Parsing

NLTK 8. Analyzing Sentence Structure

http://www.nltk.org/book/ch08.html

SLP 12. Constituency Grammars

https://web.stanford.edu/~jurafsky/slp3/12.pdf

Outline

There are an **unlimited number of possible sentences**.

1. How can we use a **formal grammar** to describe the structure of an unlimited set of sentences?
2. How do we **represent** the structure of sentences using syntax trees?
3. How do parsers **analyze a sentence** and automatically build a syntax tree?

Grammar

- Speakers of a language can make judgements about sentences, as being grammatical or ungrammatical
- The purpose of a **grammar** is to give an **explicit description of a language**
- The formal framework of "**generative grammar**" (Chomsky, 1965) considers a "language" as a collection of all grammatical sentences, and a **grammar is a formal notation** that can be used for "generating" the members of this set

Syntax

- The word syntax comes from the Greek sy´ntaxis, meaning “setting out together or arrangement”, and refers to the way words are arranged together
- We have seen various syntactic notions in previous chapters
  - regular expressions
  - probabilities for these word sequences
  - part-of-speech categories could act as a kind of equivalence class for words
- Formalism
  - **constituency grammars**
  - **dependency parsing**

Constituent structure

- The fundamental notion underlying constituency is abstraction — groups of words behaving as single units, or **constituents**
- Developing a grammar involves discovering the inventory of constituents present in the language
- A sequence of words in a well-formed sentence can be replaced by a **shorter sequence**. Each sequence that forms a unit can be replaced by a **single word**, ending up with just two elements

the	little	bear	saw	the	fine	fat	trout	in	the	brook
the	bear		saw	the	trout			in	it	
He			saw	it				there		
He			ran					there		
He			ran							

Constituent structure

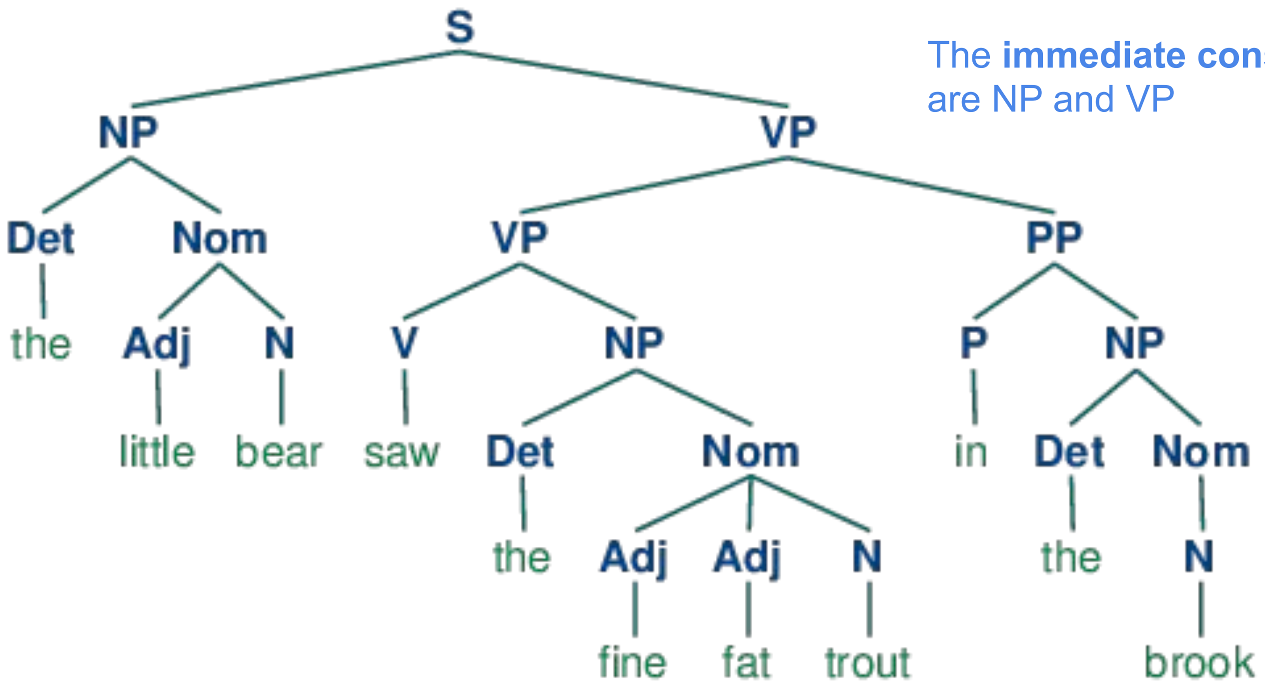
- Grammatical category labels to the words

Det the	Adj little	N bear	V saw	Det the	Adj fine	Adj fat	N trout	P in	Det the	N brook
Det the	Nom bear		V saw	Det the	Nom trout			P in	NP it	
NP He			V saw	NP it				PP there		
NP He			VP ran					PP there		
NP He			VP ran							

Syntactic Categories

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

- A **phrase structure tree** can be obtained by removing the words apart from the topmost row, adding an S node, and flipping the figure over
- Each node in this tree is called a **constituent**. e.g., NP and VP are constituents of S



The **immediate constituents** of S are NP and VP

Context-Free Grammar (CFG)
or
Phrase-Structure Grammars

Context-Free Grammar (CFG)

- The most widely used formal system for modeling constituent structure in English
- CFGs are the backbone of many formal models of the syntax of natural language (and of computer languages)
- CFGs are integral to many computational applications, including grammar checking, semantic interpretation, dialogue understanding, and machine translation
- They are powerful enough to express sophisticated relations among the words in a sentence, yet computationally tractable enough that efficient algorithms exist for parsing sentences with them
- Also called Phrase-Structure Grammars, and the formalism is equivalent to Backus-Naur Form (BNF). The idea of basing a grammar on constituent structure dates back to the psychologist Wilhelm Wundt (1900) but was not formalized until Chomsky (1956) and, independently, Backus (1959)

Context-Free Grammar (CFG)

- Consists of a set of **rules** or **productions**, each of which expresses the ways that symbols of the language can be grouped and ordered together
- The symbols are divided into two classes
  - **Terminal symbols** denote words
  - **Non-terminals** express abstractions over these terminals
- All well-formed trees must have **S (sentence) as the start-symbol** of the grammar

```
S -> NP VP
VP -> V NP | V NP PP
PP -> P NP
V -> "saw" | "ate" | "walked"
NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "man" | "dog" | "cat" | "telescope" | "park"
P -> "in" | "on" | "by" | "with"
```

