Übersicht

IV Partielles Parsing

13 Partielles Parsing; Komplexität natürlicher Sprachen

13.1 Partielles Parsing (Chunking)
- 13.1.1 Chunking mit regulärer Grammatik
- 13.1.2 Kaskadierende Chunker
- 13.1.3 Evaluation von Chunkern
- 13.1.4 Lernbasierte Chunking-Modelle

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

Teil IV.

Partielles Parsing

13. Partielles Parsing; Komplexität natürlicher Sprachen

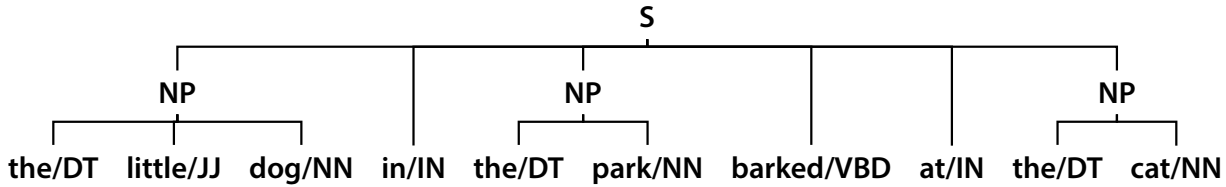
13.1. Partielles Parsing (Chunking)

- für viele Anwendungen: **keine syntaktische Vollanalyse notwendig**
- **Partielles Parsing** als **unvollständige, flache Syntaxanalyse**
  $\rightarrow$ nur die Konstituententypen mit **Inhaltswort als Kopf:**
  `NP, VP, PP, AdjP`
  $\rightarrow$ 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

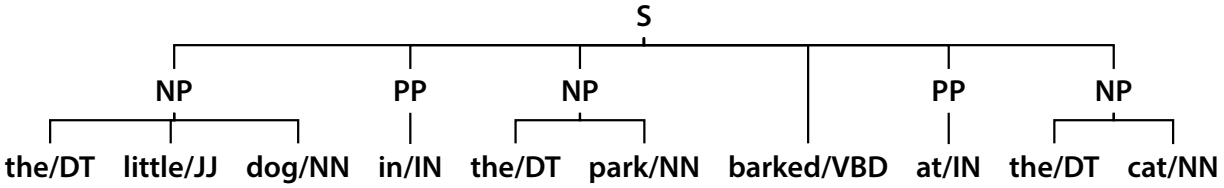
- **unvollständige Analyse:**
  - nur die Wörter berücksichtigt, die **Element der relevanten syntaktischen Einheiten** sind
  - **andere Wörter** werden **nicht syntaktisch annotiert**

    $\Rightarrow$ **Partielle syntaktische Analyse**

  $\rightarrow$ z. B. nur NP-Chunks:



- **flache Analyse:**
  - flache, **nicht-hierarchische syntaktische Analyse**
    $\rightarrow$ Chunk-Bäume haben **Tiefe 1**
    $\rightarrow$ **keine komplexen**, verschachtelten **Phrasen**
  - **Chunk als *base phrase*:**
    $\rightarrow$ **kleinere Einheiten als vollständige Phrasen**
    $\rightarrow$ NP-Chunks **ohne Rechtsattribute**
    $\rightarrow$ 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***

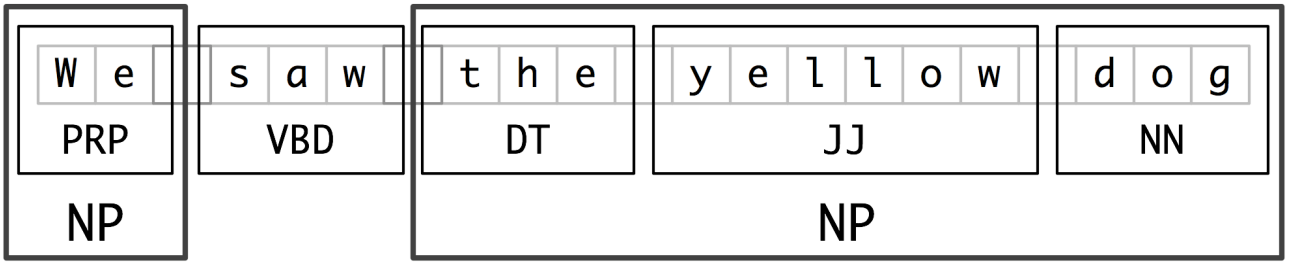


Abbildung 1: *Segmentierung und Labeling auf Token- und Chunk-Ebene (http://www.nltk.org/images/chunk-segmentation.png)*

- **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**)

