

13 Partielles Parsing; Komplexität natürlicher Sprachen

13.1 Partielles Parsing (Chunking)

- 13.1.1 Grundlagen
- 13.1.2 Chunking mit regulärer Grammatik
- 13.1.3 Kaskadierende Chunker
- 13.1.4 Evaluation von Chunkern
- 13.1.5 Lernbasierte Chunking-Modelle / IOB-Tags

13.2 Komplexität formaler und natürlicher Sprachen

- 13.2.1 Chomsky-Hierarchie
- 13.2.2 *Center-embedding*-Konstruktionen
- 13.2.3 *Cross-serial dependencies*
- 13.2.4 *Garden-path*-Sätze

13 Partielles Parsing; Komplexität natürlicher Sprachen

13.1 Partielles Parsing (Chunking)

13.1.1 Grundlagen

- für viele Anwendungen: **keine syntaktische Vollanalyse notwendig**
- **Partielles Parsing** als **unvollständige, flache Syntaxanalyse**
 - nur die Konstituententypen mit **Inhaltswort als Kopf**:
NP, VP, PP, AdjP
 - auch als ***Chunking*** bezeichnet (Abney 1991: '*parsing by chunks*')
- Partielle-Parsing-Methoden u.a. entwickelt für:
 - **Informationsextraktion**: Finden semantischer Einheiten
 - ***Information Retrieval***: Phrasen als Indexterme

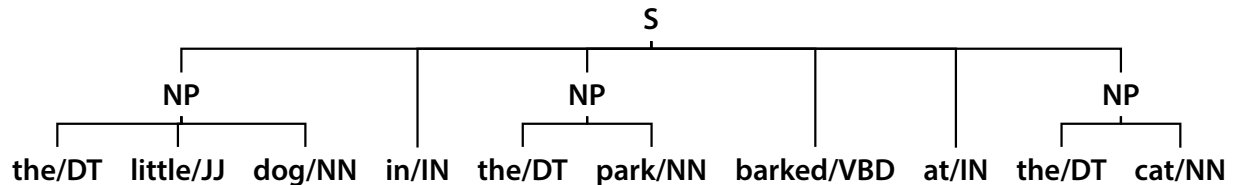
- **unvollständige Analyse:**

- nur die Wörter berücksichtigt, die **Element der relevanten syntaktischen Einheiten** sind

- **andere Wörter werden nicht syntaktisch annotiert**

⇒ **Partielle syntaktische Analyse**

→ z. B. nur NP-Chunks:



- **flache Analyse:**

- flache, **nicht-hierarchische syntaktische Analyse**

- Chunk-Bäume haben **Tiefe 1**

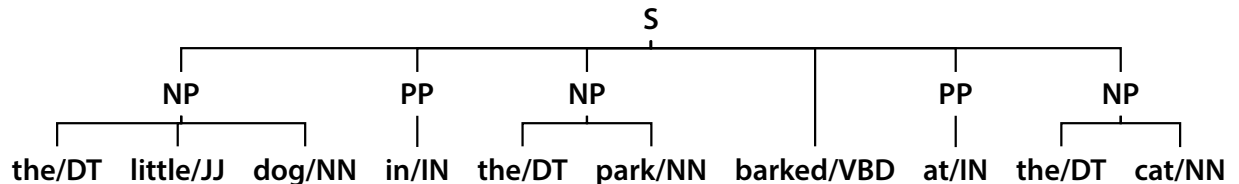
- **keine komplexen, verschachtelten Phrasen**

- **Chunk als *base phrase*:**

- **kleinere Einheiten als vollständige Phrasen**

- NP-Chunks **ohne Rechtsattribute**

- **PP-Chunks bestehen nur aus Präposition:**



- **2 Aufgaben für Chunk-Parser:**
 - ***Segmentierung*** = **Grenzen der Chunks finden**
 - Identifizierung von **Sequenzen von Tokens** als Chunks
 - ***Labeling*** der Chunks
 - **Klassifikation der Einheiten** = ***Tagging***

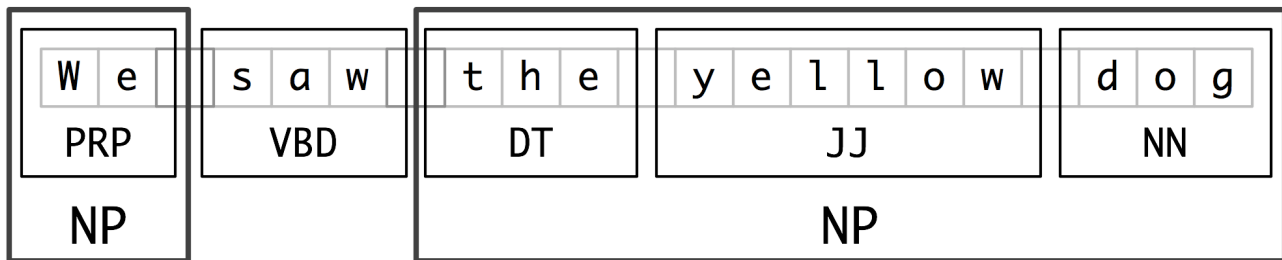


Abbildung 1: *Segmentierung und Labeling auf Token- und Chunk-Ebene*
(<http://www.nltk.org/images/chunk-segmentation.png>)