CFG for analyzing and generating sentences

- **Analyze** if the following sentences are grammatical or not using the CFG below (assuming case insensitive)
  - “man saw a dog”
  - “My cat walked the telescope”
- **Generate** a sentence using the CFG on the left side

```
S -> NP VP
VP -> V NP | V NP PP
PP -> P NP
V -> "saw" | "ate" | "walked"
NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "man" | "dog" | "cat" | "telescope" | "park"
P -> "in" | "on" | "by" | "with"
```

Analyzing a sentence

Grammar Definition

```
grammar1 = nltk.CFG.fromstring("""
  S -> NP VP
  VP -> V NP | V NP PP
  PP -> P NP
  V -> "saw" | "ate" | "walked"
  NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
  Det -> "a" | "an" | "the" | "my"
  N -> "man" | "dog" | "cat" | "telescope" | "park"
  P -> "in" | "on" | "by" | "with"
  """)
```

Parsing

```
>>> sent = "Mary saw Bob".split()
>>> rd_parser = nltk.RecursiveDescentParser(grammar1)
>>> for tree in rd_parser.parse(sent):
...     print(tree)
(S (NP Mary) (VP (V saw) (NP Bob)))
```

Ambiguity

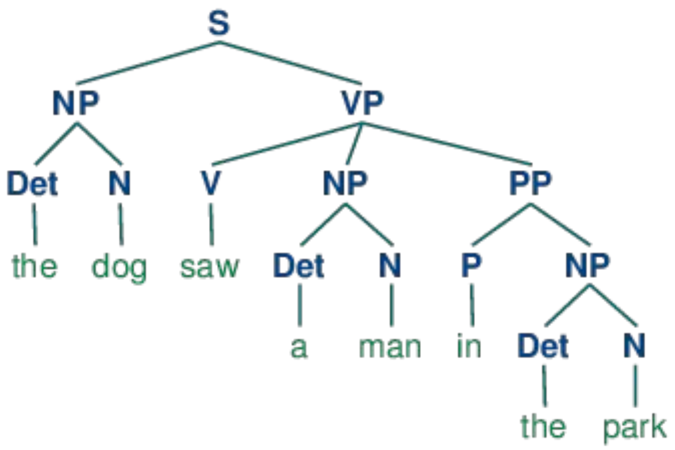
- The sentence "the dog saw a man in the park" is **structurally ambiguous**, generating two trees
- This is an example of prepositional phrase **attachment ambiguity**
- **Coordination ambiguity** involves conjunction such as and
    - ‘Old men and women’ can be interpreted as [old [men and women]], [old men and old women], [old men] and [women]

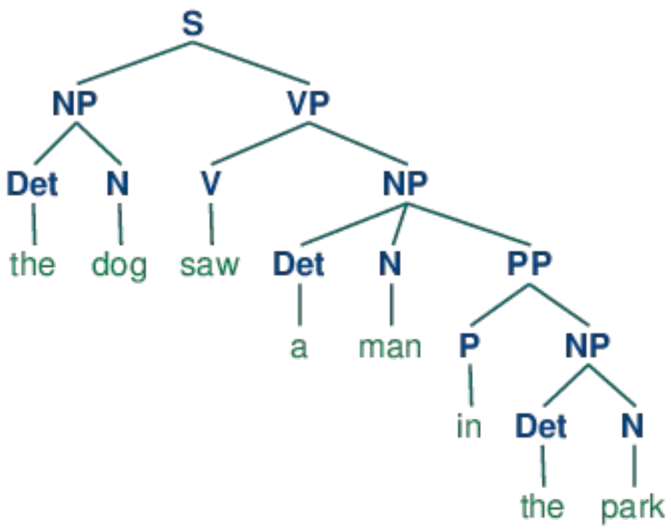
```
grammar1 = nltk.CFG.fromstring("""
  S -> NP VP
  VP -> V NP | V NP PP
  PP -> P NP
  V -> "saw" | "ate" | "walked"
  NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
  Det -> "a" | "an" | "the" | "my"
  N -> "man" | "dog" | "cat" | "telescope" | "park"
  P -> "in" | "on" | "by" | "with"
  """)

>>> parser = nltk.ChartParser(grammar1)
>>> for tree in parser.parse(sent):
...     print(tree)

(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)))))
```

PP Attachment Ambiguity

```
(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))))))
```

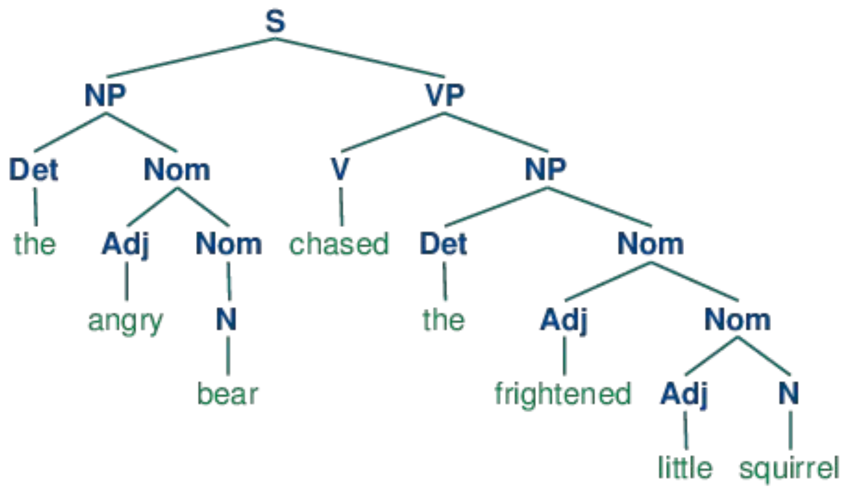




Recursion in Syntactic Structure

- A grammar is **recursive** if a category occurring on the left hand side of a production also appears on the right-hand side of a production
- *Nom -> Adj Nom* involves direct recursion on the category *Nom*

```
S  -> NP VP
NP -> Det Nom | PropN
Nom -> Adj Nom | N
VP -> V Adj | V NP | V S | V NP PP
PP -> P NP
PropN -> 'Buster' | 'Chatterer' | 'Joe'
Det -> 'the' | 'a'
N -> 'bear' | 'squirrel' | 'tree' | 'fish' | 'log'
Adj  -> 'angry' | 'frightened' |  'little' | 'tall'
V ->  'chased'  | 'saw' | 'said' | 'thought' | 'was' | 'put'
P -> 'on'
```



