

Problems L3

Fundamentals of Natural Language

Authors:

Sonia Espinilla, 1708919
Agustina Inés Lazzati, 1707964

Autonomous University of Barcelona

March 23, 2025

✓ Graded Exercises

These exercises will count towards your Problems mark and must be delivered through Campus Virtual. You must deliver a single PDF with all answers in groups of two. Answers should be succinct and to the point.

Make sure to install NLTK. Install an Anaconda Distribution and then run the following command in a terminal:

```
#conda install nltk
```

If you are using Windows, you have to either use the specialised Anaconda Prompt terminal or the Anaconda navigator program to install it. You can also use Spyder's integrated terminal by prefixing the command with a !

Once installed, download its data using:

```
import nltk
nltk.download("popular")
nltk.download('punkt_tab')
nltk.download('averaged_perceptron_tagger_eng')
```



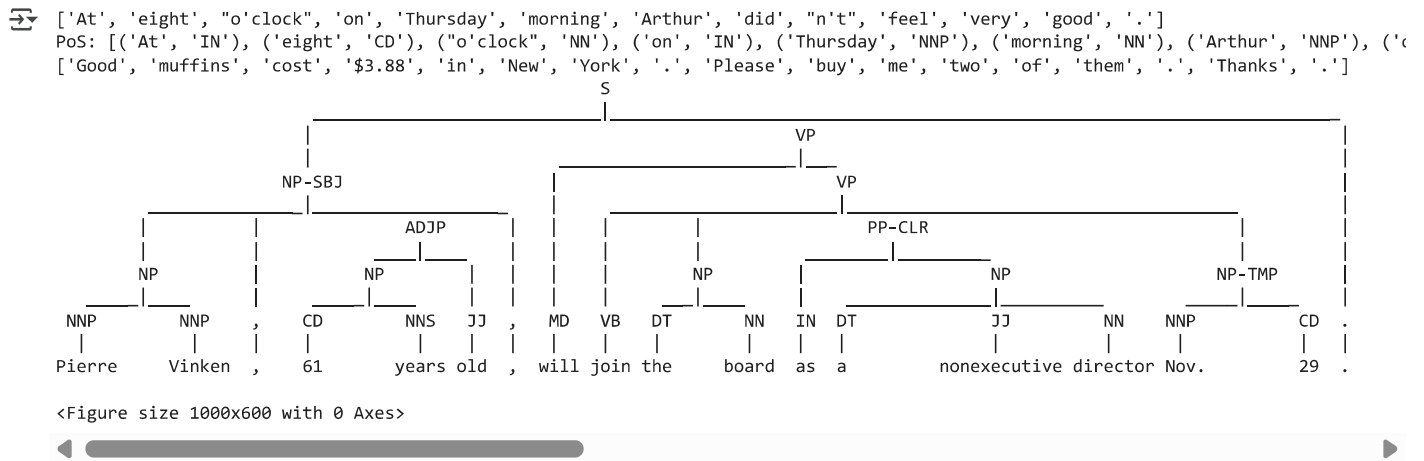
Mostrar salida oculta

```
import nltk
import matplotlib.pyplot as plt
from nltk.corpus import treebank
from nltk import Tree

sentence = "At eight o'clock on Thursday morning Arthur didn't feel very good."
tokens = nltk.word_tokenize(sentence) # Tokenize
print(tokens)
tagged = nltk.pos_tag(tokens) # PoS tagging
print("PoS:", tagged)
from nltk.tokenize import RegexpTokenizer
s = "Good muffins cost $3.88\nin New York. Please buy me two of them.\n\nThanks."
tokenizer = RegexpTokenizer(r'\w+|\$[\d\.]+|\S+')
output = tokenizer.tokenize(s)
print(output)

#print the tree, because t.draw() doesn't work
from nltk.corpus import treebank
t = treebank.parsed_sents('wsj_0001.mrg')[0]
tree_str = str(t)
tree = Tree.fromstring(tree_str)

plt.figure(figsize=(10, 6))
tree.pretty_print()
plt.show()
```



✓ Graded Exercise 1 (1 point)

Tokenise and compute the PoS of the following sentences. Show both the code that you used and the results.