- **Methoden zur Erstellung von Chunk-Parsern**
 - **regelbasiert** mit regulärer Grammatik
 - *supervised machine learning*
- zentrales Merkmal in beiden Ansätzen: **POS-Tags**
- **Darstellung:** als flache **Bäume** oder als **Tags (IOB-Format)**

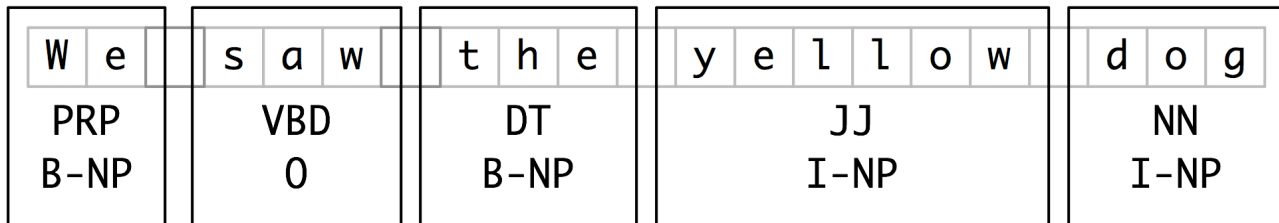


Abbildung 2: Tag-Repräsentation von Chunks im IOB-Format
(<http://www.nltk.org/images/chunk-tagrep.png>)

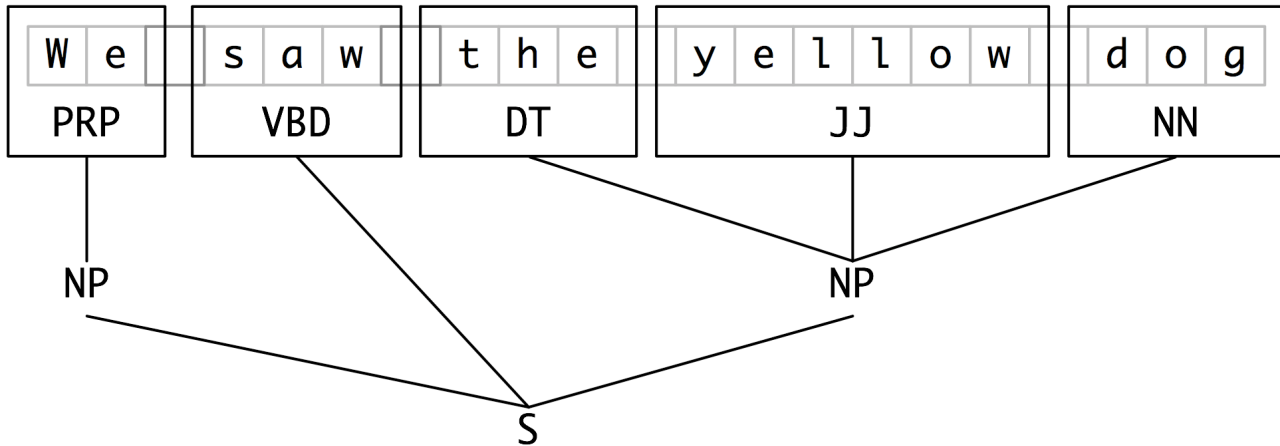


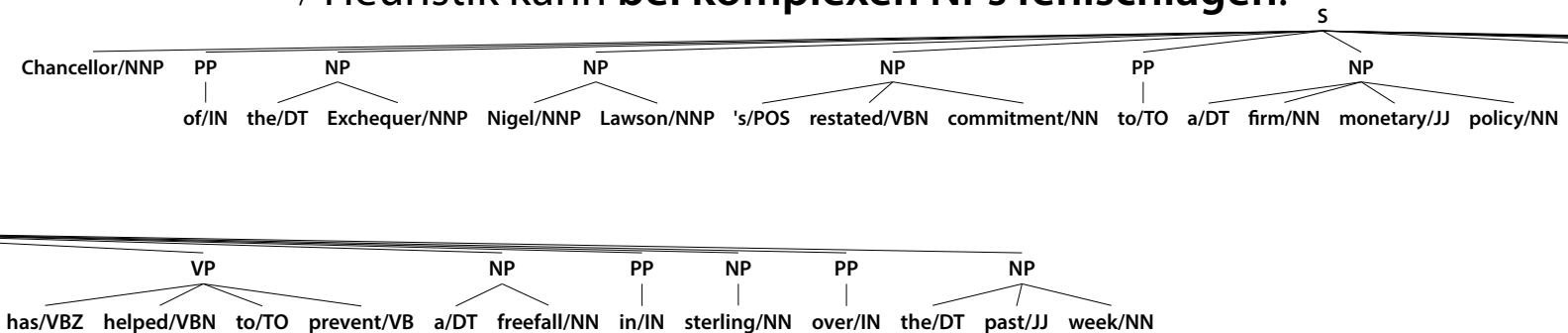
Abbildung 3: *Baum-Repräsentation von Chunks*
(<http://www.nltk.org/images/chunk-treerep.png>)

Partielles vs. Vollständiges Parsing

- **Vorteile:**
 - **einfaches Modell**, in vielen Anwendungsfällen ausreichend
 - **weniger Disambiguierungsprobleme** als bei PSG-Grammatiken
→ z. B. **Vermeidung PP-Attachment-Ambiguität** durch nicht-hierarchische Analyse
 - **bessere Performance** gegenüber CFGs oder Unifikationsgrammatiken, z.B. **Chunker als reguläre Grammatik**
 - **hohe Accuracy** aufgrund flacher, unvollständiger Strukturanalyse