Parser

Parsing With Context Free Grammar

- A **grammar** is a declarative specification of **well-formedness**
- A parser processes input sentences **according to the productions of a grammar**, and builds constituent structures that conform to the grammar
- A **parser** is an algorithm which **searches** through the space of trees to find one that has the required sentence along its fringe
- **Syntactic parsing** is the task of assigning a syntactic structure to a sentence
- Parsers employ these grammars to efficiently produce correct parse trees
- Simple parsing algorithms
  - **top-down method**: **recursive descent parsing**
  - **bottom-up method: shift-reduce parsing**

```
grammar1 = nltk.CFG.fromstring("""
  S -> NP VP
  VP -> V NP | V NP PP
  PP -> P NP
  V -> "saw" | "ate" | "walked"
  NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
  Det -> "a" | "an" | "the" | "my"
  N -> "man" | "dog" | "cat" | "telescope" | "park"
  P -> "in" | "on" | "by" | "with"
  """)
```

Recursive Descent Parsing

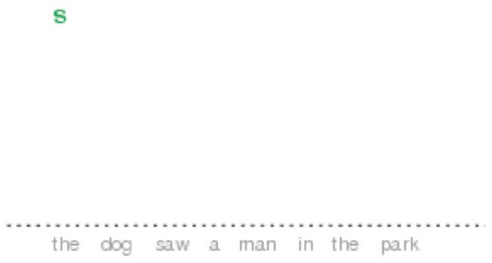
- Breaks a high-level goal into lower-level subgoals
  - The **top-level goal** is to find S
  - *S → NP VP* permits the parser to replace *S* with two **subgoals**: find NP, then find VP. Replace each of these subgoals in turn by sub-sub-goals, using productions that have NP and VP on their left-hand side
  - If there is no match the parser must back up and try a different alternative
- The recursive descent parser builds a parse tree during the above process
- The parse tree is extended downwards (hence the name recursive descent)

```
>>> rd_parser = nltk.RecursiveDescentParser(grammar1)
>>> sent = 'Mary saw a dog'.split()
>>> for tree in rd_parser.parse(sent):
...     print(tree)
(S (NP Mary) (VP (V saw) (NP (Det a) (N dog))))
```

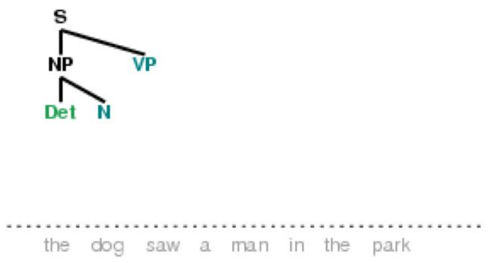
>>> nltk.app.rdparser()

- experiment with changing the sentence to be parsed by selecting Edit Text in the Edit menu.
- modify the grammar in the recursive descent parser demo by selecting Edit Grammar in the Edit menu. Change the second expansion production, namely NP -> Det N PP, to NP -> NP PP. Using the Step button, try to build a parse tree. What happens?