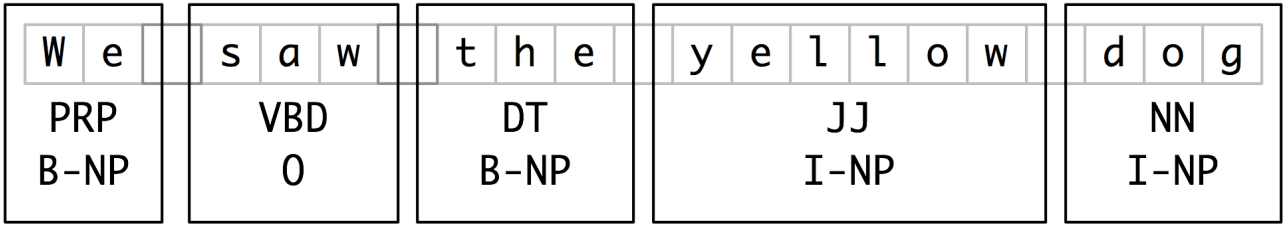


Abbildung 2: *Tag-Repräsentation von Chunks im IOB-Format (http://www.nltk.org/images/chunk-tagrep.png)*

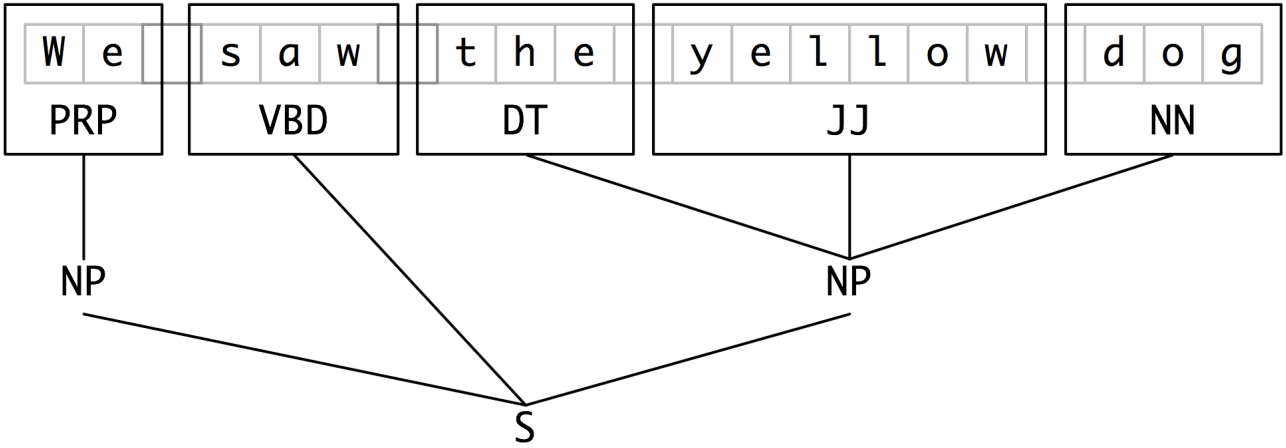
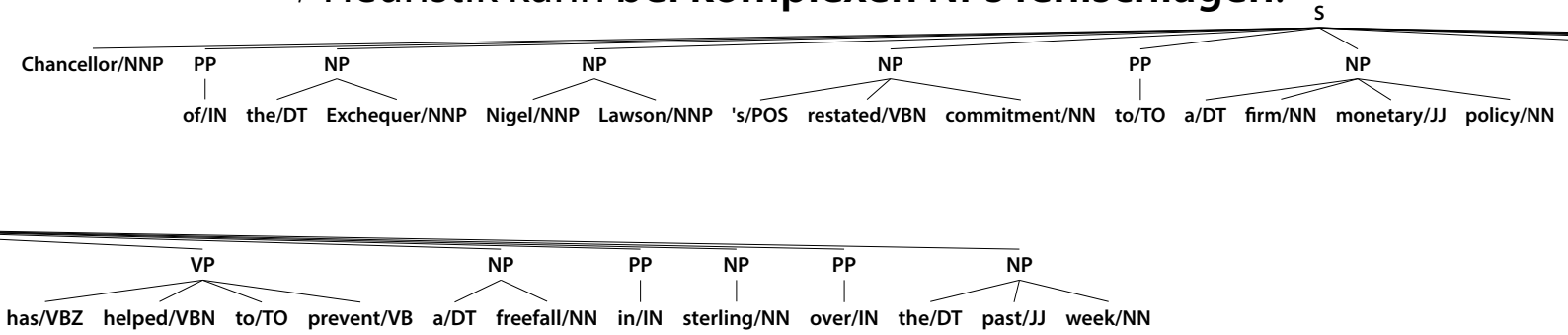


Abbildung 3: *Baum-Repräsentation von Chunks (http://www.nltk.org/images/chunk-treerep.png)*

Partielles vs. Vollständiges Parsing

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

- **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)*

```
#(S
#  Chancellor/NNP
#  (PP of/IN)
#  (NP the/DT Exchequer/NNP)
#  (NP Nigel/NNP Lawson/NNP)
#  (NP 's/POS restated/VBN commitment/NN)
#  (PP to/TO)
#  (NP a/DT firm/NN monetary/JJ policy/NN)
#  (VP has/VBZ helped/VBN to/TO prevent/VB)
#  (NP a/DT freefall/NN)
#  (PP in/IN)
#  (NP sterling/NN)
#  (PP over/IN)
#  (NP the/DT past/JJ week/NN)
#  ./.)
```