- The Jamaica Observer reported that Usain Bolt broke the 100m record.
- While hunting in Africa, I shot an elephant in my pajamas. How an elephant got into my pajamas I'll never know.

```

from nltk.tokenize import word_tokenize
from nltk import pos_tag

sentence1 = "The Jamaica Observer reported that Usain Bolt broke the 100m record."
sentence2 = "While hunting in Africa, I shot an elephant in my pajamas. How an elephant got into my pajamas I'll never know."

tokens1 = word_tokenize(sentence1)
tokens2 = word_tokenize(sentence2)

pos_tags1 = pos_tag(tokens1)
pos_tags2 = pos_tag(tokens2)

print("Sentence 1:")
print("Tokens:", tokens1)
print("PoS Tags:", pos_tags1)

print("\nSentence 2:")
print("Tokens:", tokens2)
print("PoS Tags:", pos_tags2)

```

Sentence 1:
 Tokens: ['The', 'Jamaica', 'Observer', 'reported', 'that', 'Usain', 'Bolt', 'broke', 'the', '100m', 'record', '.']
 PoS Tags: [('The', 'DT'), ('Jamaica', 'NNP'), ('Observer', 'NNP'), ('reported', 'VBD'), ('that', 'DT'), ('Usain', 'NNP'), ('Bolt',
 Sentence 2:
 Tokens: ['While', 'hunting', 'in', 'Africa', ',', 'I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas', '.', 'How', 'an', 'elephant
 PoS Tags: [('While', 'IN'), ('hunting', 'VBG'), ('in', 'IN'), ('Africa', 'NNP'), (',', ','), ('I', 'PRP'), ('shot', 'VBP'), ('an',

✓ Graded Exercise 2 (3 points)

Given the following code:

```

import nltk
from nltk import CFG
# Defining a grammar
groucho_grammar = CFG.fromstring("""
S -> NP VP
NP -> P NP
NP -> Det N | Det N PP | 'I' | 'You'
VP -> V NP | VP PP
Det -> 'an' | 'my'
N -> 'elephant' | 'pajamas'
V -> 'shot'
P -> 'in'
""")
# Printing the grammar
print("START grammar:", groucho_grammar.start())
print("PRODUCTIONS grammar:", groucho_grammar.productions())

```

```

text = "I shot an elephant in my pajamas"
text_tokens = nltk.word_tokenize(text)
# Parsing the text
parser = nltk.parse.chart.ChartParser(groucho_grammar, trace=2)
trees = parser.parse(text_tokens)
for t in trees:
    t.pretty_print()

```



Mostrar salida oculta

Modify it to parse this list of texts:

```

mytexts = ["John saw a man with my telescope",
           "Alex kissed the dog",
           "the man with the telescope ate a sandwich in the park"]

```

You have to:

- Create a CFG that can parse the sentences within mytexts ,
- make a loop that, for each sentence, parses it and produces the parse trees and
- if the output contains more than one parse tree, explain (max two sentences) why there is an ambiguity.

```

import nltk
from nltk import CFG
from nltk.parse.chart import ChartParser

# Define the grammar
my_grammar = CFG.fromstring("""
    S -> NP VP
    NP -> Det N | NP PP | 'John' | 'Alex' | 'the' N
    VP -> V NP | V NP PP
    PP -> P NP
    Det -> 'a' | 'the' | 'my'
    N -> 'man' | 'telescope' | 'dog' | 'sandwich' | 'park'
    V -> 'saw' | 'kissed' | 'ate'
    P -> 'with' | 'in'
""")

print("START grammar:", my_grammar.start())
print("PRODUCTIONS grammar:", my_grammar.productions())

# Sentences to parse
mytexts = [
    "John saw a man with my telescope",
    "Alex kissed the dog",
    "the man with the telescope ate a sandwich in the park"
]

# Dictionary to store parse trees
parse_trees = {}

# Loop through each sentence, parse it, and store the parse trees
for text in mytexts:
    print(f"\nParsing sentence: '{text}'")
    text_tokens = nltk.word_tokenize(text)
    parser = ChartParser(my_grammar, trace=2)
    trees = list(parser.parse(text_tokens))

    if not trees:
        print("No parse trees found.")
    else:
        parse_trees[text] = trees # Store all parse trees for the sentence
        print(f"Found {len(trees)} parse tree(s).")

```



Mostrar salida oculta

```

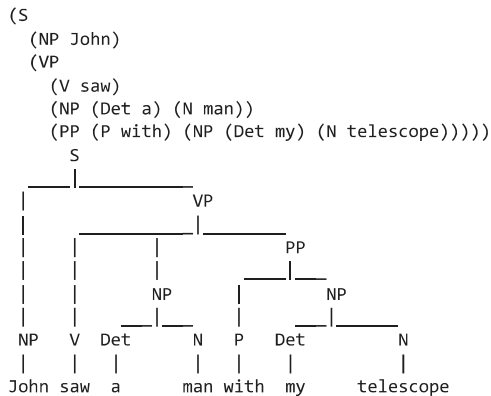
if mytexts[0] in parse_trees:
    print(f"\nSentence: '{mytexts[0]}'")
    for i, tree in enumerate(parse_trees[mytexts[0]], 1):
        print(f"\nParse Tree {i}:")
        print(tree) # Print tree object
        tree.pretty_print() # Pretty print the parse tree
else:
    print(f"Sentence '{mytexts[0]}' not found in the dictionary.")