- **Nachteile:**

- nur **unvollständige Beschreibung** syntaktischer Struktur
- **grammatische Beziehungen** in der syntaktischen Struktur **durch die flache Analyse** nicht direkt modelliert
 - nur heuristisch erfassbar, z. B.:
 - *NP vor Verb ist Subjekt oder erste NP ist Subjekt*
 - Heuristik kann **bei komplexen NPs** fehlschlagen:



Auflistung 1: *NLTK: Beispiel flach analysierter komplexer NP (aus conll2000-Korpus)*

```
1 #(S
2 #  Chancellor/NNP
3 #  (PP of/IN)
4 #  (NP the/DT Exchequer/NNP)
5 #  (NP Nigel/NNP Lawson/NNP)
6 #  (NP 's/POS restated/VBN commitment/NN)
7 #  (PP to/TO)
8 #  (NP a/DT firm/NN monetary/JJ policy/NN)
9 #  (VP has/VBZ helped/VBN to/TO prevent/VB)
10 #  (NP a/DT freefall/NN)
11 #  (PP in/IN)
12 #  (NP sterling/NN)
13 #  (PP over/IN)
```

14 # (NP the/DT past/JJ week/NN)
 15 # ./.)

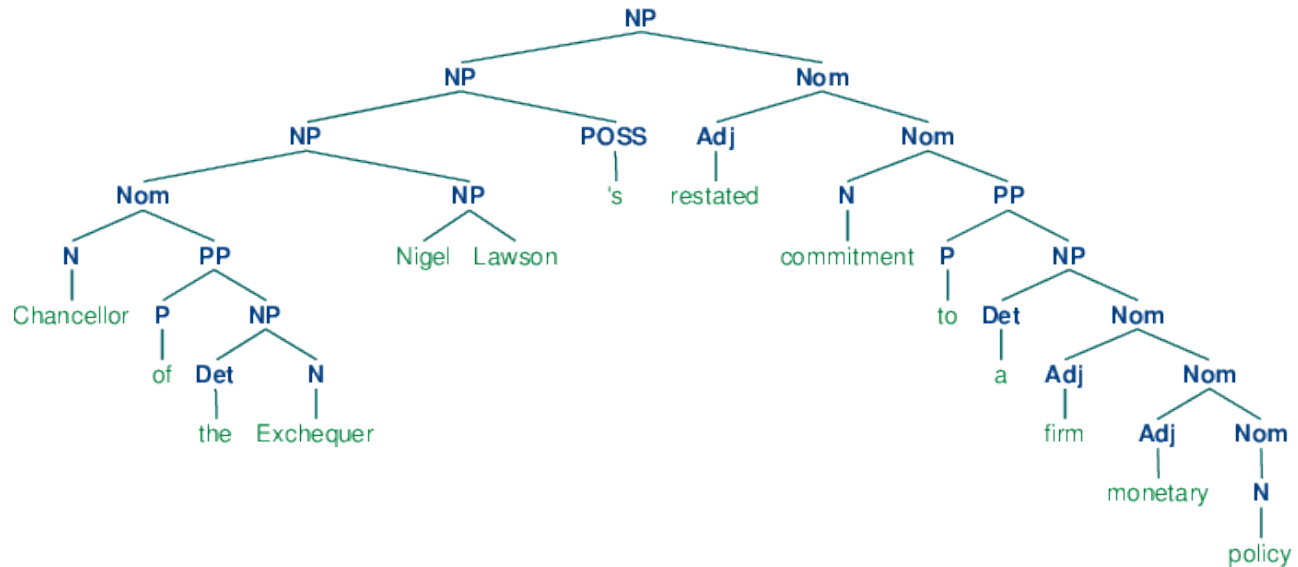


Abbildung 4: Vollständige Phrasenstrukturanalyse der komplexen Subjekt-NP
 (http://www.nltk.org/book/tree_images/ch08-extras-tree-1.png)

- in **hierarchischer Phrasenstrukturanalyse** wird die **komplexe NP unter einem NP-Knoten** zusammengefasst (statt in viele NP-Chunks aufgesplittet)
 - komplexe NPs werden **in ihrem hierarchischem Aufbau** analysiert und der **Kopf der Phrase** ist in einem X-Bar-Schema **über Strukturposition** eindeutig **identifizierbar**: NP, NOM+, N
- **Subjekt- und Objekt-Funktion** sind über **Position im Strukturbaum** repräsentiert
 - Subjekt-NP unmittelbar dominiert von S
 - Objekt-NP unmittelbar dominiert von VP

13.1.2 Chunking mit regulärer Grammatik

- Beschreibung von POS-Folgen durch reguläre Ausdrücke
- Tag-Muster = Regeln einer Chunk-Grammatik
 - NLTK-eigene Syntax: POS-Tags in eckigen Klammern:
 - z. B. Chunk als POS-Folge *Artikel - Adjektivattribute (optional)*
 - *Nomen*: <DT><JJ>*<NN>
- Chunking als **sukzessive Anwendung der Regeln** einer Chunk-Grammatik auf die Sätze eines POS-getaggten Textes
- **FSTs** = *finite state transducers* ermöglichen **Segmentierung und Labeling** (Output durch *transducer*): NP: <DT><JJ>*<NN>

