Introduction to Python and Natural Language Technologies

10. Syntax

Dávid Márk Nemeskey

MTA SZTAKI

November 10, 2017

Introduction

Introduction

The word is again from Ancient Greek: "arrangement" or "coordination"

- σύν (sýn): together
- τάξις (táxis): ordering

Introduction

The word is again from Ancient Greek: “arrangement” or “coordination”

- σύν (sýn): together
- τάξις (táxis): ordering

It is the next step up from morphology:

- *Morphology* is the set of rules that govern the structure of *words*
- **Syntax** is the set of rules that govern the structure of **sentences**

The terms syntax and *(formal) grammar* are usually synonymous.

Grammar vs Semantics

(1) *Colorless green ideas sleep furiously.*

(2) **Furiously sleep ideas green colorless.*

Chomsky, (1956)

Grammar vs Semantics

(1) *Colorless green ideas sleep furiously.*

(2) **Furiously sleep ideas green colorless.*

Chomsky, (1956)

Neither of the two sentences make any sense. However,

1. sentence (1) is **grammatical**;
2. sentence (2) is not.

Grammar vs Semantics

(1) *Colorless green ideas sleep furiously.*

(2) **Furiously sleep ideas green colorless.*

Chomsky, (1956)

Neither of the two sentences make any sense. However,

1. sentence (1) is **grammatical**;
2. sentence (2) is not.

Syntax concerns itself with grammaticality; whether the sentences are meaningful or not is irrelevant.

Grammar vs Semantics

(1)   *Colorless green ideas sleep furiously.*

(2)   **Furiously sleep ideas green colorless.*

Chomsky, (1956)

Neither of the two sentences make any sense. However,

1. sentence (1) is **grammatical**;
2. sentence (2) is not.

Syntax concerns itself with grammaticality; whether the sentences are meaningful or not is irrelevant.

The study of meaning is semantics, which shall be covered in later lectures.

History

1. Antiquity
   - Pāṇini (around 500BC): Aṣṭādhyāyī, a sūtra of 3,959 verses, describing the grammar of Ancient Sanskrit
2. Traditional (high school) grammar
   - The European tradition before the (middle of the) 20th century
   - Focused on grammatical relations (subject, object, etc.)
3. Phrase-structure grammar (PSG)
   - PSG and formal grammars were defined by Noah Chomsky, (1956)
   - It became the predominant linguistic theory in the second half of the 20th century
4. Modern grammars
   - The Chomskyan paradigm (the latest version is *Minimalism*) is still going strong
   - Grammatical relations have made a comeback, e.g. in *Dependency Grammar*
   - A whole landscape of various formalisms

Concepts

We need to define a few concepts to formalize the notion of grammar

- Part of speech
- Grammatical relations
- Subcategorization
- Constituency

Concepts

We need to define a few concepts to formalize the notion of grammar

- Part of speech
- Grammatical relations
- Subcategorization
- Constituency

In the following, we mainly concentrate on

- English syntax
- significant phenomena from other languages

Concepts: Part of speech

A **part of speech (PoS)** is a category of words with similar grammatical distribution. Some of these are:

- Noun: *dogs*, *John*
- Verb: *sleep*, *was*
- Adjective: *blue*, *quicker*
- Adverb: *very*, *quickly*
- Preposition: *for*, *to*
- Determiners: *the*, *that*
- ...

Concepts: Part of speech

A **part of speech (PoS)** is a category of words with similar grammatical distribution. Some of these are (with Penn Treebank POS tags):

- Noun (`NN*`): *dogs* (`NNS`), *John* (`NNP`)
- Verb (`VB*`): *sleep* (`VBZ`), *was* (`VBD`)
- Adjective (`JJ*`): *blue* (`JJ`), *quicker* (`JJR`)
- Adverb (`RB*`): *very*, *quickly*
- Preposition (`IN`): *for*, *to*
- Determiners (`DT`): *the*, *that*
- ...

Concepts: Part of speech

A **part of speech (PoS)** is a category of words with similar grammatical distribution. Some of these are (with Penn Treebank POS tags):

- Noun (`NN*`): *dogs* (`NNS`), *John* (`NNP`)
- Verb (`VB*`): *sleep* (`VBZ`), *was* (`VBD`)
- Adjective (`JJ*`): *blue* (`JJ`), *quicker* (`JJR`)
- Adverb (`RB*`): *very*, *quickly*
- Preposition (`IN`): *for*, *to*
- Determiners (`DT`): *the*, *that*
- ...

These classes can be classified as

- **Open** (the first four): new words can be added, e.g. *laser*, *to google*
- **Closed** (the last two): the membership is fixed

Concepts: grammatical relations

The center of a sentence is the **predicate**.

- In English, this is always a *verb*, and also called the **main verb**
- Other languages, such as Hungarian or Japanese, may have *nominal* or *adjectival predicates*

(1) The sky **is** blue

(2) *Az ég* ***kék***
the sky blue
'The sky is blue'

(3) 空が **青かった**
sky.SUBJ blue.PAST
'The sky was blue'

Concepts: grammatical relations

The predicate may have **arguments**: expressions the complete its meaning. They are usually obligatory, i.e. the sentence would be meaningless without them.

Their type depends on the language, predicate and other factors (voice, etc.).

(1) [$_{\text{SUBJ}}$ John] loves [$_{\text{OBJ}}$ Mary].

(2) Rose, [$_{\text{SUBJ}}$ ich] erinnere [$_{\text{ACC}}$ mich] [$_{\text{DAT}}$ an Ihren Vater] liebevoll.

Concepts: grammatical relations

The arguments that English verbs may take are listed below, with their most common semantic roles:

- Subject: the agent / theme of the predicate
- Object: the entity acted upon by the subject
  1. Direct object: entity acted upon (accusative case)
  2. Indirect object: affected by the action (dative case)
  3. Oblique object: object with a preposition (all other cases)

(1) $[_{\text{SUBJ}}$ John$]$ loves $[_{\text{DOBJ}}$ Mary$]$.

(2) $[_{\text{SUBJ}}$ John$]$ gives $[_{\text{IOBJ}}$ Mary$]$ $[_{\text{DOBJ}}$ a flower$]$.

(3) $[_{\text{SUBJ}}$ John$]$ gave $[_{\text{DOBJ}}$ a flower$]$ $[_{\text{OBL-OBJ}}$ to Mary$]$.

Concepts: grammatical relations

Note that the roles on the last slide are only **approximately** valid, and differ from verb to verb. Also they were listed with the active voice in mind.
Observe how the arguments change in the passive voice, even if the meaning of the sentence remains the same.

(1) $[_{\text{SUBJ}}$ Mary$]$ was given $[_{\text{DOBJ}}$ a flower$]$ by John.

(2) $[_{\text{SUBJ}}$ A flower$]$ was given $[_{\text{OBL-OBJ}}$ to Mary$]$ by John.

Concepts: grammatical relations

A sentence may also have **adjuncts**: phrases that give extra information but are *optional*.

Adjuncts are similar to oblique objects, but can be omitted, while **arguments** generally cannot:

(1) John gave a flower **to Mary** *in front of the café*.

(2) *John gave a flower *in front of the café*. *To whom?*

(3) John gave a flower **to Mary**.

Concepts: subcategorization

Also **valency**: the ability of verbs that specify (limit) the number and types of its arguments.

Concepts: subcategorization

Also **valency**: the ability of verbs that specify (limit) the number and types of its arguments.

One form of subcategorization is **transitivity**. Verbs can be

- **Intransitive**: no object (*John sleeps.*)
- **Transitive**: only direct subject (*John loves Mary.*)
- **Ditransitive**: direct and indirect subject (*John gave Mary a flower.*)

Concepts: subcategorization

Also **valency**: the ability of verbs that specify (limit) the number and types of its arguments.

One form of subcategorization is **transitivity**. Verbs can be

- **Intransitive**: no object (*John sleeps.*)
- **Transitive**: only direct subject (*John loves Mary.*)
- **Ditransitive**: direct and indirect subject (*John gave Mary a flower.*)

Requirements on argument types (semantics has more to say on this):

(1) I wanted a cake.

(2) I bought a cake.

(3) I wanted to go.

(4) *I bought to go.

Concepts: constituency

A **constituent** or **phrase** is a group of words that acts as a unit and fulfills one of the grammatical roles in the sentence.

Examples:

- Noun phrases (NP): *John*, *the poor dog*, *the girl I saw yesterday*
- Prepositional phrases (PP): *from him*, *in front of the café*
- Adjectival phrases (AP): *very quick*

Concepts: constituency

A **constituent** or **phrase** is a group of words that acts as a unit and fulfills one of the grammatical roles in the sentence.

Examples:

- Noun phrases (NP): *John*, *the poor dog*, *the girl I saw yesterday*
- Prepositional phrases (PP): *from him*, *in front of the café*
- Adjectival phrases (AP): *very quick*

Noun phrases, but not their parts, can be substituted for one another:

(1) **The poor dog** bit **John**.

(2) **John** bit **the poor dog**.

(3) *The John bit poor dog.

Concepts: constituency

A **constituent** or **phrase** is a group of words that acts as a unit and fulfills one of the grammatical roles in the sentence.

The substitution test you saw previously is one of the tests designed to decide whether a group of words is a constituent or not. Several such tests exist.

Concepts: constituency

A **constituent** or **phrase** is a group of words that acts as a unit and fulfills one of the grammatical roles in the sentence.

The substitution test you saw previously is one of the tests designed to decide whether a group of words is a constituent or not. Several such tests exist.

An example is the topicalization test: prepositional phases can be *topicalized*, but only whole:

(1) John gave Mary a flower **in front of the café**.

(2) **In front of the café**, John gave Mary a flower.

(3) *In front, John gave Mary a flower of the café.

Phrase-structure Grammar

Phrase-structure Grammar

The most widely known formal grammar for English is the **Phrase-structure Grammar (PSG)**.

The main units it deals with are *phrases* or *constituents*. Traditional grammatical functions are not marked explicitly.

Phrase-structure Grammar

The most widely known formal grammar for English is the **Phrase-structure Grammar (PSG)**.

The main units it deals with are *phrases* or *constituents*. Traditional grammatical functions are not marked explicitly.

It is a **generative grammar**:

- it consists of a set of **production rules**
- these rules can be used to generate a number of *sentences*
- The **language** $\mathcal{L}$ defined by the grammar is the set of sentences it can generate.

