THE UNIVERSITY of ADELAIDE

COMP SCI 7327 Concepts in Artificial Intelligence & Machine Learning -NLP

By Dr Wei Zhang

Outline

- Natural Language Processing
- Basic pre-processing
- Natural Language Understanding (NLU)
  - Context-free Grammars

Natural Language Processing

"*Natural language processing is a range of computational techniques for analyzing and representing naturally occurring texts at one or more levels of linguistic analysis for the purpose of achieving human-like languages processing for a range of particular tasks or applications*."
by Liddy (1998)

- Some other names:
  - Computational Linguistics
  - Natural Language Engineering
  - Speech and Text Processing

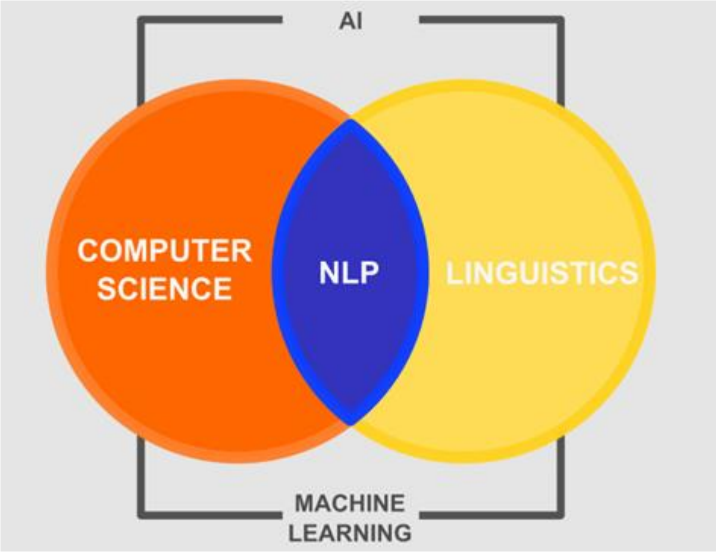
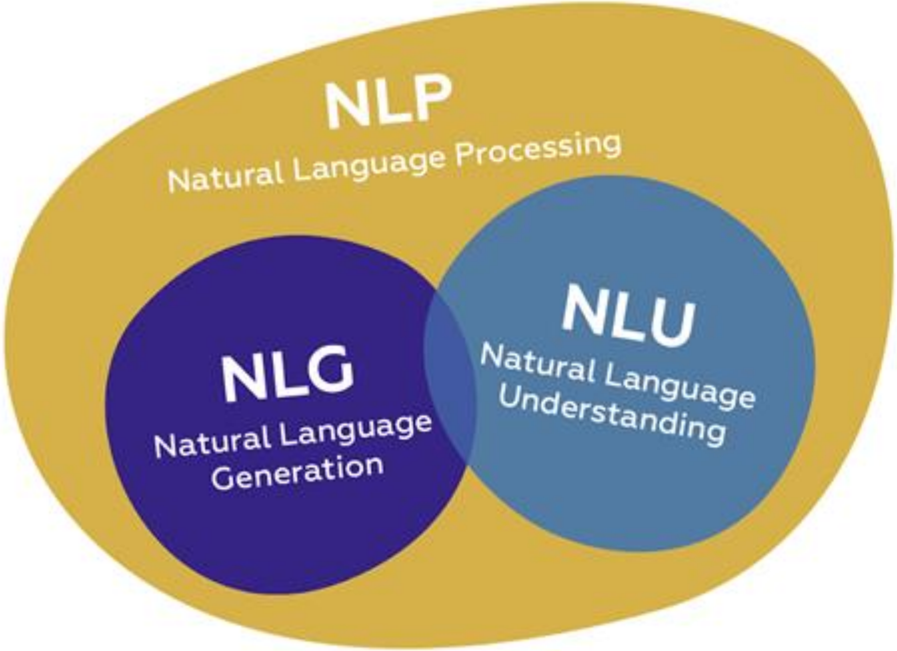


