Introduction to Text Mining

Part V: Text Mining using Grammars

Henning Wachsmuth

https://cs.upb.de/css

Text Mining using Grammars: Learning Objectives

Concepts

- Different types of formal grammars
- Use of formal grammars for text mining
- Benefits and limitations of the different types

Text analysis techniques

- Identification of numeric entities with regular expressions
- Data-driven approaches to syntactic parsing
- Syntactic parsing of sentences with the (extended) CKY algorithm

Covered text analyses

- Time expression recognition
- Constituency parsing
- Dependency parsing

Outline of the Course

I. Overview

II. Basics of Linguistics

III. Text Mining using Rules

IV. Basics of Empirical Methods

V. Text Mining using Grammars

- What Is Text Mining using Grammars?
- Regular Grammars
- Probabilistic Context-Free Grammars
- Parsing based on a PCFG
- Dependency Grammars

VI. Basics of Machine Learning

VII. Text Mining using Similarities and Clustering

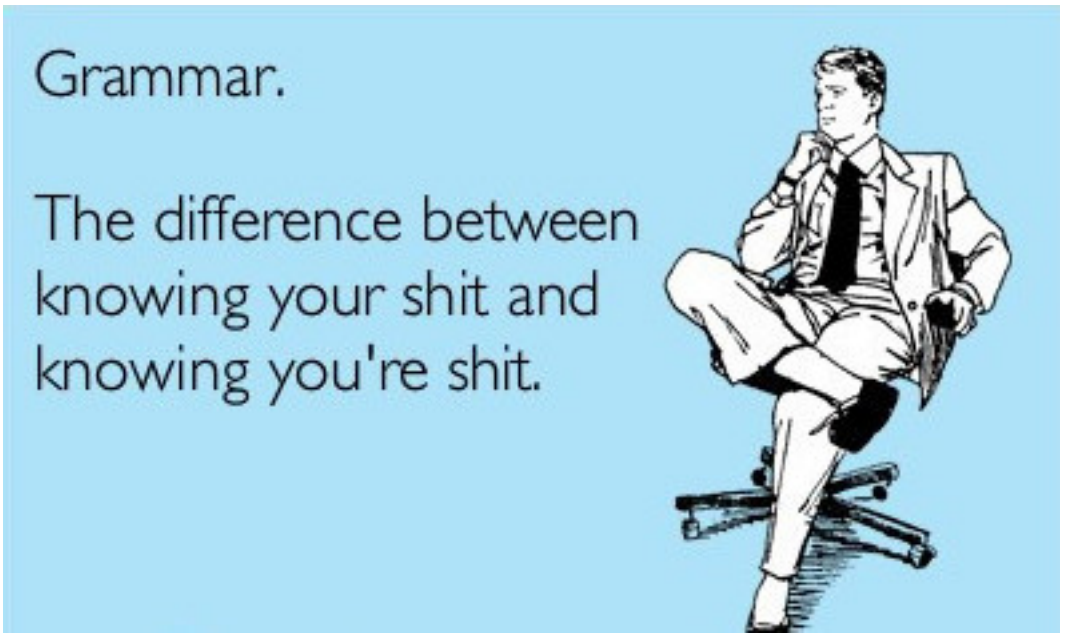
VIII. Text Mining using Classification and Regression

IX. Text Mining using Sequence Labeling

X. Practical Issues

What Is Text Mining using Grammars?

What Is Text Mining using Grammars?



What Is Text Mining using Grammars?

Grammars

What is a grammar?

- A grammar is a description of the valid structures of a language.
- One of the most central concepts of linguistics is *formal grammars*.

Formal grammars

- A formal grammar specifies a set of rules consisting of terminal and non-terminal symbols.
- Terminals. "Words" that cannot be rewritten any further.
- Non-terminals. Clusters or generalizations of terminals.

Grammar (Σ, N, S, R)

Σ An alphabet, i.e., a finite set of terminal symbols, $\Sigma = \{v_1, v_2, \ldots\}$.

N A finite set of non-terminal symbols, $N = \{W_1, W_2, \ldots\}$.

S A start non-terminal symbol, $S \in N$.

R A finite set of production rules, $R \subseteq (\Sigma \cup N)^+ \setminus \Sigma^* \times (\Sigma \cup N)^*$.

What Is Text Mining using Grammars?

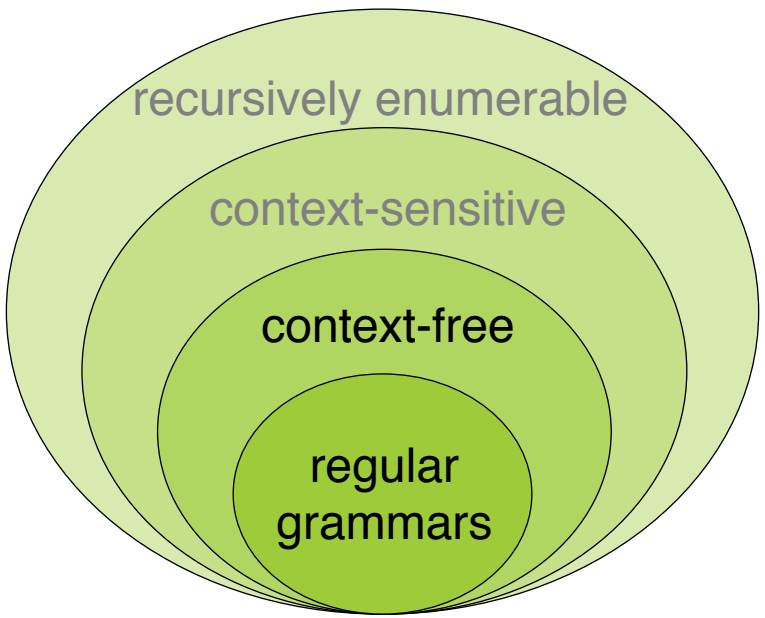
Chomsky Grammars *

Four types of formal grammars

- Chomsky-0 (recursively enumerable). Any (Σ, N, S, R) as defined.
- Chomsky-1 (context-sensitive). Only rules $U \to V$ with $|U| \leq |V|$.
- Chomsky-2 (context-free). Only rules $U \to V$ with $U \in N$.
- Chomsky-3 (regular). Only rules $U \to V$ with $U \in N$ and $V \in \{\varepsilon, v, vW\}$, $v \in \Sigma$, $W \in N$.

Grammars in text mining

- Only regular and context-free grammars are commonly used.



What Is Text Mining using Grammars?

Regular Grammars

Regular grammars in text mining

- Regular grammars tend to be effective for information whose language follows clear sequential patterns.
- Examples. Numeric entities, structural entities (e.g., eMail addresses), lexico-syntactic relations (e.g., “<NN> is a <NN>”), ...
- To infer information, texts are matched against *regular expressions*.

Numeric (and alphanumeric) entities

- Values, quantities, proportions, ranges, or similar.
- Examples are times, dates, phone numbers, monetary values, ...

“in this year” “2018-10-18” “$ 100 000” “60-68 44”

Numeric entity recognition

- The text analysis that mines numeric entities from text.
- Used in text mining within many information extraction tasks.

What Is Text Mining using Grammars?

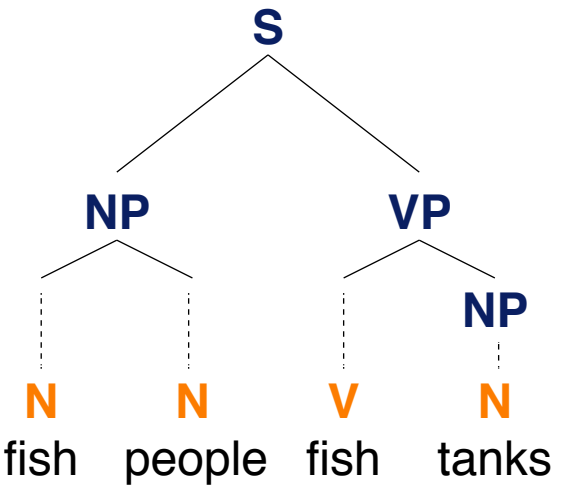
Context-Free Grammars

Context-free grammars (CFGs) in text mining

- CFGs tend to be effective for hierarchical structures of language.
- Probabilistic extensions (PCFGs) capture the likeliness of structures.
- CFGs usually define the basis of syntactic parsing.

Syntactic parsing (aka full parsing)

- The text analysis that determines the syntactic structure of a sentence.
- Used in text mining as preprocessing for many tasks, e.g., relation extraction.



Constituency vs. dependency parsing

- Constituency parsing. Infers the phrase structure of a sentence.
- Dependency parsing. Uses a *dependency grammar*, i.e., a special case of CFGs where relations are modeled directly between words.

Regular Grammars

Woodchucks *

How much wood would a woodchuck chuck, if a woodchuck could chuck wood?

So much wood as a woodchuck would,
if a woodchuck could chuck wood.

He would chuck, if he would, as much wood as he could, if a woodchuck could chuck wood.

A woodchuck would chuck no amount of wood, since a woodchuck can't chuck wood.

But if a woodchuck could and would chuck some wood, what amount of wood would a woodchuck chuck?

Even if a woodchuck could chuck wood, and even if a woodchuck would chuck wood, should a woodchuck chuck wood?

A woodchuck should chuck wood if a woodchuck could chuck wood, as long as a woodchuck would chuck wood.

Woodchucks

Mining Woodchucks from Text

How can we find all of all these in a text?

- “woodchuck”
- “Woodchuck”
- “woodchucks”
- “Woodchucks”
- “WOODCHUCK”
- “WOODCHUCKS”
- “wooooodchuck”
- “groundhog” (synonym)

... and so on

Notice

- The previous slide does *not* show really insightful examples.

Regular Grammars

What is a regular grammar?

- A grammar (Σ, N, S, R) is *regular* if all rules in R are of the form $U \to V$ with $U \in N$ and $V \in \{\varepsilon, v, vW\}$, where $v \in \Sigma$ and $W \in N$.

  ε is the empty word.
- Extended. A regular grammar is *extended*, if $v \in \Sigma^*$.

  Below, we refer to them also as *regular grammar* only.
- Right-regular. Intuitively, a structure defined by a regular grammar can be constructed from left to right.
- A language is regular, if there is a regular grammar that defines it.

Representation of regular grammars

- Every regular grammar can be represented by a *finite-state automaton*.
- Every regular grammar can be represented by a *regular expression*.

  And vice versa.

Regular Grammars

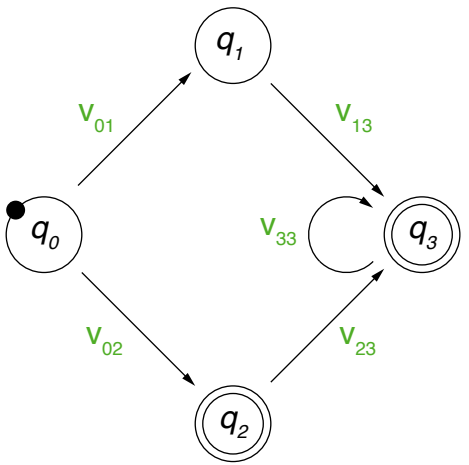
Finite-State Automata

Finite-state automaton (FSA)

- An FSA is a state machine that reads a string from a specific regular language. It represents the set of all strings belonging to the language.

An FSA as a 5-tuple $(Q, \Sigma, q_0, F, \delta)$

- Q A finite set of $n > 0$ states, $Q = \{q_0, ..., q_n\}$.
- Σ An alphabet, i.e., a finite set of terminal symbols, $\Sigma \cap Q = \emptyset$.
- q_0 A start state, $q_0 \in Q$.
- F A set of final states, $F \subseteq Q$.
- δ A transition function between states, triggered based on $v \in \Sigma$, $\delta : Q \times \Sigma \rightarrow Q$.



Regular Expressions

Regular expression (aka regex)

- A regex defines a regular language over an alphabet Σ as a sequence of characters (from Σ) and metacharacters.
- Metacharacters denote disjunction, negation, repetition, ... (see below).
- Example. The example FSA above is defined by the following regex.

$$\mathrm{v}_{02} \mid (\mathrm{v}_{01}\mathrm{v}_{13} \mid \mathrm{v}_{02}\mathrm{v}_{23})\,\mathrm{v}_{33}{}^{*}$$

Use of regular expressions

- Definition of patterns that generalize over structures of a language.
- A pattern matches all spans of text that contain any of the structures.

Regular expressions in text mining

- Regexes are a widely used technique in text mining, particularly for the extraction of numeric and similar entities.
- In machine learning, regexes often take on the role of features.

Regular Expressions

Characters and Metacharacters

Regular characters

- The default interpretation of a character sequence in a regex is a concatenation of each single character.

  `woodchuck` matches “woodchuck”

Metacharacters

- A regex uses specific metacharacters to efficiently encode specific regular-language constructions, such as negation and repetition.
- The main metacharacters are presented below in Python notation:

  `[]` `-` `|` `^` `.` `( )` `\` `*` `+` `?`

  The used metacharacters partly differ across literature and programming languages.

- Some languages also include certain non-regular constructions, e.g., `\b` matches if a word boundary is reached.

  Regexes can solve this case when given token information.

Regular Expressions

Disjunction

Disjunction of patterns

- Brackets `[]` specify a character class.

  `[wod]` matches "w" or "o" or "d" `[wW]` matches "w" or "W"

- Disjunctive ranges of characters can be specified with a hyphen `-`.

  `[a-zA-Z]` matches any letter `[0-8]` matches any digit except for "9"

- The pipe `|` specifies a disjunction of string sequences.

  `groundhog|woodchuck` matches "groundhog" and "woodchuck"

Notes on disjunctions

- Combinations of different disjunctions are often useful.

  `[gG]roundhog|[wW]oodchuck` matches "groundhog", "Woodchuck", ...

- In Python, many metacharacters are not active within brackets.

  `[wod.]` matches "w", "o", "d", and "."

Regular Expressions

Negation, Choice, Grouping

Negation

- The caret `^` inside brackets complements the specified character class.

  `[^0-9]` matches anything but digits `[^wo]` matches any character but “w”, “o”

- Outside brackets, the caret `^` is interpreted as a normal character.

  `woodchuck^` matches “woodchuck^”

Free choice

- The period `.` matches any character.

  `w..dchuck` matches “woodchuck”, “woudchuck”, ...

  To match a period, it needs to be escaped as: `\.`

Grouping

- Parentheses `()` can be used to group parts of a regex. A grouped part is treated as a single character.

  `w[^(oo)]dchuck` matches any variation of the two o’s in “woodchuck”

Regular Expressions

Whitespaces and Predefined Character Classes

Whitespaces

- Different whitespaces are referred to with different special characters.
- For instance, `\n` is the regular new-line space.

Predefined character classes

- Several specific character classes a referred to by a backslash `\` followed by a specific letter.

  `\d` Any decimal digit. Equivalent to `[0-9]`.
  `\D` Any non-digit character. Equivalent to `[^0-9]`.
  `\s` Any whitespace character. Equivalent to `[\t\n\r\f\v]`.
  `\S` Any non-whitespace character. Equivalent to `[^\t\n\r\f\v]`.
  `\w` Any alphanumeric character. Equivalent to `[a-zA-Z0-9]`.
  `\W` Any non-alphanumeric character;. Equivalent to `[^a-zA-Z0-9]`.