PSG Components

1. Production (replacement) rules:

$$\begin{array}{lcl}
\texttt{NP} & \rightarrow & \texttt{ProperNoun} \\
\texttt{NP} & \rightarrow & \texttt{Det Nominal} \\
\texttt{Nominal} & \rightarrow & \texttt{Noun | Nominal Noun}
\end{array}$$

PSG Components

1. Production (replacement) rules:

$$\begin{array}{lcl} \texttt{NP} & \rightarrow & \texttt{ProperNoun} \\ \texttt{NP} & \rightarrow & \texttt{Det Nominal} \\ \texttt{Nominal} & \rightarrow & \texttt{Noun} \mid \texttt{Nominal Noun} \end{array}$$

2. Lexicon:

$$\begin{array}{lcl} \texttt{Det} & \rightarrow & \textit{the} \mid \textit{a} \mid \textit{an} \\ \texttt{ProperNoun} & \rightarrow & \textit{John} \mid \textit{Mary} \mid \ldots \\ \texttt{Noun} & \rightarrow & \textit{dog} \mid \textit{cat} \mid \ldots \end{array}$$

PSG Components

1. Production (replacement) rules:

$$\begin{array}{lcl} \texttt{NP} & \rightarrow & \texttt{ProperNoun} \\ \texttt{NP} & \rightarrow & \texttt{Det Nominal} \\ \texttt{Nominal} & \rightarrow & \texttt{Noun} \mid \texttt{Nominal Noun} \end{array}$$

2. Lexicon:

$$\begin{array}{lcl} \texttt{Det} & \rightarrow & \textit{the} \mid \textit{a} \mid \textit{an} \\ \texttt{ProperNoun} & \rightarrow & \textit{John} \mid \textit{Mary} \mid \ldots \\ \texttt{Noun} & \rightarrow & \textit{dog} \mid \textit{cat} \mid \ldots \end{array}$$

There are two types of symbols:

- **terminals** (*and*, *John*) are part of the generated language
- **nonterminals** (`Nominal`) represent word class or constituents

Derivation

Text generation works as follows:

- We start with a string of a single nonterminal, the **start symbol** (usually `S`)
- At each step, we apply a rule:
  - We pick a nonterminal from the string
  - Choose one of the rules where that nonterminal is on the left hand side
  - Replace it in the string with the right hand side of the rule.
- The process ends when the string consists entirely of terminal symbols.

The sequence of rule applications is called the **derivation** of the final string.

Derivation

Text generation works as follows:

- We start with a string of a single nonterminal, the **start symbol** (usually `S`)
- At each step, we apply a rule:
  - We pick a nonterminal from the string
  - Choose one of the rules where that nonterminal is on the left hand side
  - Replace it in the string with the right hand side of the rule.
- The process ends when the string consists entirely of terminal symbols.

The sequence of rule applications is called the **derivation** of the final string.

Example: `NP`

Derivation

Text generation works as follows:

- We start with a string of a single nonterminal, the **start symbol** (usually `S`)
- At each step, we apply a rule:
  - We pick a nonterminal from the string
  - Choose one of the rules where that nonterminal is on the left hand side
  - Replace it in the string with the right hand side of the rule.
- The process ends when the string consists entirely of terminal symbols.

The sequence of rule applications is called the **derivation** of the final string.

Example: `NP` $\rightarrow$ `Det Nominal`

Derivation

Text generation works as follows:

- We start with a string of a single nonterminal, the **start symbol** (usually `S`)
- At each step, we apply a rule:
  - We pick a nonterminal from the string
  - Choose one of the rules where that nonterminal is on the left hand side
  - Replace it in the string with the right hand side of the rule.
- The process ends when the string consists entirely of terminal symbols.

The sequence of rule applications is called the **derivation** of the final string.

Example: `NP` $\rightarrow$ `Det Nominal` $\rightarrow$ *a* `Nominal`

Derivation

Text generation works as follows:

- We start with a string of a single nonterminal, the **start symbol** (usually `S`)
- At each step, we apply a rule:
  - We pick a nonterminal from the string
  - Choose one of the rules where that nonterminal is on the left hand side
  - Replace it in the string with the right hand side of the rule.
- The process ends when the string consists entirely of terminal symbols.

The sequence of rule applications is called the **derivation** of the final string.

Example: `NP` → `Det Nominal` → *a* `Nominal` → *a* `Noun`

Derivation

Text generation works as follows:

- We start with a string of a single nonterminal, the **start symbol** (usually `S`)
- At each step, we apply a rule:
  - We pick a nonterminal from the string
  - Choose one of the rules where that nonterminal is on the left hand side
  - Replace it in the string with the right hand side of the rule.
- The process ends when the string consists entirely of terminal symbols.

The sequence of rule applications is called the **derivation** of the final string.

Example: `NP` $\rightarrow$ `Det Nominal` $\rightarrow$ *a* `Nominal` $\rightarrow$ *a* `Noun` $\rightarrow$ *a dog*

The parse tree

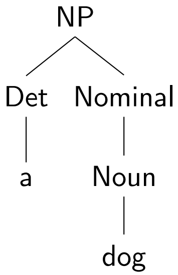
A **Parse tree** or **syntax tree** provides a graphical view on the derivation process:

- An inverted tree
- Inner nodes are nonterminal symbols
- The leaves are terminals
- Edges signify rule applications

The parse tree

A **Parse tree** or **syntax tree** provides a graphical view on the derivation process:

- An inverted tree
- Inner nodes are nonterminal symbols
- The leaves are terminals
- Edges signify rule applications



Example

This is our example grammar:

`S`	$\rightarrow$	`NP VP`	*Subject-predicate*
`NP`	$\rightarrow$	`Pronoun | ProperNoun`	*One way to list alternatives*
	|	`Det Nominal`	*Another way*
`Nominal`	$\rightarrow$	`Nominal Noun`	
	|	`Noun`	
`VP`	$\rightarrow$	`Verb`	*Intransitive verb*
	|	`Verb NP`	*Transitive verb*
	|	`Verb PP`	*Oblique / adjunct*
	|	`Verb NP PP`	*Oblique / adjunct*
`PP`	$\rightarrow$	`Preposition NP`	

Example

Rule to apply:

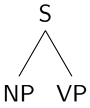
String: S

S

Example

Rule to apply: `S` $\rightarrow$ `NP VP`

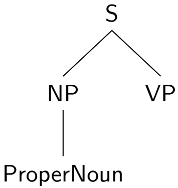
String: `NP VP`



Example

Rule to apply: `NP` $\rightarrow$ `ProperNoun`

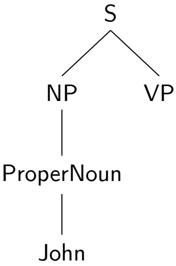
String: `ProperNoun VP`



Example

Rule to apply: `ProperNoun` $\rightarrow$ *John*

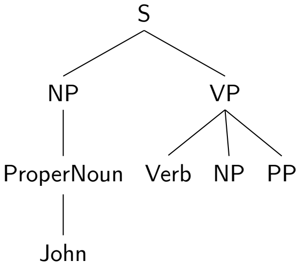
String: *John* `VP`



Example

Rule to apply: `VP` $\rightarrow$ `Verb NP PP`

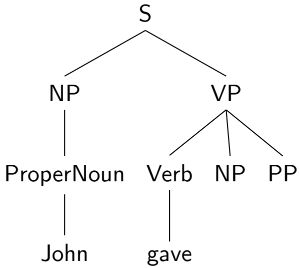
String: *John* `Verb NP PP`



Example

Rule to apply: `Verb` $\rightarrow$ *gave*

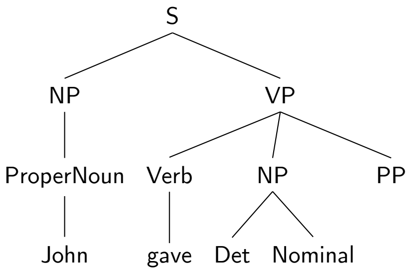
String: *John gave* `NP PP`



Example

Rule to apply: `NP` $\rightarrow$ `Det Nominal`

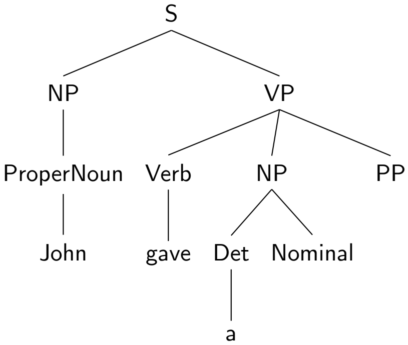
String: *John gave* `Det Nominal PP`



Example

Rule to apply: `Det` $\rightarrow$ *a*

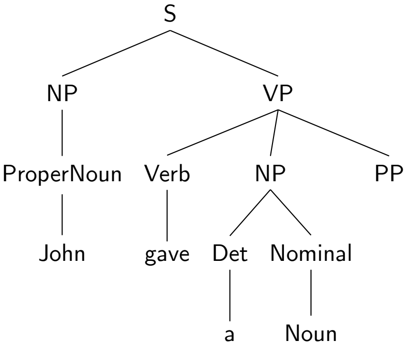
String: *John gave a* `Nominal PP`



Example

Rule to apply: `Nominal` $\rightarrow$ `Noun`

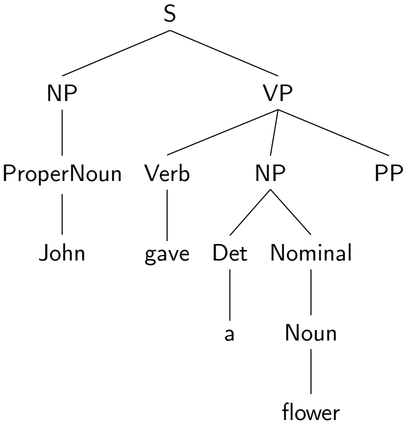
String: *John gave a* `Noun PP`



Example

Rule to apply: `Noun` $\rightarrow$ *flower*

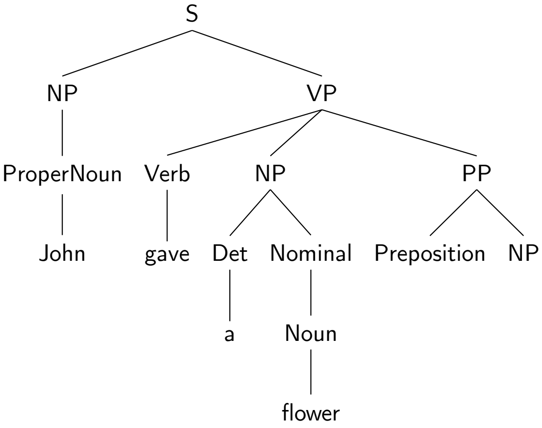
String: *John gave a flower* `PP`



Example

Rule to apply: `PP` → `Preposition NP`

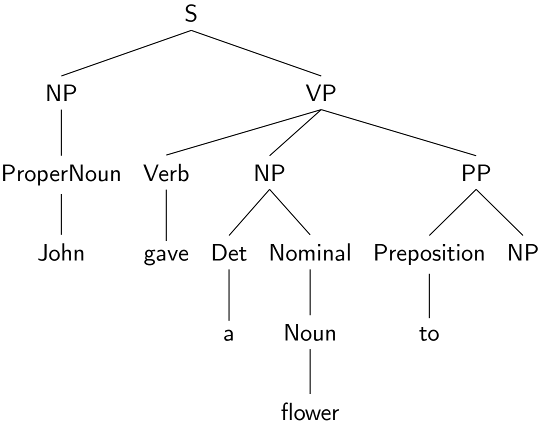
String: *John gave a flower* `Preposition NP`