Auflistung 2: NLTK: Chunk-Parser-Grammatik mit 2 NP-Chunk-Regeln

```
1 grammar = r"""
2     NP: {<DT|PP\$>?<JJ>*<NN>}
3     {<NNP>+}
4     """
5 sent = [("Rapunzel", "NNP"), ("let", "VBD"),
6         ("down", "RP"), ("her", "PP$"), ("long",
7         "JJ"), ("golden", "JJ"), ("hair", "NN")]
8 parser = nltk.RegexpParser(grammar)
9 tree = parser.parse(sent)
10 print(tree)
11 #(S
12 #  (NP Rapunzel/NNP)
13 #  let/VBD
```



```
12 | # down/RP  
13 | # (NP her/PP$ long/JJ golden/JJ hair/NN))
```

- **bereits gefundene Chunk-Elemente:** in folgenden Regeln **nicht mehr verfügbar**
 - nur **nicht-überlappende Chunks** werden gefunden
 - **Reihenfolge der Regeln** in Grammatik ist also **wichtig**
 - *siehe folgendes Beispiel: bei invertierter Regel-Abfolge wird die POSS-ADJ-NN-Folge nicht gefunden, da NN schon als NP erkannt*
- **Tracing** hilfreich beim **Entwickeln von Chunk-Grammatiken**
 - Beschreibung der Regeln durch **Kommentare**

Auflistung 3: *NLTK: Chunk-Parser-Grammatik mit verschiedener Regelreihenfolge, Kommentaren und Parsing mit tracing*

```

1 grammar = r"""
2     NP: {<DT|PP\$>?<JJ>*<NN>}    #chunk DET/POSS
        ADJ NOUN
3     {<NN.?>+}                    #chunk sequences
        of NOUNS/PROPER NOUNS
4     """
5 parser = nltk.RegexpParser(grammar, trace=1)
6 # Input:
7 <NNP> <VBD> <RP> <PP\$> <JJ> <JJ> <NN>
8 # chunk DET/POSS ADJ NOUN:
9 <NNP> <VBD> <RP> {<PP\$> <JJ> <JJ> <NN>}
10 # chunk sequences of NOUNS/PROPER NOUNS:
11 {<NNP>} <VBD> <RP> {<PP\$> <JJ> <JJ> <NN>}
12

```

```
13 print(tree)
14
15 #(S
16 #  (NP Rapunzel/NNP)
17 #  let/VBD
18 #  down/RP
19 #  (NP her/PP$ long/JJ golden/JJ hair/NN))
20
21
22 #GRAMMATIK MIT INVERTIERTER REIHENFOLGE:
23 grammar = r"""
24     NP: {<NN.?>+}                #chunk
25         sequences of NOUNS/PROPER NOUNS
26     {<DT|PP\$>?<JJ>*<NN>}  #chunk DET/POSS ADJ
27         NOUN
28     """
```

```

27 parser = nltk.RegexpParser(grammar, trace=1)
28 # Input:
29 <NNP> <VBD> <RP> <PP$> <JJ> <JJ> <NN>
30 # chunk sequences of NOUNS/PROPER NOUNS:
31 {<NNP>} <VBD> <RP> <PP$> <JJ> <JJ> {<NN>}
32 # chunk DET/POSS ADJ NOUN:
33 {<NNP>} <VBD> <RP> <PP$> <JJ> <JJ> {<NN>}
34 print(tree)
35 #(S
36 # (NP Rapunzel/NNP)
37 # let/VBD
38 # down/RP
39 # her/PP$
40 # long/JJ
41 # golden/JJ
42 # (NP hair/NN))

```

Chunking und Chinking

- **Chunking-Regel:** Definition Chunk als Muster einer POS-Folge:

→ NP: {<NNP>+}

- **Chinking-Regel:** Definition eines Musters einer POS-Folge, die **nicht in Chunk enthalten ist:**

→ }<VBD | IN>+{

[the/DT little/JJ yellow/JJ dog/NN] barked/VBD at/IN [the/DT
cat/NN]

→ **Anwendung auf bereits gefundene Chunks**

- **Möglichkeiten beim Chinking:**
 - wenn Chink-Muster in **Mitte** von Chunk
⇒ Chunk wird entsprechend **gesplittet**
 - wenn Chink-Muster **kompletter Chunk**
⇒ Chunk wird **entfernt**
 - wenn Chink-Muster am **Anfang oder Ende** von Chunk
⇒ Chunk wird entsprechend **beschnitten**

Auflistung 4: *Chinking mit NLTK*

```

1 grammar = r"""
2     NP:
3         {<.*>+}      # Chunk everything
4         }<VBD|IN>+{    # Chink sequences of VBD and IN
5     """
6 parser = nltk.RegexpParser(grammar)
7 tree = parser.parse(sent)
8 print(tree)
9 #(S
10 #  (NP the/DT little/JJ yellow/JJ dog/NN)
11 #  barked/VBD
12 #  at/IN
13 #  (NP the/DT cat/NN))

```

Split-Regeln

- **Split-Regeln**
 - **Anwendung auf gefundene Chunks**
 - z. B. **Splitten** von gefundener NP am Determinierer:
 - $\langle . * \rangle \{ \langle DT \rangle$