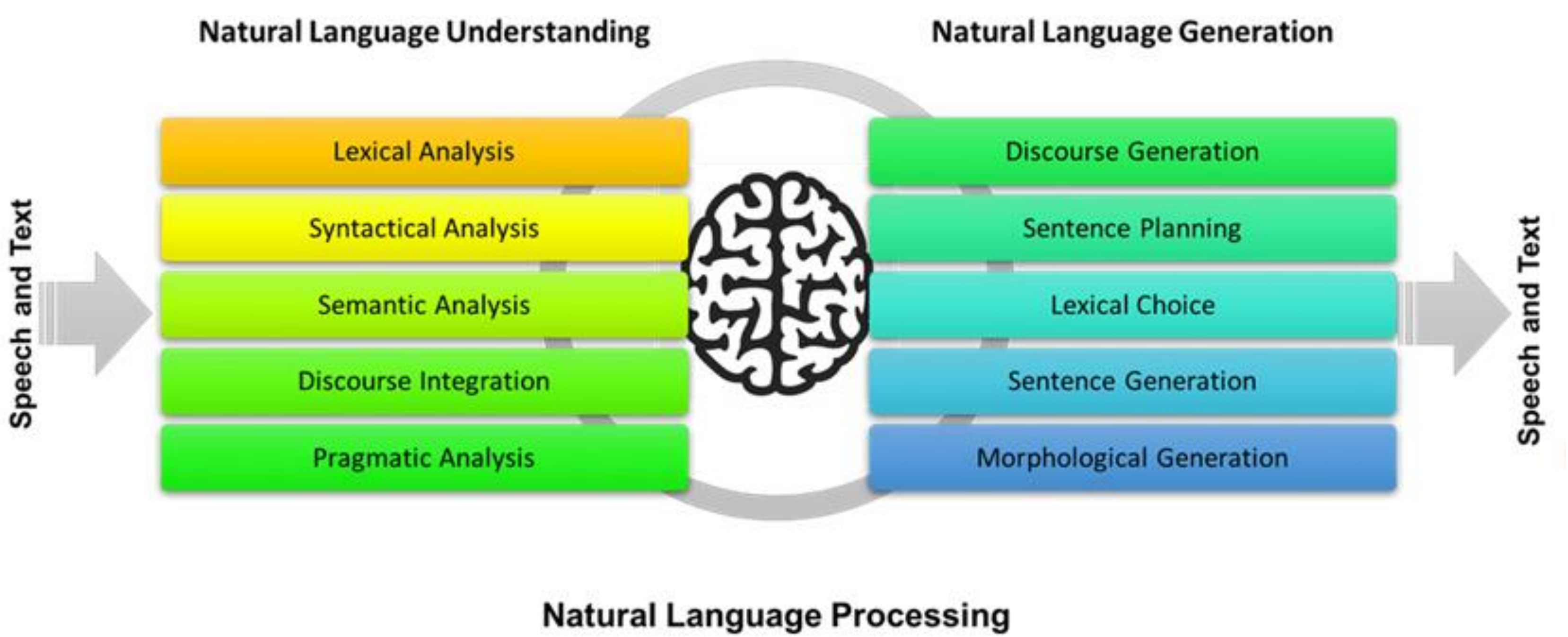
Image source: algorithmxlab.com

Natural Language Processing

- NLU
  - Syntactic parsing
  - Coreference resolution
  - Semantic parsing
  - Part-of-speech tagging (POS)
  - Named entity recognition (NER)
  - Natural language inference
  - Relation extraction
  - Text categorization
  - Sentiment analysis
  - ….
- NLU & NLG
  - Paraphrase
  - Dialogue agents
  - Question answering
  - Text summarization
  - Machine translation
  - …



Natural Language Processing



- Dialogue system:
  - A typical application to combine NLU and NLG

Image source: FinTechXpert

Outline

- Natural Language Processing
- Basic pre-processing
- Natural Language Understanding (NLU)
  - Context-free Grammars

Basic Pre-Processing

- Regular expression
- Word tokenization

Regular Expressions

- A formal language for specifying text strings
- How can we search for any of these?
  - woodchuck
  - woodchucks
  - Woodchuck
  - Woodchucks

Regular Expressions: Disjunctions

- Letters inside square brackets []

Pattern	Matches
`[wW]oodchuck`	Woodchuck, woodchuck
`[1234567890]`	Any digit

- Ranges `[A-Z]`

Pattern	Matches	Example Patterns Matched
`[A-Z]`	An upper case letter	`Drenched Blossoms`
`[a-z]`	A lower case letter	`my beans were impatient`
`[0-9]`	A single digit	`Chapter 1: Down the Rabbit Hole`

Regular Expressions: In Python

```python
import re

words = ['Woodchuck', 'woodchuck' , 'chuck', 'wWoodchuck']

pattern = re.compile(r"[wW]oodchuck")

print ("========match/search========")
for wd in words:
	t=re.match(pattern,wd)
	if (t):
		print(t.group(0))
	else:
	    print ("Not match \"%s\"" % wd)
```

```
========match/search========
Woodchuck
woodchuck
Not match "chuck"
Not match "wWoodchuck"
```

```python
strs = ['Drenched Blossoms', 'my beans were impatient', 'Chapter 1: Down the Rabbit Hole']
```

```python
pattern = re.compile(r"[A-Z]")
for s in strs:
	t=re.match(pattern,s)
	if (t):
		print(t.group(0))
	else:
	    print ("Not match \"%s\"" % s)
```

```
D
Not match "my beans were impatient"
C
```

Regular Expressions: Negation in Disjunction

- Negations `[^Ss]`
  - Carat means negation only when first in []