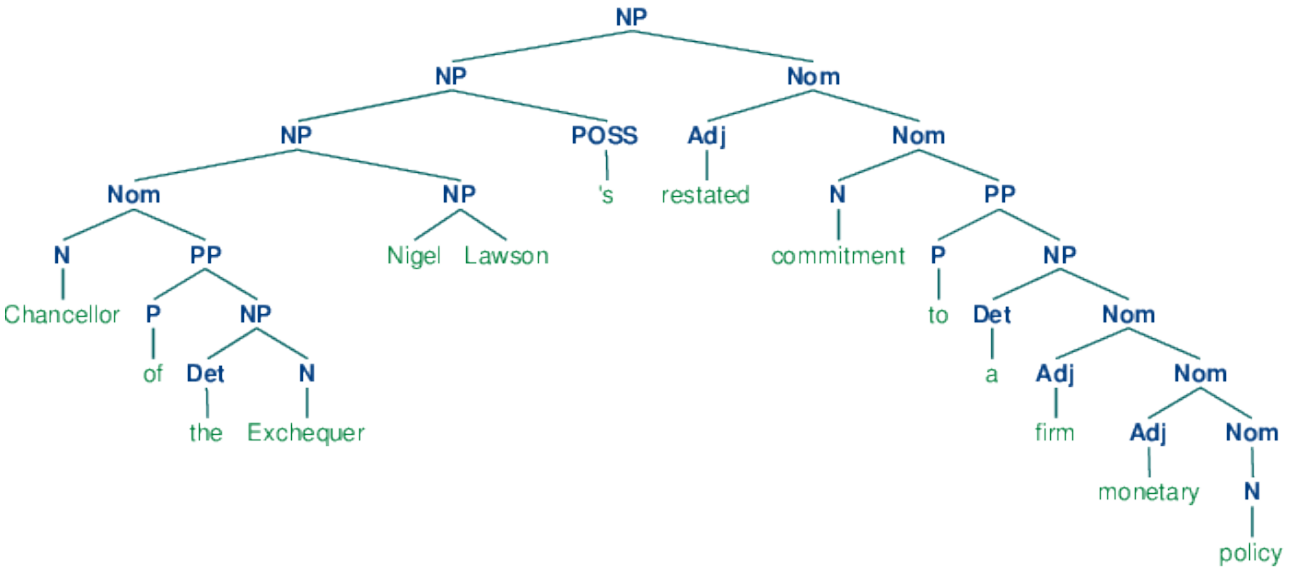


Abbildung 4: *Vollständige Phrasenstrukturanalyse der komplexen Subjekt-NP (http://www.nltk.org/book/tree_images/ch08-extras-tree-1.png)*

- in **hierarchischer Phrasenstrukturanalyse** wird die **komplexe NP unter einem NP-Knoten** zusammengefasst (statt in viele NP-Chunks aufgesplittet)
  $\rightarrow$ 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
  $\rightarrow$ Subjekt-NP unmittelbar dominiert von S
  $\rightarrow$ Objekt-NP unmittelbar dominiert von VP

13.1.1. 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*

```
grammar = r"""
    NP: {<DT|PP\$>?<JJ>*<NN>}
    {<NNP>+}
    """
sent = [("Rapunzel", "NNP"), ("let", "VBD"),
  ("down", "RP"), ("her", "PP$"), ("long",
  "JJ"), ("golden", "JJ"), ("hair", "NN")]
parser = nltk.RegexpParser(grammar)
tree = parser.parse(sent)
print(tree)
#(S
#  (NP Rapunzel/NNP)
#  let/VBD
#  down/RP
#  (NP her/PP$ 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**

- **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

```
grammar = r"""
    NP: {<DT|PP\$>?<JJ>*<NN>}   #chunk DET/POSS ADJ NOUN
    {<NN.?>+}                   #chunk sequences of NOUNS/PROPER NOUNS
    """
parser = nltk.RegexpParser(grammar, trace=1)
# Input:
 <NNP>  <VBD>  <RP>  <PP$>  <JJ>  <JJ>  <NN>
# chunk DET/POSS ADJ NOUN:
 <NNP>  <VBD>  <RP> {<PP$>  <JJ>  <JJ>  <NN>}
# chunk sequences of PROPER NOUNS:
{<NNP>} <VBD>  <RP> {<PP$>  <JJ>  <JJ>  <NN>}

```

```
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
       sequences of NOUNS/PROPER NOUNS
25     {<DT|PP\$>?<JJ>*<NN>}  #chunk DET/POSS ADJ
       NOUN
26     """
```

```
27 parser = nltk.RegexpParser(grammar, trace=1)
28 # Input:
29  <NNP>  <VBD>  <RP>  <PP$>  <JJ>  <JJ>  <NN>
30 # chunk sequences of PROPER NOUNS:
31 {<NNP>} <VBD>  <RP>  <PP$>  <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
    $\Rightarrow$ Chunk wird entsprechend **gesplittet**
  - wenn Chink-Muster **kompletter Chunk**
    $\Rightarrow$ Chunk wird **entfernt**
  - wenn Chink-Muster am **Anfang oder Ende** von Chunk
    $\Rightarrow$ Chunk wird entsprechend **beschnitten**