```

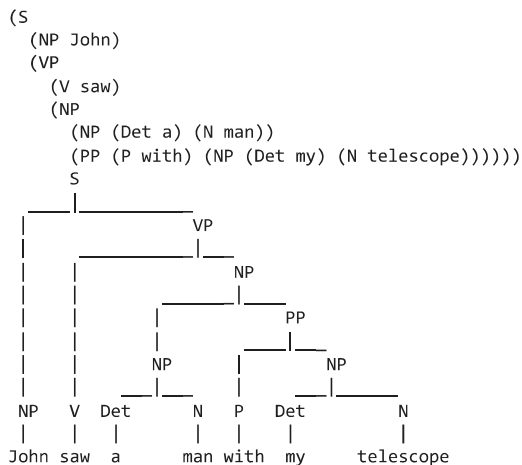


Sentence: 'John saw a man with my telescope'

Parse Tree 1:



Parse Tree 2:



```

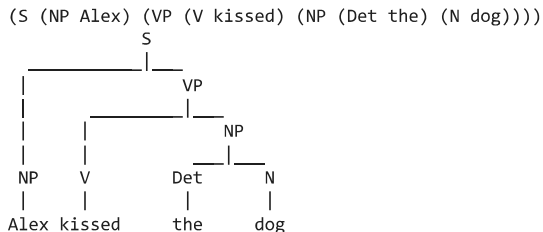
if mytexts[1] in parse_trees:
    print(f"\nSentence: '{mytexts[1]}'")
    for i, tree in enumerate(parse_trees[mytexts[1]], 1):
        print(f"\nParse Tree {i}:")
        print(tree) # Print tree object
        tree.pretty_print() # Pretty print the parse tree
else:
    print(f"Sentence '{mytexts[1]}' not found in the dictionary.")

```

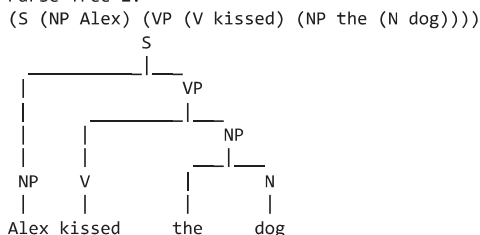


Sentence: 'Alex kissed the dog'

Parse Tree 1:



Parse Tree 2:



```

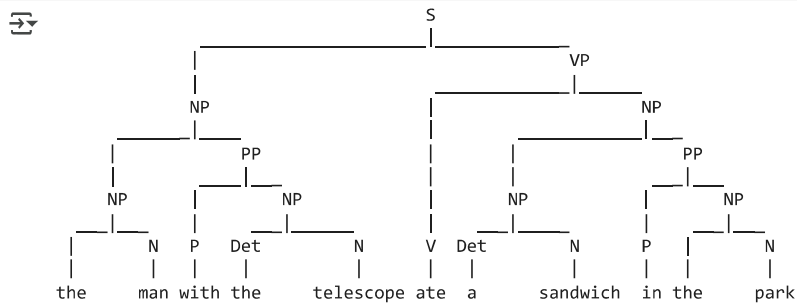
if mytexts[2] in parse_trees:
    print(f"\nSentence: '{mytexts[2]}'")

```

```

for i, tree in enumerate(parse_trees[mytexts[2]], 1):
    print(f"\nParse Tree {i}:")
    print(tree) # Print tree object
    tree.pretty_print() # Pretty print the parse tree
else:
    print(f"Sentence '{mytexts[2]}' not found in the dictionary.")