Auflistung 5: *Split-Regel mit NLTK*

```
1 sent = [("the", "DT"), ("cat", "NN"), ("the",  
    "DT"), ("dog", "NN"), ("chased", "V")]  
2  
3 #OHNE SPLIT-REGEL:  
4 grammar = r"""  
5     NP:    {<DT|NN>+}    #chunk sequences of DETs  
6             and NOUNs  
7     """  
8 parser = nltk.RegexpParser(grammar)  
9 tree = parser.parse(sent)  
10 print(tree)  
11 #(S (NP the/DT cat/NN the/DT dog/NN)  
12     chased/VBD)
```

```

13 #MIT SPLIT-REGEL:
14 grammar = r"""
15     NP:    {<DT|NN>+}    #chunk sequences of DETs
16             and NOUNs
17             <.*>}{<DT> #split chunks
18     """
19 parser = nltk.RegexpParser(grammar)
20 tree = parser.parse(sent)
21 print(tree)
22  #(S (NP the/DT cat/NN) (NP the/DT dog/NN)
23     chased/VBD)

```

13.1.3 Kaskadierende Chunker

- durch **Kombination** von (flachen) **Chunk-Parsern** können **hierarchische, tiefere Strukturen** erzeugt werden
 - **Output eines Chunkers** dient als **Input des Folgenden**
 - Regeln können **Chunk-Tags** enthalten:
 - ⇒ hierarchische Strukturen, z. B. PP: {<IN><NP>}
 - **Approximation** Output eines **PSG-Parsers**
- **Loopen eines Chunk-Parsers** über von diesem Parser erkannte Muster:
 - findet **Chunks**, die in einem ersten Durchlauf nicht erkannt wurden

Hintereinandergeschaltete Chunk-Parser für komplexe PPs:

1. Suche **Adjektivattribute (AdjP)**: <JJ>+
2. Suche **NPs**: <DT><ADJP><NN>
3. Suche **PPs**: <P><NP>

Satz	on/IN	the/DT	black/JJ	mat/NN
Chunker 1	on/IN	the/DT [ADJP black/JJ]	mat/NN	
Chunker 2	on/IN [NP the/DT [ADJP black/JJ]	mat/NN]		
Chunker 3	[PP on/IN [NP the/DT [ADJP black/JJ]	mat/NN]]		

Abbildung 5: Beispiel kaskadierender Chunk-Parser zur Analyse einer komplexen PP

Auflistung 6: *NLTK: Loopen eines 4-stufigen Chunk-Parsers erkennt in zweitem Durchgang saw/VBD als VP eines Objektsatzes*

```

1 grammar = r"""
2     NP: {<DT|JJ|NN.*>+}           # Chunk
        sequences of DT, JJ, NN
3     PP: {<IN><NP>}                 # Chunk
        prepositions followed by NP
4     VP: {<VB.*><NP|PP|CLAUSE>+$} # Chunk verbs
        and their arguments
5     CLAUSE: {<NP><VP>}             # Chunk NP, VP
6     """
7
8 sent = [("John", "NNP"), ("thinks", "VBZ"),
        ("Mary", "NN"), ("saw", "VBD"), ("the",
        "DT"), ("cat", "NN"), ("sit", "VB"), ("on",
        "IN"), ("the", "DT"), ("mat", "NN")]

```

```
9 parser = nltk.RegexpParser(grammar)
10 tree = parser.parse(sent)
11 print(tree)
12 #(S
13 #  (NP John/NNP)
14 #  thinks/VBZ
15 #  (NP Mary/NN)
16 #  saw/VBD # [_saw-vbd]
17 #  (CLAUSE
18 #    (NP the/DT cat/NN)
19 #    (VP sit/VB (PP on/IN (NP the/DT
20 #      mat/NN))))))
21
22
23
```

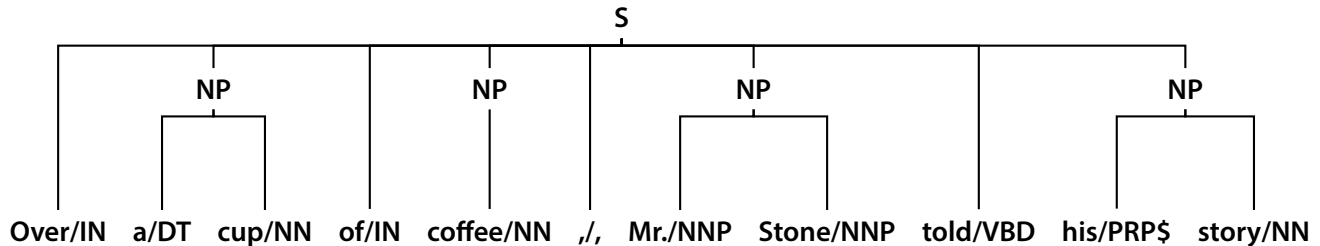
```
24 parser = nltk.RegexpParser(grammar, loop=2)
25 tree = parser.parse(sent)
26 print(tree)
27 #(S
28 #  (NP John/NNP)
29 #  thinks/VBZ
30 #  (CLAUSE
31 #    (NP Mary/NN)
32 #    (VP
33 #      saw/VBD
34 #      (CLAUSE
35 #        (NP the/DT cat/NN)
36 #        (VP sit/VB (PP on/IN (NP the/DT
    mat/NN)))))))))
```

13.1.4 Evaluation von Chunkern

- **con112000-Korpus** im NLTK (270.000 Tokens)
 - **POS- und Chunk-getaggt**es Korpus mit NPs, VPs, und PPs
 - Aufteilung in **Test- und Trainingsmenge**
- Auswahl **Chunks bestimmter Typen** mit `chunk_types`-Argument