Pattern	Matches (single character)	Example Patterns Matched
`[^A-Z]`	Not an upper case letter	`Oyfn pripetchik`
`[^Ss]`	Neither ‘S’ nor ‘s’	`I have no exquisite reason`
`[^e^]`	Neither e nor ^	`Look here`
`a^b`	The pattern a carat b	`Look up a^b now`

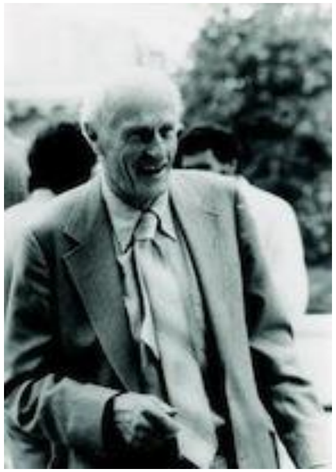
Regular Expressions: More Disjunction

- Woodchucks is another name for groundhog!
- The pipe | for disjunction

Pattern	Matches
groundhog|woodchuck	groundhog woodchuck
yours|mine	yours mine
a|b|c	= [abc]
[gG]roundhog|[Ww]oodchuck	groundhog Groundhog Woodchuck woodchuck

Regular Expressions: ? * + .

Pattern	Matches	Example Patterns Matched
colou?r	Optional previous char	color colour
oo*h!	0 or more of previous char	oh! ooh! oooh! ooooh! …
o+h!	1 or more of previous char	oh! ooh! oooh! ooooh! …
baa+	=baaa*	baa baaa baaaa baaaaa
beg.n	any character between *beg* and *n*	begin beg'n begun beg3n



Stephen C Kleene

Kleene *, Kleene +

Regular Expressions: Anchors ^ $

- The caret ^ matches the start of a line.
- The dollar sign $ matches the end of a line.

Pattern	Matches	Example Patterns Matched
`^[A-Z]`	Starting symbol is an upper case letter	Palo Alto
`^[^A-Za-z]`	Starting character is not letter.	1 "Hello"
`\.$`		The end.
`.$`		The end? The end!

Example

- Find me all instances of the word "the" in a text.

  `the`

  Misses capitalized examples: *The*

  `[tT]he`

  Incorrectly returns *other* or *theology*

  `[^a-zA-Z]*[tT]he[^a-zA-Z]|^[tT]he$`

Example in Python

```
strs = ['It is the cat, that is the dog', 'the dog', 'The dog', 'other thing', 'what is theology', 'the' ]

pattern = re.compile(r"[^a-zA-Z]*[tT]he[^a-zA-Z]*")
for s in strs:
	t=re.findall(pattern,s)
	if (t):
		print(t)
	else:
	    print ("Not match \"%s\"" % s)
```

```
[' the ', ' the ']
['the ']
['The ']
['the']
[' the']
['the']
```

Regular Expressions

- Regular expressions play a surprisingly large role
  - Sophisticated sequences of regular expressions are often the first model for any text processing text
- For many hard tasks, we use machine learning classifiers
  - But regular expressions are used as features in the classifiers
  - Can be very useful in capturing generalizations

Tokenization

- How many words?

	Tokens = N	Distinct words = $\|V\|$
Switchboard phone conversations	2.4 million	20 thousand
Shakespeare	884,000	31 thousand
Google N-grams	1 trillion	13 million

N = number of tokens

V = vocabulary = number of distinct words

$|V|$ is the size of the vocabulary

Simple Tokenization in UNIX

- Given a text file, output the word tokens and their frequencies

```
tr -sc 'A-Za-z' '\n' < shakes.txt
     | sort
     | uniq -c
```

Change all non-alpha to newlines

Sort in alphabetical order

Merge and count each type

```
1945 A
72 AARON
19 ABBESS
25 Aaron
6 Abate
1 Abates
5 Abbess
6 Abbey
3 Abbot
....
```

Simple Tokenization in Python

```
import nltk
```

```
sent = "This sentence is going to be tokenized."
```

```
sent.split()
```

```
['This', 'sentence', 'is', 'going', 'to', 'be', 'tokenized.']
```

```
from nltk import word_tokenize
word_tokenize(sent)
```

```
['This', 'sentence', 'is', 'going', 'to', 'be', 'tokenized', '.']
```

Issues in Tokenization

- Finland's capital → Finland Finlands Finland's ?
- what're, I'm, isn't → What are, I am, is not
- Hewlett-Packard → Hewlett Packard ?
- state-of-the-art → state of the art ?
- Lowercase → lower-case lowercase lower case ?
- San Francisco → one token or two?
- m.p.h., PhD. → ??