```

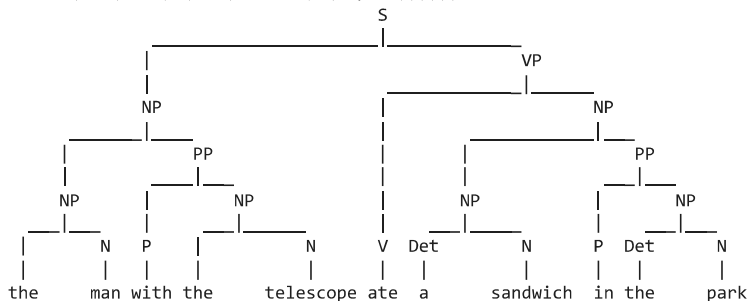


Parse Tree 15:

```

(S
  (NP (NP the (N man)) (PP (P with) (NP the (N telescope)))))
  (VP
    (V ate)
    (NP
      (NP (Det a) (N sandwich))
      (PP (P in) (NP (Det the) (N park))))))

```

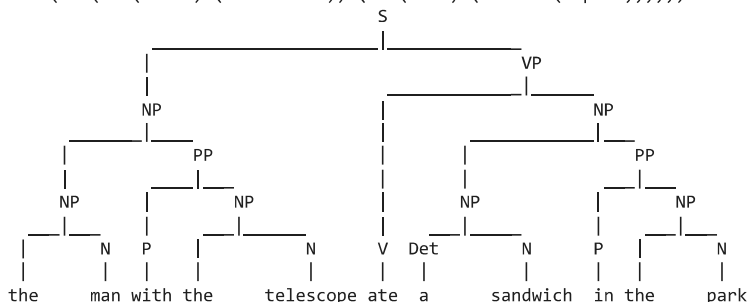


Parse Tree 16:

```

(S
  (NP (NP the (N man)) (PP (P with) (NP the (N telescope)))))
  (VP
    (V ate)
    (NP (NP (Det a) (N sandwich)) (PP (P in) (NP the (N park))))))

```



In all three sentences, multiple parse trees are generated, leading to ambiguity, as different rules can produce the same outcome:

- Sentence 1 (generating 2 parse trees): VP → V NP PP equals doing VP → V NP → NP PP
- Sentence 2 (generating 2 parse trees) & 3 (generating 16 parse trees): NP → Det N (Det → 'the') equals doing NP → 'the' N

✓ Graded Exercise 3 (1 point)

Write an example of a sentence that, although it is syntactically correct in English and uses the lexicon of the previous grammar, it cannot be parsed by the previously defined grammar.

A sentence that cannot be parsed by the previously defined grammar could be "the man ate" since there is no grammar rule for using intransitive verbs without being followed by an object. We should include in our grammar rules that allow us to parse transitive and intransitive verbs, for example:

- VP → V | V NP | V NP PP

Confirming that this sentence cannot be parsed:

```
# CONFIRMING THAT THIS SENTENCE CANNOT BE PARSED:
sentence = "the man ate"

print(f"\nParsing sentence: '{sentence}'")
text_tokens = nltk.word_tokenize(sentence)
parser = nltk.parse.chart.ChartParser(my_grammar, trace=2)
trees = list(parser.parse(text_tokens))

if not trees: # Checks if the list is empty
    print("No parse trees found.")
else:
    for i, t in enumerate(trees, start=1):
        print(f"Parse Tree {i}:")
        print(t)
        t.pretty_print()
```



```
Parsing sentence: 'the man ate'
|. the . man . ate .|
Leaf Init Rule:
|[------] . | [0:1] 'the' *
|. [------] . | [1:2] 'man' *
|. . [------] | [2:3] 'ate'
Bottom Up Predict Combine Rule:
|[------> . | [0:1] NP -> 'the' * N
|[------] . | [0:1] Det -> 'the' *
Bottom Up Predict Combine Rule:
|[------> . | [0:1] NP -> Det * N
Bottom Up Predict Combine Rule:
|. [------] . | [1:2] N -> 'man' *
Single Edge Fundamental Rule:
|[------] . | [0:2] NP -> 'the' N *
|[------] . | [0:2] NP -> Det N *
Bottom Up Predict Combine Rule:
|[------> . | [0:2] S -> NP * VP
|[------> . | [0:2] NP -> NP * PP
Bottom Up Predict Combine Rule:
|[------> . | [0:2] S -> NP * VP
|[------> . | [0:2] NP -> NP * PP
Bottom Up Predict Combine Rule:
|. . [------] | [2:3] V -> 'ate' *
Bottom Up Predict Combine Rule:
|. . [------> | [2:3] VP -> V * NP
|. . [------> | [2:3] VP -> V * NP PP
No parse trees found.
```

✓ Graded Exercise 4 (2 points)

Propose how to augment the previous parser to deal with sentences that may be incorrect, for example, containing spelling errors or mistakes arising from automatic speech recognition or handwritten text recognition (maximum 3 sentences).

Several alternatives could be implemented to augmented the parser, for example:

- Using **CKY parsing with binarization**, allowing the parser to explore multiple possible parses and select the most probable one, making it more resilient to errors caused by speech or handwriting recognition.
- Implement an **Error-Tolerant Parsing**, such as a fuzzy parser or correct misspelled words based on **Edit-Distance**.