Auflistung 7: NLTK: Auszug aus conll2000-Trainingskorpus

```
1 from nltk.corpus import conll2000
2 print(conll2000.chunked_sents('train.txt',
   chunk_types=['NP'])[99])
3 (S
4   Over/IN
5   (NP a/DT cup/NN)
6   of/IN
7   (NP coffee/NN)
8   ,/,
9   (NP Mr./NNP Stone/NNP)
10  told/VBD
11  (NP his/PRP$ story/NN)
12  ./.)
```



- Das Korpus kann zum **Evaluieren von Chunk-Parsern** verwendet werden
- **Vergleich Chunker-Output** (Hypothese) mit Test-Menge als *gold-standard*-Referenz-Korpus (annotiert von Experten)
- **korrekter Chunk = korrekte Spanne und korrektes Label**

- **Recall:** $R = \frac{(\text{Anzahl von korrekten Chunks in Chunker-Output})}{(\text{Anzahl aller Chunks in Referenz-Korpus})}$

→ Anteil der Chunks des Referenz-Korpus, die vom Chunker korrekt identifiziert wurden

- **Precision:** $P = \frac{(\text{Anzahl von korrekten Chunks in Chunker-Output})}{(\text{Anzahl aller Chunks in Chunker-Output})}$

→ Anteil der vom Chunker identifizierten Chunks, die korrekt sind

- **F-score** = $\frac{(\beta^2 + 1)PR}{\beta^2 P + R}$

→ Kombination von Precision und Recall in einem Maß

→ β : Parameter zur Gewichtung von P und R, gleichgewichtet:

$$F_1 = \frac{2PR}{P + R} \quad (\text{harmonisches Mittel von P und R})$$

Auflistung 8: NLTK: Evaluation Regexp-Parser

```
1 from nltk.corpus import conll2000
2 test_sents =
    conll2000.chunked_sents('test.txt',
        chunk_types=['NP'])
3
4 grammar = "NP: {<DT>?<JJ>*<NN>}"
5 cp = nltk.RegexpParser(grammar)
6 print(cp.evaluate(test_sents))
7 ChunkParse score:
8     IOB Accuracy:    59.7%
9     Precision:       45.3%
10    Recall:          24.2%
11    F-Measure:       31.6%
12
13
```

```
14 grammar = r"""
15     NP: {<DT|PP\$>?<JJ>*<NN>}
16     {<NNP>+}
17     """
18 cp = nltk.RegexpParser(grammar)
19 print(cp.evaluate(test_sents))
20 ChunkParse score:
21     IOB Accuracy:    67.7%
22     Precision:       48.1%
23     Recall:          37.4%
24     F-Measure:       42.1%
```

13.1.5 Lernbasierte Chunking-Modelle / IOB-Tags

- **Training eines Klassifikators, der die Wörter auf Chunk-Klassen abbildet**
- Vorgehen analog zu POS-Tagging mit *supervised machine-learning*
 - **POS-Tagging**: Abbildung Token auf POS-Tag
 - **Chunking**: Abbildung Token-POS-Tupel auf IOB-Tag (*IOB-Tagging*) = Sequenzklassifikation (*'parsing as tagging'*)
- für **Training** ist ein **Chunk-getaggttes Korpus im IOB-Format** notwendig, z. B. das con112000-Korpus im NLTK

Chunking als *supervised*-Sequenzklassifikation

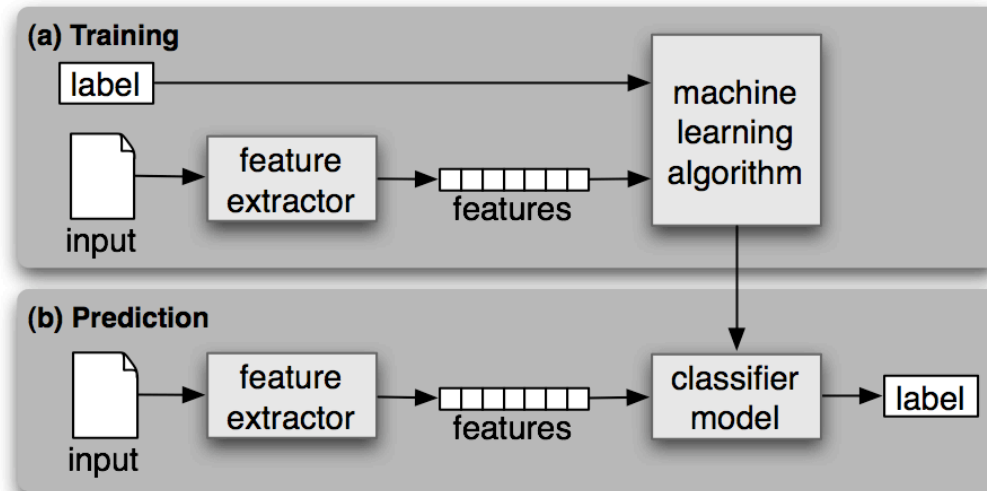
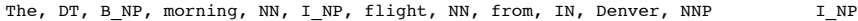


Abbildung 7: Schema supervised-Klassifikation
(<http://www.nltk.org/images/supervised-classification.png>)



