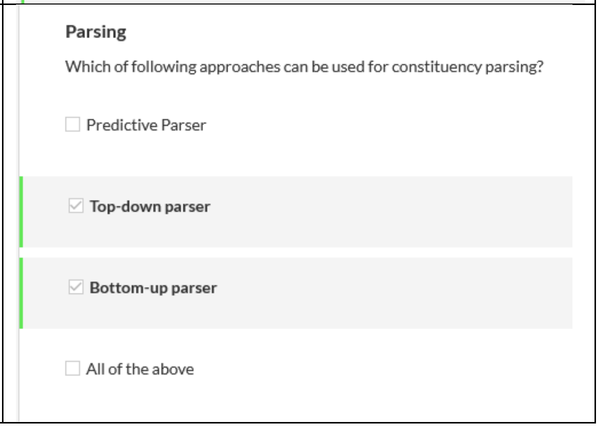

CFG

Which of following correctly justifies the reason for using term 'context-free' given the below production rule?

A -> B C

- [✓ **Grammar can be applied irrespective of the context in which A appears.**] ✓ **Correct**

  Feedback : *Production rules are applied irrespective of the context in which A appears.*
- Grammar can be applied only if A is a Noun Phrase
- Grammar can be applied irrespective of language.
- All of the above


Parsing

Which of following approaches can be used for constituency parsing?

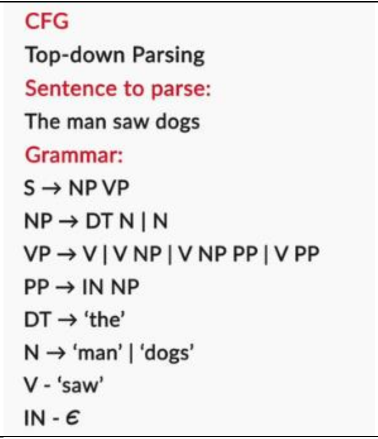
- Predictive Parser
- [✓ **Top-down parser**]
- [✓ **Bottom-up parser**]
- All of the above

Top-Down Parsing

Until now, basics of phrases, CFGs and how CFGs can be used for parsing sentences. Now algorithms for parsing. 2 broad approaches for parsing:

1. **Top-down**: Start from the starting symbol S and produce each word in the sentence
2. **Bottom-up:** Start from the individual words and reduce them to the sentence S

CFGs define what are called grammar or **production rules** of the form A > BC. Both parsing techniques use these rules to create parse trees.



CFG

Top-down Parsing

Sentence to parse:

The man saw dogs

Grammar:

S → NP VP

NP → DT N | N

VP → V | V NP | V NP PP | V PP

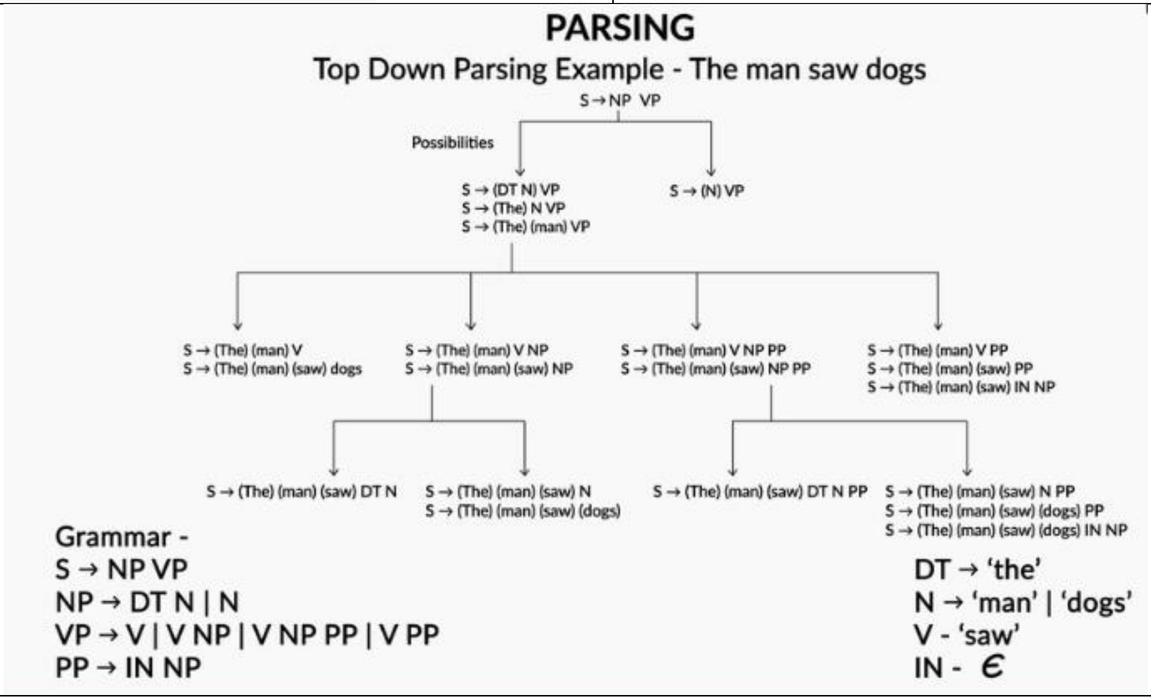
PP → IN NP

DT → 'the'

N → 'man' | 'dogs'

V - 'saw'

IN - ϵ



PARSING

Top Down Parsing Example - The man saw dogs

S→NP VP
Possibilities
S → (DT N) VP
S → (The) N VP
S → (The) (man) VP
S → (N) VP
S → (The) (man) V
S → (The) (man) (saw) dogs
S → (The) (man) V NP
S → (The) (man) (saw) NP
S → (The) (man) V NP PP
S → (The) (man) (saw) NP PP
S → (The) (man) V PP
S → (The) (man) (saw) PP
S → (The) (man) (saw) IN NP
S → (The) (man) (saw) DT N
S → (The) (man) (saw) N
S → (The) (man) (saw) (dogs)
S → (The) (man) (saw) DT N PP
S → (The) (man) (saw) N PP
S → (The) (man) (saw) (dogs) PP
S → (The) (man) (saw) (dogs) IN NP

Grammar -

S → NP VP

NP → DT N | N

VP → V | V NP | V NP PP | V PP

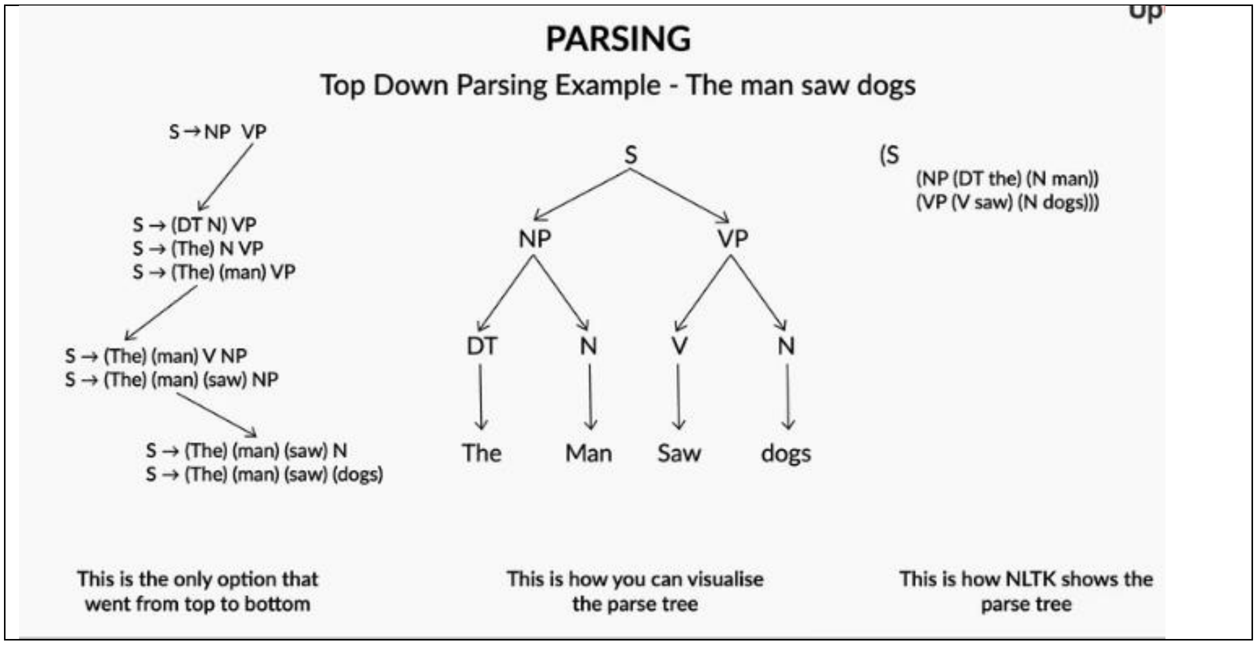
PP → IN NP

DT → 'the'

N → 'man' | 'dogs'

V - 'saw'

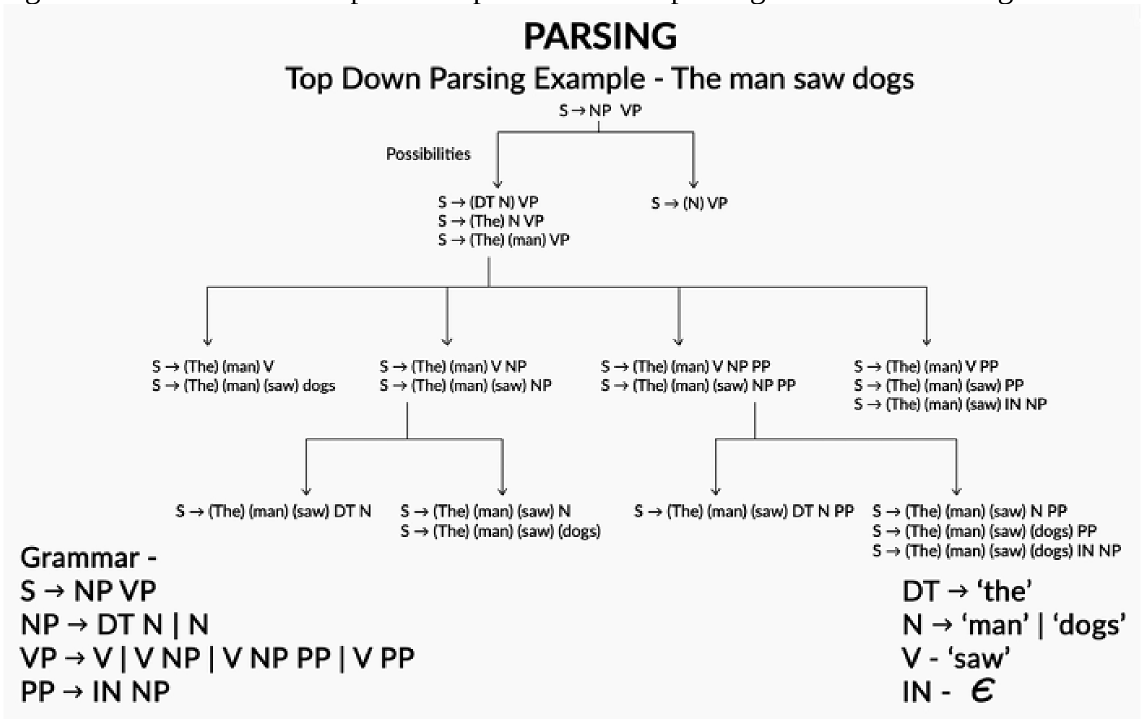
IN - ϵ



Up
PARSING
Top Down Parsing Example - The man saw dogs
S → NP VP
S → (DT N) VP
S → (The) N VP
S → (The) (man) VP
S → (The) (man) V NP
S → (The) (man) (saw) NP
S → (The) (man) (saw) N
S → (The) (man) (saw) (dogs)
S
NP
VP
DT
N
V
N
The
Man
Saw
dogs
(S
(NP (DT the) (N man))
(VP (V saw) (N dogs)))
This is the only option that went from top to bottom
This is how you can visualise the parse tree
This is how NLTK shows the parse tree

Top-down parsing starts with the start symbol S at the top and uses the production rules to parse each word one by one. Parse until all the words have been allocated to some production rule.

The figure below shows all the paths the parser tried for parsing "The man saw dogs".