✓ Graded Exercise 5 (2 points)

Given the following code:

- Execute this code, modifying the parameter trace (values from 1 to 3). Explain the differences. Print the parsing probability of the sentence "I saw the man with a telescope". Print the tree.

```
#trace 1
import nltk
from nltk import PCFG
prob = {} #we will store all the prob in this dictionary
pcfg1 = PCFG.fromstring("""
S -> NP VP [1.0]
NP -> Det N [0.5] | NP PP [0.25] | 'John' [0.1] | 'I' [0.15]
Det -> 'the' [0.8] | 'a' [0.2]
N -> 'man' [0.5] | 'telescope' [0.5]
VP -> VP PP [0.1] | V NP [0.7] | V [0.2]
V -> 'ate' [0.35] | 'saw' [0.65]
PP -> P NP [1.0]
P -> 'with' [0.61] | 'in' [0.39]
""")
```

```

text = "I saw the man with a telescope"
text_tokens = nltk.word_tokenize(text)
viterbi_parser = nltk.ViterbiParser(pcfg1,trace=1)
trees = viterbi_parser.parse(text_tokens)
for tree in trees:
    print(tree)
    tree.pretty_print()
    probb["Trace 1"]=tree.probab()

```

↔ Inserting tokens into the most likely constituents table...

Finding the most likely constituents spanning 1 text elements...

Finding the most likely constituents spanning 2 text elements...

Finding the most likely constituents spanning 3 text elements...

Finding the most likely constituents spanning 4 text elements...

Finding the most likely constituents spanning 5 text elements...

Finding the most likely constituents spanning 6 text elements...

Finding the most likely constituents spanning 7 text elements...

(S

(NP I)

(VP

(V saw)

(NP

(NP (Det the) (N man))

(PP (P with) (NP (Det a) (N telescope)))))) (p=0.000104081)

S

NP V Det N P Det N

I saw the man with a telescope

```

#trace 2
import nltk
from nltk import PCFG
pcfg1 = PCFG.fromstring("""
S -> NP VP [1.0]
NP -> Det N [0.5] | NP PP [0.25] | 'John' [0.1] | 'I' [0.15]
Det -> 'the' [0.8] | 'a' [0.2]
N -> 'man' [0.5] | 'telescope' [0.5]
VP -> VP PP [0.1] | V NP [0.7] | V [0.2]
V -> 'ate' [0.35] | 'saw' [0.65]
PP -> P NP [1.0]
P -> 'with' [0.61] | 'in' [0.39]
""")
text = "I saw the man with a telescope"
text_tokens = nltk.word_tokenize(text)
viterbi_parser = nltk.ViterbiParser(pcfg1,trace=2)
trees = viterbi_parser.parse(text_tokens)
for tree in trees:
    print(tree)
    tree.pretty_print()
    probb["Trace 2"]=tree.probab()

```

↔ Inserting tokens into the most likely constituents table...

Insert: |=.....| I

Insert: |.=.....| saw

Insert: |..=....| the

Insert: |...=...| man

Insert: |....=..| with

Insert: |.....=.| a

Insert: |.....=| telescope

Finding the most likely constituents spanning 1 text elements...

Insert: |=.....| NP -> 'I' [0.15]

Insert: |.=.....| V -> 'saw' [0.65]

Insert: |..=....| VP -> V [0.2]

Insert: |...=...| Det -> 'the' [0.8]

Insert: |....=..| N -> 'man' [0.5]

Insert: |.....=.| P -> 'with' [0.61]

Insert: |.....=| Det -> 'a' [0.2]

Insert: |.....=| N -> 'telescope' [0.5]

Finding the most likely constituents spanning 2 text elements...

Insert: |=.....| S -> NP VP [1.0]

Insert: |..=....| NP -> Det N [0.5]

Insert: |....=..| NP -> Det N [0.5]

Finding the most likely constituents spanning 3 text elements...