- These classes can be used within brackets.

  `[\s0-9]` matches any space and digit.

Regular Expressions
Repetition

Repetition

- The asterisk `*` repeats the previous character zero or more times.

  `woo*dchuck` matches “wodchuck”, “woodchuck”, “wooodchuck”, “woooodchuck”, ...

- The plus `+` repeats the previous character one or more times.

  `woo+dchuck` matches “woodchuck”, “wooodchuck”,“woooodchuck”, ...

- The question mark `?` repeats the previous character zero or one time.

  `woo?dchuck` matches “wodchuck” and “woodchuck”

Notes on repetitions

- Repetitions are implemented in a greedy manner in many programming languages, i.e., longer matches are preferred over shorter ones.

  `to*` matches “too”, not “to”o”, ...

- This may actually violate the regularity of the defined language.

  “woodchuck” needs to be processed twice for the regex `wo*odchuck`

Regular Expressions

Summary of Metacharacters

Char	Concept	Example
`[ ]`	Disjunction of characters	`[Ww]oodchuck`
`-`	Ranges in disjunctions	`There are [0-9]+ woodchucks\.`
`|`	Disjunction of regexes	`woodchuck|groundhog`
`^`	Negation	`[^0-9]`
`.`	Free choice	`What a (.)* woodchuck`
`( )`	Grouping of regex parts	`w(oo)+dchuck`
`\`	Special (sets of) characters	`\swoodchuck\s`
`*`	Zero or more repetitions	`wooo*dchuck`
`+`	One or more repetitions	`woo+dchuck`
`?`	Zero or one repetition	`woodchucks?`

Regular Expressions

Examples

The

- Regex for all variations of “the” in news article text:

```
the
```
(misses capitalized cases, matches “theology”, ...)

```
[^a-zA-Z][tT]he[^a-zA-Z]
```
(requires a character before and afterwards)

Woodchucks

- Regex for all woodchuck cases from above (and for similar):

```
[wW][oO][oO]+[dD][cC][hH][uU][cC][kK][sS]? | groundhog
```

eMail Adresses

- All eMail addresses with the German top-level domain, whose text segments contain no special characters.

```
[a-zA-Z0-9]+ @ ([a-zA-Z0-9]+\.)* [a-zA-Z0-9][a-zA-Z0-9]+ \.de
```

Time Expression Recognition with Regular Expressions

What is a time expression?

- A time expression is as an alphanumeric entity that represents a date or a period.

> "Cairo, August 25th 2010 — Forecast on Egyptian Automobile industry
>
> [...] In the next five years, revenues will rise by 97% to US-$ 19.6 bn. [...]"

Time expression recognition

- The text analysis that finds time expressions in natural language text.
- Used in text mining within temporal relation and event extraction.

Approach in a nutshell

- Model phrase structure of time expressions with a complex regex.
- Include lexicons derived from training data to match closed-class terms, such as month names and prepositions.
- Match regex with sentences of a text.

  The matching approach can easily be adapted to any other type of information.

Time Expression Recognition with Regular Expressions

Pseudocode

Signature

- Input. A text split into sentences, and a regex.
- Output. All time expressions in the text.

```
extractAllMatches(List<Sentence> sentences, Regex regex)
 1.     List<TimeExpression> matches ← ()
 2.     for each sentence ∈ sentences do
 3.         int index ← 0
 4.         while index < sentence.length - 1 do
 5.             int [] exp ← regex.match(sentence.sub(index))
 6.             if exp ≠ ⊥ then   // ⊥ represents "null"
 7.                 matches.add(new TimeExpression(exp[0], exp[1]))
 8.                 index ← exp[1]
 9.             index ← index + 1
10.     return matches
```

Notice

- Most programming languages provide explicit matching classes.

Time Expression Recognition with Regular Expressions

Complete Regex (Part 1 out of 2) *

```
((((((iI]n|[wW]ithin|[tT]o\s\s?the|[tT]o|[fF]or\s\s?the|[fF]or|[fF]rom|[sS]ince|[aA]fter|[bB]efore|[bB]etween|[aA]t|[oO]n|[
oO]ver|[pP]er)((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*([tT]he|[tT]his|[tT]hese|[tT]hose|[iI]ts))?)(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))
\s*)?(((0?[123456789]|[12]\d|3[01])(\.|/))((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)?)(((month|time(span)?(\s+(\r(\n)?|\n)?|(\r(\n)
?|\n))\s*(from(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)?)?([Jj]anuary|[Jj]an\.|[Jj]an|[Ff]ebruary|[Ff]eb\.|[Ff]eb|[Mm]arch|[Mm]ar\
.|[Mm]ar|[Aa]pril|[Aa]pr\.|[Aa]pr|[Mm]ay|[Jj]une|[Jj]un\.|[Jj]un|[Jj]uly|[Jj]ul\.|[Jj]ul|[Aa]ugust|[Aa]ug\.|[Aa]ug|[Ss]epte
mber|[Ss]ep\.|[Ss]ep|[Oo]ctober|[Oo]ct\.|[Oo]ct|[Nn]ovember|[Nn]ov\.|[Nn]ov|[Dd]ecember|[Dd]ez\.|[Dd]ez|[Ss]pring|[Ss]ummer
|[Aa]utumn|[Ff]all|[Ww]inter))|((0?[123456789]|1[012])(\.|/))((()?((19|20)?\d2))?)|((((((iI]n|[wW]ithin|[tT]o\s\s?the|[tT]o
|[fF]or\s\s?the|[fF]or|[fF]rom|[sS]ince|[aA]fter|[bB]efore|[bB]etween|[aA]t|[oO]n|[oO]ver|[pP]er)((\s+(\r(\n)?|\n)?|(\r(\n)
?|\n))\s*([tT]he|[tT]his|[tT]hese|[tT]hose|[iI]ts))?)(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*((([sS]tart|[bB]egin|[Ss]tart|[Bb]egi
n|[Ee]nd|[eE]nd|[Mm]idth|[mM]idth)((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*([tT]he|[tT]his|[tT]hese|[tT]hose|[iI]ts))?)(\s+(\r(\n)
?|\n)?|(\r(\n)?|\n))\s*)?|(([sS]tart|[bB]egin|[Ss]tart|[Bb]egin|[Ee]nd|[eE]nd|[Mm]idth|[mM]idth)((\s+(\r(\n)?|\n)?|(\r(\n)?|
\n))\s*([tT]he|[tT]his|[tT]hese|[tT]hose|[iI]ts))?)(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)([[a-z]]+(\s+(\r(\n)?|\n)?|(\r(\n)?|\n)
)\s*)?)?((((((lL]ast|[pP]receding|[pP]ast|[cC]urrent|[tT]his|[uU]pcoming|[fF]ollowing|[sS]ucceeding|[nN]ext))((\s+(\r(\n)?
|\n)?|(\r(\n)?|\n))\s*(((1|2|3|4|5|6|7|8|9)\d?|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]our|[fF]ive|[sS]ix|[s
S]even|[eE]ight|[nN]ine|[tT]en|[eE]leven|[tT]welve|[tT]wenty|[tT]hirty|[fF]ourty|[fF]orty|[fF]ifty|[sS]ixty|[sS]eventy|[eE]
ighty|[nN]inety|[hH]undred|[aA]\s\s?hundred))|((1[012]?|2|3|4|5|6|7|8|9)(\.|())|([fF]irst|[sS]econd|[tT]hird|[fF]ourth|[fF]
ifth|[sS]ixth|[sS]eventh|[eE]ighth|[nN]inth|[tT]enth|[eE]leventh))(-((1[012]?|2|3|4|5|6|7|8|9)(\.|())|([fF]irst|[sS]econd|[
tT]hird|[fF]ourth|[fF]ifth|[sS]ixth|[sS]eventh|[eE]ighth|[nN]inth|[tT]enth|[eE]leventh)))?((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\
s*((1|2|3|4|5|6|7|8|9)\d?|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]our|[fF]ive|[sS]ix|[sS]even|[eE]ight|[nN]i
ne|[tT]en|[eE]leven|[tT]welve|[tT]wenty|[tT]hirty|[fF]ourty|[fF]orty|[fF]ifty|[sS]ixty|[sS]eventy|[eE]ighty|[nN]inety|[hH]u
ndred|[aA]\s\s?hundred)))?))?(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)|((((1|2|3|4|5|6|7|8|9)\d?|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]our|[fF]ive|[sS]ix|[sS]even|[eE]ight|[nN]ine|[tT]en|[eE]leven|[tT]welve|[tT]wenty|[tT]hirty|[fF]ourty
|[fF]orty|[fF]ifty|[sS]ixty|[sS]eventy|[eE]ighty|[nN]inety|[hH]undred|[aA]\s\s?hundred))|((1[012]?|2|3|4|5|6|7|8|9)(\.|())|
([fF]irst|[sS]econd|[tT]hird|[fF]ourth|[fF]ifth|[sS]ixth|[sS]eventh|[eE]ighth|[nN]inth|[tT]enth|[eE]leventh))(-((1[012]?|2|
3|4|5|6|7|8|9)(\.|())|([fF]irst|[sS]econd|[tT]hird|[fF]ourth|[fF]ifth|[sS]ixth|[sS]eventh|[eE]ighth|[nN]inth|[tT]enth|[eE]l
eventh)))?((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*((1|2|3|4|5|6|7|8|9)\d?|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]
our|[fF]ive|[sS]ix|[sS]even|[eE]ight|[nN]ine|[tT]en|[eE]leven|[tT]welve|[tT]wenty|[tT]hirty|[fF]ourty|[fF]orty|[fF]ifty|[sS
]ixty|[sS]eventy|[eE]ighty|[nN]inety|[hH]undred|[aA]\s\s?hundred)))?)((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*(([lL]ast|[pP]recedi
ng|[pP]ast|[cC]urrent|[tT]his|[uU]pcoming|[fF]ollowing|[sS]ucceeding|[nN]ext)))?(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*))?(((Q(1
|2|3|4)|H(1|2)(\/(19|20)?\d2)?|(((\w([a-z])*(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)?(year|quarter))([a-z])*))|((month|time(span)
?(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*(from(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)?)?([Jj]anuary|[Jj]an\.|[Jj]an|[Ff]ebruary|[Ff]eb\
.|[Ff]eb|[Mm]arch|[Mm]ar\.|[Mm]ar|[Aa]pril|[Aa]pr\.|[Aa]pr|[Mm]ay|[Jj]une|[Jj]un\.|[Jj]un|[Jj]uly|[Jj]ul\.|[Jj]ul|[Aa]ugust
|[Aa]ug\.|[Aa]ug|[Ss]eptember|[Ss]ep\.|[Ss]ep|[Oo]ctober|[Oo]ct\.|[Oo]ct|[Nn]ovember|[Nn]ov\.|[Nn]ov|[Dd]ecember|[Dd]ez\.|[
Dd]ez|[Ss]pring|[Ss]ummer|[Aa]utumn|[Ff]all|[Ww]inter))|(([Rr]eported\s\s?time\s\s?span|[Rr]eported\s\s?time\s\s?span|[Rr]e
ported\s\s?time|[rR]eported\s\s?time|[Tt]ime\s\s?span|[tT]ime\s\s?span|[Ss]pan|[sS]pan|[Dd]ecade|[dD]ecade)))((\s+(\r(\n)?|\
n)?|(\r(\n)?|\n))\s*((19|20)\d2(/(19|20)?\d2)?|\d2/\d2))?|((19|20)\d2(/(19|20)?\d2)?|\d2/\d2)))|((((((lL]ast|[pP]receding|[p
P]ast|[cC]urrent|[tT]his|[uU]pcoming|[fF]ollowing|[sS]ucceeding|[nN]ext))((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*(((1|2|3|4|5|6|
7|8|9)\d?|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]our|[fF]ive|[sS]ix|[sS]even|[eE]ight|[nN]ine|[tT]en|[eE]le
ven|[tT]welve|[tT]wenty|[tT]hirty|[fF]ourty|[fF]orty|[fF]ifty|[sS]ixty|[sS]eventy|[eE]ighty|[nN]inety|[hH]undred|[aA]\s\s?h
undred))|((1[012]?|2|3|4|5|6|7|8|9)(\.|())|([fF]irst|[sS]econd|[tT]hird|[fF]ourth|[fF]ifth|[sS]ixth|[sS]eventh|[eE]ighth|[n
N]inth|[tT]enth|[eE]leventh))(-((1[012]?|2|3|4|5|6|7|8|9)(\.|())|([fF]irst|[sS]econd|[tT]hird|[fF]ourth|[fF]ifth|[sS]ixth|[
sS]eventh|[eE]ighth|[nN]inth|[tT]enth|[eE]leventh)))?((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*((1|2|3|4|5|6|7|8|9)\d?|([oO]ne|[sS]
everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]our|[fF]ive|[sS]ix|[sS]even|[eE]ight|[nN]ine|[tT]en|[eE]leven|[tT]welve|[tT]went
y|[tT]hirty|[fF]ourty|[fF]orty|[fF]ifty|[sS]ixty|[sS]eventy|[eE]ighty|[nN]inety|[hH]undred|
```

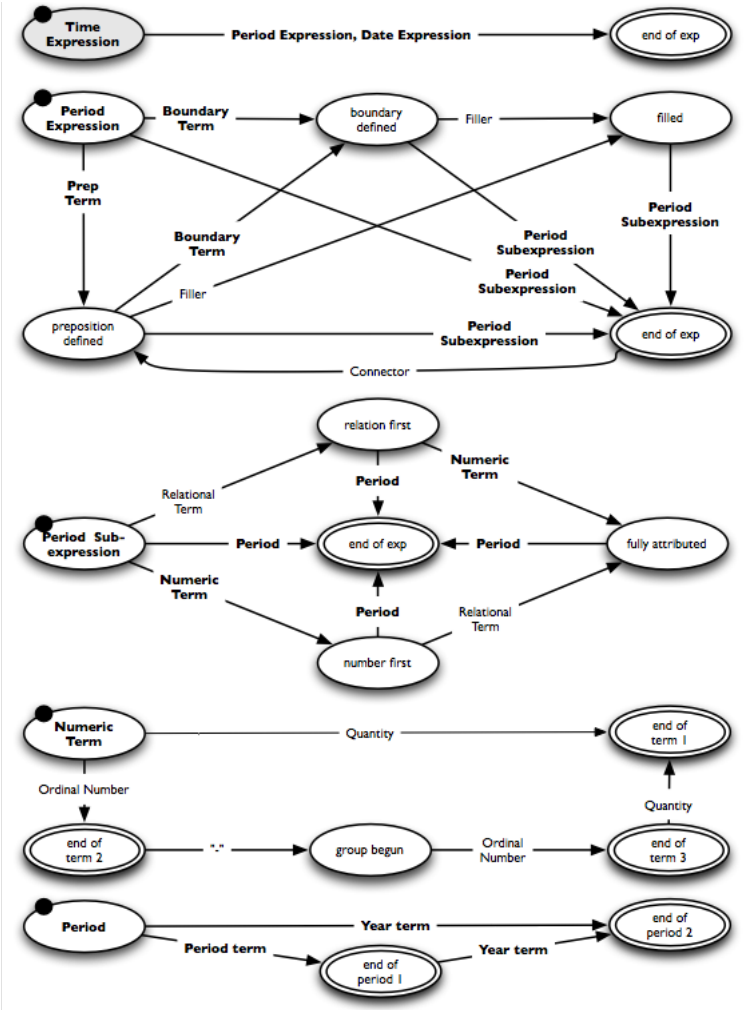
Time Expression Recognition with Regular Expressions