Auflistung 4: *Chinking mit NLTK*

```
grammar = r"""
  NP:
    {<.*>+}     # Chunk everything
    }<VBD|IN>+{ # Chink sequences of VBD and IN
  """
parser = nltk.RegexpParser(grammar)
tree = parser.parse(sent)
print(tree)
#(S
#  (NP the/DT little/JJ yellow/JJ dog/NN)
#  barked/VBD
#  at/IN
#  (NP the/DT cat/NN))
```

Split-Regeln

- **Split-Regeln**
  → **Anwendung auf gefundene Chunks**
  → z. B. **Splitten** von gefundener **NP am Determinierer**:
  → `<.*>}{<DT>`

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
       and NOUNs
6    """
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)
     chased/VBD)
12
```

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

13.1.2. 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*

```
grammar = r"""
  NP: {<DT|JJ|NN.*>+}          # Chunk
    sequences of DT, JJ, NN
  PP: {<IN><NP>}               # Chunk
    prepositions followed by NP
  VP: {<VB.*><NP|PP|CLAUSE>+$} # Chunk verbs
    and their arguments
  CLAUSE: {<NP><VP>}           # Chunk NP, VP
  """

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
   mat/NN)))))
20
21
22
23
```

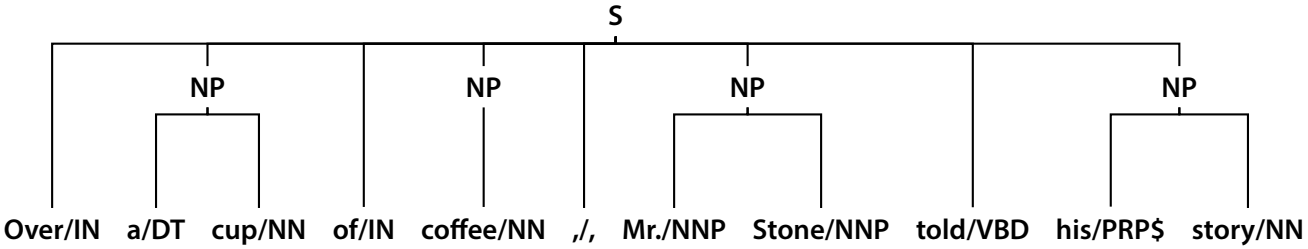
```
24 parser = nltk.RegexpParser(grammar, loop=2)
25 tree = parser.parse(sent)
26 print(tree)
27 #(S
28 #  (NP John/NNP)
29 #  thinks/VBZ
30 #  (CLAUSE
31 #    (NP Mary/NN)
32 #    (VP
33 #      saw/VBD
34 #      (CLAUSE
35 #        (NP the/DT cat/NN)
36 #        (VP sit/VB (PP on/IN (NP the/DT
   mat/NN))))))))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

13.1.3. Evaluation von Chunkern

- `conll2000`-**Korpus** im NLTK (270.000 Tokens)
  $\rightarrow$ **POS- und Chunk-getaggtes Korpus mit NPs, VPs, und PPs**
  $\rightarrow$ 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:** $R = \frac{(\text{Anzahl von korrekten Chunks in Chunker-Output})}{(\text{Anzahl aller Chunks in Referenz-Korpus})}$

  $\rightarrow$ Anteil der Chunks des Referenz-Korpus, die vom Chunker korrekt identifiziert wurden

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

  $\rightarrow$ Anteil der vom Chunker identifizierten Chunks, die korrekt sind

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

  $\rightarrow$ Kombination von Precision und Recall in einem Maß

  $\rightarrow \beta$: Parameter zur Gewichtung von P und R, gleichgewichtet:

  $F_1 = \frac{2PR}{P + R}$ (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%
```

13.1.4. Lernbasierte Chunking-Modelle

- **Training eines Klassifikators**, der die **Wörter auf Chunk-Klassen abbildet**
- Vorgehen **analog zu POS-Tagging** mit ***supervised machine-learning***
  $\rightarrow$ **POS-Tagging:** Abbildung **Token auf POS-Tag**
  $\rightarrow$ **Chunking:** Abbildung **Token-POS-Tupel auf IOB-Tag *(IOB-Tagging)***
  = **Sequenzklassifikation** ('*parsing as tagging*')
- für **Training** ist ein **Chunk-getaggtes Korpus im IOB-Format** notwendig, z. B. das `conll2000`**-Korpus im NLTK**

IOB-Format

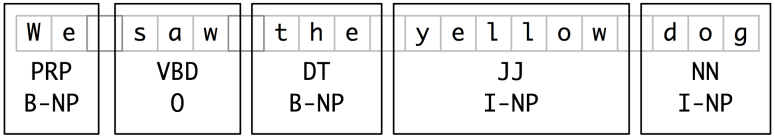


Abbildung 6: *IOB-Tags als Chunk-Tags (http://www.nltk.org/images/chunk-tagrep.png)*

- **IOB = 'Inside–Outside–Beginning'**
- IOB-Tag repräsentiert gleichzeitig **Segmentierung+Label**
- **wortweise** Auszeichnung von **verbundenen Sequenzen**
- **Ende eines Chunks implizit kodiert:**
  $\rightarrow$ **Übergang von** `I` **oder** `B` **zu** `B` (neuer Chunk) **oder** `O` (Outside)

Auflistung 9: *IOB-Format (Ausschnitt `conll2000`-Korpus)*