Example

Rule to apply: `Preposition` $\rightarrow$ *to*

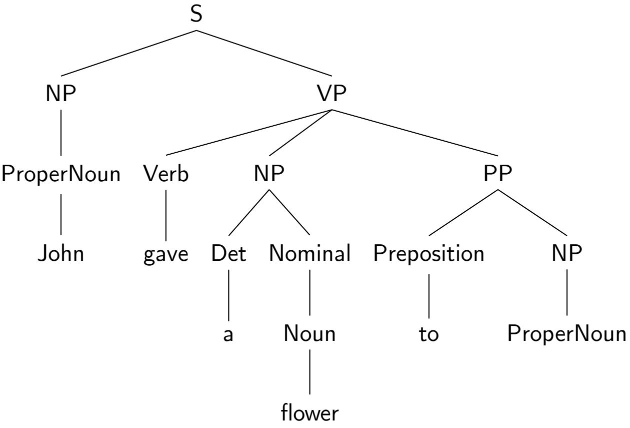
String: *John gave a flower to* `NP`



Example

Rule to apply: `NP` $\rightarrow$ `ProperNoun`

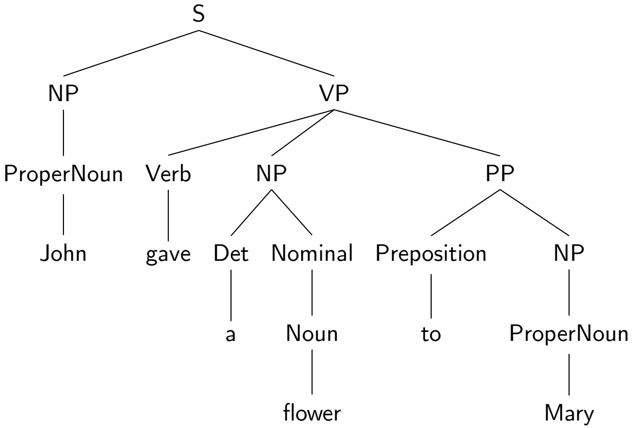
String: *John gave a flower to* `ProperNoun`



Example

Rule to apply: `ProperNoun` $\rightarrow$ *Mary*

String: *John gave a flower to Mary*



Bracketed notation

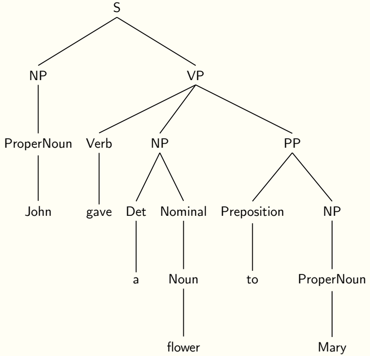
The **bracketed notation** is a way to represent the parse tree in text. This format is used by software (such as the online version of the Stanford Parser) to process trees.

Bracketed notation

The **bracketed notation** is a way to represent the parse tree in text. This format is used by software (such as the online version of the Stanford Parser) to process trees.

Example:

```
(S
  (NP
    (ProperNoun John))
  (VP
    (Verb gave)
    (NP
      (Det a)
      (Nominal
        (Noun flower)))
    (PP
      (TO to)
      (NP
        (ProperNoun Mary)))))
```



The Chomsky Normal Form

A grammar is in the **Chomsky Normal Form (CNF)**, if all its rules take one of two forms:

$$\begin{aligned} \mathtt{A} &\rightarrow \mathtt{B\ C} \\ \mathtt{A} &\rightarrow \mathtt{a} \end{aligned}$$

The Chomsky Normal Form

A grammar is in the **Chomsky Normal Form (CNF)**, if all its rules take one of two forms:

$$\begin{array}{lcl} \mathtt{A} & \rightarrow & \mathtt{B\ C} \\ \mathtt{A} & \rightarrow & \mathtt{a} \end{array}$$

CNF has two interesting properties:

- CNF grammars produce binary parsing trees
- Every PSG grammar can be converted into CNF

The Chomsky Normal Form

A grammar is in the **Chomsky Normal Form (CNF)**, if all its rules take one of two forms:

$$\begin{aligned} \texttt{A} &\rightarrow \texttt{B C} \\ \texttt{A} &\rightarrow \texttt{a} \end{aligned}$$

CNF has two interesting properties:

- CNF grammars produce binary parsing trees
- Every PSG grammar can be converted into CNF

Some algorithms only work on CNF grammars.

Parsing

What can we do with grammars?

- Obtain a grammar
  - The lexicon can be trivially derived from dictionaries or morphological lexicons
  - Production rules are usually written by linguists (but see next section...)

What can we do with grammars?

- Obtain a grammar
  - The lexicon can be trivially derived from dictionaries or morphological lexicons
  - Production rules are usually written by linguists (but see next section...)
- Parse a text
  - Given a text, return the syntax tree (the "**parse**")
  - Enumerating all possible parses is not difficult
  - Picking the *most correct* parse is

What can we do with grammars?

- Obtain a grammar
  - The lexicon can be trivially derived from dictionaries or morphological lexicons
  - Production rules are usually written by linguists (but see next section...)
- Parse a text
  - Given a text, return the syntax tree (the "**parse**")
  - Enumerating all possible parses is not difficult
  - Picking the *most correct* parse is
- Generate a text
  - The "inverse" of parsing (see previous example)
  - We are not going into details in this course

Parsing

Input: The text sentence

Output: A parse tree

Parsing

Input: The text sentence

Output: A parse tree

There are two directions we can build the tree from:

1. Top-down
    - We start from the root nonterminal, `S`
    - Apply rules until we reach the terminals (as if we were generating)
    - Parsing finishes when the whole sentence is generated

Parsing

Input: The text sentence

Output: A parse tree

There are two directions we can build the tree from:

1. Top-down
    - We start from the root nonterminal, `S`
    - Apply rules until we reach the terminals (as if we were generating)
    - Parsing finishes when the whole sentence is generated
2. Bottom-up
    - Start from the sentence
    - Apply rules backwards: if we have a sub-graph that matches the right-hand side of a rule, add the left-hand side node, and connect it with the right-hand side
    - Parsing finishes if we reach `S` and have a single tree

Challenges

There are two main challenges that parsing algorithms must face:

- **Nondeterminism** of the grammar
- **Ambiguity** of the natural language

Challenge #1: Nondeterminism

Natural language grammars are nondeterministic:

- Each node could be expanded in several ways
- Examples:
  - `VP` → `Verb | Verb NP | Verb PP | Verb NP PP`
  - `Verb` → `give | take | see | ...`
- There is only one (or at most few) correct choice for a particular sentence

Challenge #1: Nondeterminism

Natural language grammars are nondeterministic:

- Each node could be expanded in several ways
- Examples:
  - `VP → Verb | Verb NP | Verb PP | Verb NP PP`
  - `Verb → give | take | see | ...`
- There is only one (or at most few) correct choice for a particular sentence

Building the tree can be rephrased into a graph search problem:

- Each node in the search graph represents a partially completed parse tree
- Nodes are visited according to a **search strategy**: BFS, DFS, A^*, ...
- When the node is not compatible with the sentence, the algorithm **backtracks**

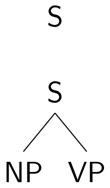
Top-down parse example

Let's parse "*John loves Mary.*" with our toy grammar.

S

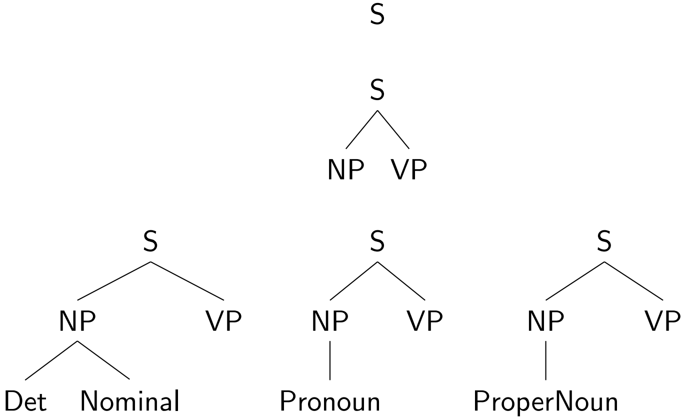
Top-down parse example

Let's parse "*John loves Mary.*" with our toy grammar.



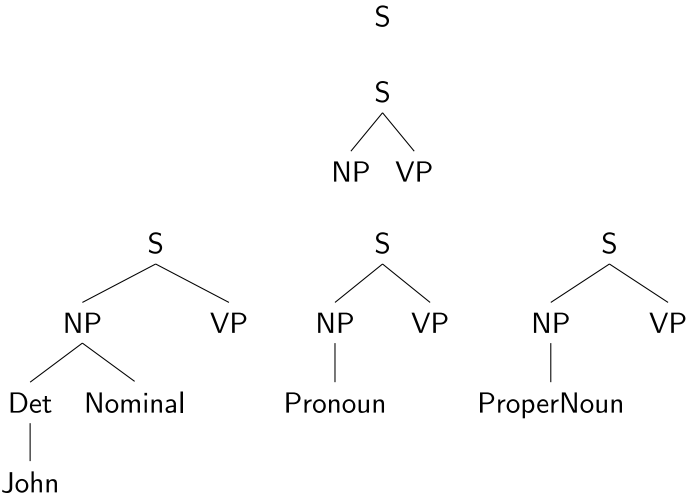
Top-down parse example

Let's parse "*John loves Mary.*" with our toy grammar.



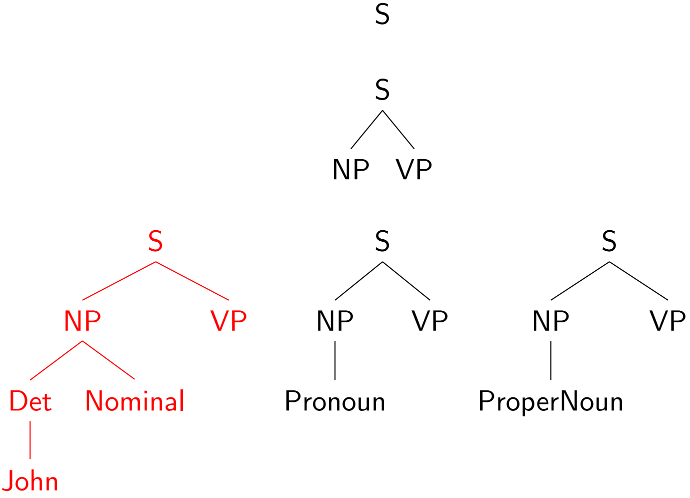
Top-down parse example

Let's parse "*John loves Mary.*" with our toy grammar.