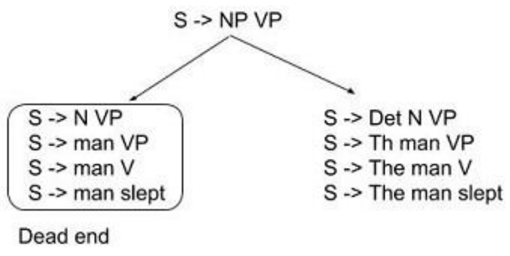
PARSING
Top Down Parsing Example - The man saw dogs
S → NP VP
Possibilities
S → (DT N) VP
S → (The) N VP
S → (The) (man) VP
S → (N) VP
S → (The) (man) V
S → (The) (man) (saw) dogs
S → (The) (man) V NP
S → (The) (man) (saw) NP
S → (The) (man) V NP PP
S → (The) (man) (saw) NP PP
S → (The) (man) V PP
S → (The) (man) (saw) PP
S → (The) (man) (saw) IN NP
S → (The) (man) (saw) DT N
S → (The) (man) (saw) N
S → (The) (man) (saw) (dogs)
S → (The) (man) (saw) DT N PP
S → (The) (man) (saw) N PP
S → (The) (man) (saw) (dogs) PP
S → (The) (man) (saw) (dogs) IN NP
Grammar -
S → NP VP
NP → DT N | N
VP → V | V NP | V NP PP | V PP
PP → IN NP
DT → 'the'
N → 'man' | 'dogs'
V - 'saw'
IN - ϵ

Top-down Parsed Tree

In the process, we often encounter **dead ends**, i.e. points where no production rule can be applied to produce a right-hand side from the left-hand side. In such cases, the algorithm needs to **backtrack** and try some other rule.

Let's understand this through an example of a simple sentence and production rules.

Sentence: The man slept
Grammar:
S -> NP VP
NP -> N| Det N
VP -> V | V NP
Det -> ‘the’
N -> ‘man’
V -> ‘slept’

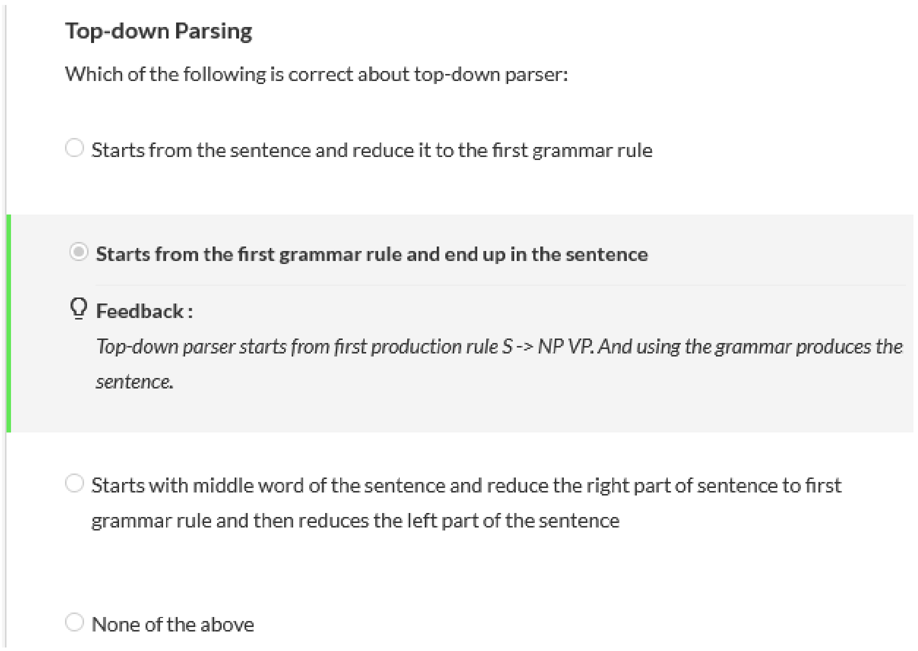


Top-down Parsing

The sentence couldn’t be parsed using the left side of the tree since it reached a dead end. On the other hand, the grammar used on the right side is able to parse the sentence completely.

The NLTK library in Python will show the parse tree as:

```
(S
        (NP (DT The) (N man))
        (VP (V slept)))
```



Top-down Parsing

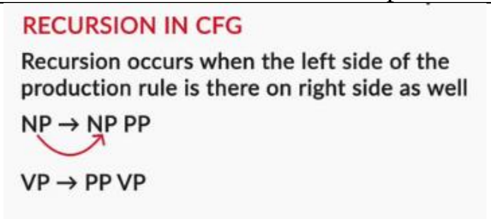
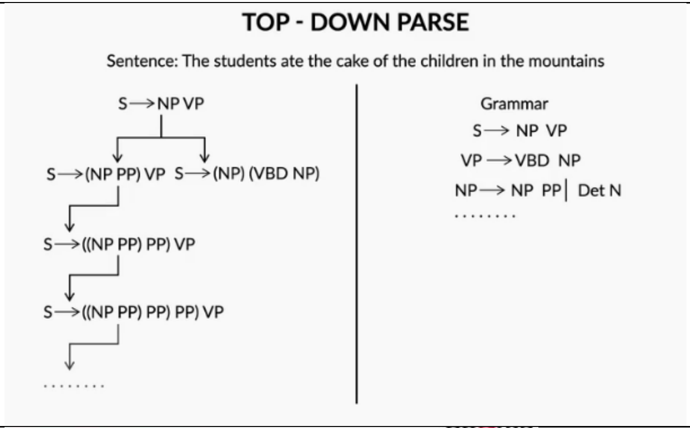
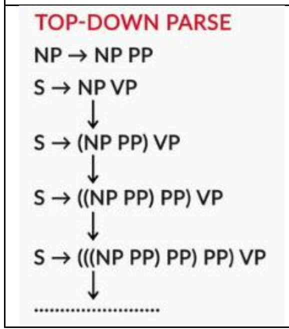
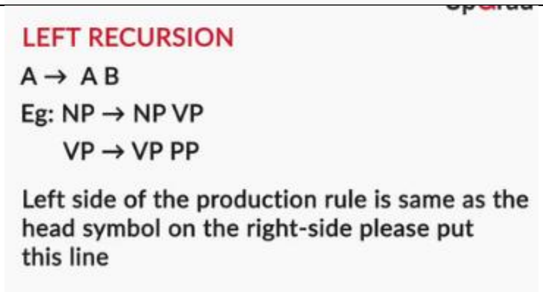
Which of the following is correct about top-down parser:

- Starts from the sentence and reduce it to the first grammar rule
- [✓ **Starts from the first grammar rule and end up in the sentence**]

  Feedback :

  Top-down parser starts from first production rule S -> NP VP. And using the grammar produces the sentence.
- Starts with middle word of the sentence and reduce the right part of sentence to first grammar rule and then reduces the left part of the sentence
- None of the above

You learnt how to parse a sentence in the previous lecture. But top-down parsers have a specific limitation. Next is where the top-down parser fails.

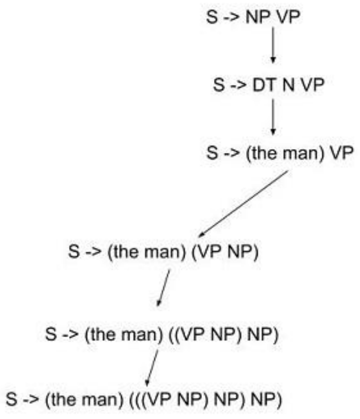
 RECURSION IN CFG Recursion occurs when the left side of the production rule is there on right side as well $NP \rightarrow NP\ PP$ $VP \rightarrow PP\ VP$	 TOP - DOWN PARSE Sentence: The students ate the cake of the children in the mountains S→NP VP S→(NP PP) VP S→(NP) (VBD NP) S→((NP PP) PP) VP S→((NP PP) PP) PP) VP .... Grammar S→ NP VP VP →VBD NP NP→ NP PP | Det N ....
 TOP-DOWN PARSE NP → NP PP S → NP VP S → (NP PP) VP S → ((NP PP) PP) VP S → (((NP PP) PP) PP) VP	 LEFT RECURSION $A \rightarrow A\ B$ Eg: $NP \rightarrow NP\ VP$ $VP \rightarrow VP\ PP$ Left side of the production rule is same as the head symbol on the right-side please put this line

Recursive descent parser is a type of top-down parser, though the terms are sometimes used equivalently.

Problem of **left-recursion** using an example:
Sentence: The man saw a car.

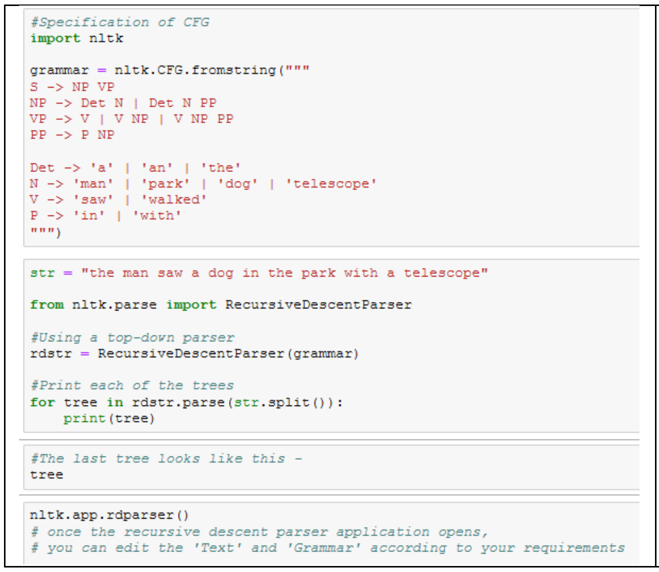
Grammar:
S -> NP VP
NP -> DT N
VP -> VP NP| V
DT -> the
N -> man

The top-down parse structure of the sentence will look as follows:



Recursion in Top-down parsing

The rule VP -> VP NP runs into an **infinite loop** and no parse tree will be obtained. This is the problem of left recursion. This problem can be resolved using the bottom-up approach.

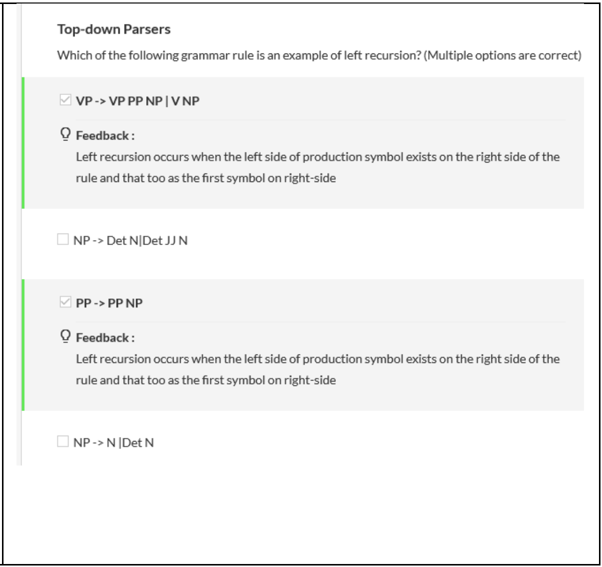


```
#Specification of CFG
import nltk

grammar = nltk.CFG.fromstring("""
S -> NP VP
NP -> Det N | Det N PP
VP -> V | V NP | V NP PP
PP -> P NP

Det -> 'a' | 'an' | 'the'
N -> 'man' | 'park' | 'dog' | 'telescope'
V -> 'saw' | 'walked'
P -> 'in' | 'with'
""")
```

```
str = "the man saw a dog in the park with a telescope"

from nltk.parse import RecursiveDescentParser

#Using a top-down parser
rdstr = RecursiveDescentParser(grammar)

#Print each of the trees
for tree in rdstr.parse(str.split()):
    print(tree)
```

```
#The last tree looks like this -
tree
```

```
nltk.app.rdparser()
# once the recursive descent parser application opens,
# you can edit the 'Text' and 'Grammar' according to your requirements
```



Top-down Parsers

Which of the following grammar rule is an example of left recursion? (Multiple options are correct)

[✓ VP -> VP PP NP | V NP]