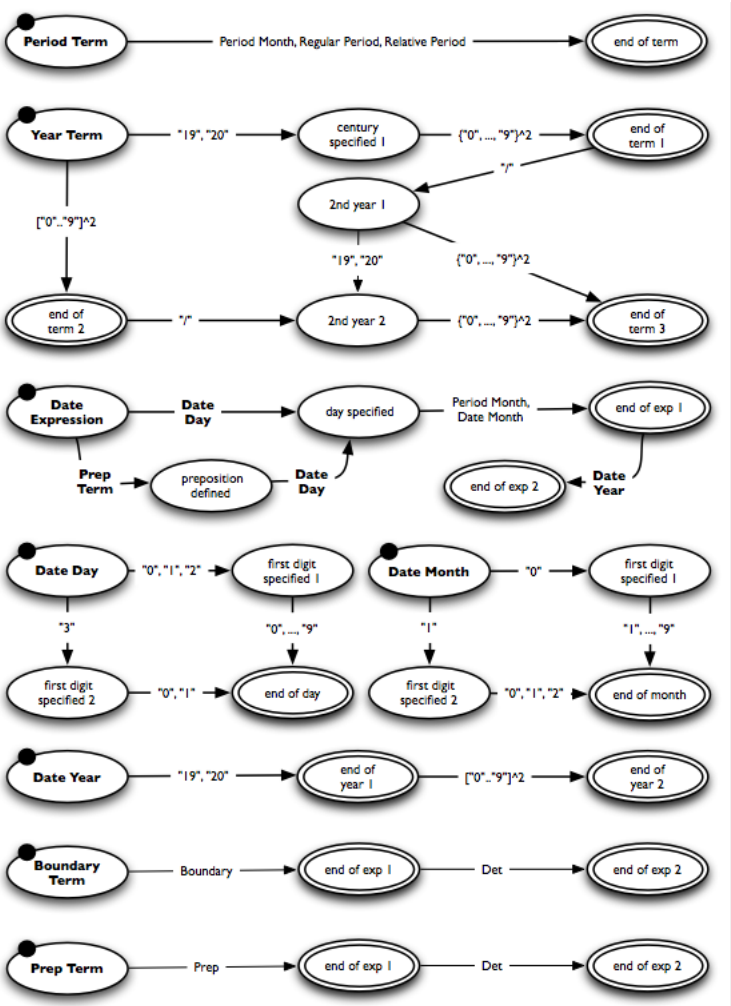
Complete Regex (Part 2 out of 2) *

```
[aA]\s\s?hundred)))?))?(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)|((((1|2|3|4|5|6|7|8|9)\d?|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]
wo|[tT]hree|[fF]our|[fF]ive|[sS]ix|[sS]even|[eE]ight|[nN]ine|[tT]en|[eE]leven|[tT]welve|[tT]wenty|[tT]hirty|[fF]ourty|[fF]o
rty|[fF]ifty|[sS]ixty|[sS]eventy|[eE]ighty|[nN]inety|[hH]undred|[aA]\s\s?hundred))|((1[012]?|2|3|4|5|6|7|8|9)(\.|())|([fF]i
rst|[sS]econd|[tT]hird|[fF]ourth|[fF]ifth|[sS]ixth|[sS]eventh|[eE]ighth|[nN]inth|[tT]enth|[eE]leventh))(-((1[012]?|2|3|4|5|
6|7|8|9)(\.|())|([fF]irst|[sS]econd|[tT]hird|[fF]ourth|[fF]ifth|[sS]ixth|[sS]eventh|[eE]ighth|[nN]inth|[tT]enth|[eE]leventh
)))?((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*((1|2|3|4|5|6|7|8|9)\d?|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]our|[f
F]ive|[sS]ix|[sS]even|[eE]ight|[nN]ine|[tT]en|[eE]leven|[tT]welve|[tT]wenty|[tT]hirty|[fF]ourty|[fF]orty|[fF]ifty|[sS]ixty|
[sS]eventy|[eE]ighty|[nN]inety|[hH]undred|[aA]\s\s?hundred))?)((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*(([lL]ast|[pP]receding|[pP]
]ast|[cC]urrent|[tT]his|[uU]pcoming|[fF]ollowing|[sS]ucceeding|[nN]ext)))?(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*))?(((Q(1|2|3|4
)|H(1|2)(\/(19|20)?\d2)?|(((\w([a-z])*(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)?(year|quarter))([a-z])*))|((month|time(span)?(\s+(
\r(\n)?|\n)?|(\r(\n)?|\n))\s*(from(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)?)?([Jj]anuary|[Jj]an\.|[Jj]an|[Ff]ebruary|[Ff]eb\.|[Ff]
eb|[Mm]arch|[Mm]ar\.|[Mm]ar|[Aa]pril|[Aa]pr\.|[Aa]pr|[Mm]ay|[Jj]une|[Jj]un\.|[Jj]un|[Jj]uly|[Jj]ul\.|[Jj]ul|[Aa]ugust|[Aa]u
g\.|[Aa]ug|[Ss]eptember|[Ss]ep\.|[Ss]ep|[Oo]ctober|[Oo]ct\.|[Oo]ct|[Nn]ovember|[Nn]ov\.|[Nn]ov|[Dd]ecember|[Dd]ez\.|[Dd]ez|
[Ss]pring|[Ss]ummer|[Aa]utumn|[Ff]all|[Ww]inter))|(([Rr]eported\s\s?time\s\s?span|[Rr]eported\s\s?time\s\s?span|[Rr]eported\
s\s?time|[rR]eported\s\s?time|[Tt]ime\s\s?span|[tT]ime\s\s?span|[Ss]pan|[sS]pan|[Dd]ecade|[dD]ecade)))((\s+(\r(\n)?|\n)?|(\r
(\n)?|\n))\s*((19|20)\d2(/(19|20)?\d2)?|\d2/\d2))?|((19|20)\d2(/(19|20)?\d2)?|\d2/\d2))))((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*(
[tT]o|[aA]nd|[oO]r|[oO]n|[aA]t|[oO]f\s\s?the|[oO]f|[tT]he|[tT]his|[iI]ts|[iI]nstead\s\s?of)((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\
s*(([sS]tart|[bB]egin|[Ss]tart|[Bb]egin|[Ee]nd|[eE]nd|[Mm]idth|[mM]idth)((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*([tT]he|[tT]his|
[tT]hese|[tT]hose|[iI]ts))?))?((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*[[a-z]]+)?(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*((((((lL]ast|[p
P]receding|[pP]ast|[cC]urrent|[tT]his|[uU]pcoming|[fF]ollowing|[sS]ucceeding|[nN]ext))((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*((
(1|2|3|4|5|6|7|8|9)\d?|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]our|[fF]ive|[sS]ix|[sS]even|[eE]ight|[nN]ine|
[tT]en|[eE]leven|[tT]welve|[tT]wenty|[tT]hirty|[fF]ourty|[fF]orty|[fF]ifty|[sS]ixty|[sS]eventy|[eE]ighty|[nN]inety|[hH]undr
ed|[aA]\s\s?hundred))|((1[012]?|2|3|4|5|6|7|8|9)(\.|())|([fF]irst|[sS]econd|[tT]hird|[fF]ourth|[fF]ifth|[sS]ixth|[sS]eventh
|[eE]ighth|[nN]inth|[tT]enth|[eE]leventh))(-((1[012]?|2|3|4|5|6|7|8|9)(\.|())|([fF]irst|[sS]econd|[tT]hird|[fF]ourth|[fF]if
th|[sS]ixth|[sS]eventh|[eE]ighth|[nN]inth|[tT]enth|[eE]leventh)))?((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*((1|2|3|4|5|6|7|8|9)\d?
|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]our|[fF]ive|[sS]ix|[sS]even|[eE]ight|[nN]ine|[tT]en|[eE]leven|[tT]w
elve|[tT]wenty|[tT]hirty|[fF]ourty|[fF]orty|[fF]ifty|[sS]ixty|[sS]eventy|[eE]ighty|[nN]inety|[hH]undred|[aA]\s\s?hundred)))
?))?(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)|((((1|2|3|4|5|6|7|8|9)\d?|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]our
|[fF]ive|[sS]ix|[sS]even|[eE]ight|[nN]ine|[tT]en|[eE]leven|[tT]welve|[tT]wenty|[tT]hirty|[fF]ourty|[fF]orty|[fF]ifty|[sS]ix
ty|[sS]eventy|[eE]ighty|[nN]inety|[hH]undred|[aA]\s\s?hundred))|((1[012]?|2|3|4|5|6|7|8|9)(\.|())|([fF]irst|[sS]econd|[tT]h
ird|[fF]ourth|[fF]ifth|[sS]ixth|[sS]eventh|[eE]ighth|[nN]inth|[tT]enth|[eE]leventh))(-((1[012]?|2|3|4|5|6|7|8|9)(\.|())|([f
F]irst|[sS]econd|[tT]hird|[fF]ourth|[fF]ifth|[sS]ixth|[sS]eventh|[eE]ighth|[nN]inth|[tT]enth|[eE]leventh)))?((\s+(\r(\n)?|\
n)?|(\r(\n)?|\n))\s*((1|2|3|4|5|6|7|8|9)\d?|([oO]ne|[sS]everal|[sS]ome|[bB]oth|[tT]wo|[tT]hree|[fF]our|[fF]ive|[sS]ix|[sS]e
ven|[eE]ight|[nN]ine|[tT]en|[eE]leven|[tT]welve|[tT]wenty|[tT]hirty|[fF]ourty|[fF]orty|[fF]ifty|[sS]ixty|[sS]eventy|[eE]igh
ty|[nN]inety|[hH]undred|[aA]\s\s?hundred)))?)((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*(([lL]ast|[pP]receding|[pP]ast|[cC]urrent|[t
T]his|[uU]pcoming|[fF]ollowing|[sS]ucceeding|[nN]ext)))?(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*))?(((Q(1|2|3|4)|H(1|2)(\/(19|20)?
\d2)?|(((\w([a-z])*(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)?(year|quarter))([a-z])*))|((month|time(span)?(\s+(\r(\n)?|\n)?|(\r(\n
)?|\n))\s*(from(\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*)?)?([Jj]anuary|[Jj]an\.|[Jj]an|[Ff]ebruary|[Ff]eb\.|[Ff]eb|[Mm]arch|[Mm]ar
\.|[Mm]ar|[Aa]pril|[Aa]pr\.|[Aa]pr|[Mm]ay|[Jj]une|[Jj]un\.|[Jj]un|[Jj]uly|[Jj]ul\.|[Jj]ul|[Aa]ugust|[Aa]ug\.|[Aa]ug|[Ss]ept
ember|[Ss]ep\.|[Ss]ep|[Oo]ctober|[Oo]ct\.|[Oo]ct|[Nn]ovember|[Nn]ov\.|[Nn]ov|[Dd]ecember|[Dd]ez\.|[Dd]ez|[Ss]pring|[Ss]umme
r|[Aa]utumn|[Ff]all|[Ww]inter))|(([Rr]eported\s\s?time\s\s?span|[Rr]eported\s\s?time\s\s?span|[Rr]eported\s\s?time|[rR]eport
ed\s\s?time|[Tt]ime\s\s?span|[tT]ime\s\s?span|[Ss]pan|[sS]pan|[Dd]ecade|[dD]ecade)))((\s+(\r(\n)?|\n)?|(\r(\n)?|\n))\s*((19|
20)\d2(/(19|20)?\d2)?|\d2/\d2))?|((19|20)\d2(/(19|20)?\d2)?|\d2/\d2))))*))
```

Time Expression Recognition with Regular Expressions

Complete Regex as a Finite-State Automaton *





Time Expression Recognition with Regular Expressions

Top-level FSA of Complete Regex *

Notice

- Bold edge labels indicate sub-FSAs, regular ones indicate lexicons.
- Below, the FSA of period expressions is decomposed top-down.
  The regex for date expressions is left out for brevity.
- During development, building a regex usually rather works bottom-up.

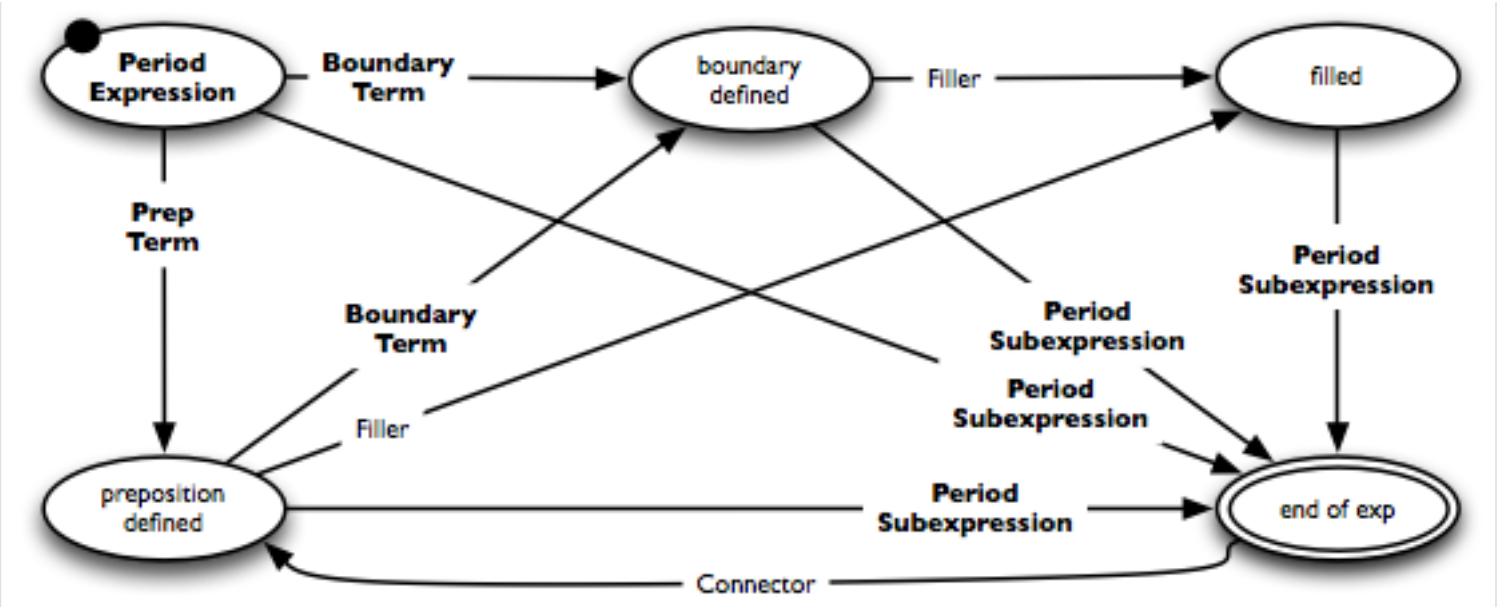
Example

- “From the very end of last year to the 2nd half of 2019”