```
1  Mr. NNP B-NP
2  Meador NNP I-NP
3  had VBD B-VP
4  been VBN I-VP
5  executive JJ B-NP
6  vice NN I-NP
7  president NN I-NP
8  of IN B-PP
9  Balcor NNP B-NP
10 . . O
```

Chunking als *supervised*-Sequenzklassifikation

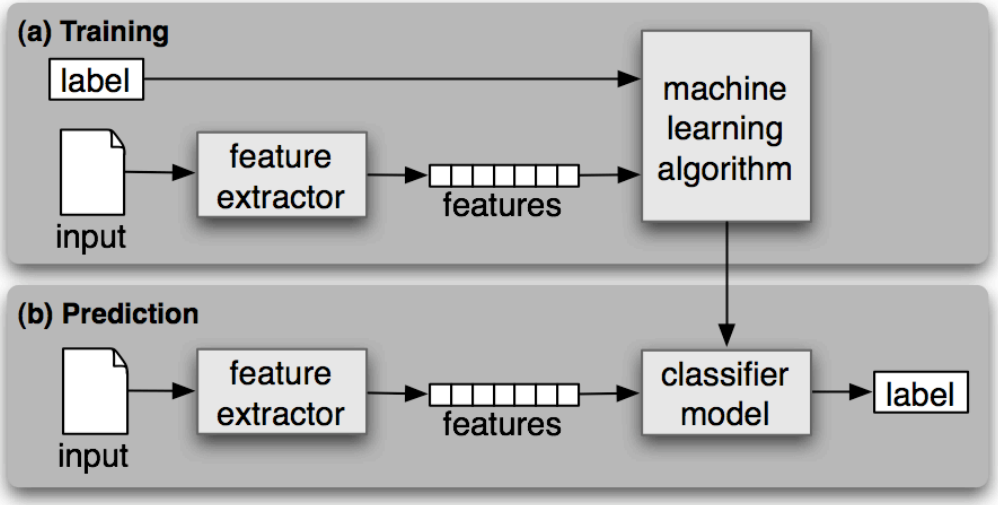


Abbildung 7: *Schema* supervised-*Klassifikation*
(http://www.nltk.org/images/supervised-classification.png)

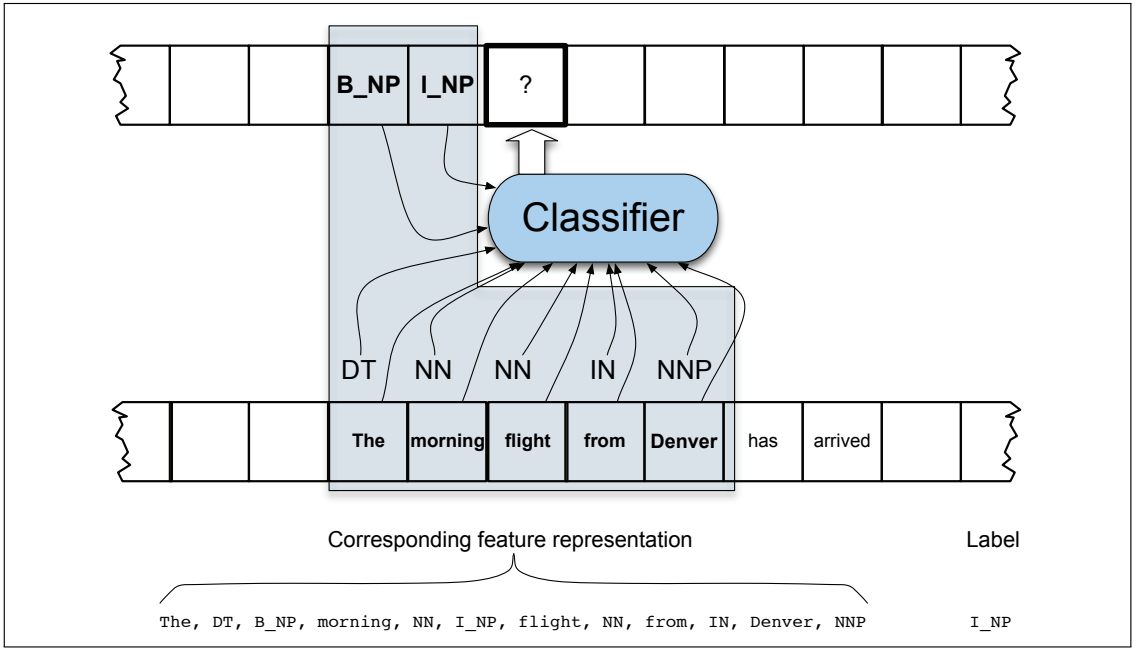


Abbildung 8: *Chunking als Sequenzklassifikation (momentaner Zustand: Tagging von* flight*)*
Input wird durch eine aus Kontextfenster extrahierte Feature-Menge repräsentiert (Abbildung nach: Speech and Language Processing. Daniel Jurafsky & James H. Martin. Draft of August 28, 2017, Figure 12.8, p. 12, https://web.stanford.edu/∼jurafsky/slp3/12.pdf)

NLTK: Unigram-Chunker

- berücksichtigt **nur POS-Tag des zu taggenden Wortes**
- ordnet jedem Token **basierend auf** seinem **POS-Tag das für dieses POS-Tag wahrscheinlichste IOB-Tag** zu
- `tree2conlltags`: transformiert Bäume zu `(word, tag, chunk)`-Tripel
- **Training:** Extraktion von `tag`, `chunk`-Tupeln (= `self.tagger`)
- **Tagging:** Extraktion der POS-Tags aus Input-Satz, Tagging der POS-Tags mit Chunk-Tags durch den trainierten `self.tagger` $\rightarrow$ Erstellung von `conll`-Tripel, Transformation in Baum

Auflistung 10: *NLTK: Definition UniGramChunker*