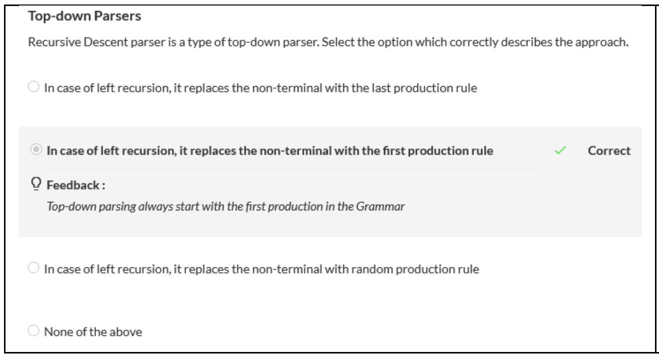
Feedback :
Left recursion occurs when the left side of production symbol exists on the right side of the rule and that too as the first symbol on right-side

☐ NP -> Det N|Det JJ N

[✓ PP -> PP NP]

Feedback :
Left recursion occurs when the left side of production symbol exists on the right side of the rule and that too as the first symbol on right-side

☐ NP -> N |Det N

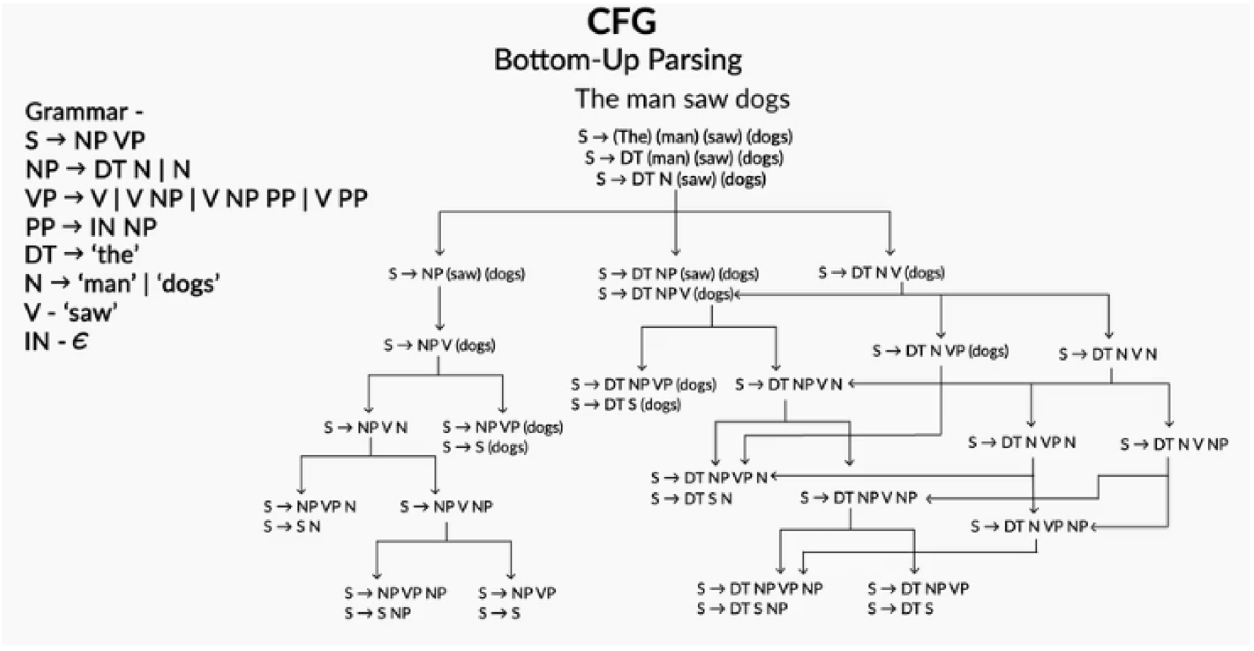
 **Top-down Parsers** Recursive Descent parser is a type of top-down parser. Select the option which correctly describes the approach. ○ In case of left recursion, it replaces the non-terminal with the last production rule [✓ **In case of left recursion, it replaces the non-terminal with the first production rule**] ✓ Correct **Feedback :** *Top-down parsing always start with the first production in the Grammar* ○ In case of left recursion, it replaces the non-terminal with random production rule ○ None of the above	

Additional Reading

- Although the naive implementation of the recursive descent parser suffers from the problem of left-recursion, alternate algorithms have been suggested which can handle this problem. The algorithms use predictive techniques to 'look ahead' and avoid going down the infinite loop. Read more about it here.

Bottom-Up Parsing

Previous segment had top-down parser and problem of left-recursion. This segment has another approach to parse a sentence- the **bottom-up** approach.



Bottom-up approach reduces each terminal word to a production rule, i.e. reduces the right-hand-side of the grammar to the left-hand-side. It continues the reduction process until the entire sentence has been reduced to the start symbol S.

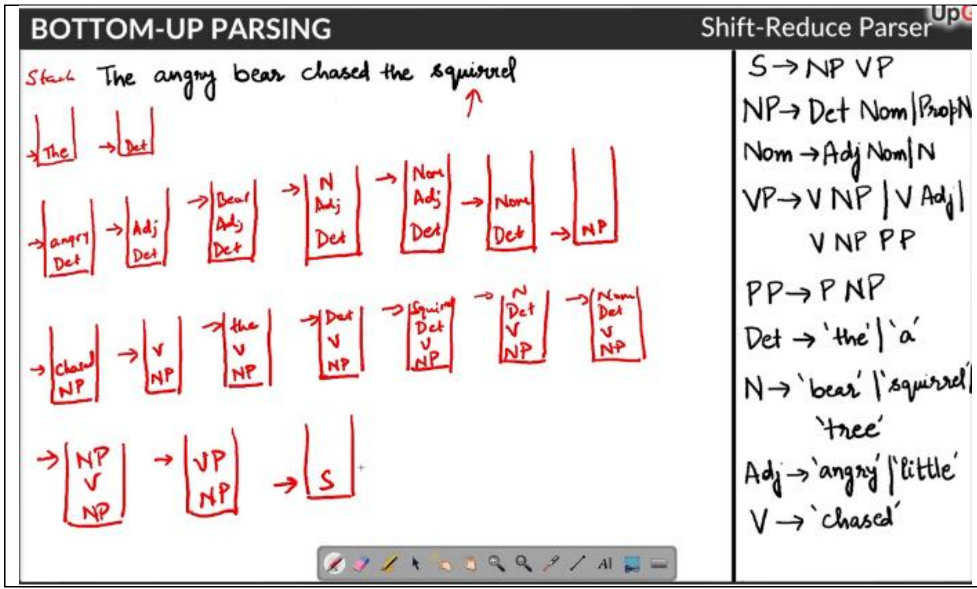
There are many types of bottom-up parsers. The **shift-reduce parser** is one of the most commonly used bottom-up parsing algorithms. Let's consider the following sentence and grammar:

Sentence: The angry bear chased the squirrel

Grammar:

```
S -> NP VP
NP -> Det Nom | PropN
Nom -> Adj Nom| N
VP -> V NP | V Adj|V NP PP
PP -> P NP
Det -> 'the' | 'a'
N -> 'bear' | 'squirrel'
Adj -> 'angry'| 'little'
V -> 'chased'
```

Following lecture explains how bottom-up parsing is done using the **shift-reduce algorithm.** Example sentence is "The angry bear chased the squirrel" and use some production rules to implement the shift-reduce algorithm.

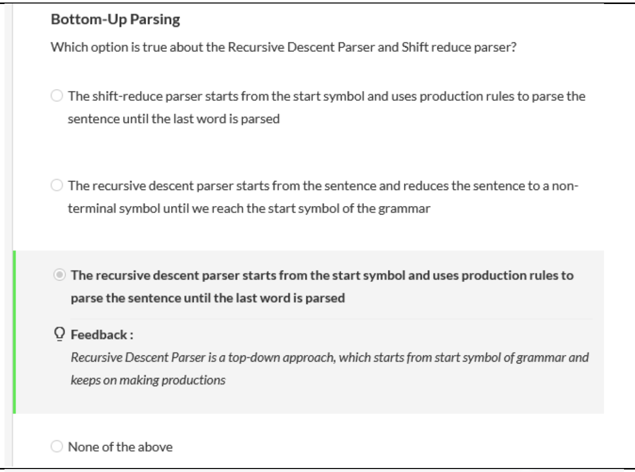
 BOTTOM-UP PARSING Shift-Reduce Parser Stack The angry bear chased the squirrel S → NP VP NP → Det Nom | PropN Nom → Adj Nom | N VP → V NP | V Adj | V NP PP PP → P NP Det → 'the' | 'a' N → 'bear' | 'squirrel' | 'tree' Adj → 'angry' | 'little' V → 'chased'	Shift-reduce parser parses the words of the sentence one-by-one either by **shifting** a word to the stack or **reducing** the stack by using the production rules.

Next is Python implementation of the shift-reduce parser.

```
#Specification of CFG
import nltk

grammar = nltk.CFG.fromstring("""
S -> NP VP
NP -> Det N | Det N PP
VP -> V | V NP | V NP PP
PP -> P NP

Det -> 'a' | 'an' | 'the'
N -> 'man' | 'park' | 'dog' | 'telescope'
V -> 'saw' | 'walked'
P -> 'in' | 'with'
""")
```

```
str = "the man saw a dog in the park with a telescope"

# ShiftReduce Parser is a bottom up parser
from nltk.parse import ShiftReduceParser
srp = ShiftReduceParser(grammar)

for t in srp.parse(str.split()):
    print(t)

# Shift Reduce parser tries to aggregate the string to the start symbol,
# since it's a bottom-up parser
# It is not able to find the parse tree even if it exists
```

```
nltk.app.srparser()
# once the shift reduce parser application opens,
# you can edit the 'Text' and 'Grammar' according to your requirements
```

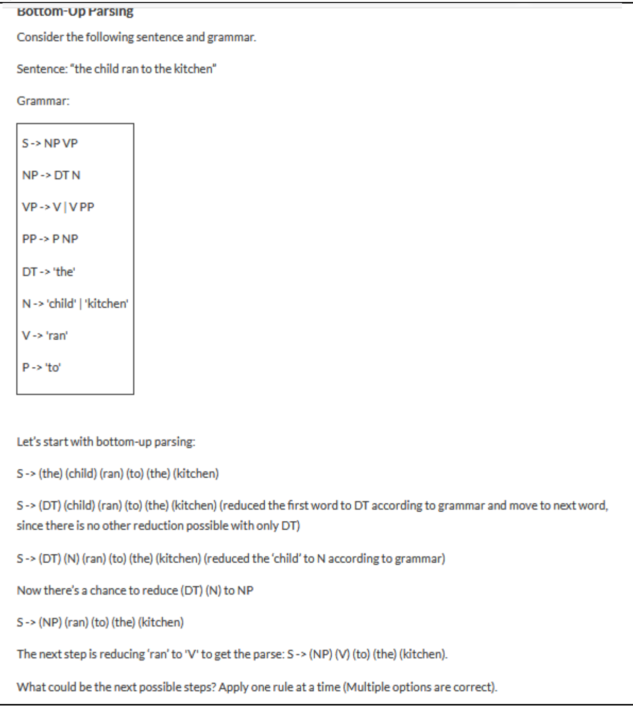


Bottom-Up Parsing

Which option is true about the Recursive Descent Parser and Shift reduce parser?

- The shift-reduce parser starts from the start symbol and uses production rules to parse the sentence until the last word is parsed
- The recursive descent parser starts from the sentence and reduces the sentence to a non-terminal symbol until we reach the start symbol of the grammar
- [✓ **The recursive descent parser starts from the start symbol and uses production rules to parse the sentence until the last word is parsed**]

  Feedback :
  Recursive Descent Parser is a top-down approach, which starts from start symbol of grammar and keeps on making productions
- None of the above



Bottom-Up Parsing

Consider the following sentence and grammar.

Sentence: "the child ran to the kitchen"

Grammar:

```
S -> NP VP
NP -> DT N
VP -> V | V PP
PP -> P NP
DT -> 'the'
N -> 'child' | 'kitchen'
V -> 'ran'
P -> 'to'
```

Let's start with bottom-up parsing:

S -> (the) (child) (ran) (to) (the) (kitchen)

S -> (DT) (child) (ran) (to) (the) (kitchen) (reduced the first word to DT according to grammar and move to next word, since there is no other reduction possible with only DT)

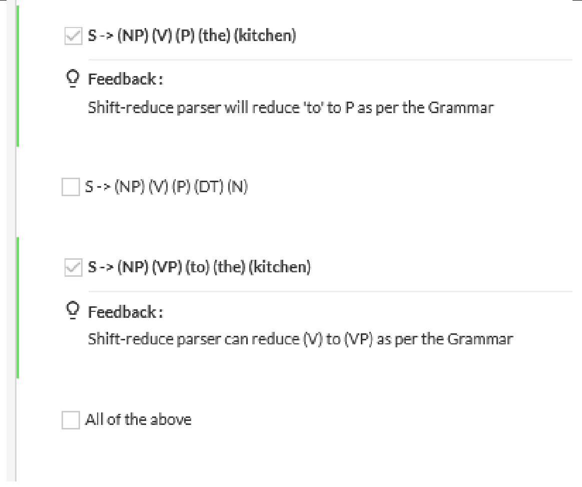
S -> (DT) (N) (ran) (to) (the) (kitchen) (reduced the 'child' to N according to grammar)

Now there's a chance to reduce (DT) (N) to NP

S -> (NP) (ran) (to) (the) (kitchen)

The next step is reducing 'ran' to 'V' to get the parse: S -> (NP) (V) (to) (the) (kitchen).