  prep filler boundary relational period connector ordinal period year

Time Expression Recognition with Regular Expressions

Sub-FSA for Period Expressions *

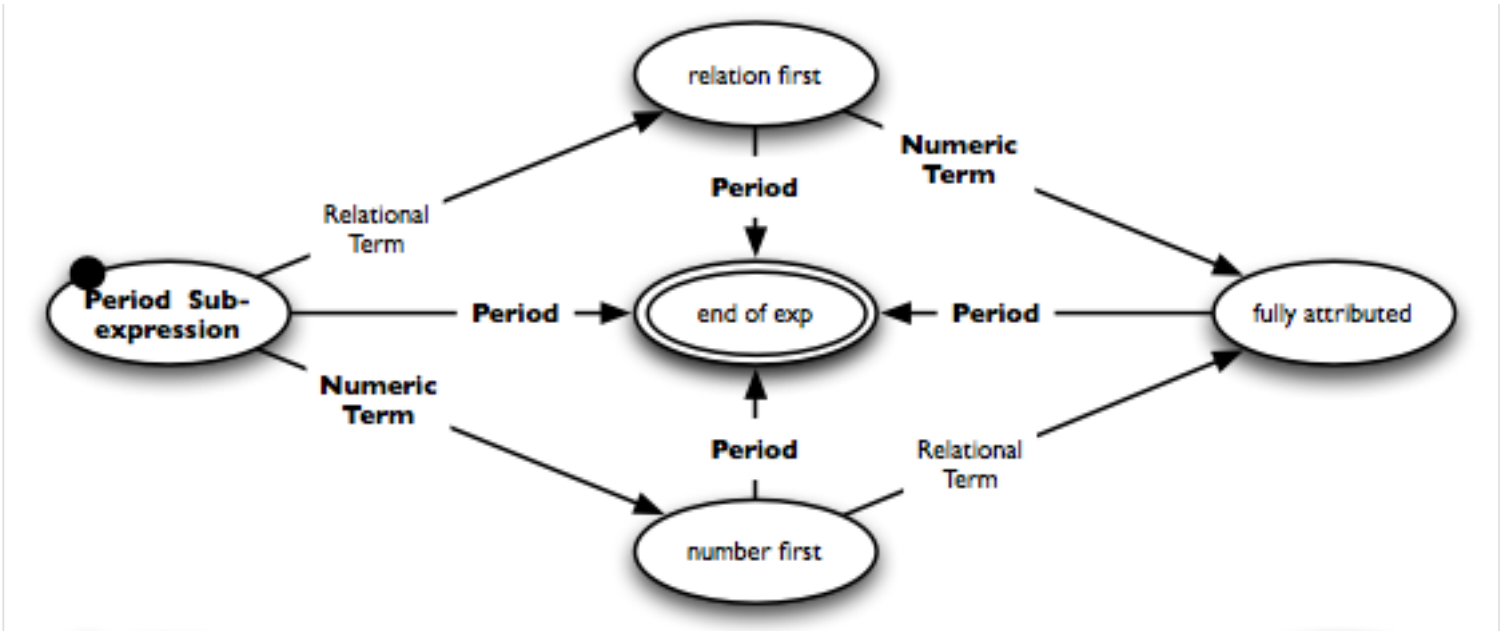


Lexicons

- Connector lexicon. “to the”, “to”, “and”, “of the”, “of”, ...
- Fillers. *Any* single word, such as “very” in the example above.

Time Expression Recognition with Regular Expressions

Sub-FSA for Period Subexpressions *

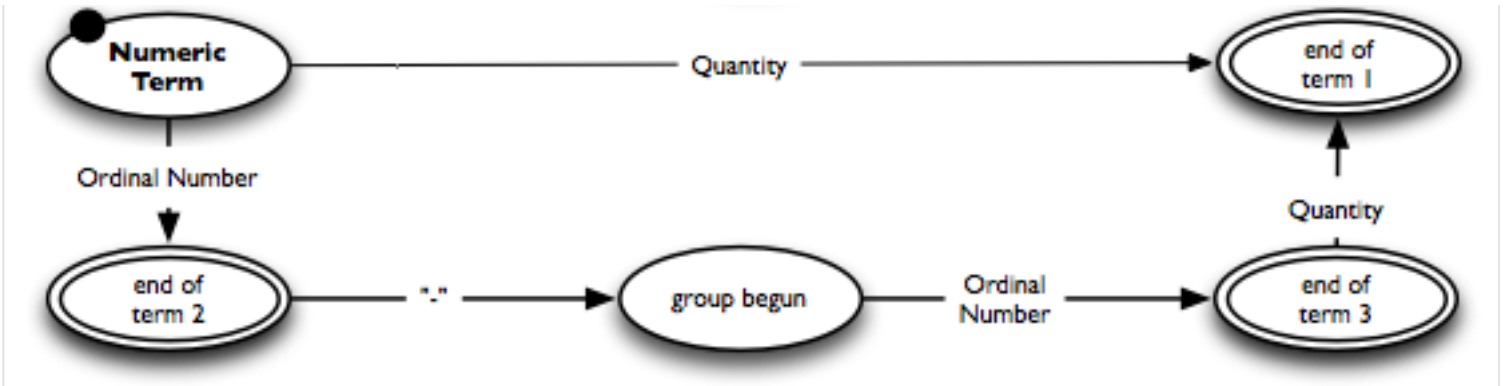


Lexicons

- Relational term lexicon. “last”, “preceding”, “past”, “current”, “this”, “upcoming”, “next”, ...

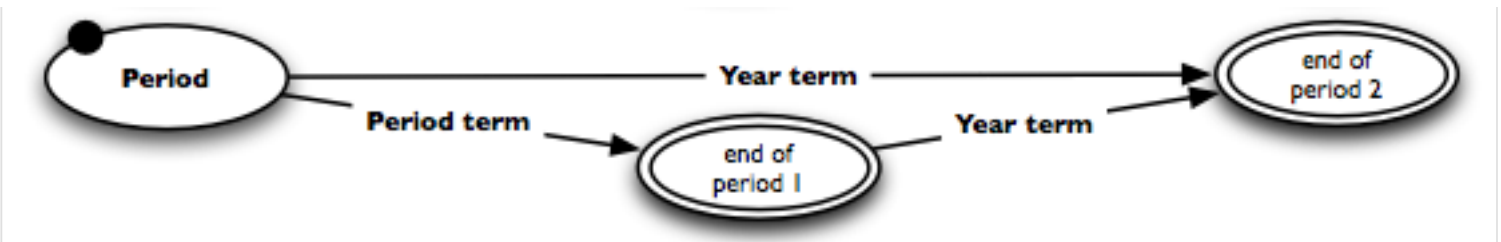
Time Expression Recognition with Regular Expressions

Sub-FSAs for Numeric Terms and Periods *



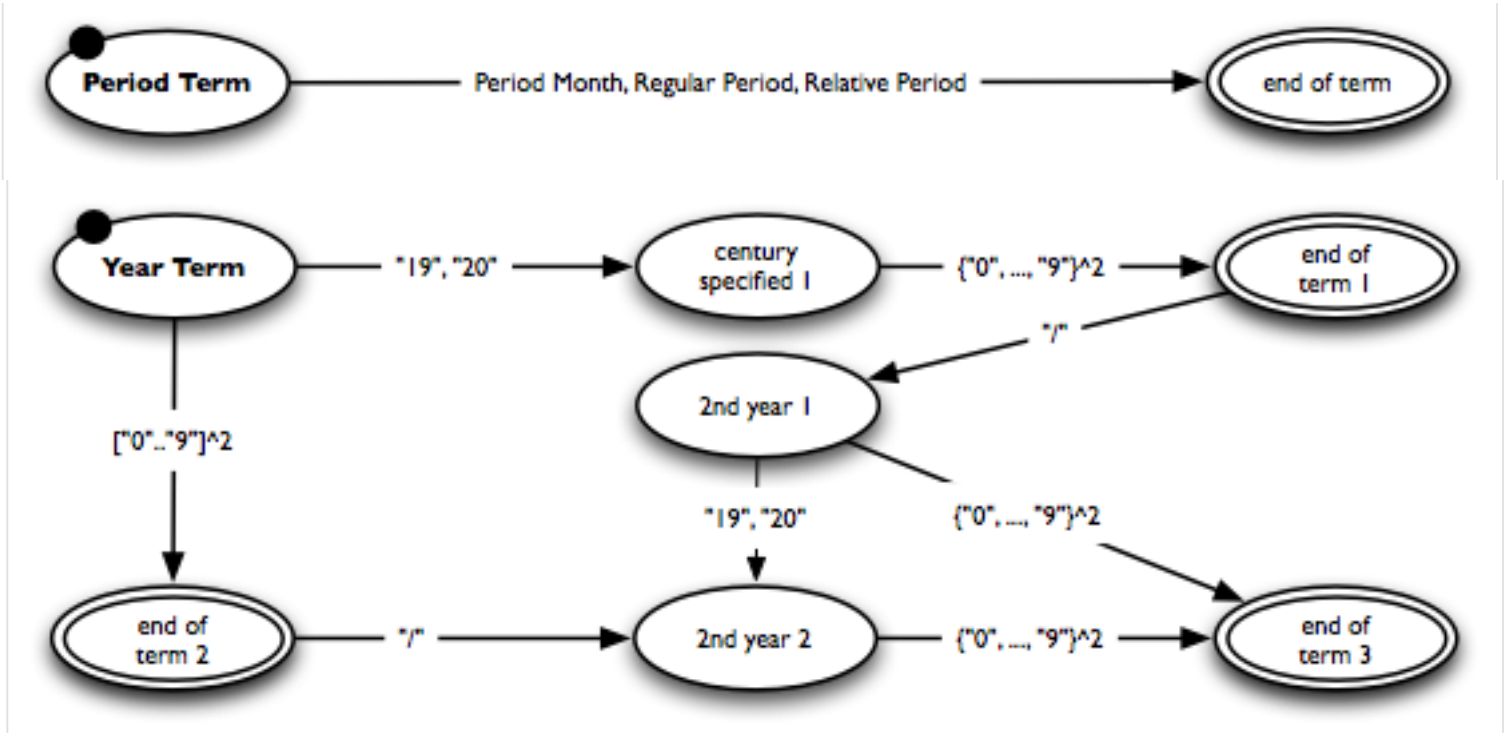
Lexicons

- Ordinal number lexicon. “first”, “1st”, “second”, “2nd”, “third”, “3rd”, ...
- Quantity lexicon. “one”, “two”, “three”, “both”, “several”, “a hundred”, ...



Time Expression Recognition with Regular Expressions

Sub-FSAs for Period Terms and Year Terms *

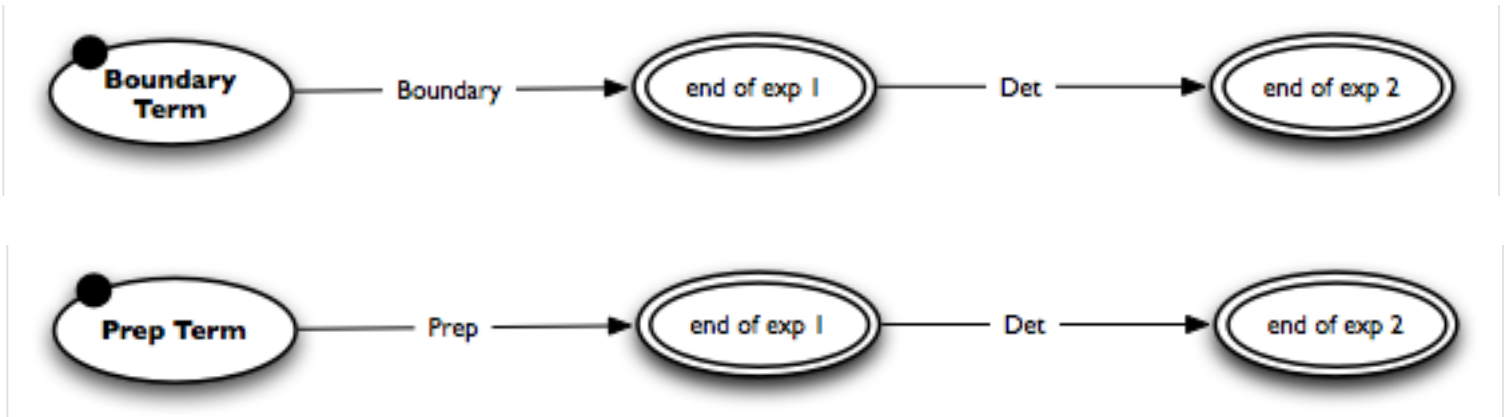


Lexicon

- Period month lexicon. “March”, “Mar.”, “Mar”, “Fall”, “fall”, “Autumn”, ...
- Regular period lexicon. “year”, “month”, “quarter”, “half”, ...
- Relative period lexicon. “decade”, “reported time”, “time span”, ...

Time Expression Recognition with Regular Expressions

Sub-FSAs for Boundary Terms and Prepositional Terms *



Lexicons

- Boundary lexicon. “Beginning”, “beginning”, “End”, “end”, “Midth”, ...
- Prep lexicon. “in”, “within”, “to”, “for”, “from”, “since”, ...
- Det lexicon. “the”, “a”, “an”

Time Expression Recognition with Regular Expressions

Evaluation

How well does the regex perform?

- Originally developed for German texts; only this version was evaluated.
- Data. Test set of the *InfexBA Revenue corpus* with 6038 sentences from business news articles.
- Evaluation measures. Precision, recall, F_1-score, run-time per sentence.
  Run-time measured on a standard computer from 2009.

Results

Approach	Precision	Recall	F_1-score	ms/sentence
Regex	0.91	0.97	0.94	0.36

Conclusion

- Regexes for semi-closed-class entity types such as time expressions can achieve very high effectiveness and efficiency.
- Their development is complex and time-intensive, though.