Input wird durch eine aus Kontextfenster extrahierte Feature-Menge repräsentiert

13.2 Komplexität formaler und natürlicher Sprachen

13.2.1 Chomsky-Hierarchie

- Klassifizierung formaler Grammatiken bzgl. **Regeleinschränkung**
→ je **stärker** eine Grammatik **eingeschränkt** desto **geringer** ihre **Erzeugungsmächtigkeit**
→ desto **geringer** die **Komplexität** der erzeugten Sprache
- 4 Typen von **Typ 0** (rekursiv aufzählbar = ohne Einschränkung) bis **Typ 3** (regulär = am stärksten eingeschränkt)
- klassische Phrasenstrukturgrammatiken: **kontext-frei** (Typ 2)
- einige Syntaxformalismen sind **kontextsensitiv** (Typ 1) (TAG, CCG) bzw. **rekursiv aufzählbar** (Typ 0) (HPSG, LFG)

Die 4 Typen der Chomsky-Hierarchie:

- **rekursiv aufzählbar (Typ 0):** $\alpha \rightarrow \beta$
→ *ohne Einschränkung bzgl. α, β ; $\alpha, \beta \in Alphabet = \{T, NT\}$*
- **kontext-sensitiv (Typ 1):** $\alpha \rightarrow \beta, length(\alpha) \leq length(\beta)$
→ bzw. auch: $lXr \rightarrow l\beta r$ ($X \in NT$; $l, r, \beta \in \{T, NT\}$)
- **kontext-frei (Typ 2):** $X \rightarrow \beta$ ($X \in NT$; $\beta \in \{T, NT\}$)
→ *LHS: nur 1 Nicht-Terminal*
- **regulär (Typ 3):** $X \rightarrow a, X \rightarrow aY$ ($X, Y \in NT, a \in T$)
→ *LHS: nur 1 Nicht-Terminal*
→ *RHS: 0-n Terminale und 0-1 Nicht-Terminale (links oder rechts)*

Komplexität natürlicher Sprachen

- **Chomsky:** Kann natürliche Sprache mit regulärer Grammatik (endlichen Automaten) modelliert werden?
- es gibt **nicht-reguläre Phänomene** in natürlicher Sprache
 - z. B. *center-embedding*-Rekursion
 - benötigt **kontextfreie Regel**
- allerdings: die Konstruktionen, die eine natürliche Sprache **nicht-regulär** machen, sind **für den Menschen schwer zu parsen**

- mathematisch-formal: **Großteil der Syntax menschlicher Sprache** mit **regulärer Grammatik** modellierbar
- aber: **kontextfreie Grammatiken** geben **beschreibungsadäquaterer Struktur**
 - **linguistisch adäquates Modell**
 - **wichtig für weitere Verarbeitung** (semantische Analyse)
- **einige Sprachen** enthalten **Konstruktionen**, die sie **kontextsensitiv** machen: *cross-serial dependencies* im Schweizerdeutschen
- **Hinweise**, dass auch **menschliches Parsing** **Wahrscheinlichkeiten berücksichtigt**: *garden-path*-Sätze

13.2.2 *Center-embedding*-Konstruktionen

- ***center-embedding*-Rekursion:** $X \rightarrow \alpha X \beta$
 - **rekursive Regel:** Nichtterminal erweitert zu selbem Nichtterminal, umgeben von Strings
- ***center-embedding*-Regel ist nicht-regulär:**
 - reguläre Grammatik: **nur links- oder rechtslineare Regeln:**
 $X \rightarrow Xa$ oder $X \rightarrow aX$
 - entsprechende **Einbettung nicht möglich**

Beispiel einer *center-embedding*-Konstruktion

- **Rekursive Einbettung von Relativsätzen** als nominales Attribut

(Das Kind,)