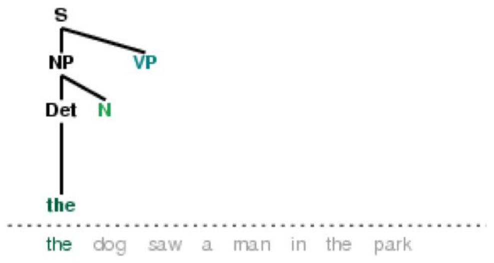
1. Initial stage



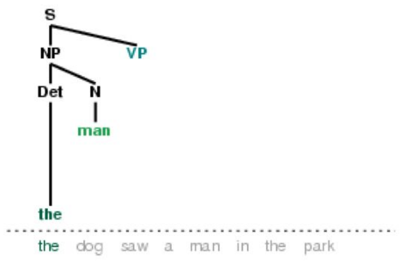
2. Second production



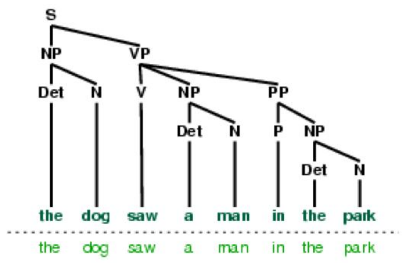
3. Matching *the*



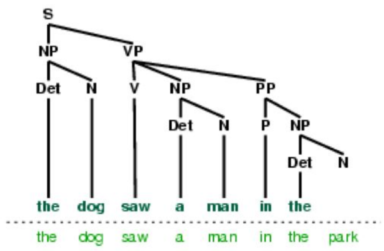
4. Cannot match *man*



5. Completed parse



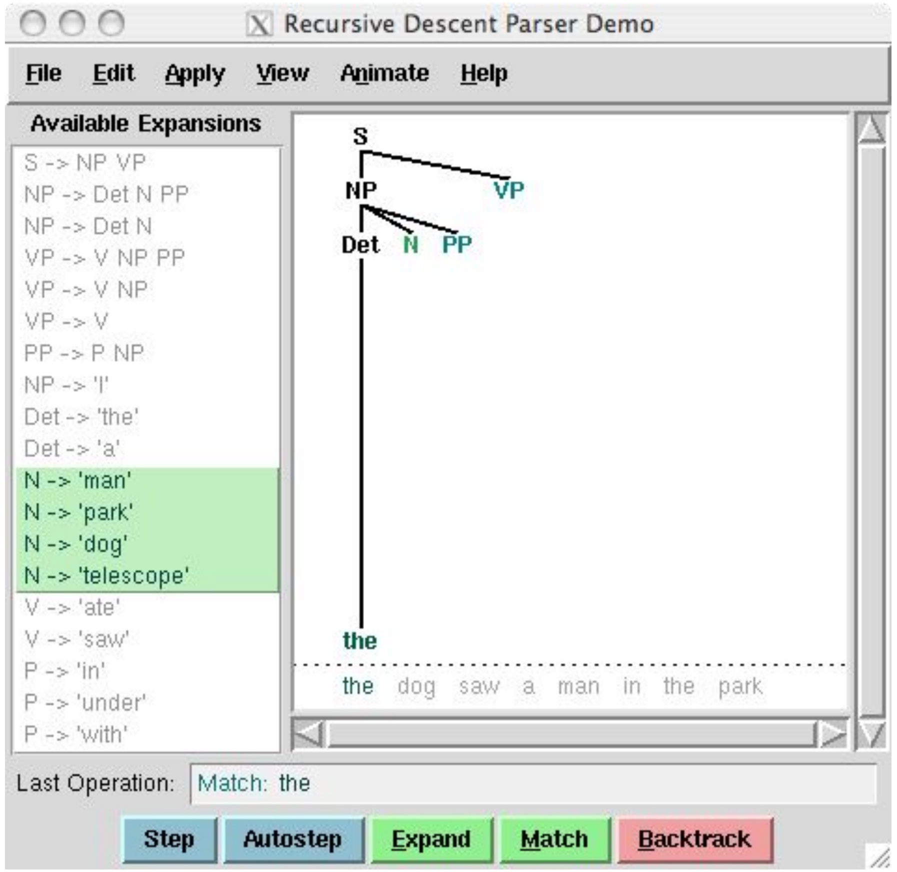
6. Backtracking



Recursive Descent Parser Demo

Demo shows the operation of a recursive descent parser as it grows the parse tree and matches it against the input words

```
>>> nltk.app.rdparser()
```



S -> NP VP
NP -> Det N PP
NP -> Det N
VP -> V NP PP
VP -> V NP
VP -> V
PP -> P NP
NP -> 'I'
Det -> 'the'
Det -> 'a'
N -> 'man'
N -> 'park'
N -> 'dog'
N -> 'telescope'
V -> 'ate'
V -> 'saw'
P -> 'in'
P -> 'under'
P -> 'with'

the dog saw a man in the park

Last Operation: Match: the

RDP shortcomings

- Left-recursive productions like NP -> NP PP send it into an **infinite loop**
- **Wastes a lot of time** considering words and structures that do not correspond to the input sentence
- The backtracking process may **discard parsed constituents that will need to be rebuilt again** later
    - E.g., backtracking over VP -> V NP will discard the subtree created for the NP. If the parser then proceeds with VP -> V NP PP, then the NP subtree must be created all over again.

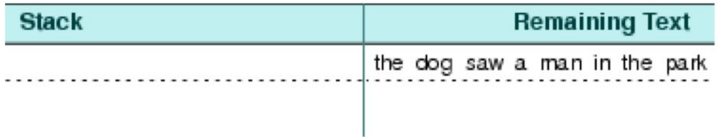
Shift-Reduce Parsing (Aho and Ullman, 1972)

- **Find** sequences of words and phrases that correspond to the **right hand side of a grammar production**, and **replaces them with the left-hand side**, until the whole sentence is reduced to S
- **Shift operation**: Repeatedly **push the next input word onto a stack**
- **Reduce operation: pop the top *n* items** on the stack **matching the *n* items on the right hand side** of some production, and **push** the item on the **left-hand side of the production**
- It **finishes** when only S remains on the stack

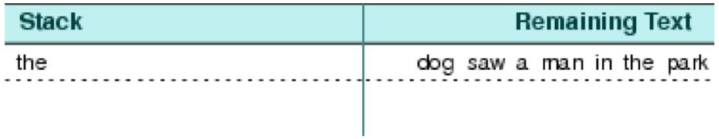
```
>>> sr_parser = nltk.ShiftReduceParser(grammar1)
>>> sent = 'Mary saw a dog'.split()
>>> for tree in sr_parser.parse(sent):
...     print(tree)
```

nltk.app.srparser()

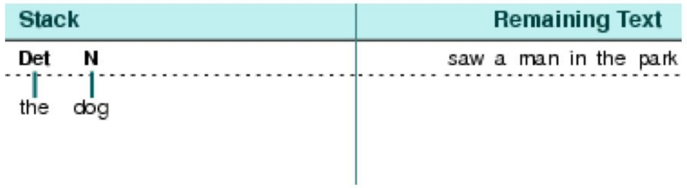
1. Initial state



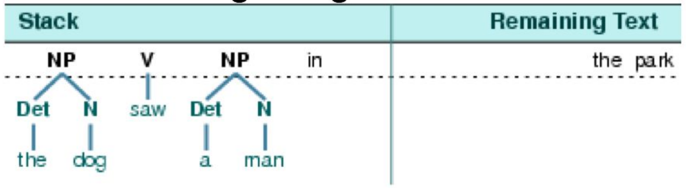
2. After one shift



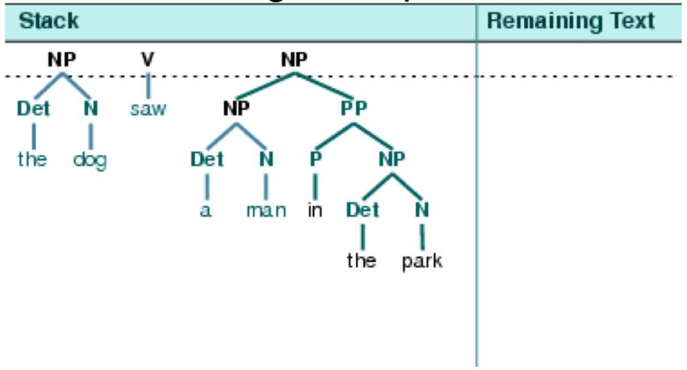
3. After reduce shift reduce



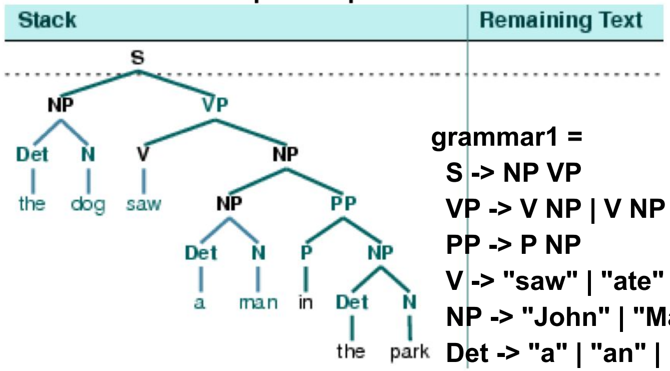
4. After recognizing the second NP



5. After building a complex NP



6. Built a complete parse tree



```
grammar1 =
  S -> NP VP
  VP -> V NP | V NP PP
  PP -> P NP
  V -> "saw" | "ate" | "walked"
  NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
  Det -> "a" | "an" | "the" | "my"
  N -> "man" | "dog" | "cat" | "telescope" | "park"
  P -> "in" | "on" | "by" | "with"
```