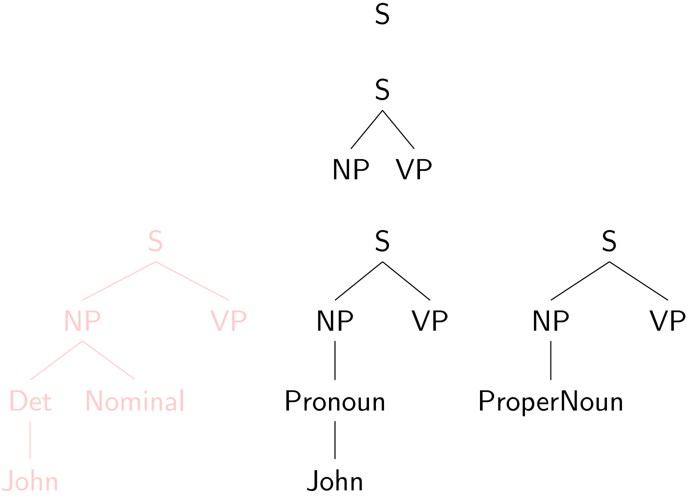
Top-down parse example

Let's parse "*John loves Mary.*" with our toy grammar.



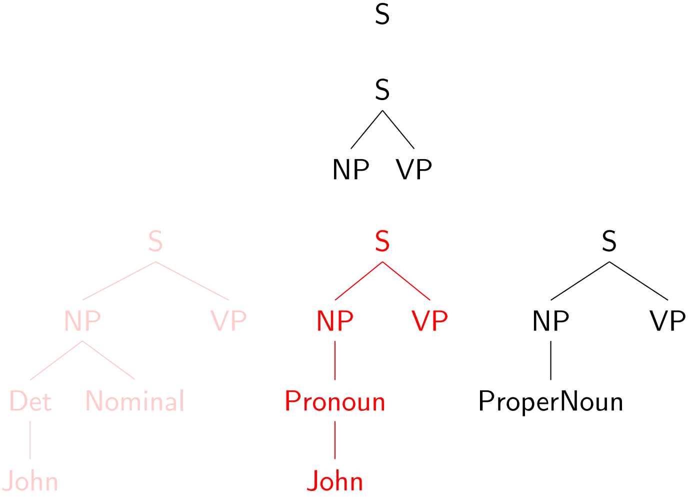
Top-down parse example

Let's parse "*John loves Mary.*" with our toy grammar.



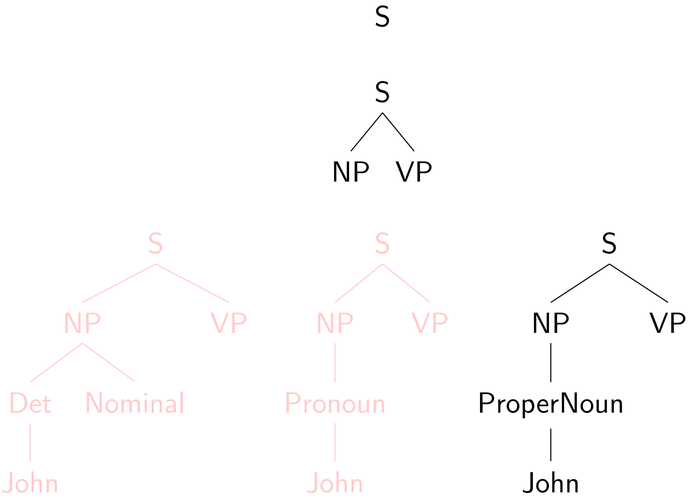
Top-down parse example

Let's parse "*John loves Mary.*" with our toy grammar.



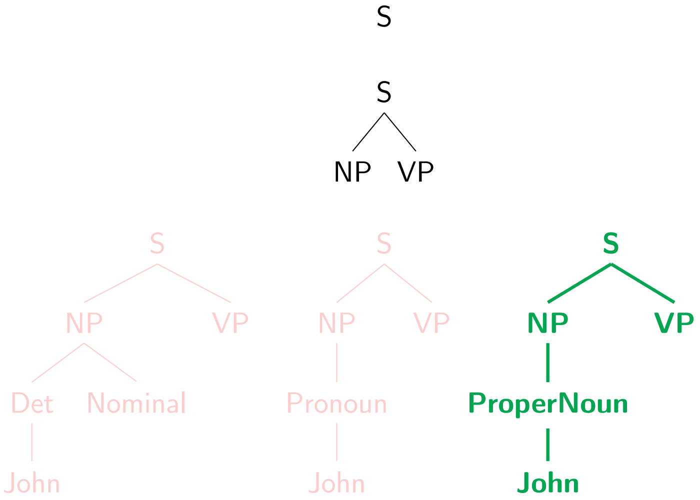
Top-down parse example

Let's parse "*John loves Mary.*" with our toy grammar.



Top-down parse example

Let's parse "*John loves Mary.*" with our toy grammar.



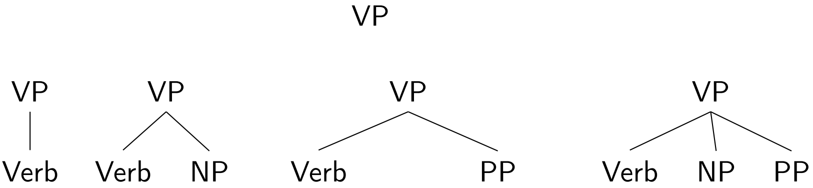
Top-down parse example – cont.

Let's parse "*... loves Mary.*"

VP

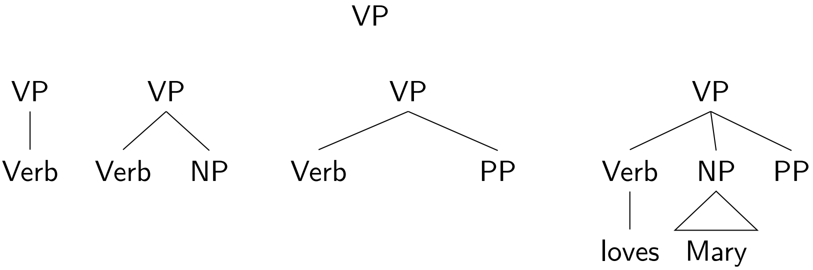
Top-down parse example – cont.

Let's parse "... *loves Mary.*"



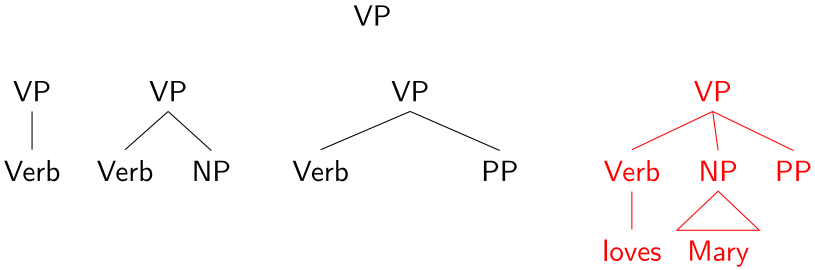
Top-down parse example – cont.

Let's parse "*... loves Mary.*"



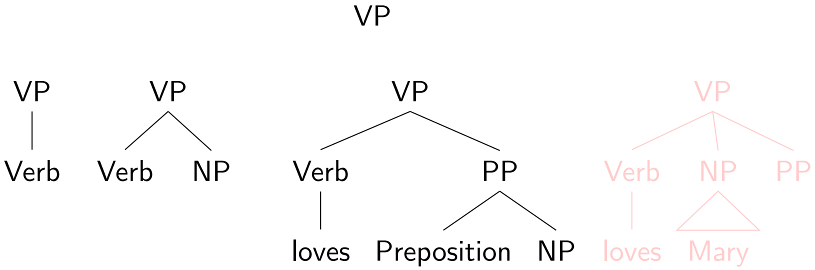
Top-down parse example – cont.

Let's parse "... *loves Mary.*"



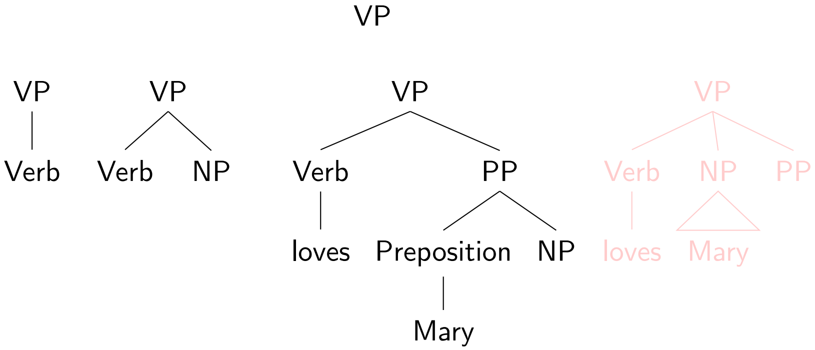
Top-down parse example – cont.

Let's parse "*... loves Mary.*"



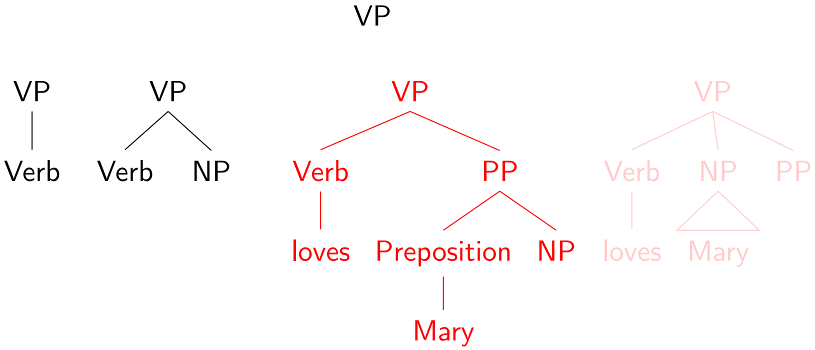
Top-down parse example – cont.