```
1  # Natural Language Toolkit:
     code_unigram_chunker
2  #http://www.nltk.org/book/ch07.html#code-unigram-ch
3
4  class UnigramChunker(nltk.ChunkParserI):
5      def __init__(self, train_sents): #
         [_code-unigram-chunker-constructor]
6          train_data = [[(t,c) for w,t,c in
             nltk.chunk.tree2conlltags(sent)]
7                        for sent in
                           train_sents]
8          self.tagger =
             nltk.UnigramTagger(train_data) #
             [_code-unigram-chunker-buildit]
9
```

```
10      def parse(self, sentence): #
          [_code-unigram-chunker-parse]
11          pos_tags = [pos for (word,pos) in
              sentence]
12          tagged_pos_tags =
              self.tagger.tag(pos_tags)
13          chunktags = [chunktag for (pos,
              chunktag) in tagged_pos_tags]
14          conlltags = [(word, pos, chunktag)
              for ((word,pos),chunktag)
15                        in zip(sentence,
                            chunktags)]
16          return
              nltk.chunk.conlltags2tree(conlltags)
```

Auflistung 11: *NLTK: Training und Evaluation UniGramChunker*

```
1 test_sents =
    conll2000.chunked_sents('test.txt',
    chunk_types=['NP'])
2 train_sents =
    conll2000.chunked_sents('train.txt',
    chunk_types=['NP'])
3 unigram_chunker = UnigramChunker(train_sents)
4 print(unigram_chunker.evaluate(test_sents))
5 ChunkParse score:
6     IOB Accuracy:  92.9%
7     Precision:     79.9%
8     Recall:        86.8%
9     F-Measure:     83.2%
```

Auflistung 12: *NLTK: Ausschnitt Prediction UniGramChunker*

```
postags = sorted(set(pos for sent in
  train_sents for (word,pos) in sent.leaves()))
print(unigram_chunker.tagger.tag(postags))

[('#', 'B-NP'), ('$', 'B-NP'), ("''", 'O'),
  ('(', 'O'), (')', 'O'),
 (',', 'O'), ('.', 'O'), (':', 'O'), ('CC',
   'O'), ('CD', 'I-NP'),
 ('DT', 'B-NP'), ('EX', 'B-NP'), ('FW',
   'I-NP'), ('IN', 'O'),
 ('JJ', 'I-NP'), ('JJR', 'B-NP'), ('JJS',
   'I-NP'), ('MD', 'O'),
 ('NN', 'I-NP'), ('NNP', 'I-NP'), ('NNPS',
   'I-NP'), ('NNS', 'I-NP'),
```

NLTK: ConsecutiveNPChunker

- Klassifikator mit **frei definierbarem *feature extractor***
  $\rightarrow$ verwendet **Maximum-Entropy-Klassifikator**
  (Sequenz-Klassifikationsalgorithmus)
- kann u.a. die **dem vorhergehenden Token zugewiesene Klasse** (IOB-Tag) berücksichtigen ($\Rightarrow$ **Kontext:** `history`)
- kann **Wortform berücksichtigen**, z. B: bei POS-Folge `DT NN NN`:

  *Joey/NN sold/VBD [**the/DT farmer/NN**] [rice/NN] ./.*

  *Nick/NN broke/VBD [**my/DT computer/NN monitor/NN**] ./.*

- mögliche **Merkmale für *feature-extraction***:
  - **Wortform und POS-Tag** des **aktuellen** Tokens
  - **Wortform und POS-Tag** der **vorhergehenden und folgenden** Tokens
  - bereits **zugewiesenes IOB-Tag** der vorhergehenden Tokens

Auflistung 13: *NLTK: Definition* feature extractor *für ConsecutiveNPChunker mit Berücksichtigung des aktuellen POS-Tag (entspricht UniGrammChunker)*