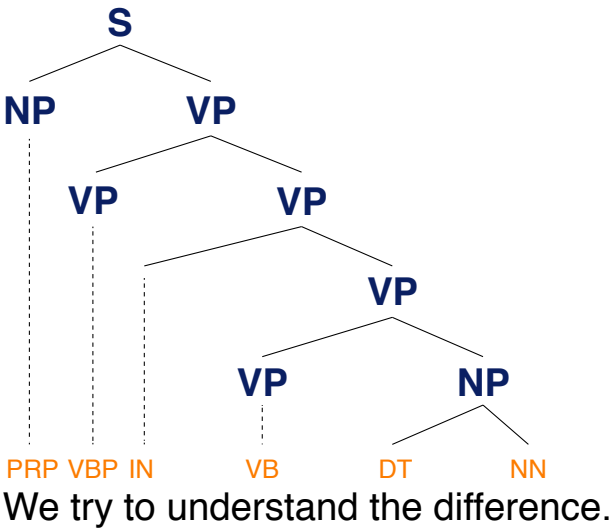
Probabilistic Context-Free Grammars

Grammars

Phrase vs. Dependency Structure (Recap)

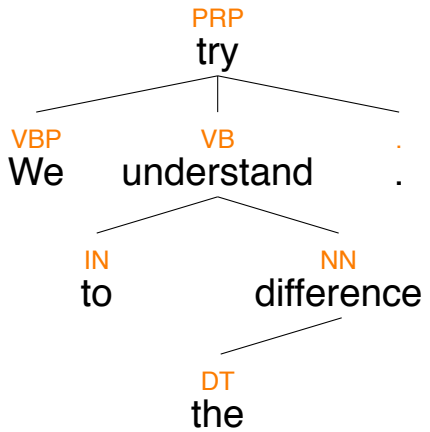
Phrase structure grammar

- Models the constituents of a sentence and how they are composed of each other.
- Constituency (parse) tree. Inner nodes are non-terminals, leafs terminals.



Dependency grammar

- Models the dependencies between the words in a sentence.
- Dependency (parse) tree. All nodes are terminals, the root is nearly always the main verb (of the first main clause).



Context-Free Grammars

What is a phrase structure grammar?

- A phrase structure grammar is a context-free grammar (CFG).
- A grammar (Σ, N, S, R) is *context-free* if all rules in R are of the form $U \to V$ with $U \in N$ and $V \in (N \cup \Sigma)^*$.
- A language is context-free, if there is a CFG that defines it.

Phrase structure interpretation of non-terminals $N = N_{phr} \cup N_{pos}$

N_{phr} Phrase types. A finite set of structural non-terminal symbols.

N_{pos} Part-of-speech tags. A finite set of lexical "pre-terminal" symbols.

$N_{phr} \cap N_{pos} = \emptyset$.

Phrase structure interpretation of rules $R = R_{phr} \cup R_{pos}$

R_{phr} A finite set of structure rules of the form $U \to V$ with $U \in N_{phr}$ and $V \in (N_{phr} \cup N_{pos})^*$.

R_{pos} A finite set of lexicon rules of the form $U \to v$ with $U \in N_{pos}$ and $v \in \Sigma$.

In addition to S, CFGs in NLP usually include an extra node ROOT at the top.

Context-Free Grammars

Example

Example CFG, represented by its rules

Structural rules			Lexical rules	
s1	$S \rightarrow NP\ VP$		l1	$N \rightarrow$ people
s2	$VP \rightarrow V\ NP$		l2	$N \rightarrow$ fish
s3	$VP \rightarrow V\ NP\ PP$		l3	$N \rightarrow$ tanks
s4	$NP \rightarrow NP\ NP$		l4	$N \rightarrow$ rods
s5	$NP \rightarrow NP\ PP$	// binary	l5	$V \rightarrow$ people
s6	$NP \rightarrow N$	// unary	l6	$V \rightarrow$ fish
s7	$NP \rightarrow \varepsilon$	// empty	l7	$V \rightarrow$ tanks
s8	$PP \rightarrow P\ NP$		l8	$P \rightarrow$ with

Example sentences created by the grammar

- “people fish tanks”
- “people fish with rods”

Context-Free Grammars

Chomsky Normal Form

Chomsky Normal Form

- A CFG is in Chomsky Normal Form if all rules in R are of the forms $U \to VW$ and $U \to v$, where $U, V, W \in N$ and $v \in \Sigma^*$.

Tansformation into normal form

- Cleaning. Empties and unaries are removed recursively.
- Binarization. n-ary rules are divided by using new non-terminals, $n > 2$.
- These transformations do not change the language defined by a CFG, but they may result in different parse trees.

Why transforming?

- Restricting a CFG in such a way is key to efficient parsing.
- Binarization is crucial for cubic time.
- Cleaning is not mandatory, but makes parsing quicker and cleaner.

Chomsky Normal Form

Pseudocode

Signature

- Input. The production rules R of a CFG.
- Output. The production rules R^* of the normalized version of the CFG.

toChomskyNormalForm(Production rules R)

```
1.  while an empty (U → ε) ∈ R do
2.      R ← R \ {U → ε}
3.      for each rule (V → V_1 ... V_k U W_1 ... W_l) ∈ R do // k, l ≥ 0
4.          R ← R ∪ {V → V_1 ... V_k W_1 ... W_l}
5.  while a unary (U → V) ∈ R do
6.      R ← R \ {U → V}
7.      if U ≠ V then
8.          for each (V → V_1 ... V_k) ∈ R do R ← R ∪ {U → V_1 ... V_k}
9.          if not (W → V_1 ... V_k V W_1 ... W_l) ∈ R then
10.             for each (V → V_1 ... V_k) ∈ R do R ← R \ {V → V_1 ... V_k}
11. while an n-ary (U → V_1 ... V_n) ∈ R do // n ≥ 3
12.     R ← (R \ {U → V_1 ... V_n}) ∪ {U → V_1 U_V_1, U_V_1 → V_2 ... V_n}
13. return R
```

Chomsky Normal Form

Example: Empties (Removal)

Structural rules		Lexical rules	
s1	S $\rightarrow$ NP VP	l1	N $\rightarrow$ people
s2	VP $\rightarrow$ V NP	l2	N $\rightarrow$ fish
s3	VP $\rightarrow$ V NP PP	l3	N $\rightarrow$ tanks
s4	NP $\rightarrow$ NP NP	l4	N $\rightarrow$ rods
s5	NP $\rightarrow$ NP PP	l5	V $\rightarrow$ people
s6	NP $\rightarrow$ N	l6	V $\rightarrow$ fish
s7	NP $\rightarrow \varepsilon$	l7	V $\rightarrow$ tanks
s8	PP $\rightarrow$ P NP	l8	P $\rightarrow$ with

Removal of empties

- Add new rules for each rule where NP occurs on the right side.

  Pseudocode lines 2–4.

Chomsky Normal Form

Example: Empties (Addition)

Structural rules		Lexical rules	
s1	S → NP VP	l1	N → people
s1'	S → VP	l2	N → fish
s2	VP → V NP	l3	N → tanks
s2'	VP → V	l4	N → rods
s3	VP → V NP PP	l5	V → people
s3'	VP → V PP	l6	V → fish
s4	NP → NP NP	l7	V → tanks
s4'	NP → NP	l8	P → with
s5	NP → NP PP		
s5'	NP → PP		
s6	NP → N		
s8	PP → P NP		
s8'	PP → P		

Chomsky Normal Form

Example: Unaries (Removal)

Structural rules		Lexical rules	
s1	S → NP VP	l1	N → people
s1'	S → VP	l2	N → fish
s2	VP → V NP	l3	N → tanks
s2'	VP → V	l4	N → rods
s3	VP → V NP PP	l5	V → people
s3'	VP → V PP	l6	V → fish
s4	NP → NP NP	l7	V → tanks
s4'	NP → NP	l8	P → with
s5	NP → NP PP		
s5'	NP → PP		
s6	NP → N		
s8	PP → P NP		
s8'	PP → P		

Chomsky Normal Form

Example: Unaries (Addition)

Structural rules		Lexical rules	
s1	S → NP VP	l1	N → people
s2	VP → V NP	l2	N → fish
s2”	S → V NP	l3	N → tanks
s2’	VP → V	l4	N → rods
s2”’	S → V	l5	V → people
s3	VP → V NP PP	l6	V → fish
s3”	S → V NP PP	l7	V → tanks
s3’	VP → V PP	l8	P → with
s3”’	S → V PP		
s4	NP → NP NP		
s4’	NP → NP		
s5	NP → NP PP		
s5’	NP → PP		
s6	NP → N		
s8	PP → P NP		
s8’	PP → P		

Chomsky Normal Form

Example: Unaries 2 (Removal)

Structural rules		Lexical rules	
s1	S → NP VP	l1	N → people
s2	VP → V NP	l2	N → fish
s2”	S → V NP	l3	N → tanks
s2’	VP → V	l4	N → rods
s2”’	S → V	l5	V → people
s3	VP → V NP PP	l6	V → fish
s3”	S → V NP PP	l7	V → tanks
s3’	VP → V PP	l8	P → with
s3”’	S → V PP		
s4	NP → NP NP		
s4’	NP → NP		
s5	NP → NP PP		
s5’	NP → PP		
s6	NP → N		
s8	PP → P NP		
s8’	PP → P		

Chomsky Normal Form

Example: Unaries 2 (Addition)

Structural rules		Lexical rules	
s1	S → NP VP	l1	N → people
s2	VP → V NP	l2	N → fish
s2”	S → V NP	l3	N → tanks
s2”’	S → V	l4	N → rods
s3	VP → V NP PP	l5	V → people
s3”	S → V NP PP	l5’	VP → people
s3’	VP → V PP	l6	V → fish
s3”’	S → V PP	l6’	VP → fish
s4	NP → NP NP	l7	V → tanks
s4’	NP → NP	l7’	VP → tanks
s5	NP → NP PP	l8	P → with
s5’	NP → PP		
s6	NP → N		
s8	PP → P NP		
s8’	PP → P		

Chomsky Normal Form

Example: Unaries 3 (Removal)

Structural rules		Lexical rules	
s1	S → NP VP	l1	N → people
s2	VP → V NP	l2	N → fish
s2”	S → V NP	l3	N → tanks
s2”’	S → V	l4	N → rods
s3	VP → V NP PP	l5	V → people
s3”	S → V NP PP	l5’	VP → people
s3’	VP → V PP	l6	V → fish
s3”’	S → V PP	l6’	VP → fish
s4	NP → NP NP	l7	V → tanks
s4’	NP → NP	l7’	VP → tanks
s5	NP → NP PP	l8	P → with
s5’	NP → PP		
s6	NP → N		
s8	PP → P NP		
s8’	PP → P		

Chomsky Normal Form

Example: Unaries 3 (Addition)

Structural rules		Lexical rules	
s1	S → NP VP	l1	N → people
s2	VP → V NP	l2	N → fish
s2”	S → V NP	l3	N → tanks
s3	VP → V NP PP	l4	N → rods
s3”	S → V NP PP	l5	V → people
s3’	VP → V PP	l5’	VP → people
s3”’	S → V PP	l5”	S → people
s4	NP → NP NP	l6	V → fish
s4’	NP → NP	l6’	VP → fish
s5	NP → NP PP	l6”	S → fish
s5’	NP → PP	l7	V → tanks
s6	NP → N	l7’	VP → tanks
s8	PP → P NP	l7”	S → tanks
s8’	PP → P	l8	P → with

Chomsky Normal Form

Example: Unaries 4–7 (Removal)

Structural rules		Lexical rules	
s1	S → NP VP	l1	N → people
s2	VP → V NP	l2	N → fish
s2”	S → V NP	l3	N → tanks
s3	VP → V NP PP	l4	N → rods
s3”	S → V NP PP	l5	V → people
s3’	VP → V PP	l5’	VP → people
s3”’	S → V PP	l5”	S → people
s4	NP → NP NP	l6	V → fish
s4’	NP → NP	l6’	VP → fish
s5	NP → NP PP	l6”	S → fish
s5’	NP → PP	l7	V → tanks
s6	NP → N	l7’	VP → tanks
s8	PP → P NP	l7”	S → tanks
s8’	PP → P	l8	P → with

Chomsky Normal Form

Example: Unaries 4–7 (Addition)

Structural rules		Lexical rules	
s1	S → NP VP	l1	NP → people
s2	VP → V NP	l2	NP → fish
s2”	S → V NP	l3	NP → tanks
s3	VP → V NP PP	l4	NP → rods
s3”	S → V NP PP	l5	V → people
s3’	VP → V PP	l5’	VP → people
s3”’	S → V PP	l5”	S → people
s4	NP → NP NP	l6	V → fish
s5	NP → NP PP	l6’	VP → fish
s5”	NP → P NP	l6”	S → fish
s8	PP → P NP	l7	V → tanks
		l7’	VP → tanks
		l7”	S → tanks
		l8	P → with
		l8’	PP → with
		l8”	NP → with

Chomsky Normal Form

Example: n-aries 1–2 (Removal)

Structural rules		Lexical rules	
s1	S → NP VP	l1	NP → people
s2	VP → V NP	l2	NP → fish
s2”	S → V NP	l3	NP → tanks
s3	VP → V NP PP	l4	NP → rods
s3”	S → V NP PP	l5	V → people
s3’	VP → V PP	l5’	VP → people
s3”’	S → V PP	l5”	S → people
s4	NP → NP NP	l6	V → fish
s5	NP → NP PP	l6’	VP → fish
s5”	NP → P NP	l6”	S → fish
s8	PP → P NP	l7	V → tanks
		l7’	VP → tanks
		l7”	S → tanks
		l8	P → with
		l8’	PP → with
		l8”	NP → with

Chomsky Normal Form

Example: n-aries 1–2 (Addition) $\rightarrow$ Results in Chomsky normal form!

Structural rules		Lexical rules	
s1	S $\rightarrow$ NP VP	l1	NP $\rightarrow$ people
s2	VP $\rightarrow$ V NP	l2	NP $\rightarrow$ fish
s2”	S $\rightarrow$ V NP	l3	NP $\rightarrow$ tanks
s3””	VP $\rightarrow$ V VP_V	l4	NP $\rightarrow$ rods
s3”””	VP_V $\rightarrow$ NP PP	l5	V $\rightarrow$ people
s3””””	S $\rightarrow$ V S_V	l5’	VP $\rightarrow$ people
s3”””””	S_V $\rightarrow$ NP PP	l5”	S $\rightarrow$ people
s3’	VP $\rightarrow$ V PP	l6	V $\rightarrow$ fish
s3”’	S $\rightarrow$ V PP	l6’	VP $\rightarrow$ fish
s4	NP $\rightarrow$ NP NP	l6”	S $\rightarrow$ fish
s5	NP $\rightarrow$ NP PP	l7	V $\rightarrow$ tanks
s5”	NP $\rightarrow$ P NP	l7’	VP $\rightarrow$ tanks
s8	PP $\rightarrow$ P NP	l7”	S $\rightarrow$ tanks
		l8	P $\rightarrow$ with
		l8’	PP $\rightarrow$ with
		l8”	NP $\rightarrow$ with

Probabilistic Context-Free Grammars

What is a probabilistic context-free grammar (PCFG)?

- A CFG where each production rule is assigned a probablility.

PCFG (Σ, N, S, R, P)

P A probability function $R \to [0, 1]$ from production rules to probabilities, such that

$$\forall U \in N : \sum_{(U \to V) \in R} P(U \to V) = 1$$

(Σ, N, S, R as before)

Probabilities

- Trees. The probability $P(t)$ of a tree t is the product of the probabilities of the rules used to generate it.
- Strings. The probability $P(s)$ of a string s is the sum of the probabilities of the trees which yield s.