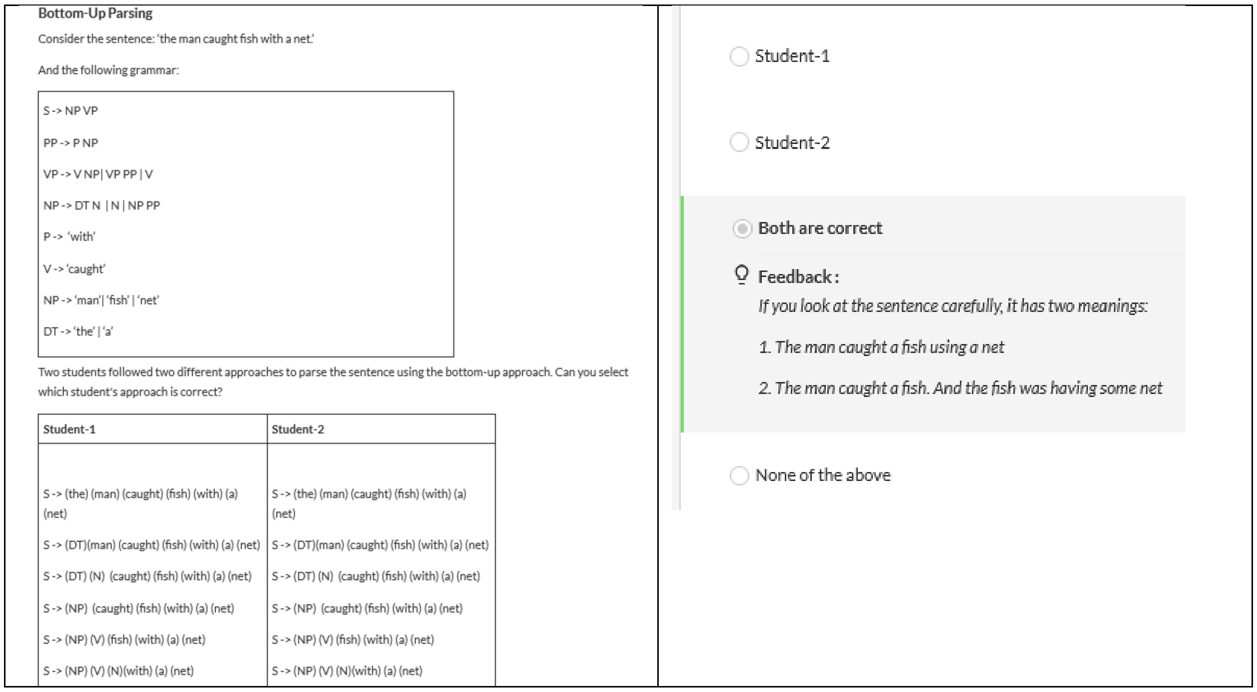
What could be the next possible steps? Apply one rule at a time (Multiple options are correct).



- [✓ **S -> (NP) (V) (P) (the) (kitchen)**]

  Feedback :
  Shift-reduce parser will reduce 'to' to P as per the Grammar
- S -> (NP) (V) (P) (DT) (N)
- [✓ **S -> (NP) (VP) (to) (the) (kitchen)**]

  Feedback :
  Shift-reduce parser can reduce (V) to (VP) as per the Grammar
- All of the above


Bottom-Up Parsing

Consider the sentence: 'the man caught fish with a net.'

And the following grammar:

```
S -> NP VP
PP -> P NP
VP -> V NP| VP PP | V
NP -> DT N  | N | NP PP
P -> 'with'
V -> 'caught'
NP -> 'man'| 'fish' | 'net'
DT -> 'the' | 'a'
```

Two students followed two different approaches to parse the sentence using the bottom-up approach. Can you select which student's approach is correct?

Student-1	Student-2
S -> (the) (man) (caught) (fish) (with) (a) (net)	S -> (the) (man) (caught) (fish) (with) (a) (net)
S -> (DT)(man) (caught) (fish) (with) (a) (net)	S -> (DT)(man) (caught) (fish) (with) (a) (net)
S -> (DT) (N) (caught) (fish) (with) (a) (net)	S -> (DT) (N) (caught) (fish) (with) (a) (net)
S -> (NP) (caught) (fish) (with) (a) (net)	S -> (NP) (caught) (fish) (with) (a) (net)
S -> (NP) (V) (fish) (with) (a) (net)	S -> (NP) (V) (fish) (with) (a) (net)
S -> (NP) (V) (N)(with) (a) (net)	S -> (NP) (V) (N)(with) (a) (net)

○ Student-1

○ Student-2

[✓ Both are correct]

Feedback:

If you look at the sentence carefully, it has two meanings:

1. The man caught a fish using a net

2. The man caught a fish. And the fish was having some net

○ None of the above

Ambiguities in Parse Trees

Sentence "The man caught fish with a net" can have two valid parses. In one parse, the prepositional phrase "with a net" is associated with "The man" whereas in another parse it is associated with "fish".

From common sense, its known that "with a net" is associated with "The man", i.e. it is the man who has the net, not the fish. But how to make the algorithm use this common sense?

Our common sense arises from the fact that it is *more likely* that the man has the net, not the fish. As always, the way to make algorithms understand the concept of *likelihood* is to use **probabilistic techniques**.

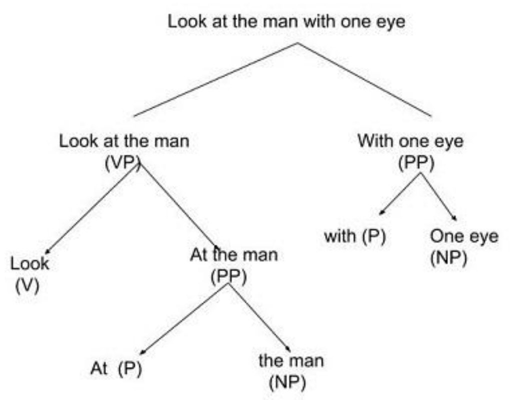
Next segment is about **probabilistic context-free grammars or PCFGs.**

Probabilistic CFG

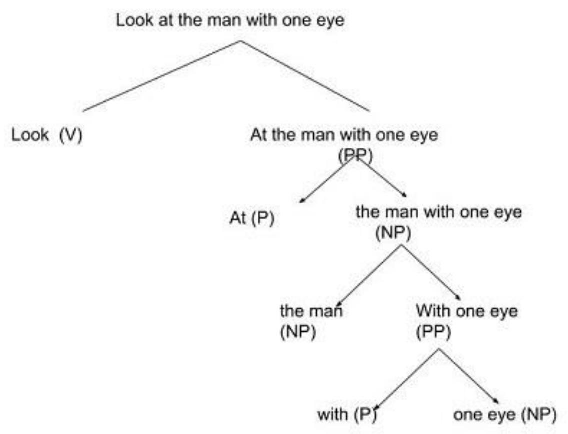
Until now, sentences resulted in a single parse tree. But what if an ambiguous sentence and grammar could lead to multiple parse trees?

For sentence: “Look at the man with one eye.” , two possible meanings of the sentence:

Look at the man using only one of your eyes



Look at the man who has one eye

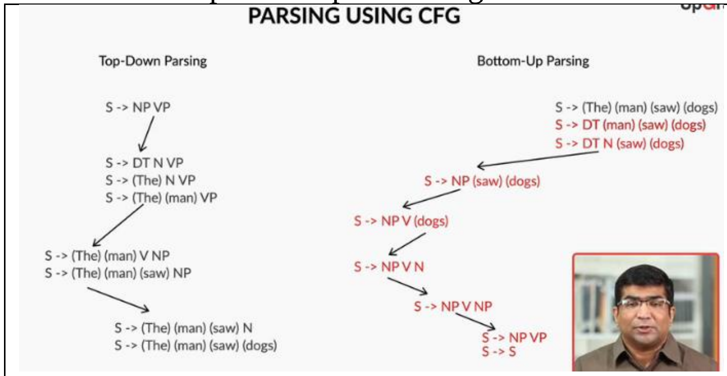


Look at the man with one eye
Look (V)
At the man with one eye (PP)
At (P)
the man with one eye (NP)
the man (NP)
With one eye (PP)
with (P)
one eye (NP)

In previous segment, we built two parse trees for another ambiguous sentence - "the man caught fish with a net".

In general, since natural languages are inherently ambiguous (at least for computers to understand), there are often cases where multiple parse trees are possible. In such cases, we need a way to make the algorithms figure out the *most likely* parse tree.

Few more examples to explain ambiguities in sentences and techniques to solve them.



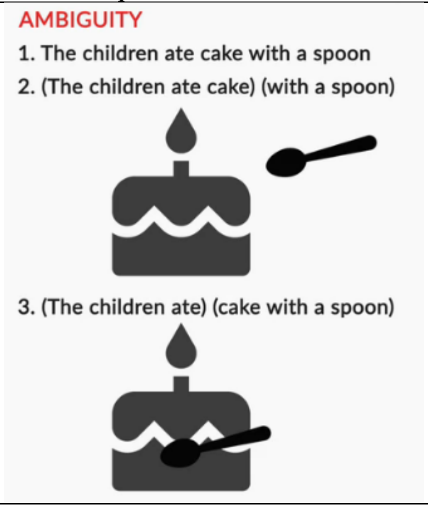
PARSING USING CFG

Top-Down Parsing

S -> NP VP
S -> DT N VP
S -> (The) N VP
S -> (The) (man) VP
S -> (The) (man) V NP
S -> (The) (man) (saw) NP
S -> (The) (man) (saw) N
S -> (The) (man) (saw) (dogs)

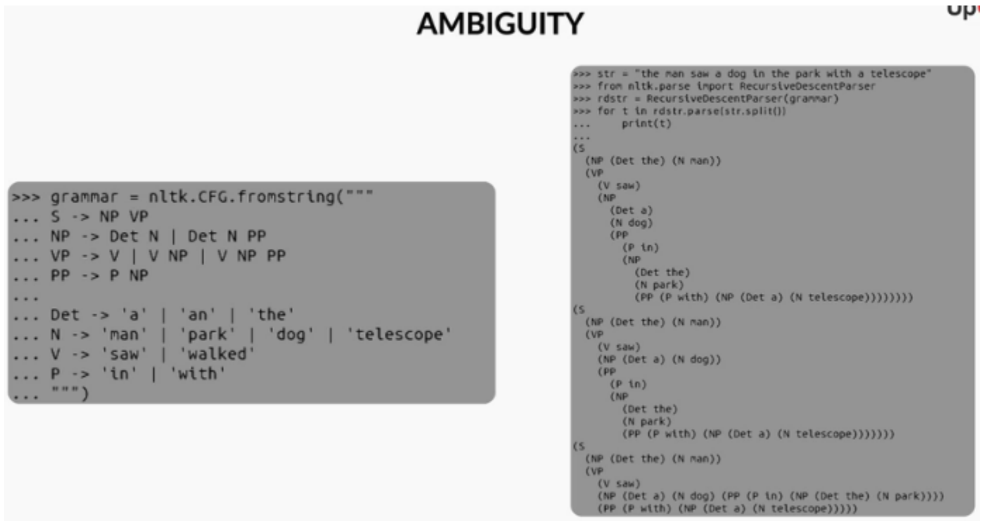
Bottom-Up Parsing

S -> (The) (man) (saw) (dogs)
S -> DT (man) (saw) (dogs)
S -> DT N (saw) (dogs)
S -> NP (saw) (dogs)
S -> NP V (dogs)
S -> NP V N
S -> NP V NP
S -> NP VP
S -> S