Let's parse "*... loves Mary.*"



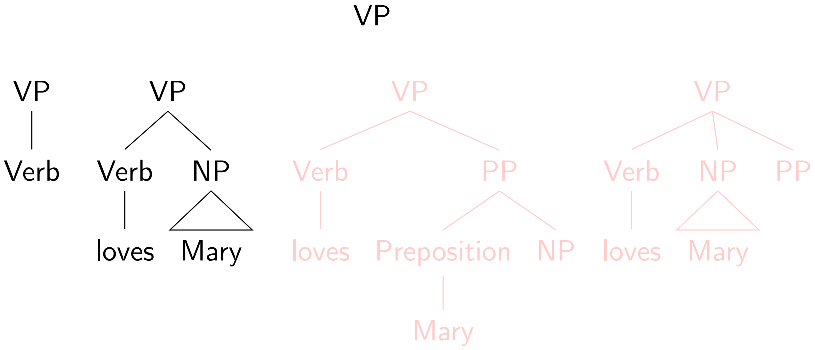
Top-down parse example – cont.

Let's parse "... *loves Mary.*"



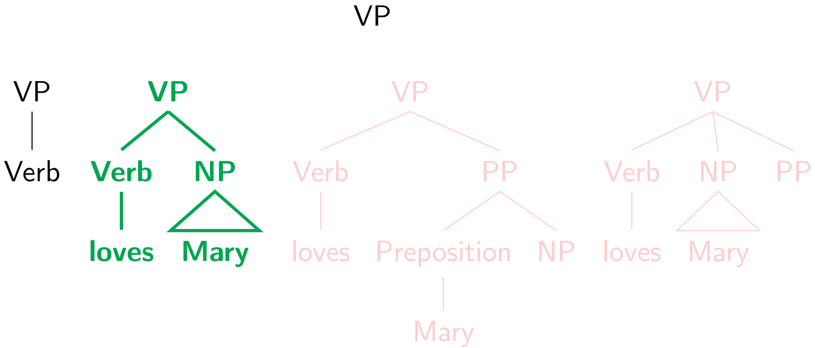
Top-down parse example – cont.

Let's parse "*... loves Mary.*"



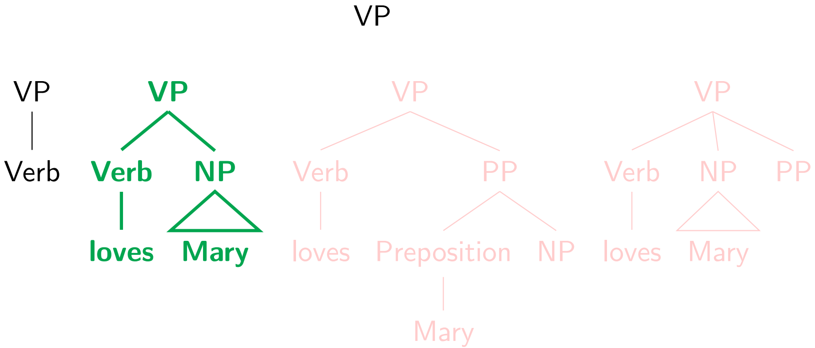
Top-down parse example – cont.

Let's parse "*... loves Mary.*"



Top-down parse example – cont.

Let's parse "*... loves Mary.*"



Even parsing a simple sentence with a toy grammar,

- top-down parsing backtracks a lot (bottom-up would, too)
- the same sub-trees are built and thrown away multiple times (`[Mary]`$_{\texttt{NP}}$)

Challenge #2: Ambiguity

A sentence is **ambiguous**, if it has more than one possible (correct) parse.

Challenge #2: Ambiguity

A sentence is **ambiguous**, if it has more than one possible (correct) parse.

One morning I shot an elephant in my pajamas.

Challenge #2: Ambiguity

A sentence is **ambiguous**, if it has more than one possible (correct) parse.

> *One morning I shot an elephant in my pajamas. How he got into my pajamas I don't know.*
> *Groucho Marx, Animal Crackers, 1930*

Challenge #2: Ambiguity

A sentence is **ambiguous**, if it has more than one possible (correct) parse.

> *One morning I shot an elephant in my pajamas. How he got into my pajamas I don't know.*
> *Groucho Marx, Animal Crackers, 1930*

- The PP "*in my pajamas*" can attach to the VP and the NP
- Disambiguation requires semantic and pragmatic knowledge – not available at this point in the usual linguistic pipeline

Challenge #2: Ambiguity

A sentence is **ambiguous**, if it has more than one possible (correct) parse.

> *One morning I **shot** an elephant **in my pajamas**. How he got into my pajamas I don't know.*
> *Groucho Marx, Animal Crackers, 1930*

- The PP "*in my pajamas*" can attach to the **VP** and the NP
- Disambiguation requires semantic and pragmatic knowledge – not available at this point in the usual linguistic pipeline

Challenge #2: Ambiguity

A sentence is **ambiguous**, if it has more than one possible (correct) parse.

> *One morning I shot an* ***elephant in my pajamas****. How he got into my pajamas I don't know.*
> *Groucho Marx, Animal Crackers, 1930*

- The PP "*in my pajamas*" can attach to the VP and the **NP**
- Disambiguation requires semantic and pragmatic knowledge – not available at this point in the usual linguistic pipeline

Challenge #2: Ambiguity

A sentence is **ambiguous**, if it has more than one possible (correct) parse.

> *One morning I shot an elephant in my pajamas. How he got into my pajamas I don't know.*
> *Groucho Marx, Animal Crackers, 1930*

- The PP "*in my pajamas*" can attach to the VP and the NP
- Disambiguation requires semantic and pragmatic knowledge – not available at this point in the usual linguistic pipeline

Challenge #2: Ambiguity

A sentence is **ambiguous**, if it has more than one possible (correct) parse.

> *One morning I shot an elephant in my pajamas. How he got into my pajamas I don't know.*
> *Groucho Marx, Animal Crackers, 1930*

- The PP "*in my pajamas*" can attach to the VP and the NP
- Disambiguation requires semantic and pragmatic knowledge – not available at this point in the usual linguistic pipeline

> *John saw the man on the mountain with a telescope.*

- This sentence is ambiguous even semantically

Consequences of ambiguity

There are two forms of ambiguity:

- **Global ambiguity**: the whole sentence is ambiguous; it has multiple parses.
- **Local ambiguity**: a constituent is ambiguous in itself, but not when part of the sentence. This induces backtracking in the parser.

Consequences of ambiguity

There are two forms of ambiguity:

- **Global ambiguity**: the whole sentence is ambiguous; it has multiple parses.
- **Local ambiguity**: a constituent is ambiguous in itself, but not when part of the sentence. This induces backtracking in the parser.

Some types of ambiguity:

- Attachment ambiguity: it is not clear where the constituent should attach. *I shot an elephant in my pajamas.*
- Coordination ambiguity: it is ambiguous what phrases a conjunction connects. *I saw old men and women*.
- Homonymy: words with different meanings or POS tag can take the same form. E.g. *loves* can be both a noun and a verb.

Dynamic programming

We need an algorithm that

- Takes a sentence and returns "the correct" parse tree(s)
- Does not backtrack much
- Does not duplicate work by repeatedly building the same subtree

Dynamic programming

We need an algorithm that

- Takes a sentence and returns "the correct" parse tree(s)
- Does not backtrack much
- Does not duplicate work by repeatedly building the same subtree

Dynamic programming is a group of algorithms that

- breaks the problem into smaller subproblems
- solves each subproblem only once
- stores the solutions to (re)use it later

Dynamic programming

We need an algorithm that

- Takes a sentence and returns "the correct" parse tree(s)
- Does not backtrack much
- Does not duplicate work by repeatedly building the same subtree

Dynamic programming is a group of algorithms that

- breaks the problem into smaller subproblems
- solves each subproblem only once
- stores the solutions to (re)use it later

In our case, these subproblems are the subtrees of the whole syntax tree.

The CKY algorithm

The **CKY** (short for **Cocke-Kasami-Younger**) algorithm is a bottom-up parsing algorithm:

- dynamic programming
- requires that it the grammar is in CNF
- runs in $\mathcal{O}(n^3)$ time.

The CKY algorithm

The **CKY** (short for **Cocke-Kasami-Younger**) algorithm is a bottom-up parsing algorithm:

- dynamic programming
- requires that it the grammar is in CNF
- runs in $\mathcal{O}(n^3)$ time.

In a nutshell:

- For a sentence of n words, CKY fills the upper triangle of an n^2 matrix
- Each cell corresponds to a number of consecutive words
- The cells record all possible constituents that cover those words
- A parse is successful, if the top right cell contains an `S`

Example

In this example, we are going to parse the sentence *The dog bit John*.

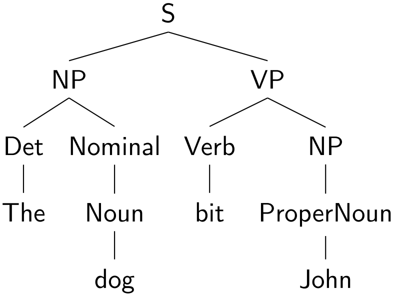
1. convert the grammar to CNF
2. run the CKY algorithm.

Example

In this example, we are going to parse the sentence *The dog bit John*.

1. convert the grammar to CNF
2. run the CKY algorithm.

The parse we are looking for is:



Example: conversion to CNF

Reminder: the CNF contains two types of rules:

$$
\begin{aligned}
\mathtt{A} &\rightarrow \mathtt{B\ C} \\
\mathtt{A} &\rightarrow \mathtt{a}
\end{aligned}
$$

Example: conversion to CNF

Reminder: the CNF contains two types of rules:

$$
\begin{aligned}
\mathtt{A} &\rightarrow \mathtt{B\ C} \\
\mathtt{A} &\rightarrow \mathtt{a}
\end{aligned}
$$

In Phrase Structure Grammars, lexical rules already conform to the second type.

Production rules that do not conform to the first type belong to two groups:

1. Right-hand side too long: $\mathtt{A} \rightarrow \mathtt{B\ C\ D}$
2. Unit productions: $\mathtt{A} \rightarrow \mathtt{B}$

Example: conversion to CNF

Rules whose right-hand side is too long:

$$\texttt{VP} \quad \rightarrow \quad \texttt{Verb NP PP}$$

Example: conversion to CNF

Rules whose right-hand side is too long:

$$\texttt{VP} \quad \rightarrow \quad \texttt{Verb NP PP}$$

are split into two by introducing a new nonterminal:

$$\begin{aligned} \texttt{VP} \quad &\rightarrow \quad \texttt{Verb VP'} \\ \texttt{VP'} \quad &\rightarrow \quad \texttt{NP PP} \end{aligned}$$

This step can be applied recursively, if needed.

Example: conversion to CNF

Unit productions:

`S`	$\rightarrow$	`NP VP`	
`NP`	$\rightarrow$	`ProperNoun`	**Unit production**

Example: conversion to CNF

Unit productions:

`S`	$\rightarrow$	`NP VP`	
`NP`	$\rightarrow$	`ProperNoun`	**Unit production**

Unit production rules are deleted, their right-hand side is "pulled up" to all rules that contain the left-hand nonterminal on their right.