Probabilistic Context-Free Grammars

Example

Example PCFG

Structural rules			Lexical rules		
s1	S → NP VP	1.0	l1	N → people	0.5
s2	VP → V NP	0.6	l2	N → fish	0.2
s3	VP → V NP PP	0.4	l3	N → tanks	0.2
s4	NP → NP NP	0.1	l4	N → rods	0.1
s5	NP → NP PP	0.2	l5	V → people	0.1
s6	NP → N	0.7	l6	V → fish	0.6
s7	PP → P NP	1.0	l7	V → tanks	0.3
			l8	P → with	1.0

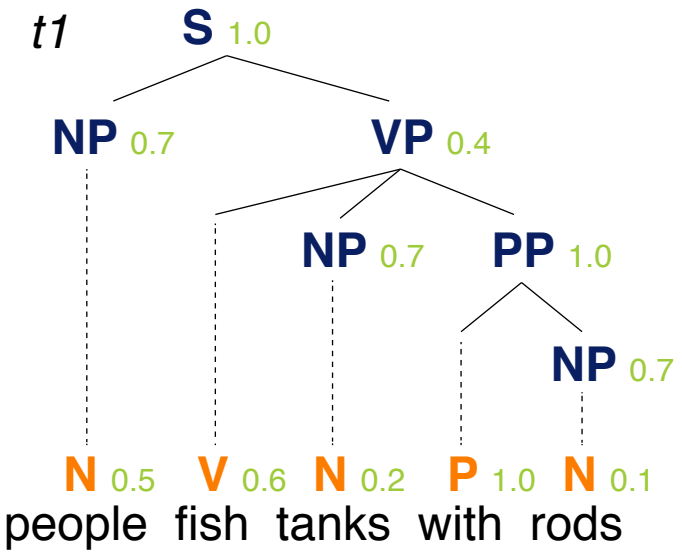
Notice

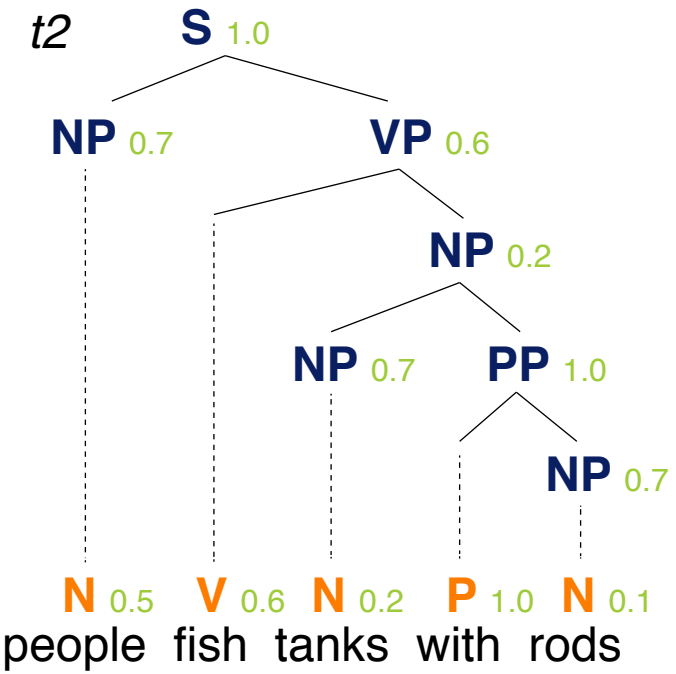
- For parsing, a PCFG should be transformed to Chomsky Normal Form or at least binarized.
- The origin of the probabilities is clarified below.

Probabilistic Context-Free Grammars

Example Probabilities

$s =$ “people fish tanks with rods”





Probabilities

$$
\begin{aligned}
P(t_1) &= 1.0 \cdot 0.7 \cdot 0.4 \cdot 0.5 \cdot 0.6 \cdot 0.7 \cdot 1.0 \cdot 0.2 \cdot 1.0 \cdot 0.7 \cdot 0.1 &&= 0.0008232 \\
P(t_2) &= 1.0 \cdot 0.7 \cdot 0.6 \cdot 0.5 \cdot 0.6 \cdot 0.2 \cdot 0.7 \cdot 1.0 \cdot 0.2 \cdot 1.0 \cdot 0.7 \cdot 0.1 &&= 0.00024696 \\
P(s) &= P(t_1) + P(t_2) &&= 0.0008232 + 0.00024696 \quad = 0.00107016
\end{aligned}
$$

Parsing based on a PCFG

Constituency Parsing

What is constituency parsing?

- The text analysis that determines the phrase structure of a sentence with respect to a given grammar.
- Often used in text mining as preprocessing where syntax is important.
- Parsing works robust across domains of well-formatted texts.

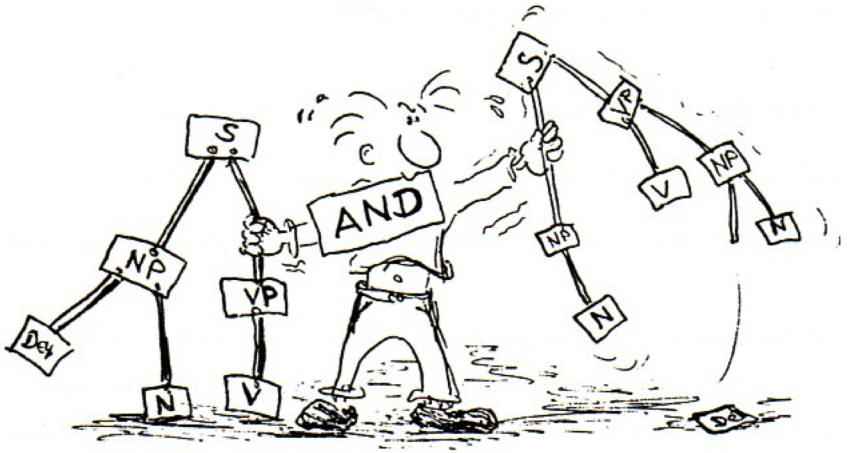
Downstream tasks based on parsing

- Named entity recognition in complex domains (e.g., biology)
- Relationship extraction, both for semantic and temporal relations
- Sentence compression with syntactic consistency
- Opinion mining regarding aspects of products or similar
- Machine translation, to analyze the source sentence
- Question answering, particularly in high-precision scenarios

... and so forth

Constituency Parsing

Parsing before ∼ 1990



Classical parsing

- Hand-crafted CFG, along with a lexicon.
- Usage of CFG-based systems to prove parses from words.
- This scales badly and fails to give high language coverage.

Example “Fed raises interest rates 0.5% in effort to control inflation”

- Minimal grammar. 36 parses
- Real-size broad-coverage grammar. Millions of parses

Constituency Parsing

Classical Parsing: Problems and Solutions

Grammars with categorical constraints

- Limit the chance for unlikely parse trees of sentences.
- But constraints reduce the coverage of a grammar.
- In classical parsing, typically $\sim$30% of all sentences cannot be parsed.

Less constrained grammars

- Can parse more sentences.
- But simple sentences end up with even more parse trees.
- No way to choose between different parse trees.

Statistical parsing

- Very loose grammars that admit millions of parse trees for sentences.
- Mechanisms that find the most likely parse tree of a sentence quickly.
- Nowadays, most parsers are based on statistics (probabilities).

Constituency Parsing

Statistical Parsing

How to build a statistical parser?

- Statistical parsers are based on PCFGs (or varations thereof).
- The rules and probabilities of the PCFGs are derived from *treebanks*.

Treebanks

- A treebank is corpus with tree-structured annotations.

  One of the most used treebanks is the *Penn Treebank*, PTB (Marcus et al., 1993).
- Building a treebank is an expensive manual process done by experts.
- Slower than building a grammar, but the benefits outweigh the costs.

Benefits of treebanks

- Statistics. Frequencies and distributional information.
- Development. Reusable for many parsers, POS taggers, etc.
- Evaluation. Basis for evaluating a developed system.
- Language. Valuable resource for linguistics in general.

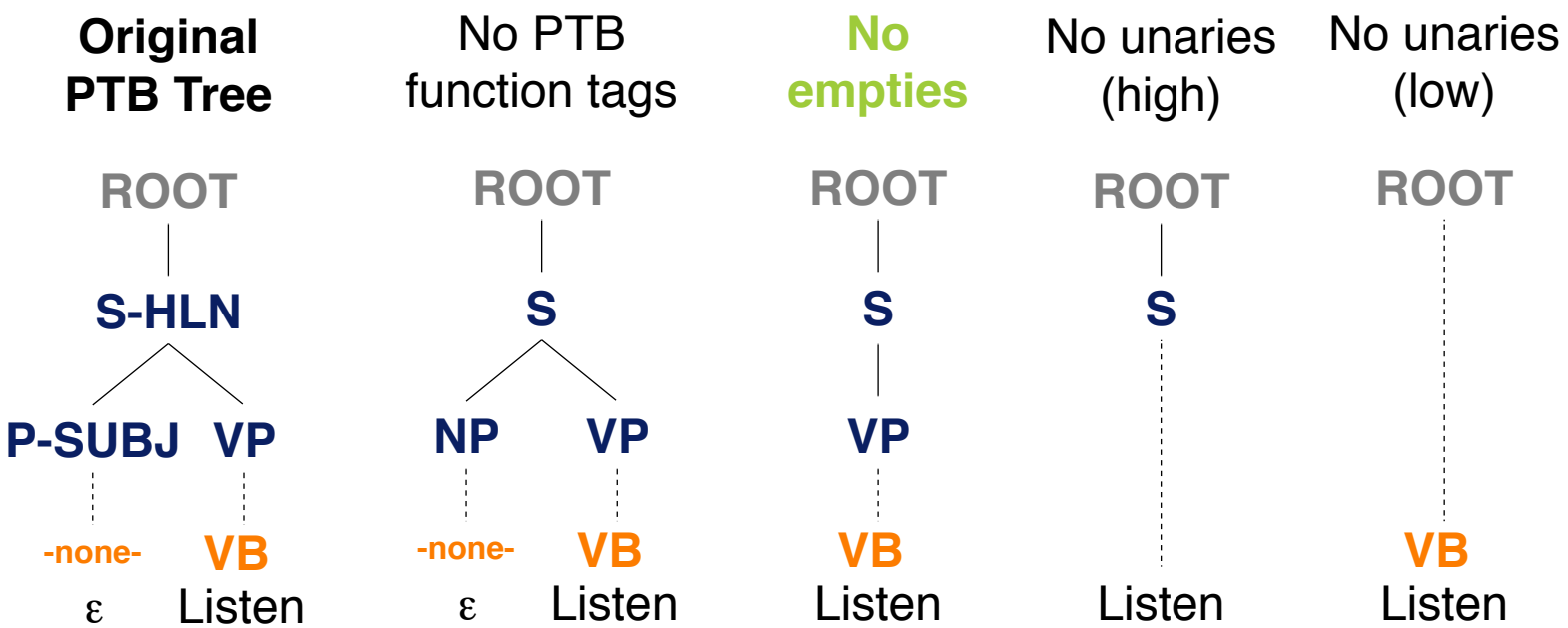
Constituency Parsing

Example PTB Sentence Representation

```
( (S
    (NP-SBJ (DT The) (NN move))
    (VP (VBD followed)
      (NP
        (NP (DT a) (NN round))
        (PP (IN of)
          (NP
            (NP (JJ similar) (NNS increases))
            (PP (IN by)
              (NP (JJ other) (NNS lenders)))
            (PP (IN against)
              (NP (NNP Arizona) (JJ real) (NN estate) (NNS loans))))))
      (, ,)
      (S-ADV
        (NP-SBJ (-NONE- *))
        (VP (VBG reflecting)
          (NP
            (NP (DT a) (VBG continuing) (NN decline))
            (PP-LOC (IN in)
              (NP (DT that) (NN market)))))))
    (. .)))
```

Constituency Parsing

From Treebank to Chomsky Normal Form



Observations

- No unaries. Low preferred over high, since it keeps lexical information.
- No empties. Enough for parsing and makes a reconstruction of the original parse tree easier.

Constituency Parsing

Attachment Ambiguity

Key parsing problem

- Correct attachment of the various constituents in a sentence, such as prepositional phrases, adverbial phrases, infinitives, coordinations, ...

"The board approved	its acquisition	→ attaches to "approved"
	by Royal Trustco Ltd.	→ attaches to "its acquisition"
	of Toronto	→ attaches to "by Royal Trustco Ltd."
	for $27 a share	→ attaches to "its acquisition"
	at its monthly meeting."	→ attaches to "approved ... for $27 a share"

How to find the correct attachment?

- Number of potential attachments grows exponentially to the number n of constituents according to *Catalan numbers*: $C_n = \frac{(2n)!}{(n+1)!\cdot n!}$
- The problem is *AI complete*.

"I saw the man with a telescope."

- Words predict attachment well.

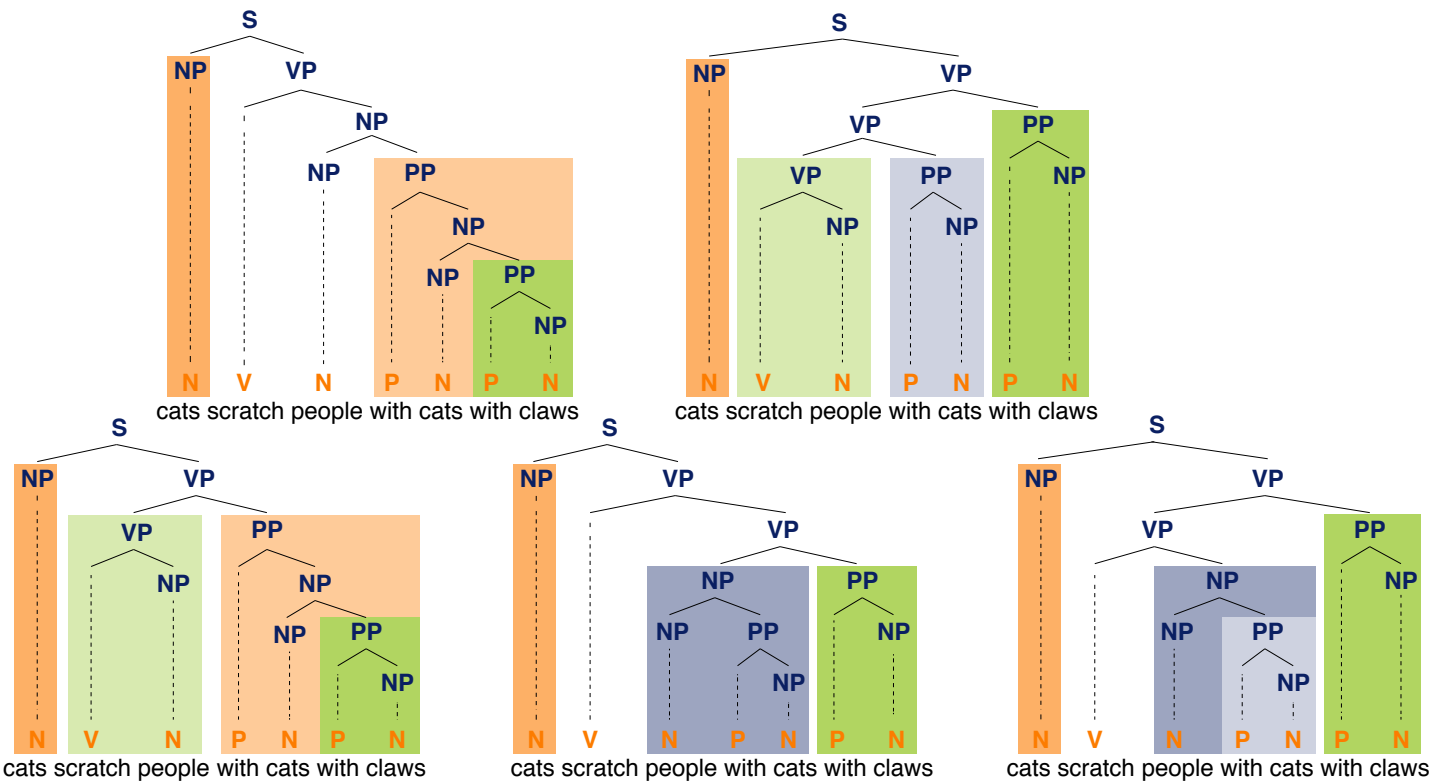
"Moscow sent more than 100,000 soldiers into Afghanistan."