AMBIGUITY

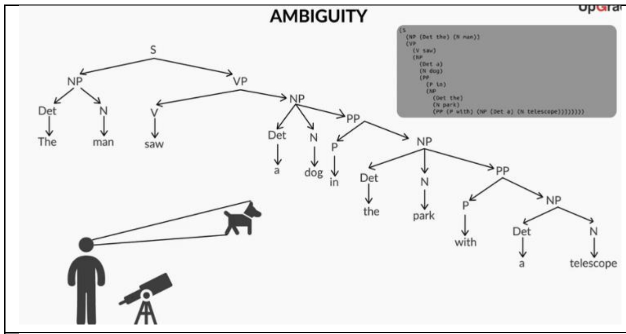
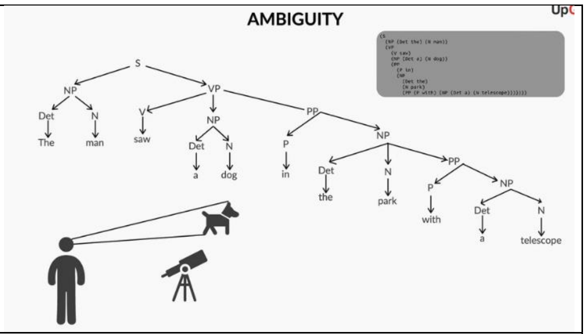
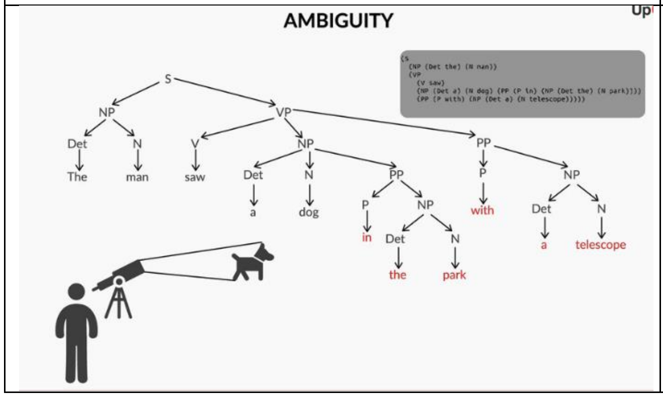
1. The children ate cake with a spoon
2. (The children ate cake) (with a spoon)
3. (The children ate) (cake with a spoon)



AMBIGUITY

```
>>> grammar = nltk.CFG.fromstring("""
... S -> NP VP
... NP -> Det N | Det N PP
... VP -> V | V NP | V NP PP
... PP -> P NP
...
... Det -> 'a' | 'an' | 'the'
... N -> 'man' | 'park' | 'dog' | 'telescope'
... V -> 'saw' | 'walked'
... P -> 'in' | 'with'
... """)
```

```
>>> str = "the man saw a dog in the park with a telescope"
>>> from nltk.parse import RecursiveDescentParser
>>> rdstr = RecursiveDescentParser(grammar)
>>> for t in rdstr.parse(str.split())
...     print(t)
...
(S
  (NP (Det the) (N man))
  (VP
    (V saw)
    (NP
      (Det a)
      (N dog)
      (PP
        (P in)
        (NP
          (Det the)
          (N park)
          (PP (P with) (NP (Det a) (N telescope))))))))
(S
  (NP (Det the) (N man))
  (VP
    (V saw)
    (NP (Det a) (N dog))
    (PP
      (P in)
      (NP
        (Det the)
        (N park)
        (PP (P with) (NP (Det a) (N telescope)))))))
(S
  (NP (Det the) (N man))
  (VP
    (V saw)
    (NP (Det a) (N dog) (PP (P in) (NP (Det the) (N park))))
    (PP (P with) (NP (Det a) (N telescope)))))
```

 AMBIGUITY S NP VP Det N The man V saw NP Det a N dog PP P in NP Det the N park PP P with NP Det a N telescope	 AMBIGUITY S NP VP Det N The man V saw NP Det a N dog PP P in NP Det the N park PP P with NP Det a N telescope
 AMBIGUITY S NP VP Det N The man V saw NP Det a N dog PP P in NP Det the N park PP P with NP Det a N telescope	

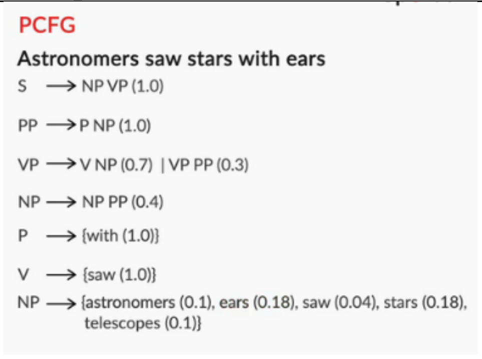
Examples of sentences where ambiguities lead to multiple parse trees. Both top-down and bottom-up techniques will generate multiple parse trees. None of these trees is *grammatically incorrect*, but some of these are *improbable* to occur in normal conversations. To identify which of these trees is the most probable, we use the notion of **probability.**

Probabilistic Context-Free Grammars (PCFGs) are used when we want to find the most probable parsed structure of the sentence. PCFGs are grammar rules along with probabilities associated with each production rule. For example, an example production rule is as follows:

NP -> Det N (0.5) | N (0.3) |N PP (0.2)

It means that the probability of an NP breaking down to a ‘Det N’ is 0.50, to an 'N' is 0.30 and to an ‘N PP’ is 0.20. Sum of probabilities is 1.00.

Concept of PCFG in detail.



PCFG

Astronomers saw stars with ears

S ⟶ NP VP (1.0)

PP ⟶ P NP (1.0)

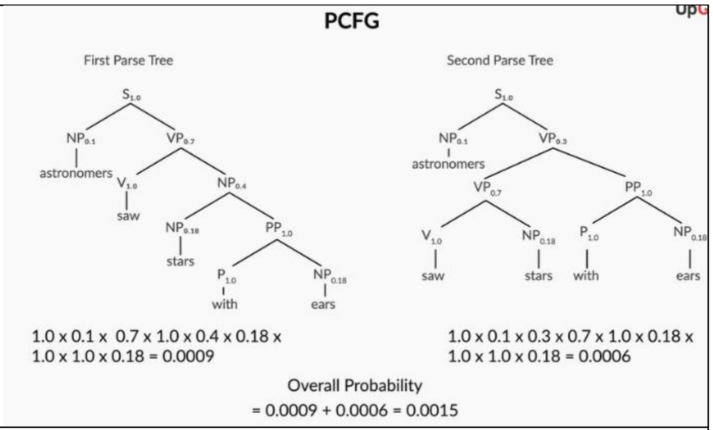
VP ⟶ V NP (0.7) | VP PP (0.3)

NP ⟶ NP PP (0.4)

P ⟶ {with (1.0)}

V ⟶ {saw (1.0)}

NP ⟶ {astronomers (0.1), ears (0.18), saw (0.04), stars (0.18), telescopes (0.1)}

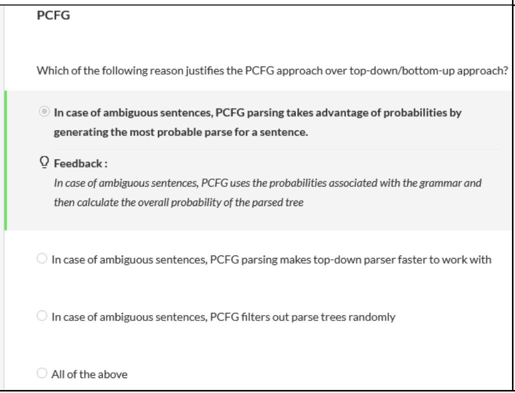


PCFG

1.0 x 0.1 x 0.7 x 1.0 x 0.4 x 0.18 x 1.0 x 1.0 x 0.18 = 0.0009

1.0 x 0.1 x 0.3 x 0.7 x 1.0 x 0.18 x 1.0 x 1.0 x 0.18 = 0.0006

Overall Probability
= 0.0009 + 0.0006 = 0.0015



PCFG

Which of the following reason justifies the PCFG approach over top-down/bottom-up approach?

[✓ In case of ambiguous sentences, PCFG parsing takes advantage of probabilities by generating the most probable parse for a sentence.]

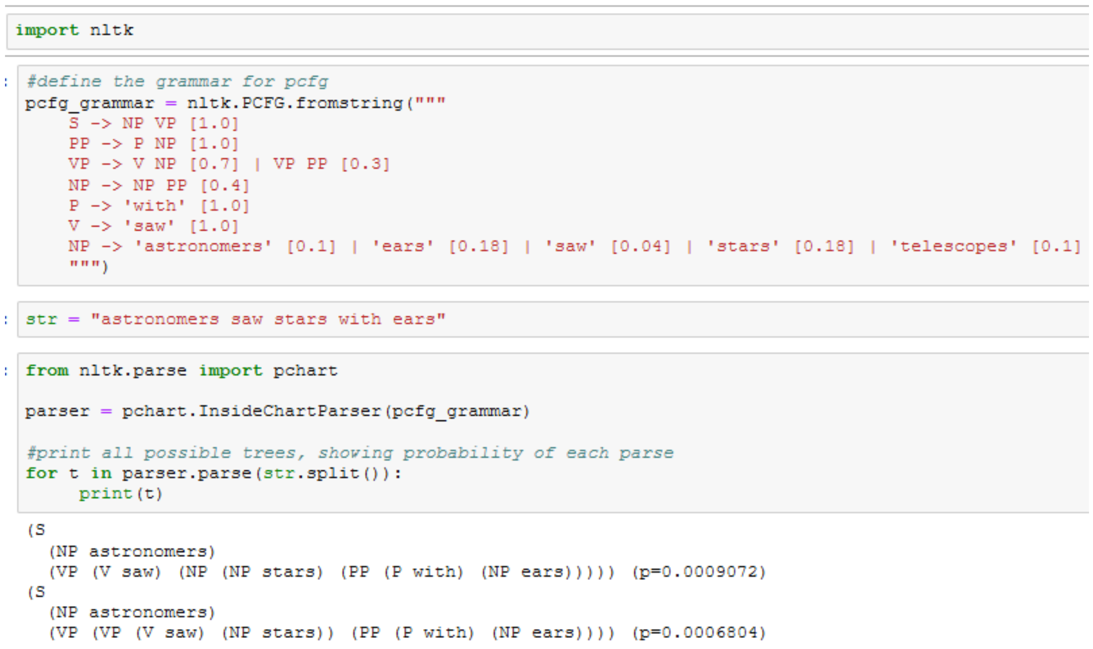
Feedback :
In case of ambiguous sentences, PCFG uses the probabilities associated with the grammar and then calculate the overall probability of the parsed tree

In case of ambiguous sentences, PCFG parsing makes top-down parser faster to work with

In case of ambiguous sentences, PCFG filters out parse trees randomly

All of the above

Parsing Using PCFG In Python



```
import nltk
```

```
#define the grammar for pcfg
pcfg_grammar = nltk.PCFG.fromstring("""
    S -> NP VP [1.0]
    PP -> P NP [1.0]
    VP -> V NP [0.7] | VP PP [0.3]
    NP -> NP PP [0.4]
    P -> 'with' [1.0]
    V -> 'saw' [1.0]
    NP -> 'astronomers' [0.1] | 'ears' [0.18] | 'saw' [0.04] | 'stars' [0.18] | 'telescopes' [0.1]
    """)
```

```
str = "astronomers saw stars with ears"
```

```
from nltk.parse import pchart

parser = pchart.InsideChartParser(pcfg_grammar)

#print all possible trees, showing probability of each parse
for t in parser.parse(str.split()):
     print(t)
```

```
(S
  (NP astronomers)
  (VP (V saw) (NP (NP stars) (PP (P with) (NP ears))))) (p=0.0009072)
(S
  (NP astronomers)
  (VP (VP (V saw) (NP stars)) (PP (P with) (NP ears)))) (p=0.0006804)
```

Idea of CFGs and PCFGs and how to build PCFG-based parse trees using NLTK. There are a few other non-trivial problems to address in PCFGs, one being that of computing the probabilities of production rules.

The solution is to **learn these probabilities** from some pre-tagged corpus where a large number of sentences have been parsed manually. Having access to such a corpus (such as the Penn Treebank), one can compute the probabilities by counting the frequencies of production rules. This is similar to how we computed the transition and emission probabilities in HMMs.

A detailed study of PCFGs is beyond the scope of this course.