`S`	$\rightarrow$	`NP VP`
`S`	$\rightarrow$	`ProperNoun VP`

Example: conversion of our toy grammar

`S`	$\rightarrow$	`NP VP`	*Subject-predicate*
`NP`	$\rightarrow$	`Pronoun | ProperNoun`	
	|	`Det Nominal`	
`Nominal`	$\rightarrow$	`Nominal Noun`	
	|	`Noun`	
`VP`	$\rightarrow$	`Verb`	*Intransitive verb*
	|	`Verb NP`	*Transitive verb*
	|	`Verb PP`	*Oblique / adjunct*
	|	`Verb NP PP`	*Oblique / adjunct*
`PP`	$\rightarrow$	`Preposition NP`	

Example: conversion of our toy grammar

`S`	$\rightarrow$	`NP VP`	*Subject-predicate*
`NP`	$\rightarrow$	`ProperNoun`	
	|	`Det Nominal`	
`Nominal`	$\rightarrow$	`Nominal Noun`	
	|	`Noun`	
`VP`	$\rightarrow$	`Verb`	*Intransitive verb*
	|	`Verb NP`	*Transitive verb*

The subset of rules needed for the example (*The dog bit John*)

Example: conversion of our toy grammar

`S`	$\rightarrow$	`NP VP`	
`NP`	$\rightarrow$	`ProperNoun`	**Unit production**
	|	`Det Nominal`	
`Nominal`	$\rightarrow$	`Nominal Noun`	
	|	`Noun`	**Unit production**
`VP`	$\rightarrow$	`Verb`	**Unit production**
	|	`Verb NP`	

Let's get rid of the rest of unit productions, `NP` $\rightarrow$ `ProperNoun` first.

Example: conversion of our toy grammar

S	→	NP VP	
	|	ProperNoun VP	
NP	→	Det Nominal	
Nominal	→	Nominal Noun	
	|	Noun	**Unit production**
VP	→	Verb	**Unit production**
	|	Verb NP	
	|	Verb ProperNoun	

... then Nominal → Noun ...

Example: conversion of our toy grammar

`S`	$\rightarrow$	`NP VP`
	|	`ProperNoun VP`
`NP`	$\rightarrow$	`Det Nominal`
	|	`Det Noun`
`Nominal`	$\rightarrow$	`Nominal Noun`
	|	`Noun Noun`
`VP`	$\rightarrow$	`Verb`
	|	`Verb NP`
	|	`Verb ProperNoun`

Unit production

… and finally `VP` $\rightarrow$ `Verb`.

Example: conversion of our toy grammar

```
S        →  NP VP
         |  ProperNoun VP
         |  NP Verb
         |  ProperNoun Verb

NP       →  Det Nominal
         |  Det Noun

Nominal  →  Nominal Noun
         |  Noun Noun

VP       →  Verb NP
         |  Verb ProperNoun
```

The CNF form of our grammar.

Example: conversion of our toy grammar

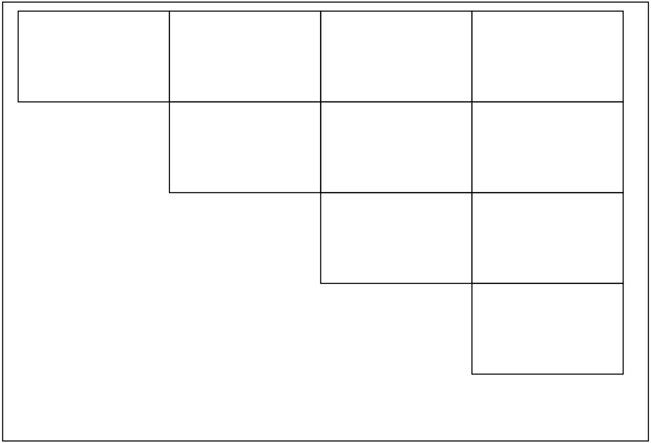
$$
\begin{array}{lcl}
\texttt{S} & \rightarrow & \texttt{NP VP} \\
 & | & \texttt{ProperNoun VP} \\
 & | & \texttt{NP Verb} \\
 & | & \texttt{ProperNoun Verb} \\
 & & \\
\texttt{NP} & \rightarrow & \texttt{Det Nominal} \\
 & | & \texttt{Det Noun} \\
 & & \\
\texttt{Nominal} & \rightarrow & \texttt{Nominal Noun} \\
 & | & \texttt{Noun Noun} \\
 & & \\
\texttt{VP} & \rightarrow & \texttt{Verb NP} \\
 & | & \texttt{Verb ProperNoun}
\end{array}
$$

The CNF form of our grammar. Note how we started out with 7 rules and ended up with 10: conversion to CNF can lead to a proliferation of rules.

Example: CKY – The dog bit John.

Now we can apply the CKY algorithm...

Example: CKY – The dog bit John.



First, we start by drawing the upper triangular matrix

Example: CKY – The dog bit John.

the			
	dog		
		bit	
			John

Each column is assigned to a word in the sentence

Example: CKY – The dog bit John.

[1]	[1–2]	[1–3]	[1–4]
the	[2]	[2–3]	[2–4]
	dog	[3]	[3–4]
		bit	[4]
			John

We note the word indices each cell corresponds to

Example: CKY – The dog bit John.

Det [1]	[1–2]	[1–3]	[1–4]
the	**Noun** [2]	[2–3]	[2–4]
	dog	**Noun** **Verb** [3]	[3–4]
		bit	**ProperNoun** [4]
			John

The diagonal records the terminal production rules (POS). Ambiguity: *bit* can also be a noun

Example: CKY – The dog bit John.

Det [1]	[1–2]	[1–3]	[1–4]
the	Noun [2]	[2–3]	[2–4]
	dog	Noun Verb [3]	[3–4]
		bit	ProperNoun [4]
			John

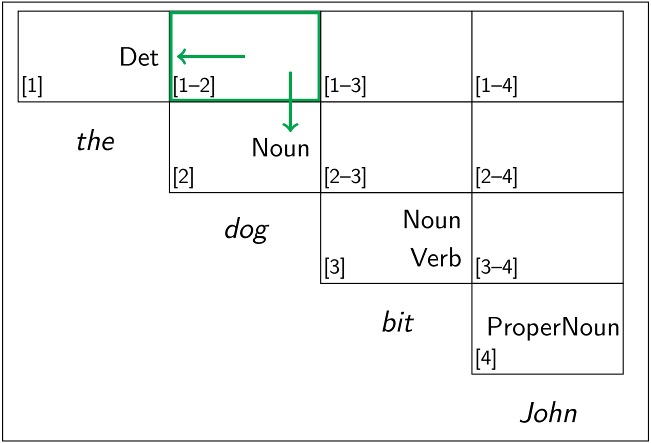
There is not much else to do in the first column

Example: CKY – The dog bit John.

Det [1]	[1–2]	[1–3]	[1–4]
the	Noun [2]	[2–3]	[2–4]
	dog	Noun Verb [3]	[3–4]
		bit	ProperNoun [4]
			John

On to the second column...

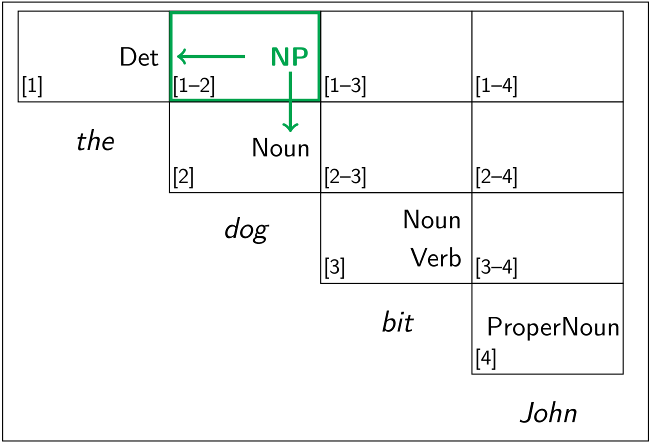
Example: CKY – The dog bit John.



Non-leaf cells correspond to binary rules:

- the first constituent on the right side of the rule is to the *left*
- the second one is *down*

Example: CKY – The dog bit John.



The first rule application: `NP` → `Det Noun`

Det [1]	NP [1–2]	[1–3]	[1–4]
the	Noun [2]	[2–3]	[2–4]
	dog	Noun Verb [3]	[3–4]
		bit	ProperNoun [4]
			John

On to the third column...

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	[1–3]	[1–4]
the	Noun [2]	**←Nominal** [2–3]	[2–4]
	dog	Noun Verb [3]	[3–4]
		bit	ProperNoun [4]
			John

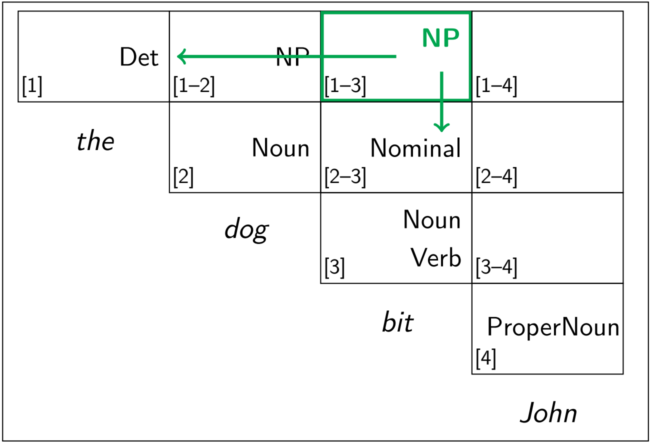
Rule application: `Nominal` $\rightarrow$ `Noun Noun`

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	[1–3]	[1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	[3–4]
		bit	ProperNoun [4]
			John

In general, non-leaf cells with N words can be split into two in $N-1$ ways.

Example: CKY – The dog bit John.



In general, non-leaf cells with N words can be split into two in $N-1$ ways. When $N=3$:

- 2 to the left, 1 down: rule application `NP` $\rightarrow$ `Det Nominal`

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	NP S [1–3]	[1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	[3–4]
		bit	ProperNoun [4]
			John

In general, non-leaf cells with N words can be split into two in $N-1$ ways. When $N=3$:

- 2 to the left, 1 down: rule application `NP` $\rightarrow$ `Det Nominal`
- 1 to the left, 2 down: rule application `S` $\rightarrow$ `NP Verb`

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	NP S [1–3]	[1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	[3–4]
		bit	ProperNoun [4]
			John

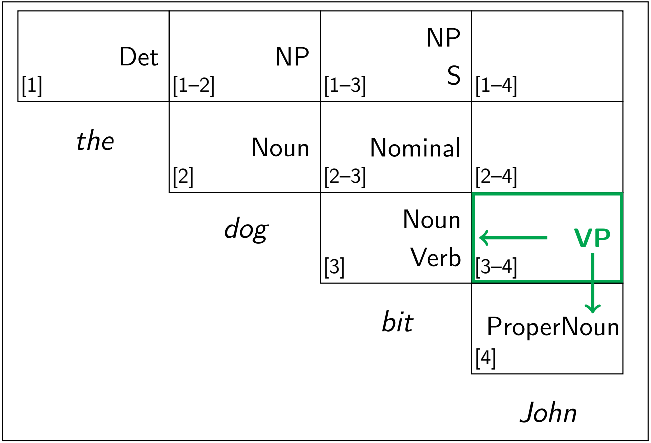
Here we found an S. However, it is not at the top right cell, so we are not done yet.