Insert: |...=...| VP -> V NP [0.7]

Insert: |.....=| PP -> P NP [1.0]

```

Finding the most likely constituents spanning 4 text elements...
Insert: |====...| S -> NP VP [1.0]
Finding the most likely constituents spanning 5 text elements...
Insert: |..=====| NP -> NP PP [0.25]
Finding the most likely constituents spanning 6 text elements...
Insert: |.=====| VP -> VP PP [0.1]
Insert: |.=====| VP -> V NP [0.7]
Discard: |.=====| VP -> VP PP [0.1]
Finding the most likely constituents spanning 7 text elements...
Insert: |=====| S -> NP VP [1.0]
(S
(NP I)
(VP
(V saw)
(NP
(NP (Det the) (N man))
(PP (P with) (NP (Det a) (N telescope)))))) (p=0.000104081)

```

```

graph TD
    S --- NP1[NP I]
    S --- VP
    VP --- V[V saw]
    VP --- NP2[NP]
    NP2 --- NP3[NP (Det the) (N man)]
    NP2 --- PP[PP (P with) (NP (Det a) (N telescope)))]
    NP3 --- Det1[Det the]
    NP3 --- N1[N man]
    PP --- P[P with]
    PP --- NP4[NP (Det a) (N telescope)]
    NP4 --- Det2[Det a]
    NP4 --- N2[N telescope]

```

```

#trace 3
import nltk
from nltk import PCFG
pcfg1 = PCFG.fromstring("""
S -> NP VP [1.0]
NP -> Det N [0.5] | NP PP [0.25] | 'John' [0.1] | 'I' [0.15]
Det -> 'the' [0.8] | 'a' [0.2]
N -> 'man' [0.5] | 'telescope' [0.5]
VP -> VP PP [0.1] | V NP [0.7] | V [0.2]
V -> 'ate' [0.35] | 'saw' [0.65]
PP -> P NP [1.0]
P -> 'with' [0.61] | 'in' [0.39]
""")
text = "I saw the man with a telescope"
text_tokens = nltk.word_tokenize(text)
viterbi_parser = nltk.ViterbiParser(pcfg1,trace=3)
trees = viterbi_parser.parse(text_tokens)

for tree in trees:
    print(tree)
    tree.pretty_print()
    probb["Trace 3"]=tree.prob()

```

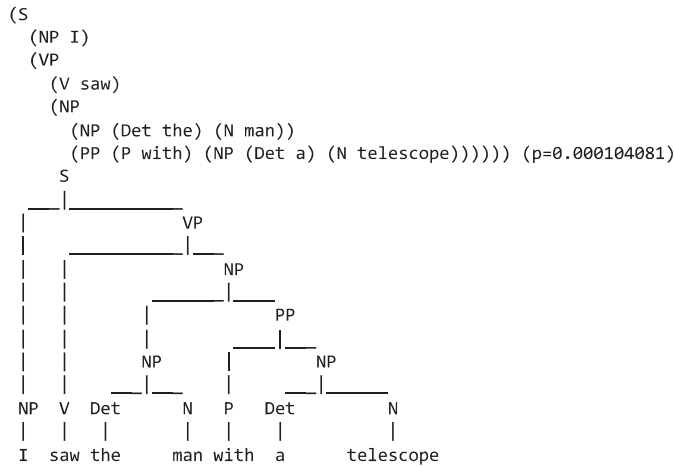
```

↔ Inserting tokens into the most likely constituents table...
Insert: |=......| I
Insert: |.=.....| saw
Insert: |..=====| the
Insert: |...=...| man
Insert: |....=.| with
Insert: |.....=.| a
Insert: |.....=| telescope
Finding the most likely constituents spanning 1 text elements...
Insert: |=......| NP -> 'I' [0.15] 0.1500000000
Insert: |.=.....| V -> 'saw' [0.65] 0.6500000000
Insert: |..=====| VP -> V [0.2] 0.1300000000
Insert: |...=...| Det -> 'the' [0.8] 0.8000000000
Insert: |....=.| N -> 'man' [0.5] 0.5000000000
Insert: |.....=.| P -> 'with' [0.61] 0.6100000000
Insert: |.....=| Det -> 'a' [0.2] 0.2000000000
Insert: |.....=| N -> 'telescope' [0.5] 0.5000000000
Finding the most likely constituents spanning 2 text elements...
Insert: |=====| S -> NP VP [1.0] 0.0195000000
Insert: |..=====| NP -> Det N [0.5] 0.2000000000
Insert: |.....=| NP -> Det N [0.5] 0.0500000000
Finding the most likely constituents spanning 3 text elements...
Insert: |.=====| VP -> V NP [0.7] 0.0910000000
Insert: |.....=| PP -> P NP [1.0] 0.0305000000
Finding the most likely constituents spanning 4 text elements...
Insert: |=====| S -> NP VP [1.0] 0.0136500000
Finding the most likely constituents spanning 5 text elements...
Insert: |..=====| NP -> NP PP [0.25] 0.0015250000
Finding the most likely constituents spanning 6 text elements...
Insert: |.=====| VP -> VP PP [0.1] 0.0002775500
Insert: |.=====| VP -> V NP [0.7] 0.0006938750
Discard: |.=====| VP -> VP PP [0.1] 0.0002775500

```


Finding the most likely constituents spanning 7 text elements...