Next is **Chomsky Normal Form.**

Additional Resources

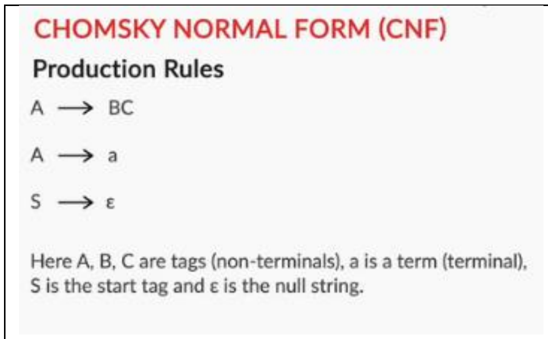
- For a further detailed study of PCFGs, refer to chapter 13, Statistical Parsing, of the book 'Speech and Language Processing, Daniel Jurafsky & James H. Martin'.

Chomsky Normal Form

Until now, we have defined multiple CFGs. Some CFGs are of the form A > B where both A and B are non-terminals (e.g. Nom > Noun), some are of the form A > B where A is a non-terminal while C is a terminal (Det > the), etc.

It is often useful to use a **standardised version of production rules** by converting a grammar to the **Chomsky Normal Form (CNF).** The CNF, proposed by the linguist Noam Chomsky, is a normalized version of the CFG with a standard set of rules defining how production rule must be written.

Rules required to convert a CFG to the corresponding Chomsky Normal Form.



CHOMSKY NORMAL FORM (CNF)

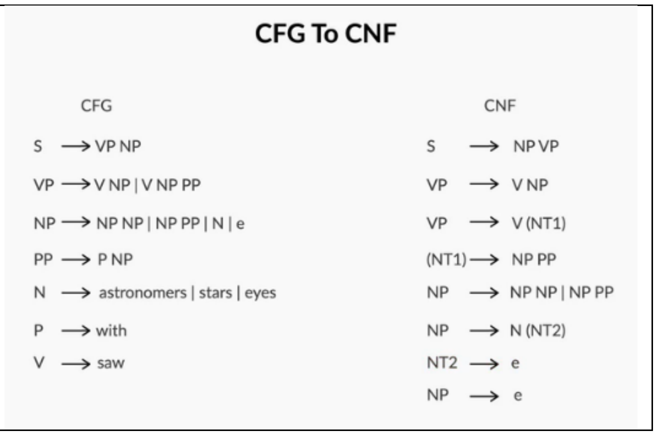
Production Rules

$A \longrightarrow BC$

$A \longrightarrow a$

$S \longrightarrow \varepsilon$

Here A, B, C are tags (non-terminals), a is a term (terminal), S is the start tag and ε is the null string.



CFG To CNF

CFG	CNF
S ⟶ VP NP	S ⟶ NP VP
VP ⟶ V NP | V NP PP	VP ⟶ V NP
NP ⟶ NP NP | NP PP | N | e	VP ⟶ V (NT1)
PP ⟶ P NP	(NT1) ⟶ NP PP
N ⟶ astronomers | stars | eyes	NP ⟶ NP NP | NP PP
P ⟶ with	NP ⟶ N (NT2)
V ⟶ saw	NT2 ⟶ e
	NP ⟶ e

The three forms of CNF rules can be written:

1. A -> B C
2. A -> a
3. S -> ε

A, B, C are non-terminals (POS tags), a is a terminal (term), S is the start symbol of the grammar and ε is the null string.

The table below shows some examples for converting CFGs to the CNF:

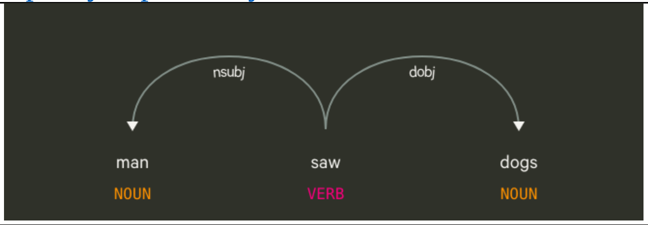
CFG	VP -> V NP PP	VP -> V
CNF	VP -> V (NP1) NP1 -> NP PP	VP -> V (VP1) VP1 -> ε

Dependency Parsing

In constituency parsing, groups of words or constituencies comprise of the basic structure of a parse tree. **Dependency grammar** is **an alternate paradigm** of grammar. It has related dependency parsing techniques.

In dependency grammar, constituencies (such as NP, VP etc.) do not form the basic elements of grammar, but rather dependencies are established between the words themselves.

Following is dependency parse tree of the sentence "man saw dogs" created using the displaCy dependency visualiser:

	The dependencies can be read as follows: 'man' is the **subject** of the sentence (the one who is doing something); 'saw' is the main **verb** (something that is being done); while 'dogs' is the **object** of 'saw' (to whom something is being done).

There is no notion of phrases or constituencies, but rather relationships are established between the words themselves.

Basic idea of **Dependency Parsing** is that *each sentence is about something*, and usually contains a subject (the doer), a verb (what is being done) and an object (to whom something is being done).

In general, **Subject-Verb-Object** (SVO) is the basic word order in present-day English (which follows a 'rigid word order' form). Of course, many sentences are far more complex to fall into this simplified SVO structure, though sophisticated dependency parsing techniques can handle most of them.

Fixed and Free-Word-Order Languages

Modern languages can be divided into two types - **fixed-word order** and **free-word order**.
To understand the nature of languages, consider the following English sentences:

The labourers built a strong wall. Subject Verb Object	The professor taught NLP to the entire class. Subject Verb Object

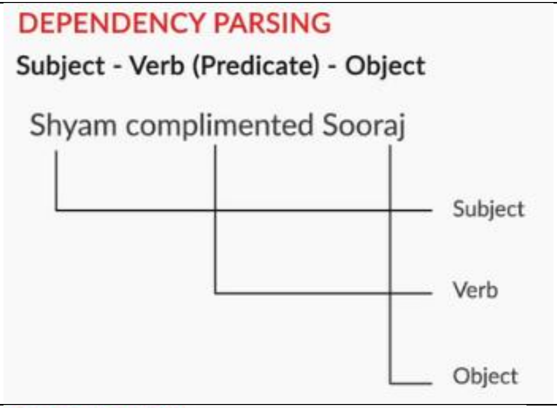
Like English, many **fixed-word order** languages follow the SVO word order.

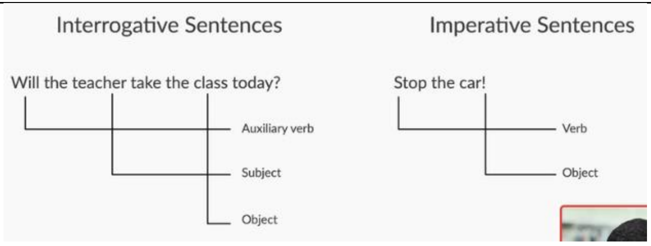
On the other hand, consider the following Hindi sentences:

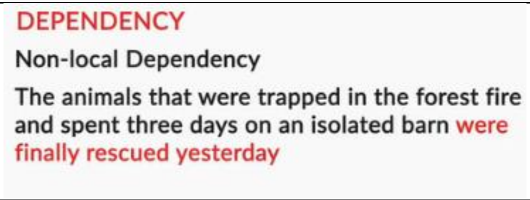
विश्व कप में भारत ने श्रीलंका को हराया Subject Object Verb	श्रीलंका को विश्व कप में भारत ने हराया Object Subject Verb

The two sentences in Hindi have the same meaning though they are written in two different word orders (SOV, OSV). There are fewer languages like Hindi (such as Spanish) which allow a **free-order** of words.

Next is the concept of ‘free’ and ‘fixed’ word order languages, why CFGs cannot handle free word order languages, and the concept of 'dependencies'.


DEPENDENCY PARSING
Subject - Verb (Predicate) - Object
Shyam complimented Sooraj
Subject
Verb
Object




DEPENDENCY

Non-local Dependency

The animals that were trapped in the forest fire and spent three days on an isolated barn were finally rescued yesterday

DEPENDENCY

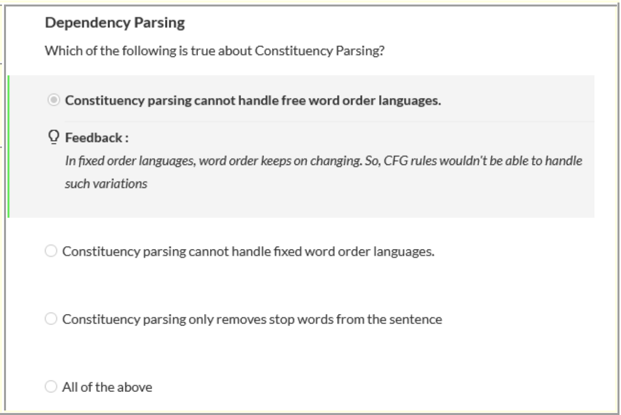
Non-local Dependency

The animals that were trapped in the forest fire and spent three days on an isolated barn were finally rescued yesterday

Free word order languages such as Hindi are difficult to parse using constituency parsing techniques. In such free-word-order languages, the order of words/constituents may change significantly while keeping the meaning exactly the same. It is thus difficult to fit the sentences into the finite set of production rules that CFGs offer.

You also saw how dependencies in a sentence are defined using the elements Subject-Verb-Object.

The following table shows examples of three types of sentences - declarative, interrogative, and imperative:

Declarative	Shyam complimented Suraj Subject Verb Object	
Interrogative	Will the teacher take the class today? Aux Subject Object (Aux: auxiliary verbs such as will, be, can)	
Imperative	Stop the car! Verb Object	

Dependency Parsing

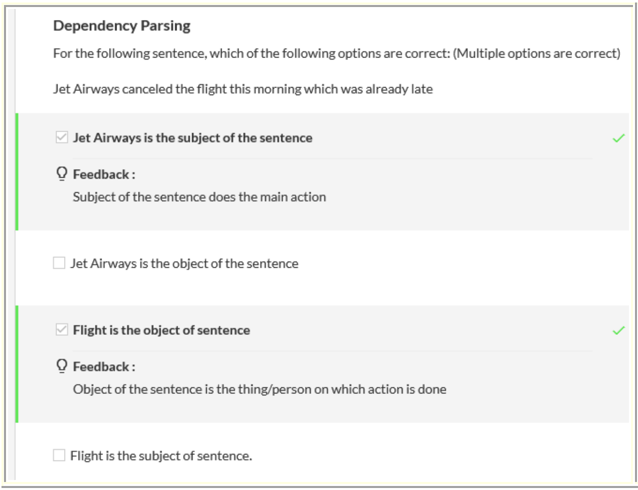
Which of the following is true about Constituency Parsing?

- [✓ Constituency parsing cannot handle free word order languages.]

  Feedback :

  In fixed order languages, word order keeps on changing. So, CFG rules wouldn't be able to handle such variations

- Constituency parsing cannot handle fixed word order languages.
- Constituency parsing only removes stop words from the sentence
- All of the above



Dependency Parsing

For the following sentence, which of the following options are correct: (Multiple options are correct)

Jet Airways canceled the flight this morning which was already late

[✓ **Jet Airways is the subject of the sentence**] ✓

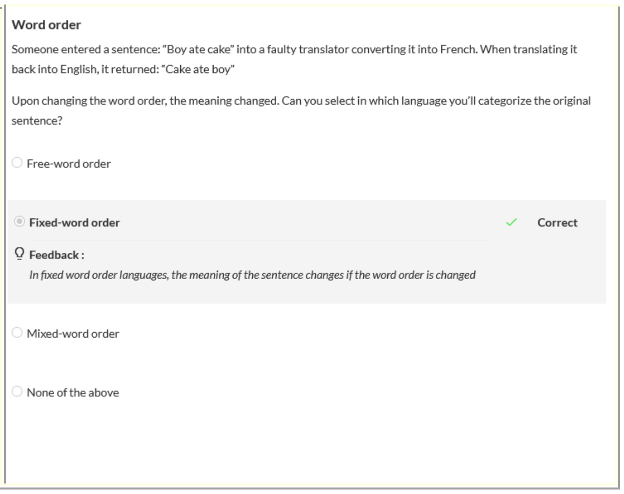
Feedback :
Subject of the sentence does the main action

☐ Jet Airways is the object of the sentence

[✓ **Flight is the object of sentence**] ✓

Feedback :
Object of the sentence is the thing/person on which action is done

☐ Flight is the subject of sentence.