Tokenization with Regular Expressions

```
import nltk
```

```
text = 'That U.S.A. poster-print costs $12.40...'
```

```
pattern = r'''(?x)          # set flag to allow verbose regexps
        (?:[A-Z]\.)+        # abbreviations, e.g. U.S.A.
      | \w+(?:-\w+)*        # words with optional internal hyphens
      | \$?\d+(?:\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
      | \.\.\.              # ellipsis
      | [][.,;"'?():_`-]    # these are separate tokens; includes ], [
    '''
```

```
nltk.regexp_tokenize(text, pattern)
```

```
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Tokenization: language issues

- Chinese and Japanese no spaces between words:
  - Sharapova now lives in US southeastern Florida
    莎拉波娃现在居住在美国东南部的佛罗里达。

  - 莎拉波娃 现在 居住 在 美国 东南部 的 佛罗里达
  - Sharapova now lives in US southeastern Florida

Word Tokenization in Chinese

- Also called **Word Segmentation**
- Chinese words are composed of characters
  - Characters are generally 1 syllable and 1 morpheme (a single unit of meaning).
  - Average word is 2.4 characters long.
- Standard baseline segmentation algorithm:
  - Maximum Matching

Maximum Matching Word Segmentation Algorithm

- Given a wordlist of Chinese, and a string.
  1) Start a pointer at the beginning of the string
  2) Find the longest word in dictionary that matches the string starting at pointer
  3) Move the pointer over the word in string
  4) Go to 2

Max-match Segmentation Illustration

- Thecatinthehat
- Thetabledownthere

the cat in the hat

the table down there

theta bled own there

- Doesn't generally work in English!

- But works astonishingly well in Chinese
  - 莎拉波娃现在居住在美国东南部的佛罗里达。
  - 莎拉波娃 现在 居住 在 美国 东南部 的 佛罗里达

  Sharapova now lives in US southeastern Florida

- Modern probabilistic segmentation algorithms even better

Outline

- Natural Language Processing
- Basic pre-processing
- Natural Language Understanding (NLU)
  - Context-free Grammars

Natural Language Understanding

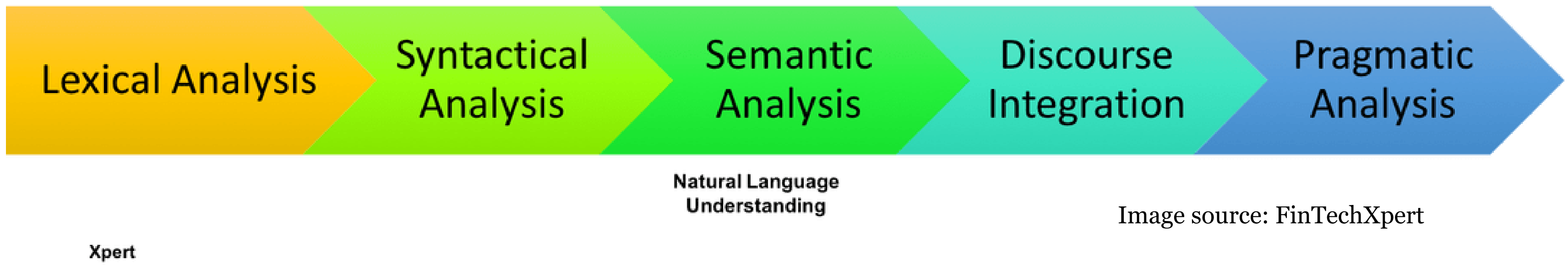


Image source: FinTechXpert

- Syntactical Analysis
  - The word syntax comes from the Greek *syntaxis*, meaning "setting out together or arrangement", and refers to the way words are arranged together.
  - Syntactic parsing is the task of recognizing a sentence and assigning a syntactic structure to it.
  - Context free grammars are used for syntactic parsing. And they are the backbone of many formal models of the syntax of natural language.

Context-free Grammars

- A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.
- A CFG consists of the following components:
  - a set of terminal symbols
  - a set of non-terminal symbols
  - a set of productions
  - a start symbol.

An Example

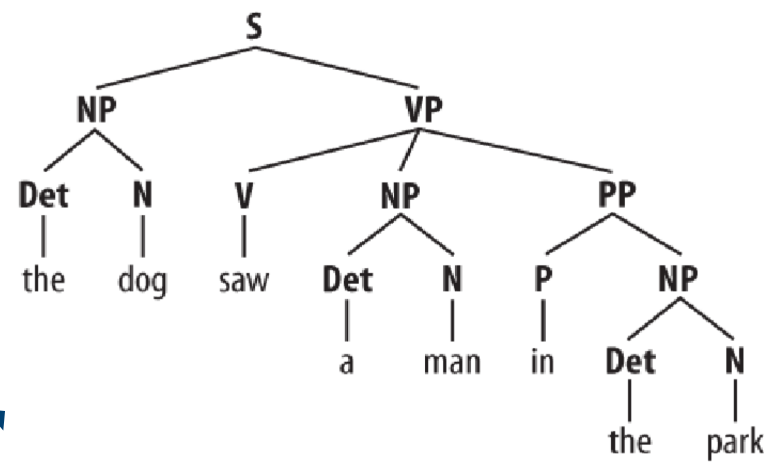
Context free grammar:

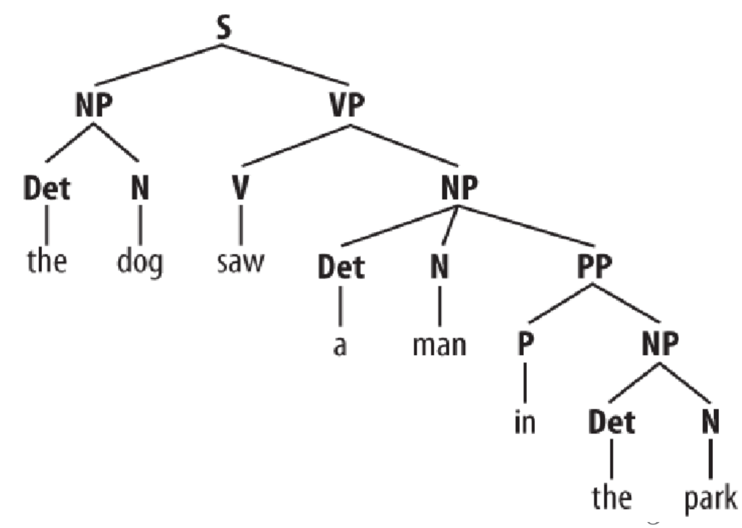
S -> NP VP
NP -> Det N | Det N PP
VP -> V NP | V NP PP
PP -> P NP
Det -> "the" | "a"
N -> "man" | "dog" | "park"
V -> "saw"
P -> "in"

The sentence to be parsed:

The dog saw a man in the park.

Syntax Trees:





Syntactic Categories

Symbol	**Meaning**	**Example**
S	sentence	*the dog saw a man in the park*
NP	noun phrase	*the dog*
VP	verb phrase	*saw a man*
PP	prepositional phrase	*in the park*
Det	determiner	*the*
N	noun	*dog*
V	verb	*saw*
P	preposition	*in*

Context-free Grammars

- Start symbol: S
- Production rule: {S -> NP VP, ...}
- Non-terminal symbols: {S, NP, VP, PP, Det, N, V, P, ...}
- Terminal symbols: {"dog", "man", "a", "saw", "in", ...}

A Context-Free Grammar in Python