Constituency Parsing

Attachment Ambiguity in Statistical Parsing

Two problems to solve in statistical parsing

1. Choose the most likely parse (according to statistics).
2. Avoid to do repeated work (algorithmically).



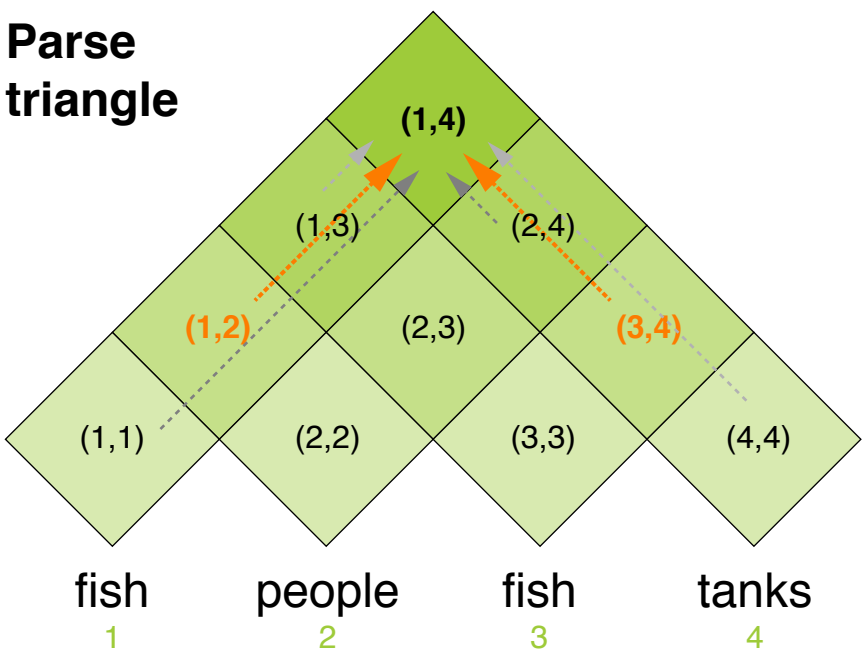
CKY Parsing

Cocke-Kasami-Younger (CKY) parsing (aka CYK parsing)

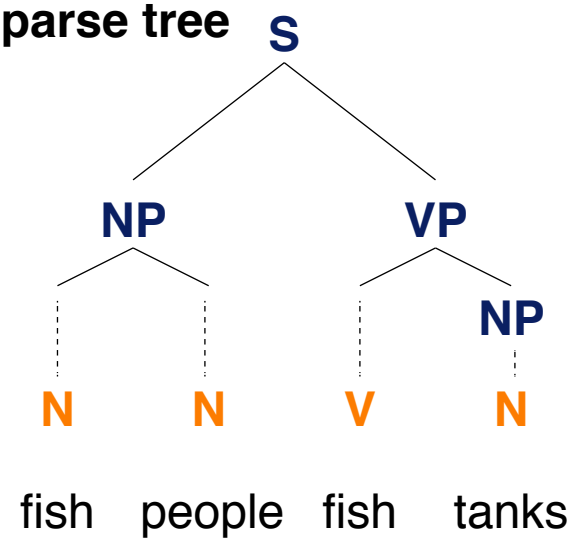
- A dynamic programming parsing algorithm from the 1960's.
- Goal. Get the most likely constituency parse tree for a sentence.
- Works with PCFGs in Chomsky Normal Form.
- Asympotically-strong: cubic time, quadratic space.

  With respect to the length of the sentence and the number of non-terminals.

Parse triangle



Most likely parse tree



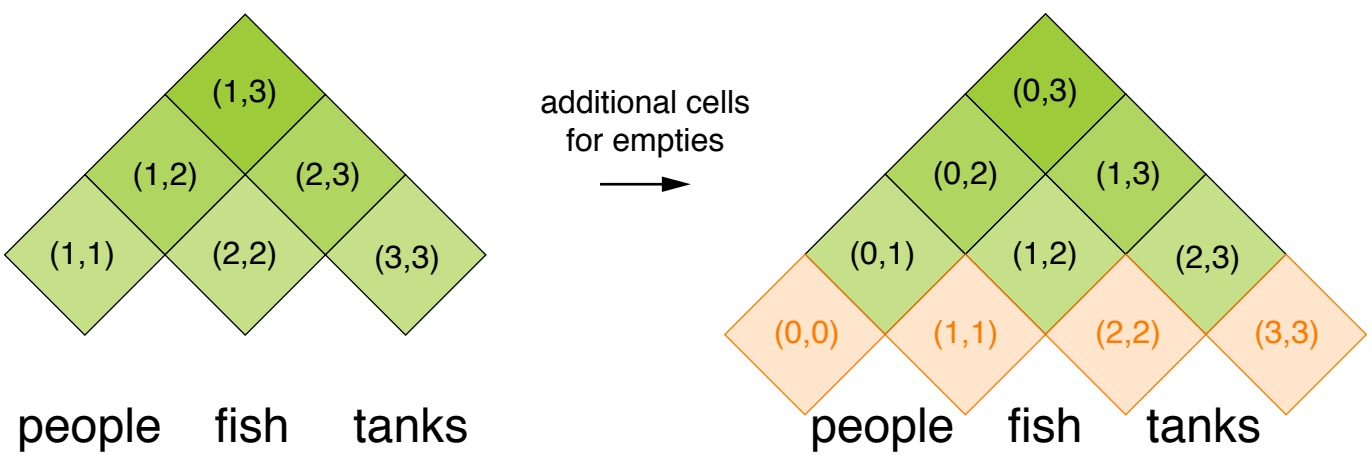
CKY Parsing

Extension

Extended CKY Parsing

- A parser for binarized PCFGs, introduced below.
- Includes unaries without increasing asymptotic complexity.

  Makes the algorithm more complicated, but keeps the grammar smaller.
- Empties are treated like unaries, after adding a cell row.



Binarization is needed for cubic time

- Without, CKY parsing does not work by concept.

  Other parsers may not require a binarized grammar, but then do binarization implicitly.

CKY Parsing

Pseudocode (1 out of 2)

Signature

- Input. A sentence (represented by a list of tokens), a binarized PCFG.
- Output. The most likely parse tree of the sentence.

extendedCKYParsing(List<Token> tokens, PCFG (Σ, N, S, R, P))

```
1.    double [][][] probs ← new double[#tokens][#tokens][#N]
2.    for int i ← 1 to #tokens do // Lexicon rules (and unaries)
3.       for each U ∈ N do
4.          if (U→tokens[i]) ∈ P then
5.             probs[i][i][U] ← P(U→tokens[i])
6.       boolean added ← 'true' // As of here: Handle unaries
7.       while added = 'true' do
8.          added ← 'false'
9.          for each U,V ∈ N do
10.            if probs[i][i][V]>0 and (U → V) ∈ P then
11.               double prob ← P(U → V) · probs[i][i][V]
12.               if prob > probs[i][i][U] then
13.                  probs[i][i][U] ← prob
14.                  added ← 'true'
15.   // ... continued on next slide...
```

CKY Parsing

Pseudocode (2 out of 2)

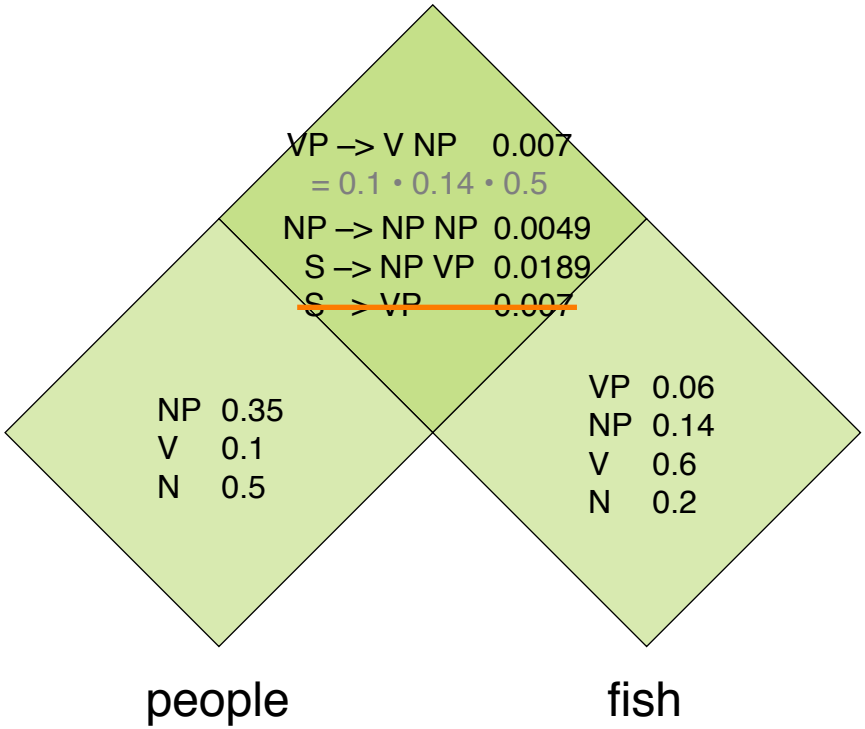
```
        // ... lines 1-14 on previous slide...
15.     for int length ← 2 to #tokens do // Structural rules
16.         for int beg ← 1 to #tokens - length + 1 do
17.             int end ← beg + length - 1
18.             for int split ← beg to end-1 do
19.                 for int U,V,W ∈ N do
20.                     int prob ← probs[beg][split][V] ·
                                   probs[split+1][end][W] · P(U → V W)
21.                     if prob > probs[beg][end][U] then
22.                         probs[beg][end][U] ← prob
23.             boolean added ← 'true' // As of here: Handle unaries
24.             while added do
25.                 added ← 'false'
26.                 for U,V ∈ N do
27.                     prob = P(U → V) · probs[beg][end][V];
28.                     if prob > probs[beg][end][U] then
29.                         probs[beg][end][U] ← prob
30.                         added ← 'true'
31.     return buildTree(probs) // Reconstruct tree from triangle
```

CKY Parsing

Example

A binarized PCFG

Structural rules		
s1	S → NP VP	0.9
s1'	S → VP	0.1
s2	VP → V NP	0.5
s2'	VP → V	0.1
s3'	VP → V VP_V	0.3
s3"	VP → V PP	0.1
s3"'	VP_V → NP PP	1.0
s4	NP → NP NP	0.1
s5	NP → NP PP	0.2
s6	NP → N	0.7
s7	PP → P NP	1.0



Filling cells

- Compute probabilities for each cell.
- Keep only highest for each left side.

CKY Parsing

Run-time Complexity

Run-time of pseudocode part 1

- $\mathcal{O}(n)$ times for-loop in lines 1–14, $n = \#$ tokens.
- $\mathcal{O}(|N|)$ times for-loop in lines 3–5.
- $\mathcal{O}(|N|^2)$ times while-loop in lines 7–14.

$\mathcal{O}(n \cdot |N|^2)$ for part 1 in total.

Run-time of pseudocode part 2

- $\mathcal{O}(n)$ times for-loop in lines 15–30.
- $\mathcal{O}(n)$ times for-loop in lines 16–30.
- $\mathcal{O}(n)$ times for-loop in lines 18–22.
- $\mathcal{O}(|N|^3)$ times for-loop in lines 19–22.
- $\mathcal{O}(|N|^2)$ times while-loop in lines 24–30.
- $\mathcal{O}(n^2)$ for building the tree in line 31.

$\mathcal{O}(n^3 \cdot |N|^3)$ for part 2 in total.

Overall run-time

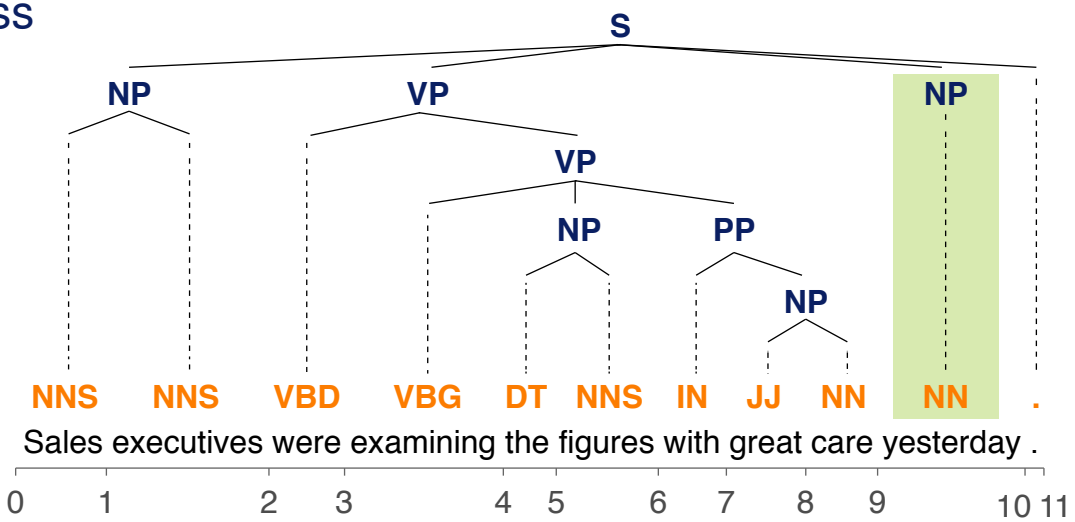
- Extended CKY parsing has a run-time of $\mathcal{O}(n^3 \cdot |N|^3)$.
- Several optimizations possible, but asymptotic complexity remains.