Word order

Someone entered a sentence: "Boy ate cake" into a faulty translator converting it into French. When translating it back into English, it returned: "Cake ate boy"

Upon changing the word order, the meaning changed. Can you select in which language you'll categorize the original sentence?

○ Free-word order

[✓ **Fixed-word order**] ✓ Correct

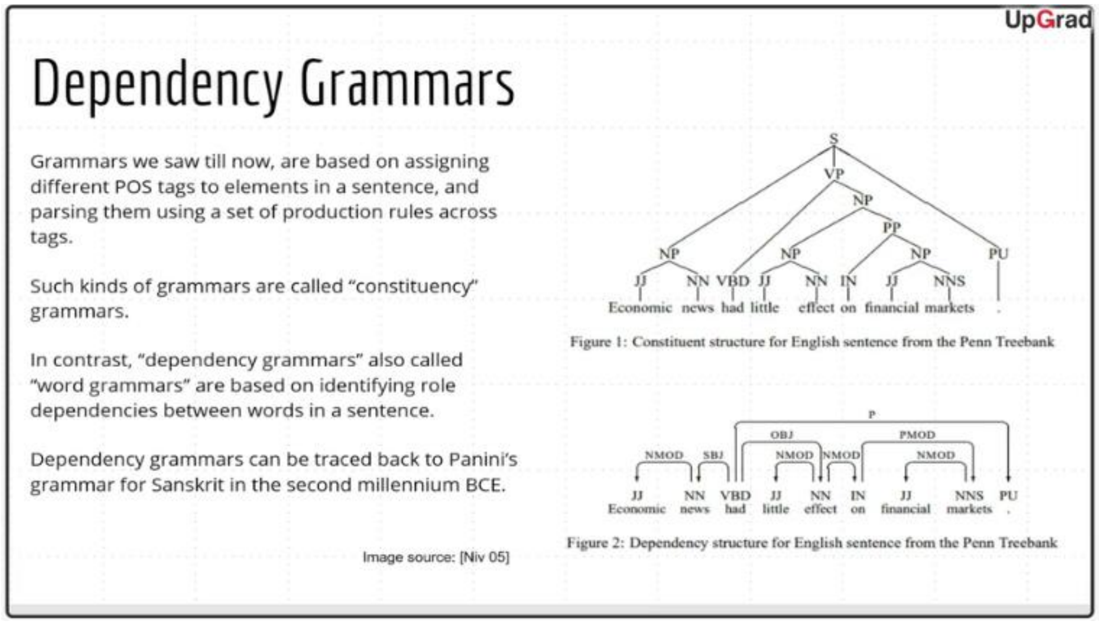
Feedback :
In fixed word order languages, the meaning of the sentence changes if the word order is changed

○ Mixed-word order

○ None of the above

Universal Dependencies

Apart from dependencies defined in the form of subject-verb-object, there's a non-exhaustive list of dependency relationships. Let's look at how a dependency parse structure looks like and how dependencies are established among the words using **universal dependencies**.



UpGrad

Dependency Grammars

Grammars we saw till now, are based on assigning different POS tags to elements in a sentence, and parsing them using a set of production rules across tags.

Such kinds of grammars are called "constituency" grammars.

In contrast, "dependency grammars" also called "word grammars" are based on identifying role dependencies between words in a sentence.

Dependency grammars can be traced back to Panini's grammar for Sanskrit in the second millennium BCE.

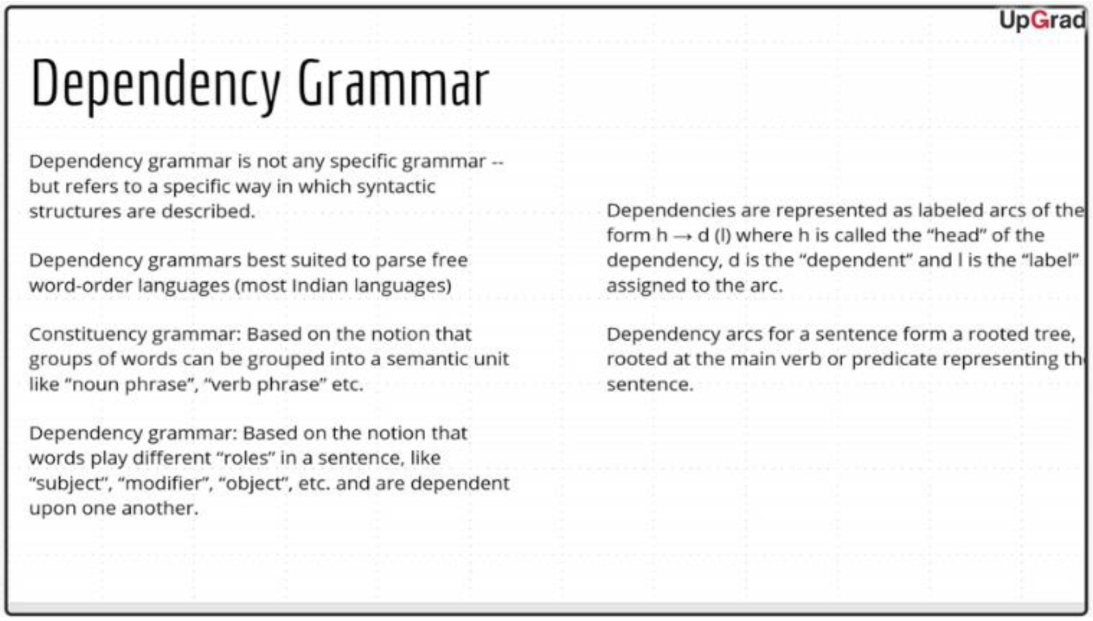
Image source: [Niv 05]

S
VP
NP
PP
NP
NP
NP
PU
JJ NN VBD JJ NN IN JJ NNS
Economic news had little effect on financial markets .

Figure 1: Constituent structure for English sentence from the Penn Treebank

P
OBJ
PMOD
NMOD
SBJ
NMOD
NMOD
NMOD
JJ NN VBD JJ NN IN JJ NNS PU
Economic news had little effect on financial markets .

Figure 2: Dependency structure for English sentence from the Penn Treebank



UpGrad

Dependency Grammar

Dependency grammar is not any specific grammar -- but refers to a specific way in which syntactic structures are described.

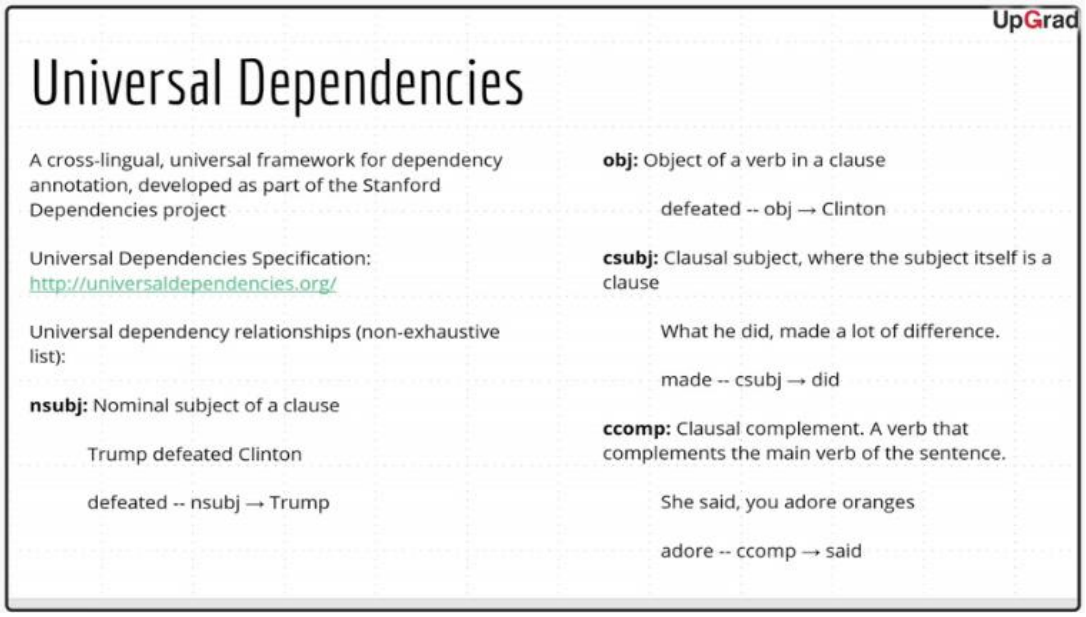
Dependency grammars best suited to parse free word-order languages (most Indian languages)

Constituency grammar: Based on the notion that groups of words can be grouped into a semantic unit like "noun phrase", "verb phrase" etc.

Dependency grammar: Based on the notion that words play different "roles" in a sentence, like "subject", "modifier", "object", etc. and are dependent upon one another.

Dependencies are represented as labeled arcs of the form h → d (l) where h is called the "head" of the dependency, d is the "dependent" and l is the "label" assigned to the arc.

Dependency arcs for a sentence form a rooted tree, rooted at the main verb or predicate representing th sentence.



UpGrad

Universal Dependencies

A cross-lingual, universal framework for dependency annotation, developed as part of the Stanford Dependencies project

Universal Dependencies Specification: http://universaldependencies.org/

Universal dependency relationships (non-exhaustive list):

nsubj: Nominal subject of a clause

Trump defeated Clinton

defeated -- nsubj → Trump

obj: Object of a verb in a clause

defeated -- obj → Clinton

csubj: Clausal subject, where the subject itself is a clause

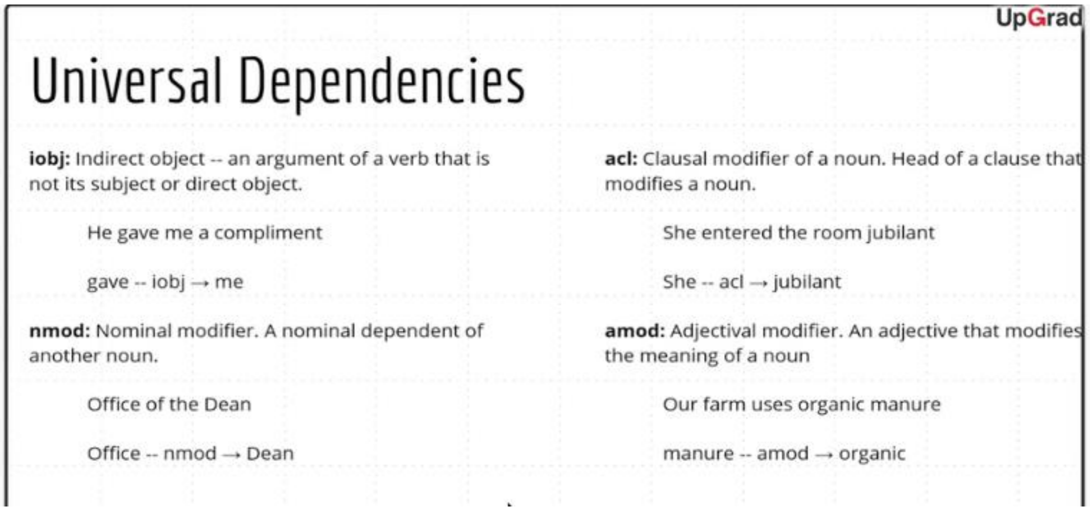
What he did, made a lot of difference.

made -- csubj → did

ccomp: Clausal complement. A verb that complements the main verb of the sentence.

She said, you adore oranges

adore -- ccomp → said



UpGrad

Universal Dependencies

iobj: Indirect object -- an argument of a verb that is not its subject or direct object.

He gave me a compliment

gave -- iobj → me

nmod: Nominal modifier. A nominal dependent of another noun.

Office of the Dean

Office -- nmod → Dean

acl: Clausal modifier of a noun. Head of a clause that modifies a noun.