das den Hund, der die Katze, die den Vogel jagt, anbellt, ausführt,

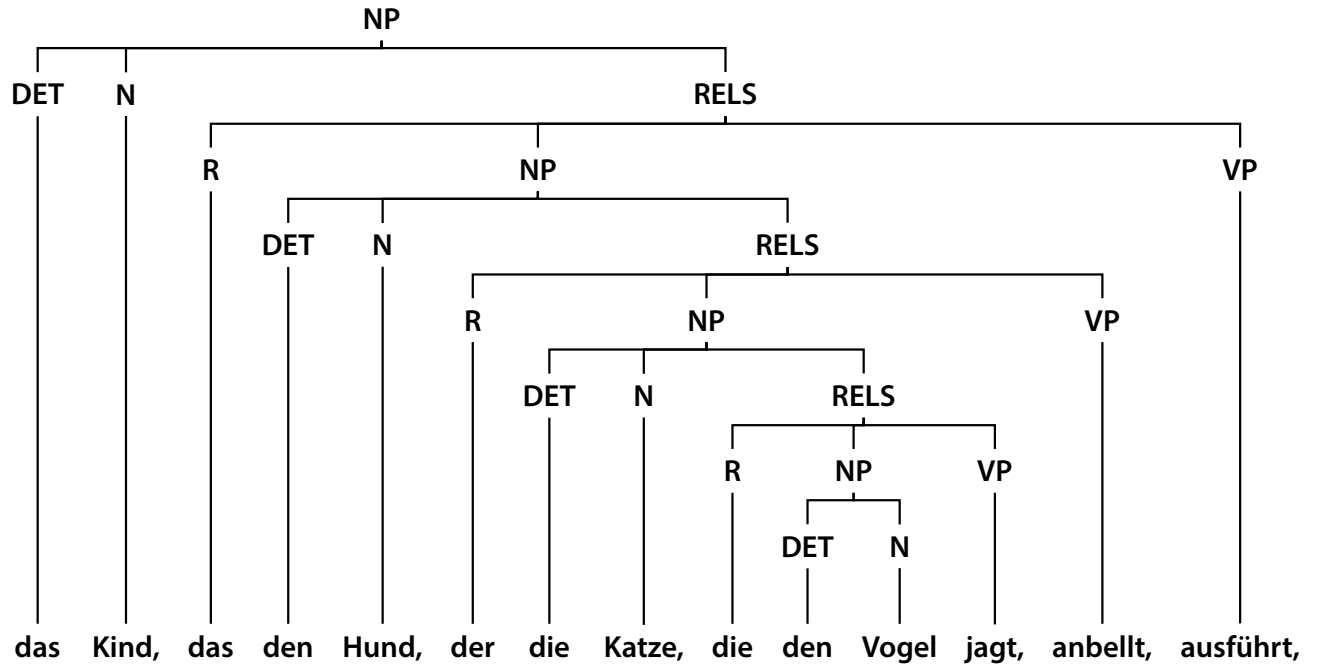
→ Schema: $N_1(N_2(N_3V_3)V_2)V_1$

RELS → R NP VP

NP → DET N ***RELS***

Ableitungen: **RELS** → R DET N **RELS** VP

RELS → R DET N **R DET N *RELS* VP VP** usw.



- psycholinguistische Experimente zeigen **Beschränkung in der Verarbeitung solcher Strukturen durch die menschliche Sprachverarbeitung** aufgrund von *memory limitations*
→ mehrfach verschachtelte *center-embedding*-Strukturen sind nur **bis zu einer begrenzten Tiefe** verarbeitbar

- **Korpus-Beispiel für *center-embedding* der Tiefe 3:**

[M Er ... war allen Gefahren ...

[C-1 welche ein jeder,

[C-2 der diese wilde Gegend zu jener Zeit,

[C-3 als diese Geschichte dort spielte,]

durchstreifte,]

gewärtig sein mußte,]

gewachsen]

(vgl. Karlsson 2007, Constraints on multiple center-embedding of clauses, Journal of Linguistics 43/2, 365-392. <http://www.ling.helsinki.fi/~fkarlssso/ceb5.pdf>)

13.2.3 *Cross-serial dependencies*

- einige Sprachen, z. B. das **Schweizerdeutsche**, besitzen eine **Konstruktion, die nicht mit kontextfreien Grammatikmodellen darstellbar ist**

→ ***cross-serial dependencies***, d. h. **Dependenzrelationen mit überkreuzenden Kanten:**

$N_1 N_2 V_1 V_2$

$N_1 N_2 N_3 V_1 V_2 V_3$

- Wörter bzw. Teilkonstituenten sind **seriell überkreuzend angeordnet**

***cross-serial*-Anordnung von Verb und Argument**

Swiss-German:

...mer em Hans es huss hälfed aastrische



English:

...we helped Hans paint the house



Abbildung 9: *Cross-serial dependencies* (by Christian Nassif-Haynes - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=28304322>)

Argument für Nicht-Kontextfreiheit des Schweizerdeutschen

- **Anzahl von Verben mit Dativ-Komplement** muss übereinstimmen mit **Anzahl von Dativ-Komplementen**
- ebenso für Akkusativ-Komplemente
- theoretisch **unbegrenzte Anzahl** solcher *cross-serial dependencies* pro Satz
- **solche Sprachen enthalten** $L' = a^m b^n c^m d^n$
- die **Sprache L' ist aber nicht-kontextfrei**
→ Nachweis über **Pumping Lemma** für kontextfreie Sprachen

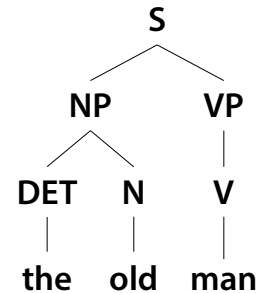
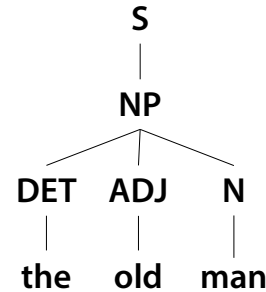
13.2.4 *Garden-path*-Sätze

- Psycholinguistik: **Parser als Modell menschlicher Sprachverarbeitung**
- **Vergleich mit statistischen Sprachmodellen gibt Hinweis, dass auch menschliches Parsing Wahrscheinlichkeiten berücksichtigt**
 - **Disambiguierung** über statistische Informationen

- Beispiel: ***garden-path*-Sätze** = Sätze mit **temporärer Ambiguität**
 - **Gesamter Satz: unambig**, nur eine Ableitung
 - **Teil des Satzes: ambig**, eine strukturelle **Lesart** wird (offensichtlich) **von der menschlichen Sprachverarbeitung bevorzugt**
 - aber: **nicht-präferierte Lesart** für den Teil ist die für die **Ableitung des Satzes korrekte**
- Beobachtung: **wahrscheinlichste Ableitung wird verfolgt**, bis sie fehlschlägt und **Backtracking (Reanalyse)** notwendig ist

Beispiel: *garden-path-Satz*

The old man the boat.



- $P(\text{man}|\text{N}) > P(\text{man}|\text{V}), P(\text{old}|\text{ADJ}) > P(\text{old}|\text{N})$