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	NP S [1–3]	[1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	[3–4]
		bit	ProperNoun [4]
			John

On to the fourth column...

Example: CKY – The dog bit John.



Rule application: `VP` $\rightarrow$ `Verb ProperNoun`

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	NP S [1–3]	[1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	VP [3–4]
		bit	ProperNoun [4]
			John

No applicable rules

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	NP S [1–3]	[1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	VP [3–4]
		bit	ProperNoun [4]
			John

No applicable rules

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	NP S [1–3]	[1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	VP [3–4]
		bit	ProperNoun [4]
			John

The top right cell represents the whole sentence.

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	NP S [1–3]	**S** [1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	VP [3–4]
		bit	ProperNoun [4]
			John

Rule application: `S` $\rightarrow$ `NP VP`

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	NP S [1–3]	**S** [1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	VP [3–4]
		bit	ProperNoun [4]
			John

Example: CKY – The dog bit John.

Det [1]	NP [1–2]	NP S [1–3]	$\boxed{\text{S}}$ [1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	VP [3–4]
		bit	ProperNoun [4]
			John

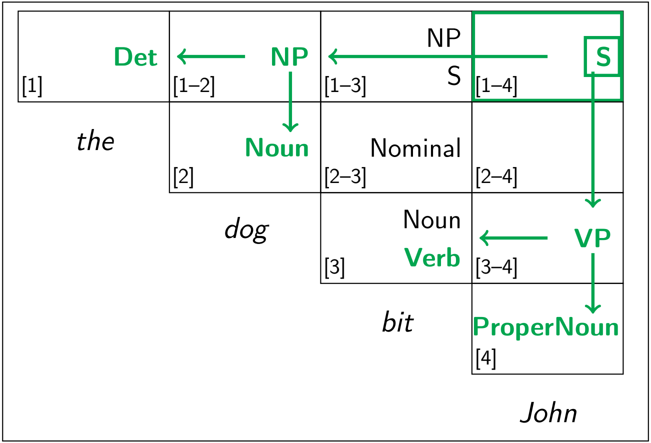
S in the top right cell: sentence accepted.

Example: CKY – The dog bit John.

Det [1]	**NP** [1–2]	NP S [1–3]	**S** [1–4]
the	Noun [2]	Nominal [2–3]	[2–4]
	dog	Noun Verb [3]	**VP** [3–4]
		bit	ProperNoun [4]
			John

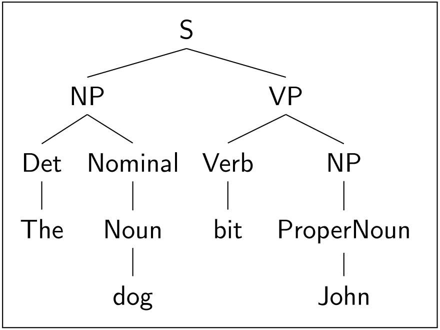
Each nonterminal maintains backpointers to its children (here: NP and VP)

Example: CKY – The dog bit John.

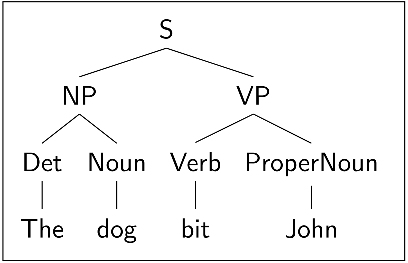


The backpointers define the syntax tree.

Example: the final tree

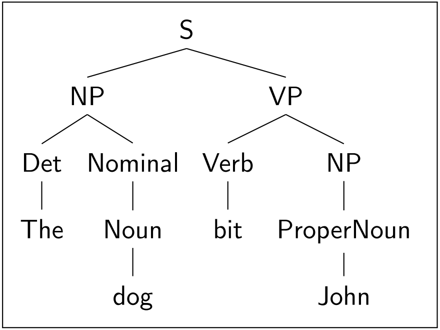


The original parse tree

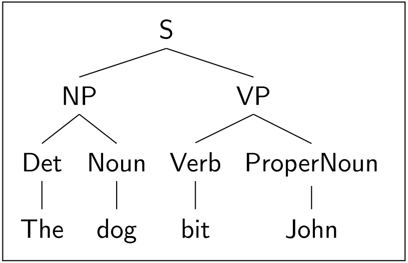


The tree found by the CKY

Example: the final tree



The original parse tree



The tree found by the CKY

Due to the CNF requirement, the CKY algorithm did not return the tree we were expecting. There are two solutions to this problem:

- convert the tree back from the CNF to the full grammar
- start from an already binary grammar

Chomsky Hierarchy

Formal languages

We have seen two formalisms for representing linguistic phenomena:

- Regular expressions and finite state automata (FSA)
- Phrase-structure grammars. In formal language theory, these are called **Context-free grammars (CFG)**

Formal languages

We have seen two formalisms for representing linguistic phenomena:

- Regular expressions and finite state automata (FSA)
- Phrase-structure grammars. In formal language theory, these are called **Context-free grammars (CFG)**

Formal languages are different in

- expressive power
- speed of inference

Formal languages

We have seen two formalisms for representing linguistic phenomena:

- Regular expressions and finite state automata (FSA)
- Phrase-structure grammars. In formal language theory, these are called **Context-free grammars (CFG)**

Formal languages are different in

- expressive power
- speed of inference

In particular,

- CFGs are more expressive than regular expressions/FSAs
- They are also much slower: $\mathcal{O}(n^3)$ vs $\mathcal{O}(n)$

Chomsky Hierarchy

Chomsky, (1956) classifies formal languages by the rules they can contain.

Type	Name	Rules allowed	Example
0	Turing complete	$\alpha \rightarrow \beta$	LFG, Python
1	Context sensitive	$\alpha A \beta \rightarrow \alpha \gamma \beta$	
-	Mildly context sensitive		CCG
2	Context free	$A \rightarrow \gamma$	PSG
3	(Right) regular	$A \rightarrow aB$ or $A \rightarrow a$	FSA, regex

Here,

- a is a terminal symbol
- A and B are non-terminals
- α, β, γ are strings of both terminal and non-terminal symbols

Formal languages: questions

Q. What is it that cannot be expressed with a regular expression?

A. **Center embedding**:

(1) The cat likes tuna fish.

(2) The cat the dog chased likes tuna fish.

(3) The cat the dog the rat bit chased likes tuna fish.

With regular expressions, we could model it with $\texttt{NP}^{\texttt{i}}\ \texttt{V}^{\texttt{i}}$ `tuna fish`. Unfortunately, the closest we can get is $\texttt{NP}^{+}\ \texttt{V}^{+}$ `tuna fish`.

Formal languages: questions

Q. What is it that cannot be expressed with a regular expression?

A. **Center embedding**:

(1) The cat likes tuna fish.

(2) The cat the dog chased likes tuna fish.

(3) The cat the dog the rat bit chased likes tuna fish.

With regular expressions, we could model it with $\texttt{NP}^{\texttt{i}}$ $\texttt{V}^{\texttt{i}}$ `tuna fish`. Unfortunately, the closest we can get is $\texttt{NP}^{+}$ $\texttt{V}^{+}$ `tuna fish`.

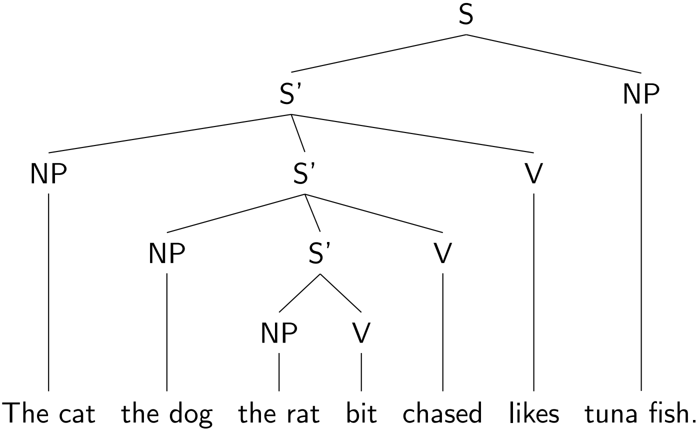
With CFG, it is easy:

$$\begin{array}{lcl} \texttt{S} & \rightarrow & \texttt{S' tuna fish} \\ \texttt{S'} & \rightarrow & \texttt{NP S' V} \mid \epsilon \end{array}$$

Formal languages: questions

Q. Can CFGs express everything in natural grammars?

A. CFG could deal with center embedding, because it could generate the `NP-V` pairs from the inside out:



Formal languages: questions

Q. Can CFGs express everything in natural grammars?

A. In Swiss German however, cross-serial dependencies also occur:

(1) *(Jan säit das) mer^1 d'chind^2 em Hans^3 es*
(Jan says that) we^1 the children/ACC^2 Hans/DAT^3 the
huus^4 haend wele^1 laa^2 hälfe^3 aastriiche^4.
house/ACC^4 have wanted^1 to let^2 help^3 paint^4.
'(Jan says that) we^1 have wanted^1 to let^2 the children^2 help^3 Hans^3 paint^4 the house^4.'

Such phenomena can only be modelled with (mildly) context-sensitive grammars.

Probabilistic PSG

Probabilistic Parsing

The parse methods introduced previously might return multiple parses due to ambiguity.

- Not all parses are born equal
- Difficult to decide which tree is the most correct

Probabilistic Context-Free Grammars (PCFG) aim to solve this problem by

- assigning a probability to each production rule
- probability a of parse tree is the product of the rule probabilities:

$$P(Tree) = \prod_{i=1}^{n} P(rule_i)$$

- the "correct" parse is the most probable tree: $\text{argmax}_i \, P(Tree_i|S)$

PCFG: training

The rule probabilities are obtained from a **treebank**:

- A text corpus annotated by linguists
- Contains the syntax tree for each sentence
- Examples:
  - English: Penn Treebank
  - Hungarian: Szeged Treebank

The probability for a rule is proportional to its normalized corpus frequency. Given a rule, e.g. $\texttt{S} \rightarrow \texttt{NP VP}$,

$$P(\texttt{S} \rightarrow \texttt{NP VP}) = P(\texttt{S} \rightarrow \texttt{NP VP}|\texttt{S}) = \frac{\#[\texttt{S} \rightarrow \texttt{NP VP}]}{\#\texttt{S}}$$

Normalization by the left hand side of the rule is important, so that

$$\sum_{\alpha} P(\texttt{S} \rightarrow \alpha) = 1$$

, i.e. the expansions of a nonterminal form a distribution.