```
1  #http://www.nltk.org/book/pylisting/code_classifier_c
2
3  def npchunk_features(sentence, i, history):
4      word, pos = sentence[i]
5      return {"pos": pos}
6
7  chunker = ConsecutiveNPChunker(train_sents)
8  print(chunker.evaluate(test_sents))
9  ChunkParse score:
10     IOB Accuracy:  92.9%
11     Precision:     79.9%
12     Recall:        86.7%
13     F-Measure:     83.2%
```

Auflistung 14: *NLTK: Definition* feature extractor *für ConsecutiveNPChunker mit Berücksichtigung von POS-Bigrammen*

```
1  def npchunk_features(sentence, i, history):
2      word, pos = sentence[i]
3      if i == 0: prevword, prevpos = "<START>",
         "<START>"
4      else: prevword, prevpos = sentence[i-1]
5      return {"pos": pos,  "prevpos": prevpos}
6  chunker = ConsecutiveNPChunker(train_sents)
7  print(chunker.evaluate(test_sents))
8  ChunkParse score:
9      IOB Accuracy:  93.3%
10     Precision:     82.3%
11     Recall:        86.8%
12     F-Measure:     84.5%
```

13.2. Komplexität formaler und natürlicher Sprachen

13.2.1. Chomsky-Hierarchie

- Klassifizierung formaler Grammatiken bzgl. **Regeleinschränkung**
  $\rightarrow$ je **stärker** eine Grammatik **eingeschränkt** desto **geringer** ihre **Erzeugungsmächtigkeit**
  $\rightarrow$ 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):** $\alpha \rightarrow \beta$
  $\rightarrow$ ***ohne Einschränkung bzgl.*** $\alpha, \beta; \alpha, \beta \in Alphabet = \{T, NT\}$
- **kontext-sensitiv (Typ 1):** $\alpha \rightarrow \beta$**,** $length(\alpha) \leq length(\beta)$
  $\rightarrow$ bzw. auch: $lXr \rightarrow l\beta r$ ($X \in NT; l, r, \beta \in \{T, NT\}$)
- **kontext-frei (Typ 2):** $X \rightarrow \beta$ ($X \in NT; \beta \in \{T, NT\}$)
  $\rightarrow$ ***LHS: nur 1 Nicht-Terminal***
- **regulär (Typ 3):** $X \rightarrow a$**,** $X \rightarrow aY$ ($X, Y \in NT$**,** $a \in T$)
  $\rightarrow$ ***LHS: nur 1 Nicht-Terminal***
  $\rightarrow$ ***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
  $\rightarrow$ z. B. ***center-embedding*-Rekursion**
  $\rightarrow$ 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 **beschreibungsadäquatere Struktur**