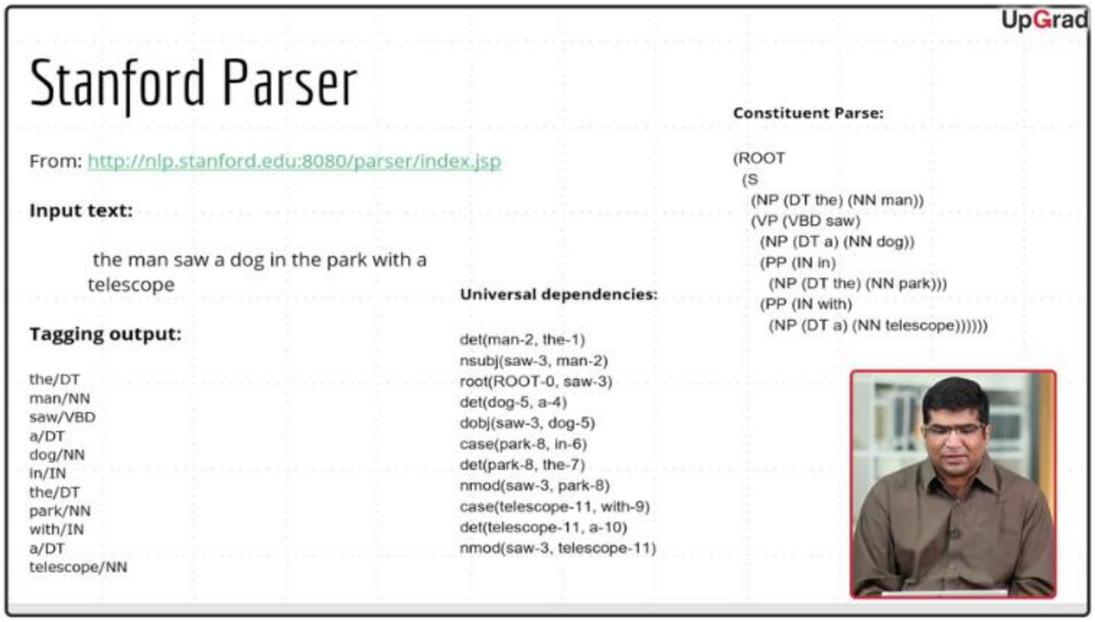
She entered the room jubilant

She -- acl → jubilant

amod: Adjectival modifier. An adjective that modifies the meaning of a noun

Our farm uses organic manure

manure -- amod → organic



UpGrad

Stanford Parser

From: http://nlp.stanford.edu:8080/parser/index.jsp

Input text:

the man saw a dog in the park with a telescope

Tagging output:

```
the/DT
man/NN
saw/VBD
a/DT
dog/NN
in/IN
the/DT
park/NN
with/IN
a/DT
telescope/NN
```

Universal dependencies:

```
det(man-2, the-1)
nsubj(saw-3, man-2)
root(ROOT-0, saw-3)
det(dog-5, a-4)
dobj(saw-3, dog-5)
case(park-8, in-6)
det(park-8, the-7)
nmod(saw-3, park-8)
case(telescope-11, with-9)
det(telescope-11, a-10)
nmod(saw-3, telescope-11)
```

Constituent Parse:

```
(ROOT
  (S
    (NP (DT the) (NN man))
    (VP (VBD saw)
      (NP (DT a) (NN dog))
      (PP (IN in)
        (NP (DT the) (NN park)))
      (PP (IN with)
        (NP (DT a) (NN telescope))))))
```

Elements of Dependency Grammar

Consider the declarative sentence: "The man jumped from the moving train into the river"**.**

In a dependency parse, we start from the **root** of the sentence, which is often a verb. In the example above, the root is the word 'jumped'. The intuition for the root is that it is the main word that describes the 'aboutness' of a sentence. Although the sentence is also about 'The man', 'the moving train' and 'the river', it is most strongly about the fact someone 'jumped' from somewhere into something.

Dependencies are represented as labelled arcs of the form h → d (l) where 'h' is called the "head" of the dependency, 'd' is the "dependent" and l is the "label" assigned to the arc.

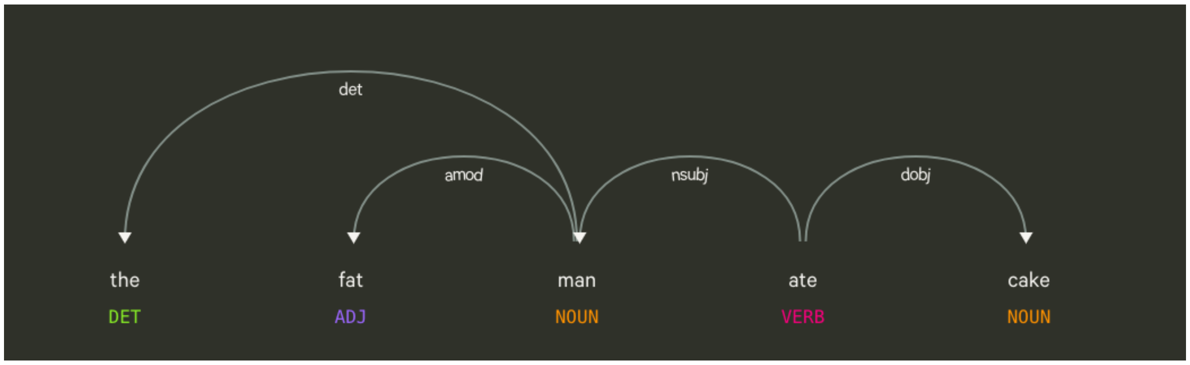
There is a non-exhaustive list of dependency roles.

You can read more about labels from the following URL.

Go through the document to make it easier to understand the dependencies mentioned later. Let's now understand the dependency graph using a small sentence.

Sentence: "The fat man ate cake."

The **root verb,** also called the **head** of the sentence, is the verb 'ate' since it describes what the sentence is about. All the other words are dependent on the root word 'ate', as shown by the arcs directed from 'ate' to the other words.



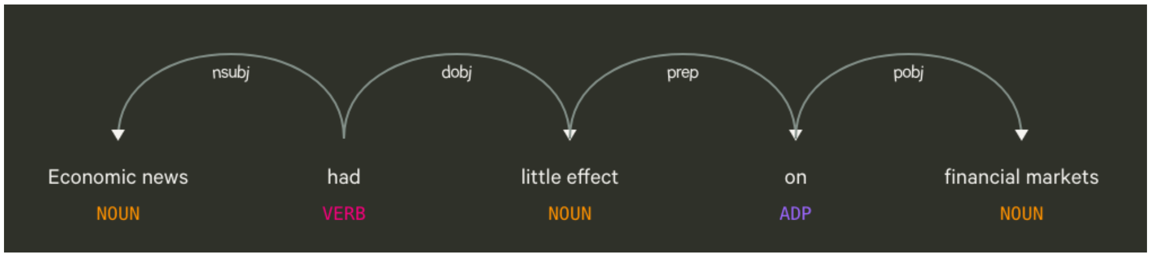
The dependencies from the root to the other words are as follows.

- **nsubj** (man) is the nominal subject of the verb 'ate'
- The word 'fat' modifies the word 'man' and is the **adjective modifier** (**amod**) of nsubj
- The word 'the' is a determiner associated with the word 'man'
- The **direct object** of the verb (**dobj)** is 'cake'

Let's now understand the dependency parse tree for the sentence shown in the lecture:

Sentence: "Economic news had little effect on financial markets"

You can visualise the dependency parse of this sentence here. Also, in the parse shown below, we have merged the phrases such as 'Economic news', 'little effect' etc.



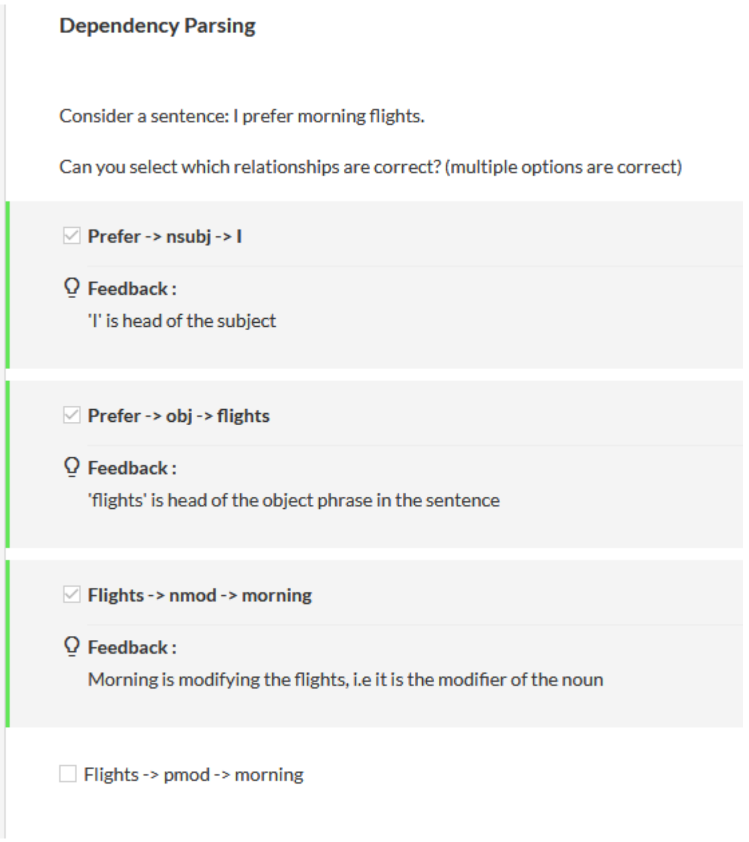
Let's identify the role of each word one by one, starting with the **root verb**

- The word 'had' is the root

- The phrase ‘Economic news’ is the nominal subject (nsubj)
- The phrase 'little effect' is the direct object (dobj) of the verb 'had'
- The word 'on' is a preposition associated with 'little effect'
- The noun phrase 'financial markets' is the object of 'on'

Let’s now look at the role of each word in the parse.

- The word 'Economic' is the modifier of ‘news’. This is represented as:
    - news -> nmod -> economic
- The words ‘financial’ and ‘little’ modify the words 'markets' and 'effect' respectively:
    - effect -> amod -> little
    - markets -> nmod -> financial
- The two words ‘on’ and ‘markets’ have no incoming arcs. The word ‘on’ is dependent on the word ‘effect’ as a nominal modifier:
    - effect -> prep -> on
- The word ‘markets’ is an object of the word ‘on’:
    - on -> pobj -> markets



Dependency Parsing

Consider a sentence: I prefer morning flights.

Can you select which relationships are correct? (multiple options are correct)

☑ **Prefer -> nsubj -> I**

Feedback :
'I' is head of the subject

☑ **Prefer -> obj -> flights**

Feedback :
'flights' is head of the object phrase in the sentence

☑ **Flights -> nmod -> morning**

Feedback :
Morning is modifying the flights, i.e it is the modifier of the noun

☐ Flights -> pmod -> morning

Dependency Parsing - Optional Content

Dependency parsing is a fairly advanced topic whose study involves a much deeper understanding of the English grammar and parsing algorithms.

Advanced topics in dependency parsing have been covered as part of the optional content .

The topics covered in optional content should provide you with sufficient background to study further advanced topics, such as the one provided below.

Additional Reading

- Read more on Dependency Parsing from Chapter 13, Dependency Parsing, Speech and Language Processing. Daniel Jurafsky & James H. Martin.

Summary

Two most commonly used paradigms of parsing - **constituency parsing** and **dependency parsing.**

In constituency parsing, basic **idea of constituents** as grammatically meaningful groups of words, or phrases, such as noun phrase, verb phrase etc. Idea of **context-free grammars** or **CFGs** which specify a set of **production rules**. Two broad approaches to constituency parsing:

- Top-down parsing
- Bottom-up parsing

Left-recursion in grammar rules causes the **top-down parser** to run into an infinite loop. The alternative, in this case, is **bottom-up** parsing. **Shift-reduce algorithm** can be used for bottom-up parsing.

The parsers based solely on CFGs often generate multiple parses of sentences, though only some of them are likely to occur in the real world. To deal with such ambiguous sentences, exploit the idea of probabilities using **probabilistic context-free grammars or PCFGs**. The probabilities associated with each rule help the algorithm decide the most probable parse tree of an ambiguous sentence.

How to convert any CFG to the **Chomsky Normal Form (CNF).** The CNF helps reduce the theoretically wide range of possible grammars into a standardised form, thereby resulting in convenience in writing parsing algorithms.

Dependency parsing which is based on an alternative paradigm of grammar called **dependency grammar**. Dependency parsing is useful in dealing with **free-word-order** languages while constituency parsing techniques are confined to performing well only on **fixed-word-order** languages. Basic elements of dependency parse grammar are subject, verb, object etc.

Next session will apply syntactic analysis techniques and build a natural language flight-booking-system via NLP techniques I**nformation Extraction**, **Named Entity Recognition**, and sophisticated sequence modelling techniques such as **Conditional Random Fields.**