This parser is not guaranteed to find a valid parse for the input even if one exists, and builds substructure without checking whether it is globally consistent with the grammar

Applications

- Parse trees are directly useful in applications such as **grammar checking** in word-processing systems
- Parse trees serve as an intermediate stage of representation for semantic analysis and thus play an important role in applications like **question answering** and **information extraction**
    - E.g., to answer the question ‘What books were written by British women authors before 1800?’
    - Need to know the subject of the sentence

Dependency Parsing

NLTK 8. Analyzing Sentence Structure

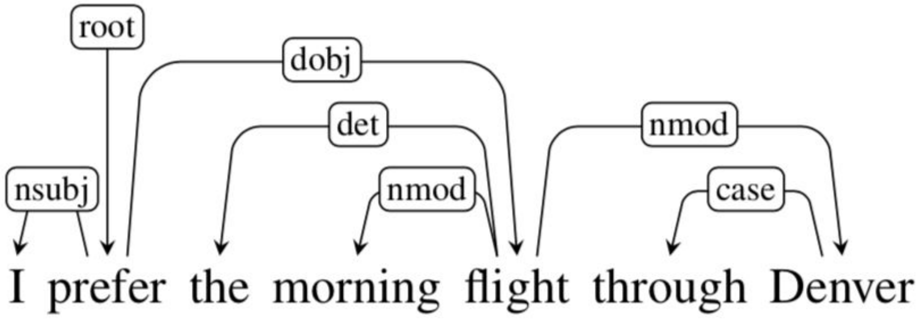
http://www.nltk.org/book/ch08.html

SLP 15. Dependency Parsing

https://web.stanford.edu/~jurafsky/slp3/15.pdf

Dependencies and Dependency Grammar

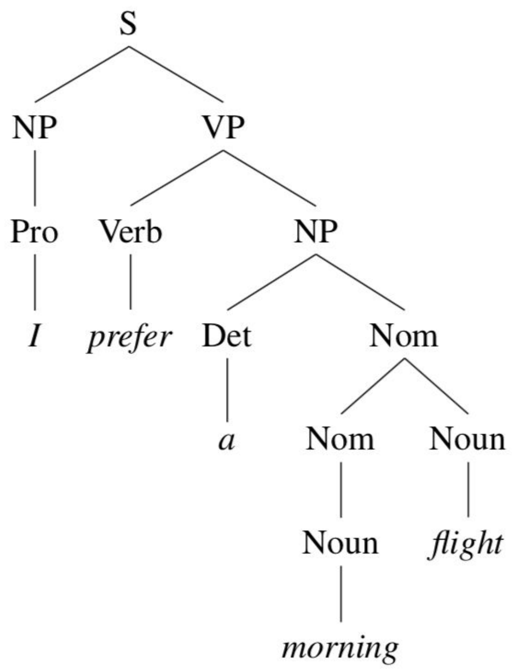
- **Constituency grammar** is concerned with how words combine to form constituents
- **Dependency grammar** focuses on **how words relate to other words**
- Dependency is a binary asymmetric relation that holds between a **head** and its **dependents.** The **head** of a sentence is usually the tensed **verb**, and every **other word is either dependent** on the sentence head, or **connects to it through a path of dependencies**
- A **dependency representation** is a **labeled directed graph**, where the nodes are the lexical items and the labeled arcs represent **dependency relations from heads to dependents**

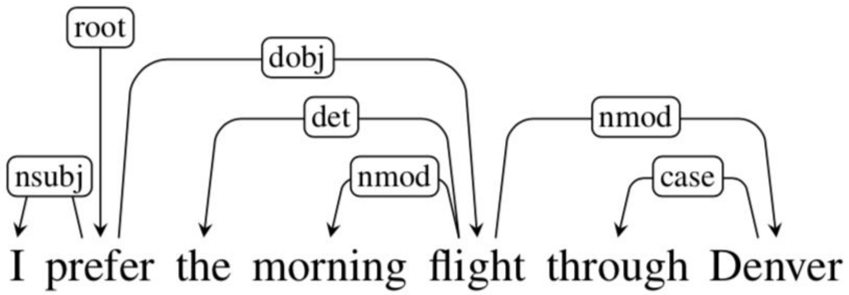


Phrasal constituents and phrase-structure rules do not play a direct role. Instead, the syntactic structure of a sentence is described solely in terms of the words in a sentence.

CFG vs. Dependency Parsing

I prefer the morning flight through Denver





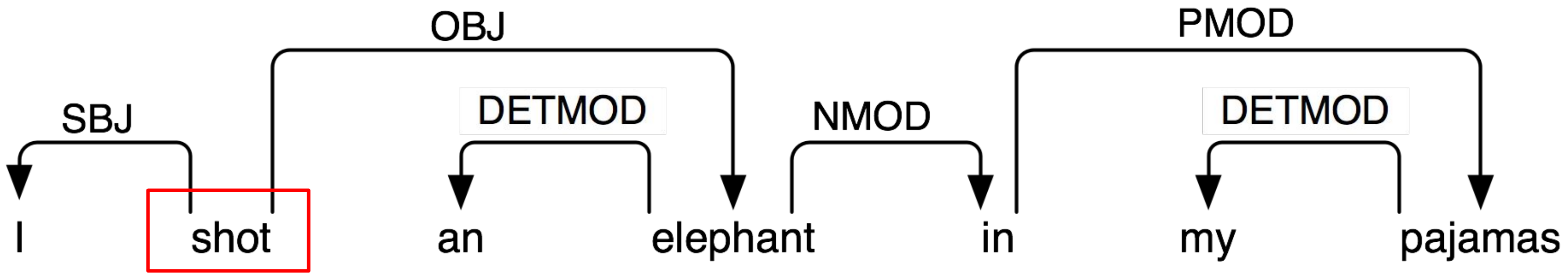
Phrasal constituents and phrase-structure rules do not play a direct role. Instead, the syntactic structure of a sentence is described solely in terms of the words in a sentence.