  $\rightarrow$ **linguistisch adäquates Modell**
  $\rightarrow$ **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:** $X \rightarrow \alpha X \beta$
  $\rightarrow$ **rekursive Regel**: Nichtterminal erweitert zu selbem Nichtterminal, umgeben von Strings

- *center-embedding*-Regel ist **nicht-regulär**:
  $\rightarrow$ reguläre Grammatik: **nur links- oder rechtslineare Regeln**:
  $X \rightarrow Xa$ oder $X \rightarrow aX$
  $\rightarrow$ 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,

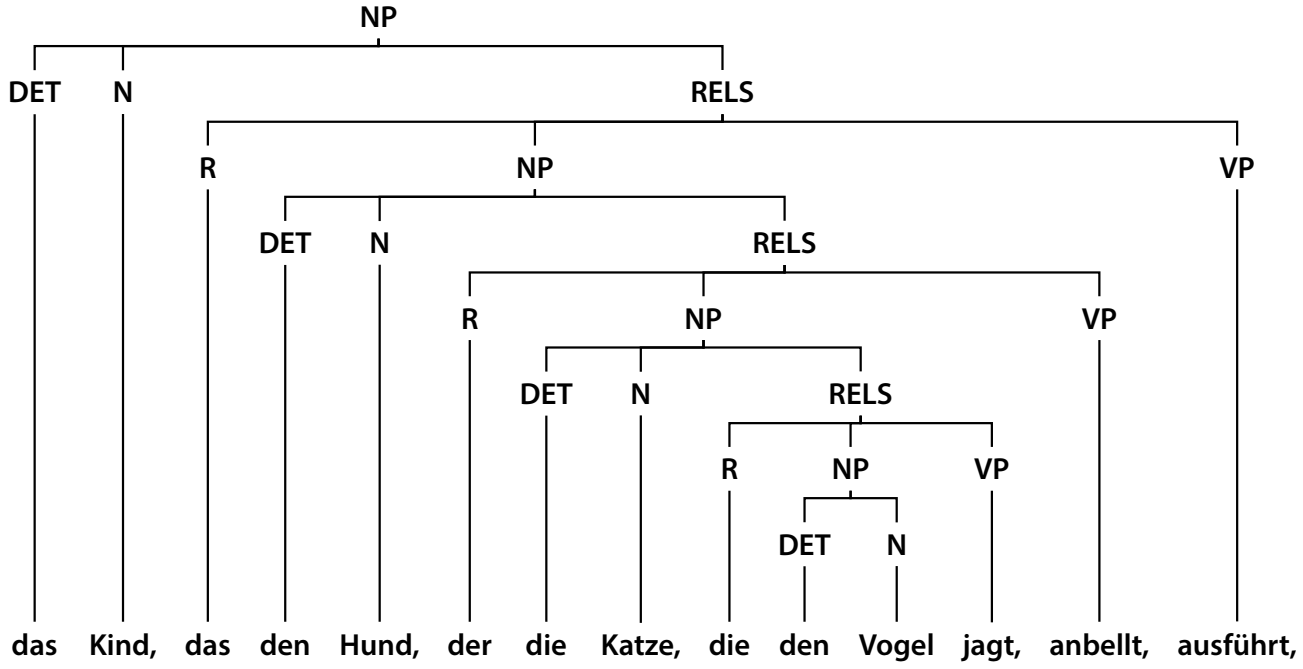
  $\rightarrow$ Schema: $N_1(N_2(N_3V_3)V_2)V_1$

  RELS $\rightarrow$ R **NP** VP

  NP $\rightarrow$ DET N ***RELS***

  Ableitungen: RELS $\rightarrow$ R DET N **RELS** VP

  RELS $\rightarrow$ R DET N **R DET N *RELS* VP** VP usw.


NP
DET
N
RELS
R
NP
VP
DET
N
RELS
R
NP
VP
DET
N
RELS
R
NP
VP
DET
N
das
Kind,
das
den
Hund,
der
die
Katze,
die
den
Vogel
jagt,
anbellt,
ausführt,

- psycholinguistische Experimente zeigen **Beschränkung in der Verarbeitung solcher Strukturen durch die menschliche Sprachverarbeitung** aufgrund von ***memory limitations***
  $\rightarrow$ 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/~fkarlsso/ceb5.pdf`)

13.2.3. *Cross-serial dependencies*

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

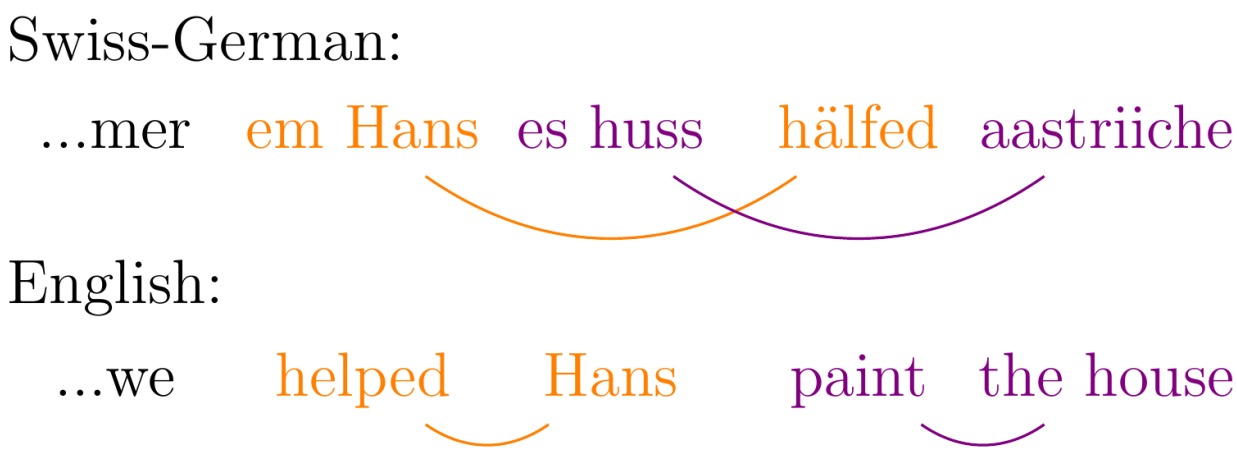


Abbildung 9: *Cross-serial dependencies (by Christian Nassif-Haynes - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=28304322)*

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**
  $\rightarrow$ Nachweis über **Pumping Lemma** für kontextfreie Sprachen

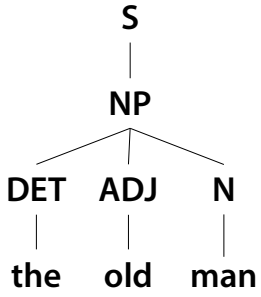
13.2.4. *Garden-path*-Sätze

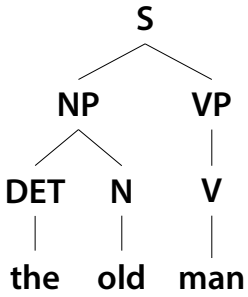
- Psycholinguistik: **Parser als Modell menschlicher Sprachverarbeitung**
- **Vergleich mit statistischen Sprachmodellen** gibt **Hinweis, dass auch menschliches Parsing Wahrscheinlichkeiten berücksichtigt**
  $\rightarrow$ **Disambiguierung** über statistische Informationen

- 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(*man*|N) > P(*man*|V), P(*old*|ADJ)>P(*old*|N)