```python
from nltk import CFG
cfg = CFG.fromstring("""
S -> NP VP
NP -> Det N | Det N PP
VP -> V NP | V NP PP
PP -> P NP
Det -> "the" | "a"
N -> "man" | "dog" | "park"
V -> "saw"
P -> "in"
""")

print(cfg)
```

Output

```
Grammar with 13 productions (start state = S)
    S -> NP VP
    NP -> Det N
    NP -> Det N PP
    VP -> V NP
    VP -> V NP PP
    PP -> P NP
    Det -> 'the'
    Det -> 'a'
    N -> 'man'
    N -> 'dog'
    N -> 'park'
    V -> 'saw'
    P -> 'in'
```

Recursive Descent Parser

- A recursive descent parser is a top-down parser that breaks a high-level goal into several lower-level sub-goals.
- The top-level goal is to find **`S`**.
- The **`S -> NP VP`** production allows the parser to replace this goal with two sub-goals: find an **`NP`**, then find a **`VP`**.
- Each sub-goal can in turn be replaced by sub-sub-goals.
- Eventually, this process leads to sub-goals such as: find the determiner **`the`**.

Recursive Descent Parser

- The recursive descent parser builds a parse tree during this process.
- With the initial goal (find an **`S`**), the **`S`** root node is created.
- As the process recursively expands its goals using the production rules of the grammar, the parse tree is extended downwards.
- During this process, the parser is often forced to choose between alternative productions.
- When a particular production does not work, then the parser has to backtrack.

Recursive Descent Parsing

- Initial stage:

```
S -> NP VP
NP -> Det N
NP -> Det N PP
VP -> V NP
VP -> V NP PP
PP -> P NP
Det -> 'the'
Det -> 'a'
N -> 'man'
N -> 'dog'
N -> 'park'
V -> 'saw'
P -> 'in'
```

Recursive Descent Parsing

- Second production:

```
S -> NP VP
NP -> Det N
NP -> Det N PP
VP -> V NP
VP -> V NP PP
PP -> P NP
Det -> 'the'
Det -> 'a'
N -> 'man'
N -> 'dog'
N -> 'park'
V -> 'saw'
P -> 'in'
```

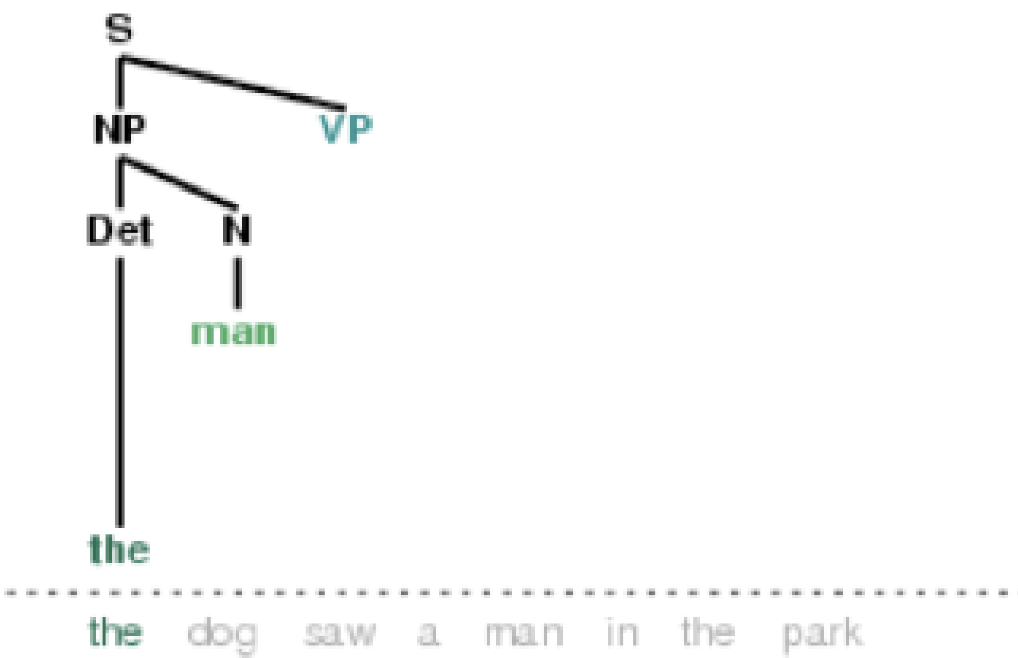
Recursive Descent Parsing

- Matching determiner *the*:


```
S -> NP VP
NP -> Det N
NP -> Det N PP
VP -> V NP
VP -> V NP PP
PP -> P NP
Det -> 'the'
Det -> 'a'
N -> 'man'
N -> 'dog'
N -> 'park'
V -> 'saw'
P -> 'in'
```

Recursive Descent Parsing

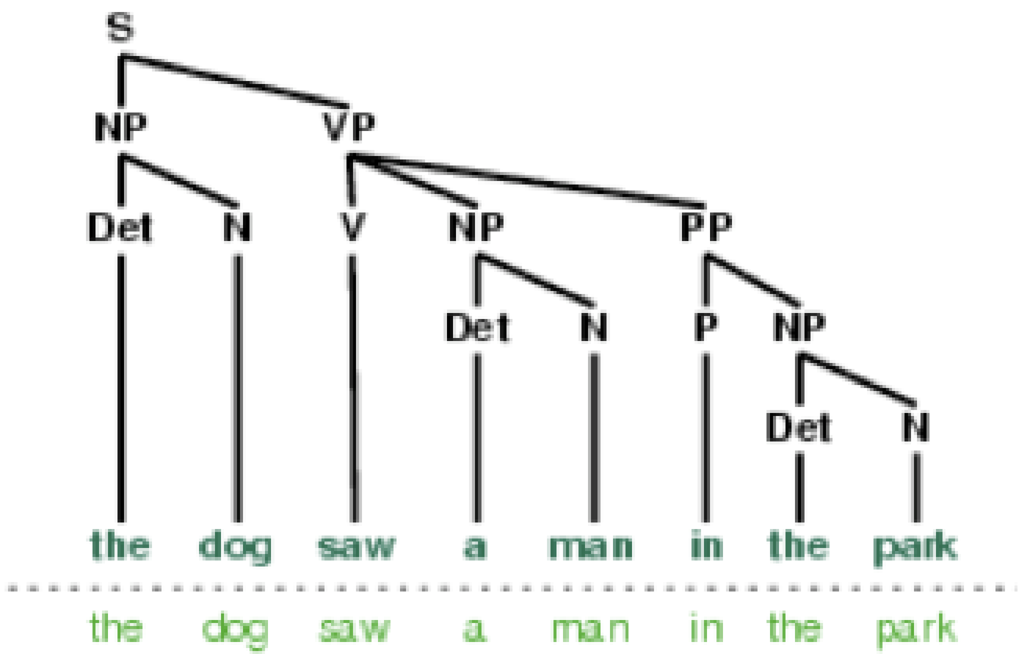
- Cannot match *man*:




```
S -> NP VP
NP -> Det N
NP -> Det N PP
VP -> V NP
VP -> V NP PP
PP -> P NP
Det -> 'the'
Det -> 'a'
N -> 'man'
N -> 'dog'
N -> 'park'
V -> 'saw'
P -> 'in'
```

Recursive Descent Parsing

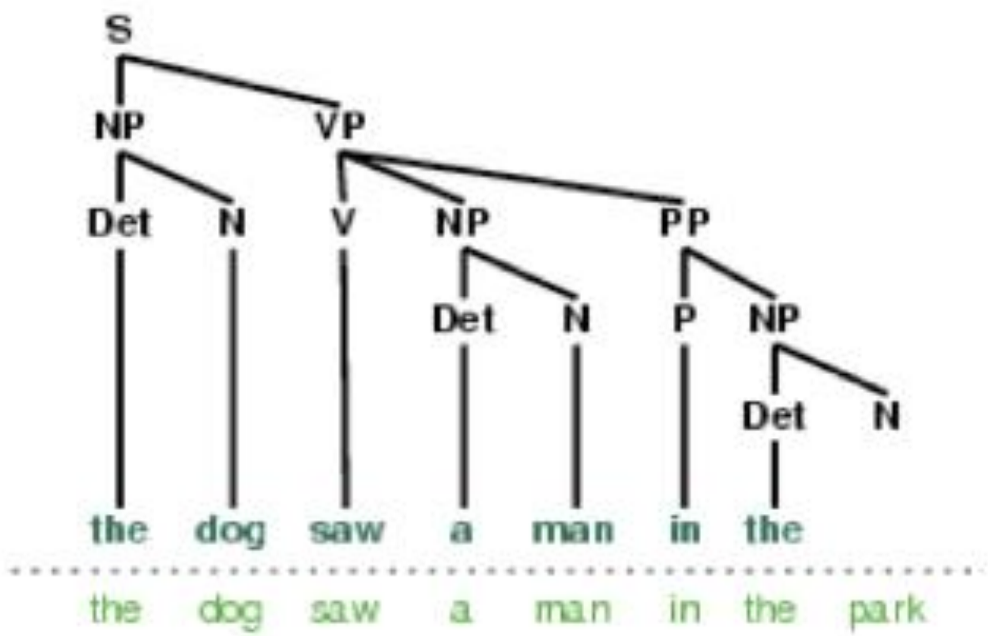
- Complete parse:




```
S -> NP VP
NP -> Det N
NP -> Det N PP
VP -> V NP
VP -> V NP PP
PP -> P NP
Det -> 'the'
Det -> 'a'
N -> 'man'
N -> 'dog'
N -> 'park'
V -> 'saw'
P -> 'in'
```

Recursive Descent Parsing

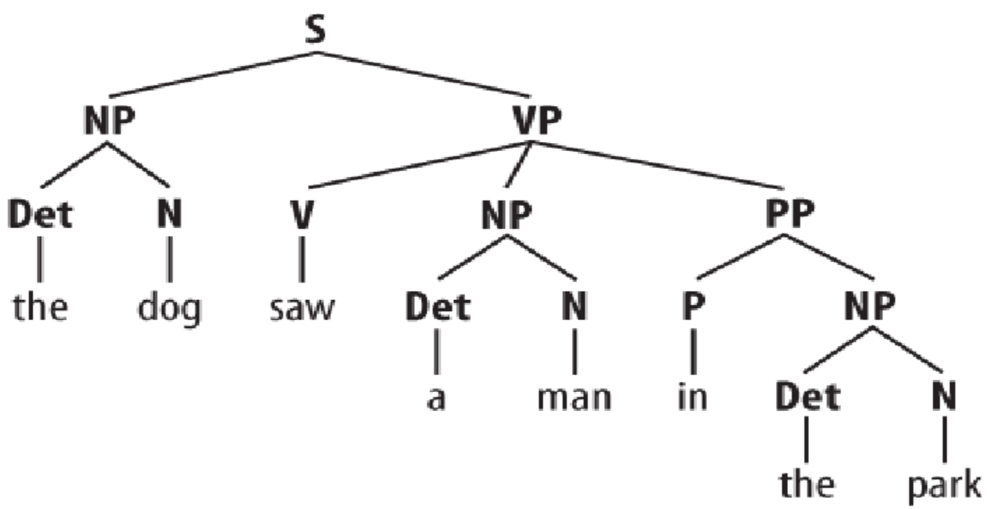
- Backtracking for second interpretation:




```
S -> NP VP
NP -> Det N
NP -> Det N PP
VP -> V NP
VP -> V NP PP
PP -> P NP
Det -> 'the'
Det -> 'a'
N -> 'man'
N -> 'dog'
N -> 'park'
V -> 'saw'
P -> 'in'
```

First Syntax Tree

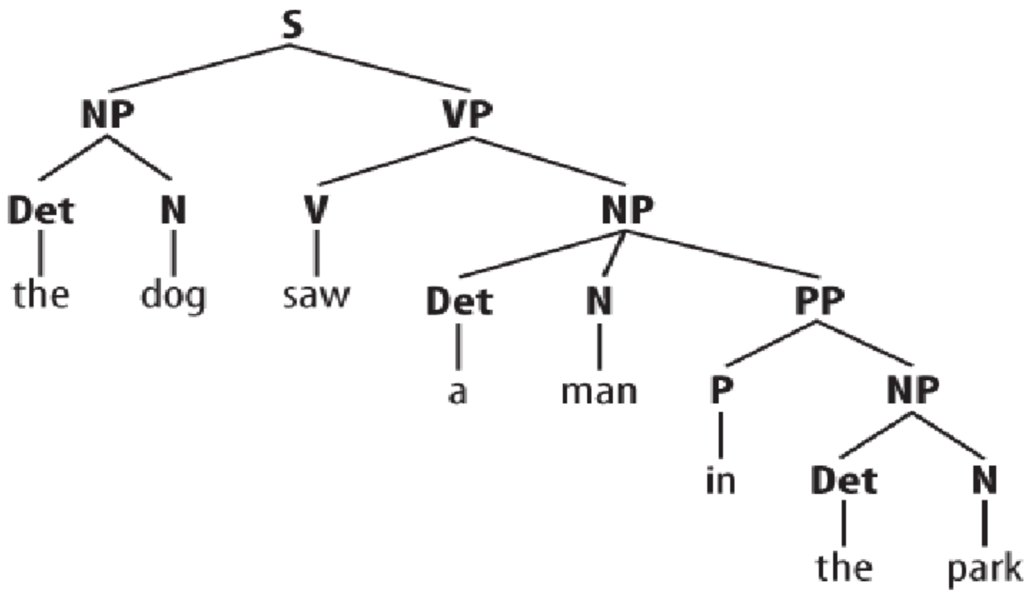
- Ambiguity: prepositional phrase attachment




```
S -> NP VP
NP -> Det N
NP -> Det N PP
VP -> V NP
VP -> V NP PP
PP -> P NP
Det -> 'the'
Det -> 'a'
N -> 'man'
N -> 'dog'
N -> 'park'
V -> 'saw'
P -> 'in'
```

Second Syntax Tree

- Ambiguity: prepositional phrase attachment




```
S -> NP VP
NP -> Det N
NP -> Det N PP
VP -> V NP
VP -> V NP PP
PP -> P NP
Det -> 'the'
Det -> 'a'
N -> 'man'
N -> 'dog'
N -> 'park'
V -> 'saw'
P -> 'in'
```

A Recursive Descent Parser in Python

```python
# Recursive Descent Parser (top-down, depth-first)
from nltk import RecursiveDescentParser

sent = "the dog saw a man in the park".split()

rd_parser = RecursiveDescentParser(cfg)

for tree in rd_parser.parse(sent):
    print(tree)
```

Output

```
(S
  (NP (Det the) (N dog))
  (VP
    (V saw)
    (NP (Det a) (N man) (PP (P in) (NP (Det the) (N park))))))

(S
  (NP (Det the) (N dog))
  (VP
    (V saw)
    (NP (Det a) (N man))
    (PP (P in) (NP (Det the) (N park)))))
```

Probabilistic Context-Free Grammars

- A probabilistic context-free grammar (or *PCFG*) is a context-free grammar that associates a probability with each of its productions.
- It generates the same set of parses for a text that the corresponding context-free grammar does, and assigns a probability to each parse.
- The probability of a parse generated by a PCFG is simply the product of the probabilities of the productions used to generate it.

A PCFG in Python

```python
from nltk import PCFG
pcfg = PCFG.fromstring("""
S -> NP VP      [1.0]
NP -> Det N     [0.7]
NP -> Det N PP [0.3]
VP -> V NP      [0.6]
VP -> V NP PP  [0.4]
PP -> P NP      [1.0]
Det -> "the"    [0.6]
Det -> "a"      [0.4]
N -> "man"      [0.4]
N -> "dog"      [0.3]
N -> "park"     [0.3]
V -> "saw"      [1.0]
P -> "in"       [1.0]
""")
```

Viterbi Parser in Python

```
from nltk import ViterbiParser

sent = "the dog saw a man in the park".split()

viterbi_parser = ViterbiParser(pcfg, trace=2)

for tree in viterbi_parser.parse(sent):
    print(tree)
```

Output

```
(S
  (NP (Det the) (N dog))
  (VP
    (V saw)
    (NP (Det a) (N man))
    (PP (P in) (NP (Det the) (N park))))) (p=0.000711245)
```

Some Online Parsers

- Stanford Parser:
  - http://nlp.stanford.edu:8080/parser/index.jsp

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

  - http://www.link.cs.cmu.edu/link/submit-sentence-4.html

```
(S (NP the dog)
   (VP saw
       (NP a man)
       (PP in
           (NP the park))))
```

References