PCFG: parsing and evaluation

Parsing is done with the probabilistic version of the parsing algorithms, e.g. PCKY.

Evaluation:

- On the "test" split of Treebanks
- The standard evaluation measure is the PARSEVAL metric
- It measures how much of the *constituents* are correct, meaning:
  - the constituent spans the same words as in the gold standard
  - it has the same label (NP, VP, etc.)
- Correctness is given by the F_1-score. The state of the art is 92-93%.

Problems with (P)CFG: #1

Problem: it does not support subcategorization.

(1) John ate a cake.

(2) *John slept a cake.

These two sentences were generated by the same $\texttt{VP} \rightarrow \texttt{Verb NP}$ rule.

Solution: **Lexicalization**

- Annotate constituents with its lexical head:
  - $\texttt{VP[eat]} \rightarrow \texttt{Verb[eat] NP}$ will appear during training
  - $\texttt{VP[sleep]} \rightarrow \texttt{Verb[sleep] NP}$ will not
- Introduces the **sparsity problem**
  - Smoothing: interpolate with the raw probability $P(\texttt{VP[eat]} \rightarrow \texttt{Verb[eat] NP}) + \lambda P(\texttt{VP} \rightarrow \texttt{Verb NP})$
  - Better handled via semantic methods

Problems with (P)CFG: #2

Problem:

- (P)CFG has strong independence assumptions: a rule can be applied whenever its left-hand nonterminal is present, irrespective of context
- Language is full of interdependence:

(1) The dog was chasing the cat.

(2) *The dog were chasing the cat.

Solution: annotate constituents with grammatical features, and propagate them up the tree

- E.g. number and person
- A sentence is grammatical iff the features agree (*dog* and *was* are both `[Sg]`)
- This technique is **mildly context sensitive**

Dependency Grammar

Dependency Grammar – Motivation

Phrase structure grammars are a departure from traditional linguistics.

- Traditional categories are missing (subject, object, etc.)
- Relationships are hidden in the tree structure
- Interpreting the tree structure requires effort
- English-centric: order of constituents matter

Dependency Grammar – Motivation

Phrase structure grammars are a departure from traditional linguistics.

- Traditional categories are missing (subject, object, etc.)
- Relationships are hidden in the tree structure
- Interpreting the tree structure requires effort
- English-centric: order of constituents matter

Dependency Grammars (DG) take a more traditional stance:

- Relationships, such as verb arguments, are directly encoded
- A more semantic view on syntax
- Handles free word order

Dependency Grammar in a nutshell

Similarly to PSG, a dependency parse is a tree:

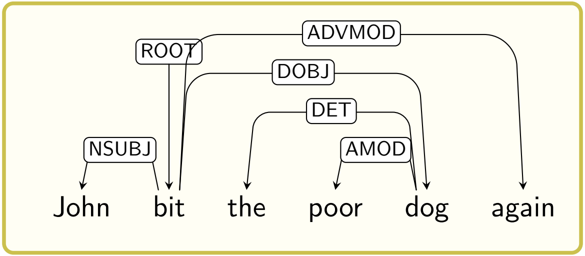
- Nodes are the words in the sentence
- Edges are grammatical relations between them
- The `ROOT` is a virtual node outside the sentence

Dependency Grammar in a nutshell

Similarly to PSG, a dependency parse is a tree:

- Nodes are the words in the sentence
- Edges are grammatical relations between them
- The `ROOT` is a virtual node outside the sentence

Example:



Dependency Grammar in a nutshell

The infrastructure is similar to PSG/PCFGs:

- Treebanks (sometimes converted from PCFG treebanks)
- Dynamic programming parsing algorithm (e.g. Eisner algorithm)
- Evaluation metrics (edge accuracy, F1)

Dependency Grammar in a nutshell

The infrastructure is similar to PSG/PCFGs:

- Treebanks (sometimes converted from PCFG treebanks)
- Dynamic programming parsing algorithm (e.g. Eisner algorithm)
- Evaluation metrics (edge accuracy, F1)

The most popular grammar formalism in NLP today.

Outlook

Other formalisms

Besides PSG and DG, a large number of formal grammars exist. Some of the most interesting formalisms are

- Lexical-Functional Grammar (LFG)
- Combinatory Categorial Grammar (CCG)
- Link Grammar (LG)

All **lexicalized grammars**, which put the brunt of syntax into the lexicon.

Other formalisms

Besides PSG and DG, a large number of formal grammars exist. Some of the most interesting formalisms are

- Lexical-Functional Grammar (LFG)
- Combinatory Categorial Grammar (CCG)
- Link Grammar (LG)

All **lexicalized grammars**, which put the brunt of syntax into the lexicon.

Others not detailed here

- Head-driven Phrase Structure Grammar (HPSG)
- Tree-adjoining Grammar (TAG)
- Constraint Grammar (CG)
- ...

Other formalisms

Besides PSG and DG, a large number of formal grammars exist. Some of the most interesting formalisms are

- Lexical-Functional Grammar (LFG)
- Combinatory Categorial Grammar (CCG)
- Link Grammar (LG)

All **lexicalized grammars**, which put the brunt of syntax into the lexicon.

Others not detailed here

- Head-driven Phrase Structure Grammar (HPSG)
- Tree-adjoining Grammar (TAG)
- Constraint Grammar (CG)
- ...

There isn't really just one "right tool" for the job!

Other formalisms: LFG

Lexical-functional grammar provides a multi-dimensional view on language.

- *c-structure*: the constituent structure
- *f-structure*: the functional (predicate) structure
- Optional structures: semantic, morphologic, etc.
- Relations between words in the lexicon (e.g. active–passive)

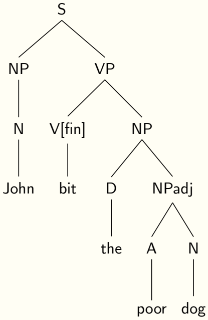
Other formalisms: LFG

Lexical-functional grammar provides a multi-dimensional view on language.

- *c-structure*: the constituent structure
- *f-structure*: the functional (predicate) structure
- Optional structures: semantic, morphologic, etc.
- Relations between words in the lexicon (e.g. active–passive)

Example:

```
[pred         'bite<[7:John], [2:dog]>'
 tns-adp      [tense past, prog -, perf -, mood indicative]
              [pred     'John'
 subj          ntype    [nsem   [proper  [proper-type name,
                                          name-type first_name]]
                         nsyn   proper]
               pers 3,  num sg, human +, gend-sem male, case nom]
 obj          ...
 vtype main,  passive -, clause-type decl]
```



Other formalisms: (C)CG

Categorial grammars are based on the idea of *compositionality*.

- Lexicon: each word has a *lexical category*
  - Basic: N, NP, S
  - Function: `the: NP/N` means: "*the*" is a function that takes a N to produce an NP
  - Argument on left: $S\backslash NP$ ($\equiv$ VP), right: N/N ($\equiv$ Adj)
- Inference rules: language-agnostic
- The most popular formalism is Combinatory Categorial Grammar (CCG), which is based on *combinatory logic*.

Other formalisms: (C)CG

Categorial grammars are based on the idea of *compositionality*.

- Lexicon: each word has a *lexical category*
  - Basic: N, NP, S
  - Function: `the: NP/N` means: "*the*" is a function that takes a N to produce an NP
  - Argument on left: $S\backslash NP$ ($\equiv$ VP), right: N/N ($\equiv$ Adj)
- Inference rules: language-agnostic
- The most popular formalism is Combinatory Categorial Grammar (CCG), which is based on *combinatory logic*.

Example:

$$\dfrac{\dfrac{\dfrac{\textit{John}}{NP}\quad\dfrac{\textit{bit}}{(S\backslash NP)/NP}}{S\backslash NP}\qquad\dfrac{\dfrac{\textit{the}}{NP/N}\quad\dfrac{\dfrac{\textit{poor}}{N/N}\quad\dfrac{\textit{dog}}{N}}{N}}{NP}}{S}$$

John	NP
bit	(S\NP)/NP
the	NP/N
poor	N/N
dog	N

Other formalisms: LG

Link Grammar is a theory of syntax and (optionally) morphology.

- Like DG, it builds labelled relations between words
- Unlike DG, it is highly lexicalized:
  - the main resource is the *dictionary*, which lists *words*
  - along with their *linking requirements*
- Each word is like a jigsaw/domino piece, and parsing is akin to assembling a puzzle
- Word order matters: extensions for free word order languages

Other formalisms: LG

Link Grammar is a theory of syntax and (optionally) morphology.

- Like DG, it builds labelled relations between words
- Unlike DG, it is highly lexicalized:
  - the main resource is the *dictionary*, which lists *words*
  - along with their *linking requirements*
- Each word is like a jigsaw/domino piece, and parsing is akin to assembling a puzzle
- Word order matters: extensions for free word order languages

Example:

```
    +----------------Xp---------------+
    |            +-------Os-------+   |
    |            |    +-----Ds----+   |
    +---Wd--+-Ss-+    |     +--A--+   |
    |       |    |    |     |     |   |
LEFT-WALL John bit.v the poor.a dog.n .
```

John	`S+ & O-`
bit	`S- & O+`
the	`D+`
poor	`A+`
dog	`D- & S+ & O-`
.	`Xp-`

Resources

- (P)CFG and DG
  - For English:
    - Online version of the Stanford Parser
    - The Stanford CoreNLP library
  - For Hungarian:
    - `e-magyar.hu`
    - The hunlp-GATE library
- LFG
  - XLE-Web, an online LFG parser
- CCG
  - A primer to CCG (Steedman and Baldridge, 2011)
  - The OpenCCG parser
- Link Grammar
  - Webpage with online parser: http://www.link.cs.cmu.edu/link/
  - Original report (Sleator and Temperley, 1991)

Appendix: bibliography

Chomsky, Noam (1956). "Three models for the description of language". In: *IRE Transactions on Information Theory* 2, pp. 113–124.

Sleator, Daniel and Davy Temperley (1991). *Parsing English with a Link Grammar*. Tech. rep. url: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/link/pub/www/papers/ps/tr91-196.pdf.

Steedman, M. and J. Baldridge (2011). "Combinatory Categorial Grammar". In: *Nontransformational Syntax: A Guide to Current Models. Blackwell, Oxford*. url: http://homepages.inf.ed.ac.uk/steedman/papers/ccg/SteedmanBaldridgeNTSyntax.pdf.