Advantages of Dependency Parsing

- **Directed relations between lexical items** in the sentence directly encode important information that is buried in the phrase-structure parses
    - E.g., the arguments to the verb *prefer* are directly linked to it in the dependency structure, while their connection to the main verb is more distant in the phrase-structure tree. Similarly, *morning* and *Denver*, modifiers of flight, are linked to it directly in the dependency structure
- Can deal with languages that are morphologically rich and have a relatively **free word order**
    - E.g.,, word order in Czech can be much more flexible than in English; a grammatical object might occur before or after a location adverbial. A phrase-structure grammar would need a separate rule for each possible place in the parse tree where such an adverbial phrase could occur. A dependency-based approach would just have one link type representing this particular adverbial relation.
- The head-dependent relations provide an approximation to the **semantic relationship** between predicates and their arguments that makes them directly useful for many applications such as coreference resolution, question answering and information extraction

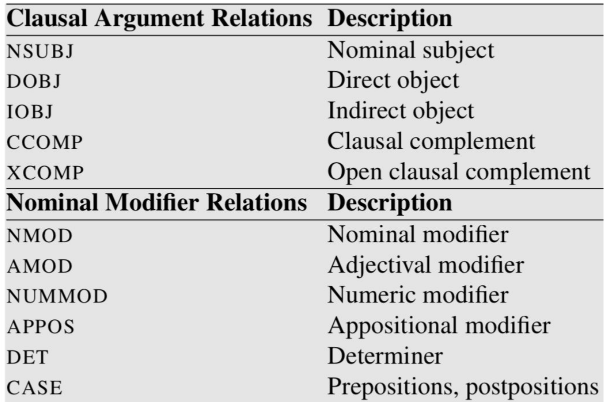
Typed Dependency Structure

- The arcs are **labeled with the grammatical function** that holds between a dependent and its head
    - *I* is the SBJ (subject) of shot (which is the head of the whole sentence)
    - *in* is an NMOD (noun modifier of elephant)

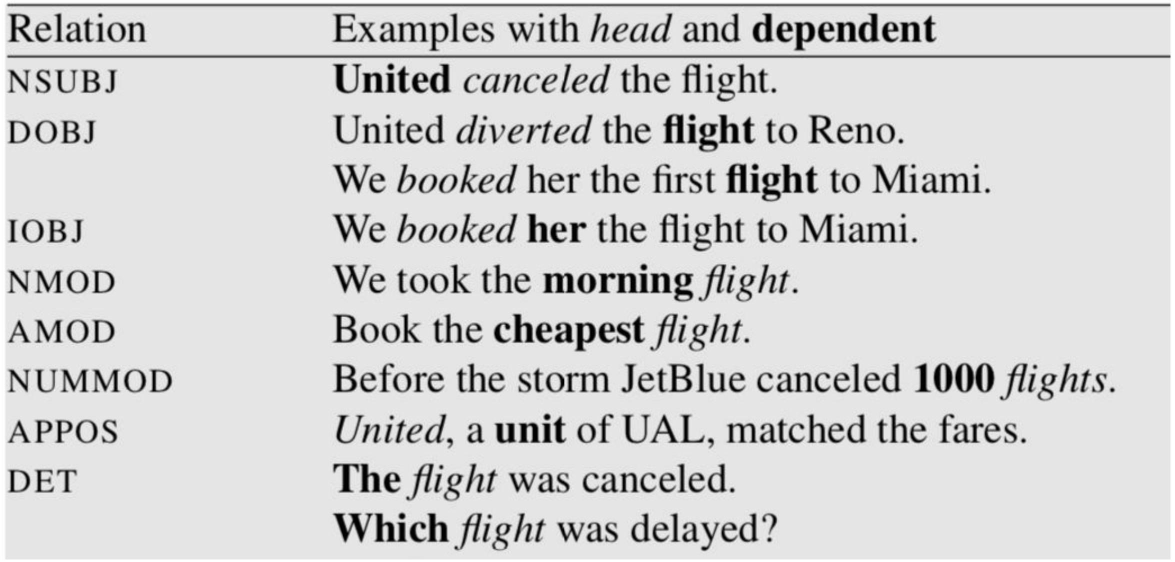


Dependency Relations

- Dependency grammars allow us to further classify the kinds of grammatical relations, or grammatical function



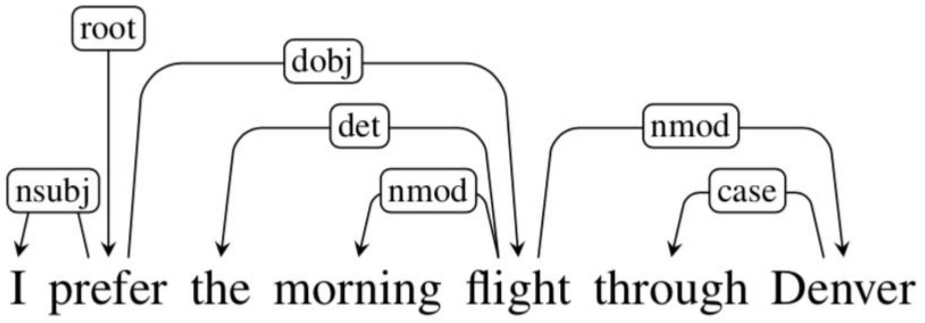
Clausal Argument Relations	**Description**
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions

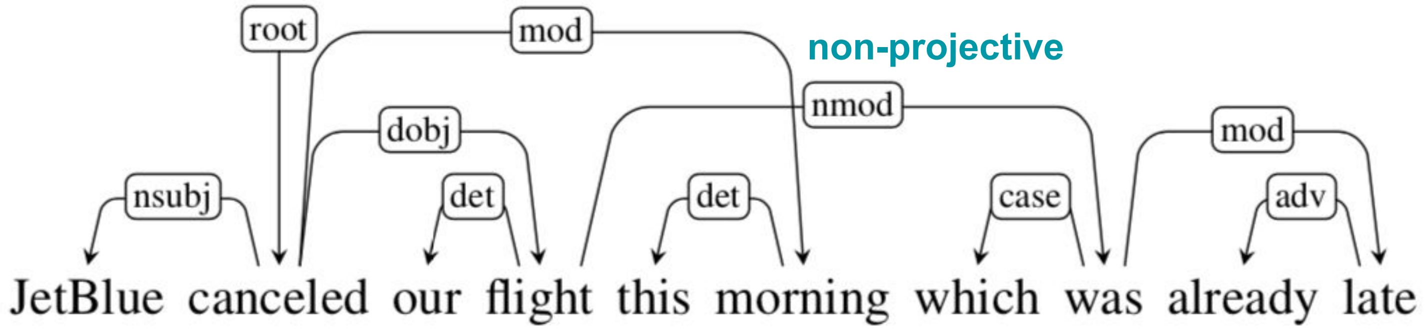


Relation	Examples with *head* and **dependent**
NSUBJ	**United** *canceled* the flight.
DOBJ	United *diverted* the **flight** to Reno.
	We *booked* her the first **flight** to Miami.
IOBJ	We *booked* **her** the flight to Miami.
NMOD	We took the **morning** *flight*.
AMOD	Book the **cheapest** *flight*.
NUMMOD	Before the storm JetBlue canceled **1000** *flights*.
APPOS	*United*, a **unit** of UAL, matched the fares.
DET	**The** *flight* was canceled.
	Which *flight* was delayed?

Projective

- A dependency graph is **projective** if the edges can be drawn above the words without crossing



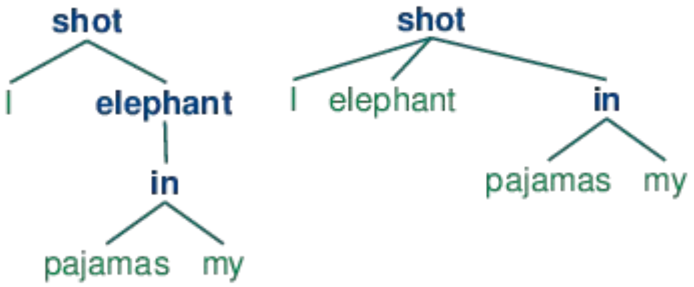

root
mod
nmod
dobj
mod
nsubj
det
det
case
adv
JetBlue canceled our flight this morning which was already late

non-projective

Ambiguity

- groucho_dep_grammar provides an alternative approach to capturing the **attachment ambiguity**

```
'shot' -> 'I' | 'elephant' | 'in'
'elephant' -> 'an' | 'in'
'in' -> 'pajamas'
'pajamas' -> 'my'
```



```
>>> groucho_dep_grammar = nltk.DependencyGrammar.fromstring("""
... 'shot' -> 'I' | 'elephant' | 'in'
... 'elephant' -> 'an' | 'in'
... 'in' -> 'pajamas'
... 'pajamas' -> 'my'
... """)
>>> print(groucho_dep_grammar)
Dependency grammar with 7 productions
  'shot' -> 'I'
  'shot' -> 'elephant'
  'shot' -> 'in'
  'elephant' -> 'an'
  'elephant' -> 'in'
  'in' -> 'pajamas'
  'pajamas' -> 'my'
>>> pdp = nltk.ProjectiveDependencyParser(groucho_dep_grammar)
>>> sent = 'I shot an elephant in my pajamas'.split()
>>> trees = pdp.parse(sent)
>>> for tree in trees:
...     print(tree)
(shot I (elephant an (in (pajamas my))))
(shot I (elephant an) (in (pajamas my)))
```

Heads vs Dependents

- The head word of a constituent is the central organizing word of a larger constituent (e.g, the primary noun in a noun phrase, or verb in a verb phrase)
- Decide **what is the head** H and **what is the dependent** D in a construction C
  - H determines the distribution class of C; or, the external syntactic properties of C are due to H
  - H determines the semantic type of C
  - H is obligatory while D may be optional
  - H selects D and determines whether it is obligatory or optional
  - The morphological form of D is determined by H (e.g. agreement or case government)

Summary

- A **grammar** is a **formal model** for describing whether a given phrase can be assigned a particular constituent or dependency structure
- A **context-free grammar** uses a set of productions to say how a phrase of some category *A* can be analyzed into a sequence of smaller parts α1 ... αn
- A **dependency grammar** uses productions to specify the dependents of a lexical head
- **Syntactic ambiguity** arises when one sentence has more than one syntactic analysis
- A **parser** finds one or more trees corresponding to a grammatically well-formed sentence
- A simple **top-down parser** is the **recursive descent parser**, which recursively expands the start symbol with the help of the grammar productions, and tries to match the input sentence
- A simple bottom-up parser is the **shift-reduce parser**, which shifts input onto a stack and tries to match the items at the top of the stack with the right hand side of grammar productions
- Context-sensitive grammar can be found at https://www.cis.upenn.edu/~jean/gbooks/cis51104sl13pdf.pdf

Text

I shot an elephant in my pajamas

Parser

Recursive descent parser

Shift-reduce parser

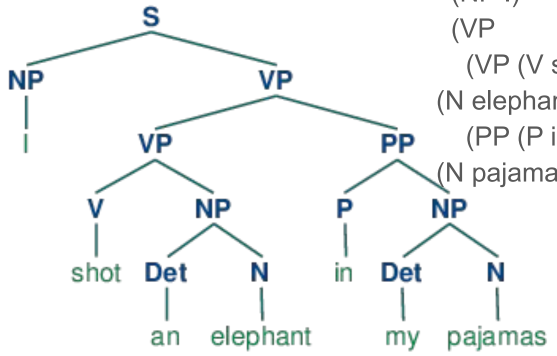
Dependency parser

Grammar

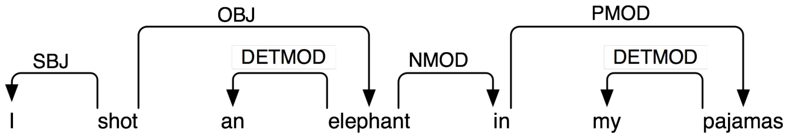
```
S -> NP VP
PP -> P NP
NP -> Det N | Det N PP | 'I'
VP -> V NP | VP PP
Det -> 'an' | 'my'
N -> 'elephant' | 'pajamas'
V -> 'shot'
P -> 'in'
```

```
'shot' -> 'I' | 'elephant' | 'in'
'elephant' -> 'an' | 'in'
'in' -> 'pajamas'
'pajamas' -> 'my'
```