Insert: |=====| S -> NP VP [1.0] 0.0001040812



trace=1: Basic progress information. Basic debugging, you process all the token

trace=2: More detailed information about rule applications and probabilities. Such as the probabilities associated with different parses and the steps taken to arrive at the final parse.

trace=3: Most detailed information, including internal state and edge additions. This includes all the information from trace=1 and trace=2, plus even more detailed information about the internal state of the parser, such as the contents of the chart and the edges being added.

```

print("Probabilities of final trees:")
for key, value in prob.items():
    print(f"{key}: {value}")

```

```

➡ Probabilities of final trees:
Trace 1: 0.00010408124999999999
Trace 2: 0.00010408124999999999
Trace 3: 0.00010408124999999999

```

• Since the sentence has two possible parse trees, modify the grammar probabilities to force the other parsing tree to be more probable. Print the new probability and the tree.

Lets see which 2 parse tree are possible!

```

pcfg1 = PCFG.fromstring("""
S -> NP VP [1.0]
NP -> Det N [0.5] | NP PP [0.25] | 'John' [0.1] | 'I' [0.15]
Det -> 'the' [0.8] | 'a' [0.2]
N -> 'man' [0.5] | 'telescope' [0.5]
VP -> VP PP [0.1] | V NP [0.7] | V [0.2]
V -> 'ate' [0.35] | 'saw' [0.65]
PP -> P NP [1.0]
P -> 'with' [0.61] | 'in' [0.39]
""")

text = "I saw the man with a telescope"
text_tokens = nltk.word_tokenize(text)

# We create a ChartParser with the PCFG
parser = ChartParser(pcfg1)

trees = list(parser.parse(text_tokens))
print(f"Number of parse trees found: {len(trees)}")

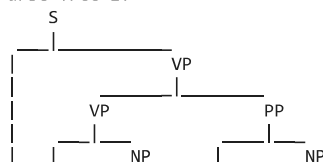
# Pretty-print each parse tree
for i, tree in enumerate(trees, 1):
    print(f"Parse Tree {i}:")
    tree.pretty_print()
    print()

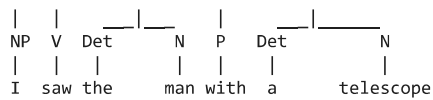
```

```

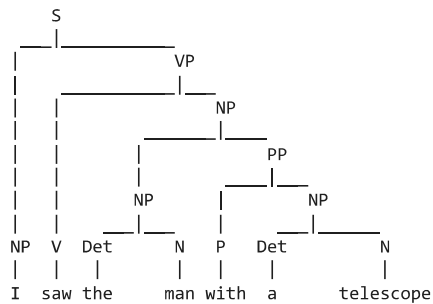
➡ Number of parse trees found: 2
Parse Tree 1:

```





Parse Tree 2:



We want to print the first one as the most probable, so to do that we have to modify that VP PP has more probability of appearing than V NP. And also add a little bit probability of appearing NP PP. Changing this probability we can obtain this parse tree as the most probable.

```
pcfg1 = PCFG.fromstring("""
S -> NP VP [1.0]
NP -> Det N [0.6] | NP PP [0.1] | 'John' [0.1] | 'I' [0.2]
Det -> 'the' [0.8] | 'a' [0.2]
N -> 'man' [0.5] | 'telescope' [0.5]
VP -> VP PP [0.6] | V NP [0.3] | V [0.1]
V -> 'ate' [0.2] | 'saw' [0.8]
PP -> P NP [1.0]
P -> 'with' [0.61] | 'in' [0.39]
""") #change NP and VP

text = "I saw the man with a telescope"
text_tokens = nltk.word_tokenize(text)
viterbi_parser = nltk.ViterbiParser(pcfg1, trace=3)
trees = viterbi_parser.parse(text_tokens)

for tree in trees:
    print(tree)
    tree.pretty_print()
    probb["New Parsing Tree"]=tree.prob()

print("Probability of New Parsing Tree:", probb["New Parsing Tree"])
```

```
➡ Inserting tokens into the most likely constituents table...
Insert: |=.....| I
Insert: |=.....| saw
Insert: |=.....| the
Insert: |=.....| man
Insert: |=.....| with
Insert: |=.....| a
Insert: |=.....| telescope
Finding the most likely constituents spanning 1 text elements...
Insert: |=.....| NP -> 'I' [0.2] 0.2000000000
Insert: |=.....| V -> 'saw' [0.8] 0.8000000000
Insert: |=.....| VP -> V [0.1] 0.0800000000
Insert: |=.....| Det -> 'the' [0.8] 0.8000000000
Insert: |=.....| N -> 'man' [0.5] 0.5000000000
Insert: |=.....| P -> 'with' [0.61] 0.6100000000
Insert: |=.....| Det -> 'a' [0.2] 0.2000000000
Insert: |=.....| N -> 'telescope' [0.5] 0.5000000000
Finding the most likely constituents spanning 2 text elements...
Insert: |=.....| S -> NP VP [1.0] 0.0160000000
Insert: |=.....| NP -> Det N [0.6] 0.2400000000
Insert: |=.....| NP -> Det N [0.6] 0.0600000000
Finding the most likely constituents spanning 3 text elements...
Insert: |=.....| VP -> V NP [0.3] 0.0576000000
Insert: |=.....| PP -> P NP [1.0] 0.0366000000
Finding the most likely constituents spanning 4 text elements...
Insert: |=.....| S -> NP VP [1.0] 0.0115200000
Finding the most likely constituents spanning 5 text elements...
Insert: |=.....| NP -> NP PP [0.1] 0.0008784000
Finding the most likely constituents spanning 6 text elements...
Insert: |=.....| VP -> VP PP [0.6] 0.0012648960
Discard: |=.....| VP -> V NP [0.3] 0.0002108160
Discard: |=.....| VP -> V NP [0.3] 0.0002108160
Finding the most likely constituents spanning 7 text elements...
Insert: |=.....| S -> NP VP [1.0] 0.0002529792
(S
(NP I)
```

```
(VP
  (VP (V saw) (NP (Det the) (N man)))
  (PP (P with) (NP (Det a) (N telescope))))) (p=0.000252979)
S
├── NP
│   └── I
├── VP
│   ├── VP
│   │   ├── V
│   │   │   └── saw
│   │   ├── NP
│   │   │   ├── Det
│   │   │   │   └── the
│   │   │   └── N
│   │   │       └── man
│   ├── PP
│   │   ├── P
│   │   │   └── with
│   │   ├── NP
│   │   │   ├── Det
│   │   │   │   └── a
│   │   │   └── N
│   │   │       └── telescope
└──
```

Probability of New Parsing Tree: 0.0002529792

Graded Exercise 6 (1 point)

Given the following code for learning and using a grammar:

```
import nltk
from nltk.corpus import treebank
productions=[]
S=nltk.Nonterminal('S')


for f in treebank.fileids():
    for tree in treebank.parsed_sents(f):
        productions+=tree.productions()

grammar=nltk.induce_pcfg(S,productions)

for p in grammar.productions()[1:25]:
    print(p)

# Trace level 1
myparser_1 = nltk.ViterbiParser(grammar, trace=1)
text = "the boy jumps over the board"
mytokens = nltk.word_tokenize(text)
myparsing_1, = myparser_1.parse(mytokens)

# Trace level 2
myparser_2 = nltk.ViterbiParser(grammar, trace=2)
mytokens = nltk.word_tokenize(text)
myparsing_2, = myparser_2.parse(mytokens)
```


 [Mostrar salida oculta](#)

Execute this code, modifying the parameter trace (values from 1 to 2). Print the output tree and the probability.

As we now, when using trace=1, the parser prints a summary of parsing steps, including rule insertions, discards, and probability calculations. However, with trace=2, the output becomes even more detailed, displaying step-by-step parsing decisions for each possible structure. This results in hundreds of lines of output (that we decided to hide) as the parser evaluates different rule applications before selecting the most probable parse.

```
print("\nParse Tree with Trace 1:")
myparsing_1.pretty_print()
print("\nProbability:", myparsing_1.prob())

print("\nParse Tree with Trace 2:")
myparsing_2.pretty_print()
print("\nProbability:", myparsing_2.prob())
```

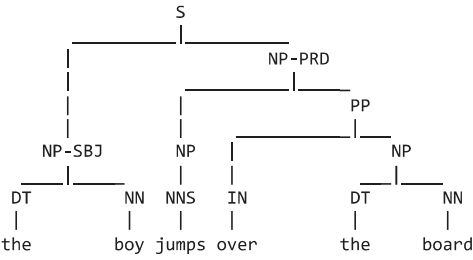
 Parse Tree with Trace 1:

```

      S
     / \
  NP-SBJ NP-PRD
  /  \   / \
DT  NN NNS PP
|   |   |  / \
the boy jumps over DT NN
                  the board
```

Probability: 1.2500082459723783e-20

Parse Tree with Trace 2:



Probability: 1.2500082459723783e-20