CKY Parsing

Evaluation of Effectiveness

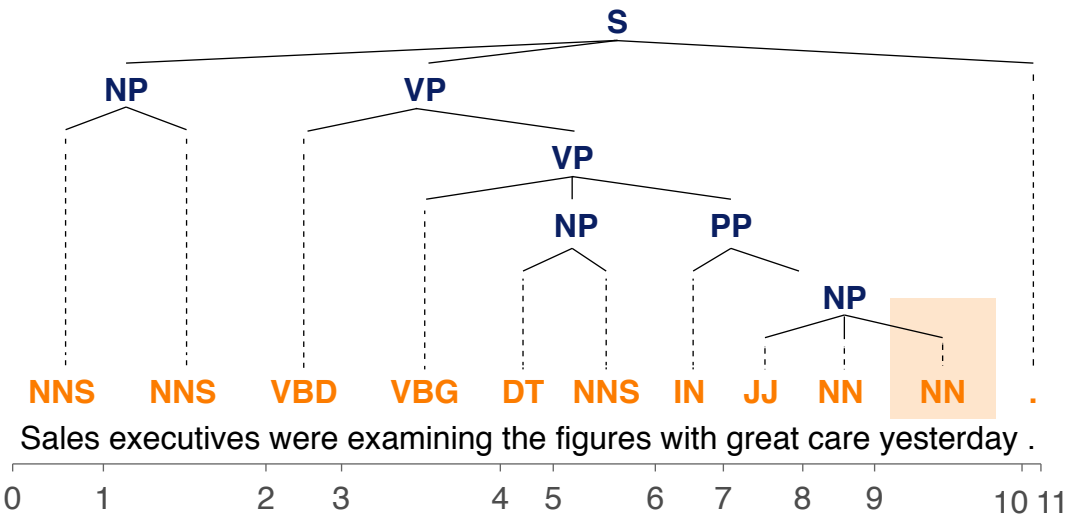
Gold standard brackets

S-(0:11), NP-(0:2),
VP-(2:9), VP-(3:9),
NP-(4:6), PP-(6:9),
NP-(7:9), NP-(9:10)



Candidate brackets

S-(0:11), NP-(0:2),
VP-(2:10), VP-(3:10),
NP-(4:6), PP-(6:10),
NP-(7:10)



CKY Parsing

Evaluation of Effectiveness

8 gold standard brackets

S-(0:11), NP-(0:2), VP-(2:9), VP-(3:9), NP-(4:6), PP-(6:9), NP-(7:9), NP-(9:10)

7 candidate brackets

S-(0:11), NP-(0:2), VP-(2:10), VP-(3:10), NP-(4:6), PP-(6:10), NP-(7:10)

Effectiveness in the example

- Labeled precision (LP). $0.429 = 3 / 7$
- Labeled recall (LR). $0.375 = 3 / 8$
- Labeled F_1-score. $0.400 = 2 \cdot \text{LP} \cdot \text{LR} / (\text{LP} + \text{LR})$
- POS tagging accuracy. $1.000 = 11 / 11$

Effectiveness of CKY in general (Charniak, 1997)

- Labeled F_1 $\sim$0.73 when trained and tested on Penn Treebank.
- CKY is robust, i.e., it parses almost anything (but with low probabilities).

Lexicalized Parsing

Limitations of standard PCFGs

- PCFGs assume that the plausibility of structures is independent of the words in a sentence, i.e., each rule has a fixed probability.

$$P(VP \rightarrow V\ NP\ NP) = 0.00151$$

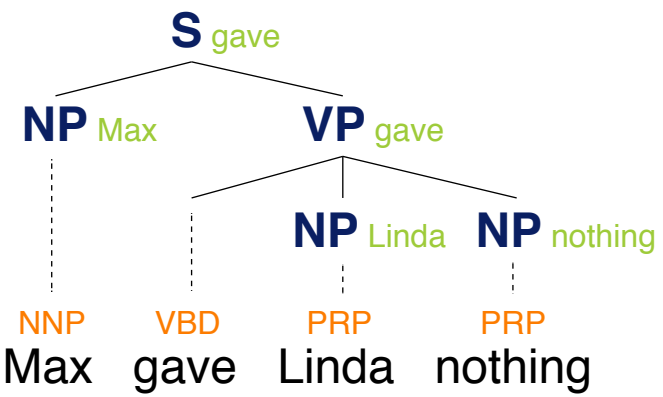
- However, specific words may make certain rules particularly (un)likely.

Lexicalization of PCFGs (Collins, 1999)

- Condition the probability of a rule on the *head word* of the given phrase.

$$P(VP \rightarrow V\ NP\ NP \mid \text{“said”}) = 0.00001$$
$$P(VP \rightarrow V\ NP\ NP \mid \text{“gave”}) = 0.01980$$

- Rationale. The head word represents the phrase's structure and meaning well.



Lexicalized parsing

- Constituency parsing based on a lexicalized PCFG.

  Seen as a breakthrough in the late 1990's.

Lexicalized Parsing

"Unlexicalization" of PCFGs

Hypothesis

- Lexical selection between content words is not crucial for parsing.
- More important are grammatical features, such as verb form, presence of a verb auxiliary, ...

Unlexicalized parsing (Klein and Manning, 2003)

- Rules are not systematically specified down to the level of lexical items.
- No semantic lexicalization for nouns, such as "NP_{stocks}".
- Instead: Structural "lexicalization", such as "NP^{S}_{CC}".
  Meaning: Parent node is "S" and noun phrase is coordinating.
- Keep functional lexicalization of closed-class words, such as "VB-have".

Learned unlexicalized parsing (Petrov and Knight, 2007)

- Learn extra information stored for a non-terminal based on training data.
- Parse based on unlexicalized PCFG.

Constituency Parsing

Evaluation Results *

Comparison of the different approaches

- All in exactly the same setting on the Penn Treebank.

Approach	Source	Labeled F_1
Extended CKY parsing	Charniak (1997)	0.73
Lexicalized parsing	Collins (1999)	0.89
Unlexicalized parsing	Klein and Manning (2003)	0.86
Learned unlexicalized parsing	Petrov and Klein (2007)	0.90
Combining parsers	Fossum and Knight (2009)	**0.92**

Notice

- These results are from a decade ago.
- Research has come up with many more approaches since then, but they are beyond the scope of this course.

  Particularly, neural approaches are successful nowadays.

Dependency Grammars

Dependency Grammars

Dependency grammar

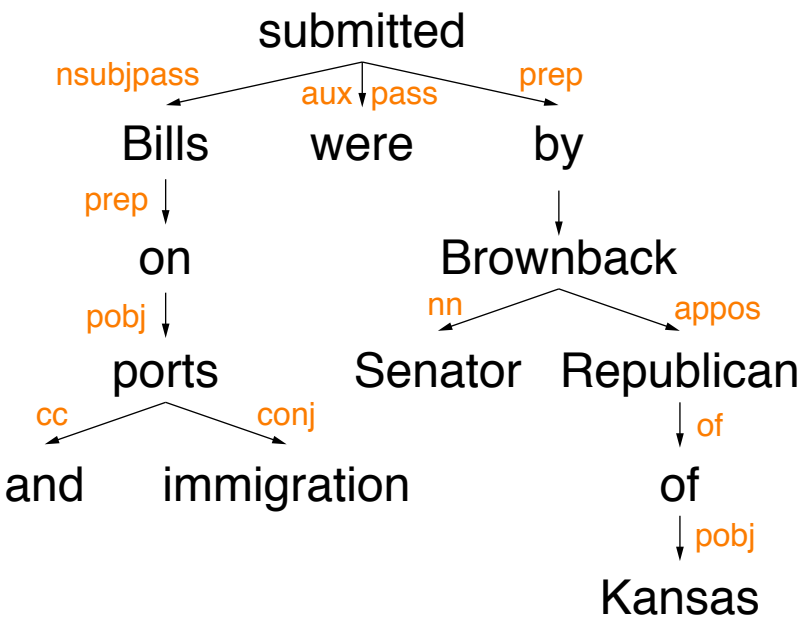
- Models the syntactic structure of a sentence by linking its tokens with binary asymmetric relations.
- Relations are called *dependencies* and define grammatical connections.

  Subject, prepositional object, apposition, etc.

Graph representation

- Each node is a token.
- An edge connects a *head* with a *dependent* node.
- The nodes and edges form a fully connected, acyclic tree with a single head.

  If available, the main verb of the first main clause is the head.



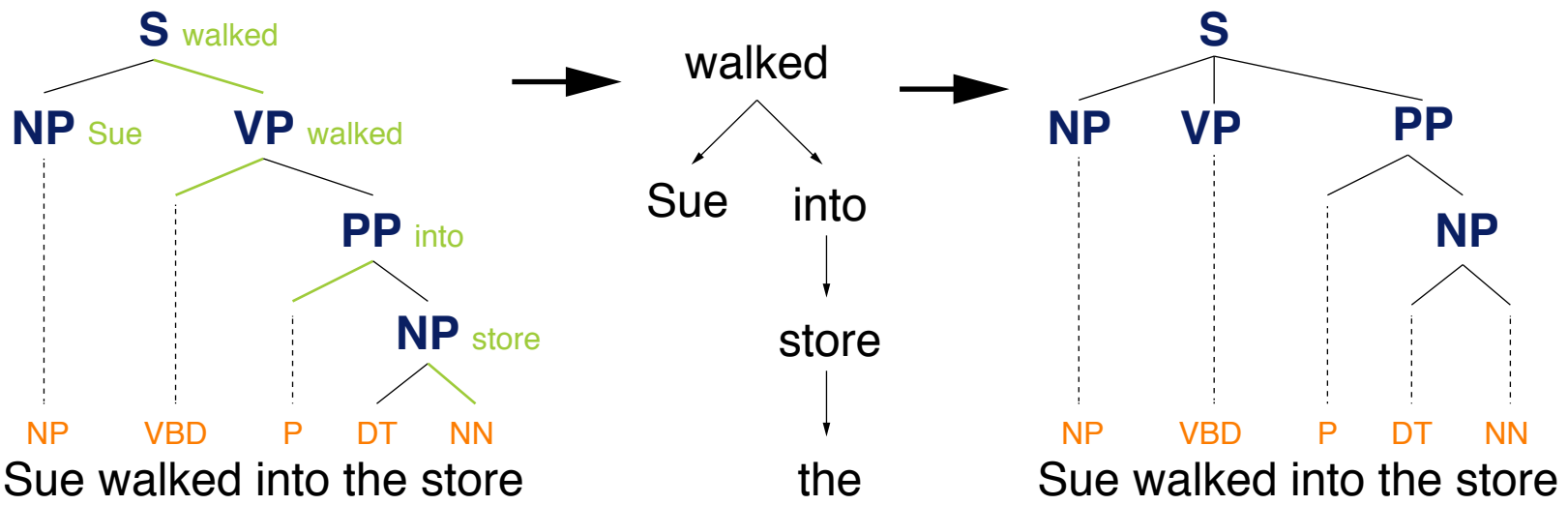
Dependency Grammars

Dependency Grammars vs. Phrase Structure Grammars

Dependency vs. phrase structure

- CFGs do not have the notion of a head — officially.
- All modern statistical parsers include hand-written phrasal “head rules”.

  For example, the head of an NP is a noun, number, adjective, ...
- Given head rules, constituencies can be converted to dependencies.
- Dependencies can be converted back to constituencies, but a word’s dependents will be on the same level.

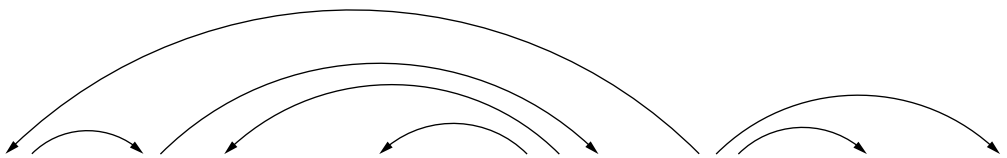


Dependency Grammars

Identification of Dependencies

Text features that can be exploited in dependency parsing

- Bilexical affinities. Some dependencies are more plausible than others.
  For example “issues → the” rather than “the → issues”.
- Distance. Dependencies more often hold between nearby words.
- Breaks. Dependencies rarely span intervening verbs or punctuation.
- Valency. Usual numbers of dependents for a head on each side.



Example “Retail sales drop in April cools afternoon market trading.”

“sales”	dependent of?	→ “drop”
“April”	dependent of?	→ “in”
“afternoon”	dependent of?	→ “trading”
“trading”	dependent of?	→ “cools”

Dependency Grammars

Parsing Methods *

Dynamic programming (Eisner, 1996)

- Lexicalized PCFG parsing, similar to CKY would need $\mathcal{O}(n^5)$ steps.
- When forcing parse structures with heads at the ends, $\mathcal{O}(n^3)$ is possible.

Graph algorithms (McDonald et al., 2005)

- Build a maximum spanning tree for a sentence and score dependencies independently using machine learning. $\rightarrow \mathcal{O}(n^3)$.
- More accurate on long dependencies and dependencies near the root.

Transition-based parsing (Nivre et al. 2008)

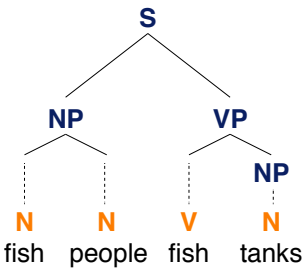
- Shift from left to right over a sentence and make greedy choices of attachments guided by a machine learning classifier. $\rightarrow \mathcal{O}(n)$
- More accurate on short dependencies and disambiguation of core grammatical functions.

Conclusion

Summary

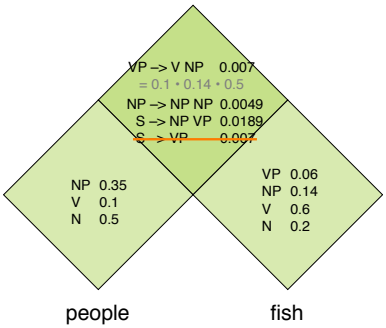
Text mining using grammars

- Text analysis based on formal language grammars.
- Grammars model sequential and hierarchical structure.
- Grammars may be based on statistics.



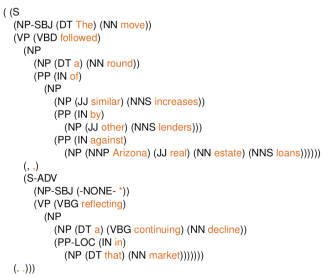
Types of grammars

- Regular grammars for numeric entities and similar.
- Probabilistic CFGs for constituency parsing.
- Dependency grammars for dependency parsing.



Benefits and limitations

- Grammars just model the ways syntax is constructed.
- Statistical grammars are a key technique in text mining.
- Creation of large-scale treebanks is very expensive.



References

Much content and many examples taken from

- Daniel Jurafsky and Christopher D. Manning (2016). Natural Language Processing. Lecture slides from the Stanford Coursera course. https://web.stanford.edu/~jurafsky/NLPCourseraSlides.html.
- Daniel Jurafsky and James H. Martin (2009). Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics. Prentice-Hall, 2nd edition.
- Friedhelm Meyer auf der Heide (2010). Einführung in Berechenbarkeit, Komplexität und Formale Sprachen. Begleitmaterial zur Vorlesung. https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Algorithmen/Lehre/Vorlesungsarchiv/WS_2009_10/Einfuehrung_in_die_Berechenbarkeit_K_u_f_S/skript.pdf
- Henning Wachsmuth (2015): Text Analysis Pipelines — Towards Ad-hoc Large-scale Text Mining. LNCS 9383, Springer.