Parsed structure

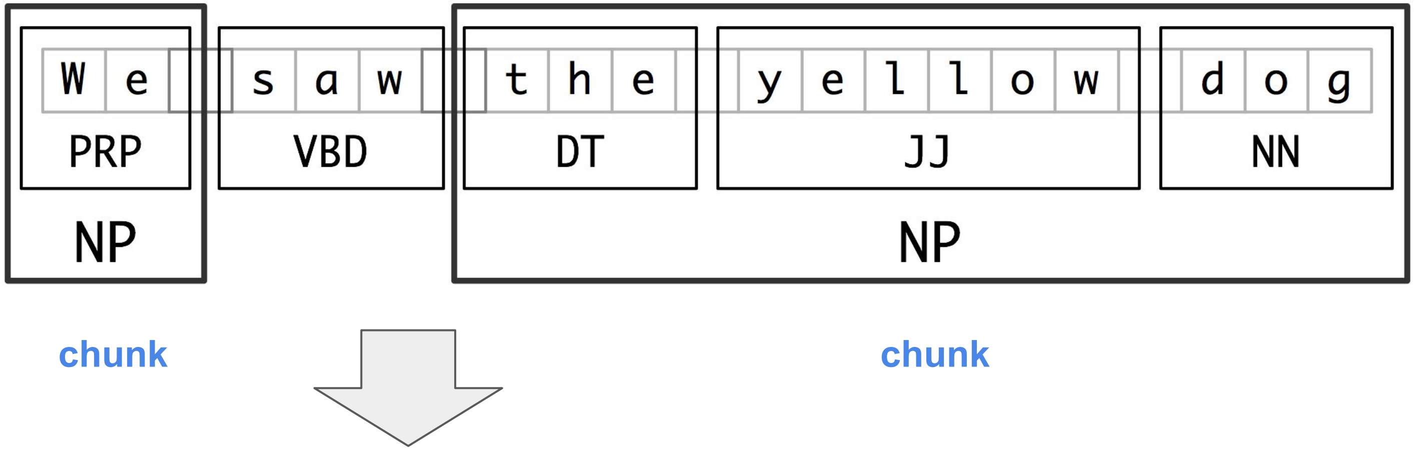


```
(S
  (NP I)
  (VP
    (VP (V shot) (NP (Det an)
(N elephant)))
    (PP (P in) (NP (Det my)
(N pajamas)))))
```



(shot I (elephant an (in (pajamas my))))

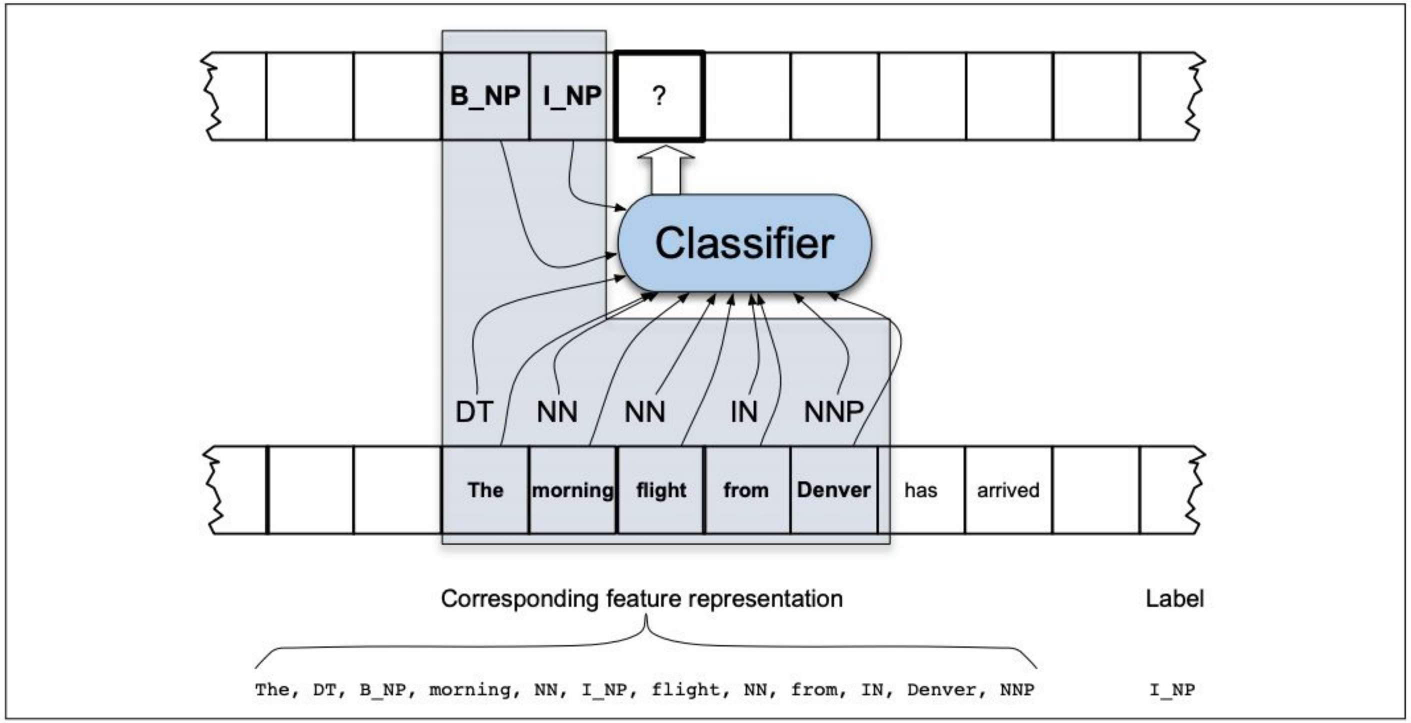
Chunking



We PRP B-NP
saw VBD O
the DT B-NP
yellow JJ I-NP
dog NN I-NP

IOB Tags

Machine Learning-Based Approaches to Chunking


B_NP
I_NP
?
Classifier
DT
NN
NN
IN
NNP
The
morning
flight
from
Denver
has
arrived
Corresponding feature representation
Label
The, DT, B_NP, morning, NN, I_NP, flight, NN, from, IN, Denver, NNP
I_NP

https://web.stanford.edu/~jurafsky/slp3/13.pdf