Syntax natürlicher Sprachen

Tutorium

CFG-Parsing

Sarah Anna Uffelmann

17.11.2023

Analogie: Verben als Funktionen

```python

def intransitiv(verb, subjekt):
    return f'{subjekt} {verb}'

def transitiv(verb, subjekt, objekt):
    return f'{subjekt} {verb} {objekt}'

def transitiv_fakultativ(verb, subjekt, objekt="{}"):
    return f'{subjekt} {verb} {objekt}'

def ditransitiv(verb, subjekt, objekt, indirektes_objekt="{}"):
    return f'{subjekt} {verb} {indirektes_objekt} {objekt}'

print("intransitiv:\n", intransitiv("schläft", "er"))
print("\n")
print("transitiv:\n", transitiv("betrachtet", "er", "das Bild"))
print("\n")
print("transitiv, fakultatives Komplement weggelassen:\n", transitiv_fakultativ("sieht", "er"))
print("\n")
print("transitiv, fakultatives Komplement eingefügt:\n", transitiv_fakultativ("sieht", "er", "das Bild"))
print("\n")
print("ditransitiv, fakultatives Komplement weggelassen:\n", ditransitiv("übergibt", "er", "das Paket"))
print("\n")
print("ditransitiv, fakultatives Komplement eingefügt:\n", ditransitiv("übergibt", "er", "das Paket", "dem Kunden"))

```

Output:

```
intransitiv:
 er schläft


transitiv:
 er betrachtet das Bild


transitiv, fakultatives Komplement weggelassen:
 er sieht {}


transitiv, fakultatives Komplement eingefügt:
 er sieht das Bild


ditransitiv, fakultatives Komplement weggelassen:
 er übergibt {} das Paket


ditransitiv, fakultatives Komplement eingefügt:
 er übergibt dem Kunden das Paket
```

```python
# intransitiv: schlafen, lachen, gehen, stehen, gähnen, regnen ...
# transitiv und Komplement darf NICHT weggelassen werden: betrachten, beantworten, bearbeiten, ansehen, beantragen, begreifen ...
# transitiv und Komplement darf weggelassen werden (= fakultatives Komplement): sehen, essen, tanzen, helfen, einkaufen, fahren ...
# ditransitiv (indirektes Objekt ist fakultativ): geben, übergeben, schenken, bringen, vermitteln ...
```

Top-Down vs. Bottom-Up Parsing

Top-Down: Ausgehend vom Startsymbol wird versucht, mit Hilfe der Produktionsregeln den gegebenen Satz abzuleiten.

-> Recursive Descent
-> Earley

Bottom-Up: Ausgehend von den Terminalsymbolen wird versucht, diese zu größeren syntaktischen Einheiten zu verbinden, bis man beim Startsymbol angelangt ist.

-> Shift-Reduce

Recursive Decent Parser

2 Operationen: Predict & Scan

- Probiert jede anwendbare Regel aus
- Führt die Regel nicht zum Erfolg, nutzt der Parser Backtracking und probiert die nächste Regel aus, etc.

Recursive Decent Parser

S → NP VP
VP → VP PP
VP → V NP
NP → Det N
NP → NP PP
NP → Pron
PP → P NP

Pron → Er
V → sieht
N → Huhn
N → Fernglas
Det → das
Det → dem
P → mit

Beispielsatz: „Er sieht das Huhn mit dem Fernglas“

```
...
--> 549                 children = [cls.convert(child) for child in tree]
    550                 return cls(tree._label, children)
    551             else:

RecursionError: maximum recursion depth exceeded
```

Parser gerät in Endlosschleife
wegen der linksrekursiven Regeln

Recursive Decent Parser

S → NP VP
VP → VP PP
VP → V NP PP
VP → V NP
NP → Det N
NP → NP PP
NP → Det N PP
NP → Pron
PP → P NP

Pron → Er
V → sieht
N → Huhn
N → Fernglas
Det → das
Det → dem
P → mit

Beispielsatz: „Er sieht das Huhn mit dem Fernglas“

Ohne die linksrekursiven Regeln werden beide Parse-Bäume für den ambigen Satz gefunden.

Auszug aus dem Tracing-Output von NLTK:

```
Parsing 'er sieht das Huhn mit dem Fernglas'
Start:
    [ * S ]
Expand: S -> NP VP
    [ * NP VP ]
Expand: NP -> Det N PP
    [ * Det N PP VP ]
Expand: Det -> 'das'
    [ * 'das' N PP VP ]
Backtrack: 'er' match failed
Expand: Det -> 'dem'
    [ * 'dem' N PP VP ]
Backtrack: 'er' match failed
```

```
Expand: Det -> 'dem'
    [ * 'dem' N PP VP ]
Backtrack: 'er' match failed
Expand: NP -> Det N
    [ * Det N VP ]
Expand: Det -> 'das'
    [ * 'das' N VP ]
Backtrack: 'er' match failed
Expand: Det -> 'dem'
    [ * 'dem' N VP ]
Backtrack: 'er' match failed
```

```
Match: 'er'
    [ 'er' * VP ]
Expand: VP -> V NP PP
    [ 'er' * V NP PP ]
Expand: V -> 'sieht'
    [ 'er' * 'sieht' NP PP ]
Match: 'sieht'
    [ 'er' 'sieht' * NP PP ]
Expand: NP -> Det N PP
    [ 'er' 'sieht' * Det N PP PP ]
Expand: Det -> 'das'
    [ 'er' 'sieht' * 'das' N PP PP ]
```

Recursive Decent Parser

2 Operationen: Predict & Scan

- Probiert jede anwendbare Regel aus
- Führt die Regel nicht zum Erfolg, nutzt der Parser Backtracking und probiert die nächste Regel aus, etc.

Probleme:

- Wenn es für ein Nicht-Terminal viele verschiedene Produktionsregeln gibt, müssen im schlimmsten Fall alle diese Regeln ausprobiert werden (exponentieller Blow-Up).
- Dabei werden viele Teilstrukturen erzeugt, die gar nicht erfolgreich sein können
- Endlosschleife bei linksrekursiven Regeln!

Shift-Reduce Parser

2 Operationen: Shift & Reduce

Verwendet einen Stack (Stapel) und verschiebt eingelesene Wörter auf den Stapel, um sie auf passende Produktionsregeln zurückzuführen.

Shift-Reduce Parser

S → NP VP
VP → VP PP
VP → V NP
NP → Det N
NP → NP PP
NP → Pron
PP → P NP

Pron → Er
V → sieht
N → Huhn
N → Fernglas
Det → das
Det → dem
P → mit

Beispielsatz: „Er sieht das Huhn mit dem Fernglas“

Der Parser findet keine Ableitung.

Er findet zwar “Er sieht das Huhn“, versucht dann aber vergebens, die PP an S anzuhängen.

Tracing-Output von NLTK:

```
Parsing 'er sieht das Huhn mit dem Fernglas'
    [ * er sieht das Huhn mit dem Fernglas]
  S [ 'er' * sieht das Huhn mit dem Fernglas]
  R [ Pron * sieht das Huhn mit dem Fernglas]
  R [ NP * sieht das Huhn mit dem Fernglas]
  S [ NP 'sieht' * das Huhn mit dem Fernglas]
  R [ NP V * das Huhn mit dem Fernglas]
  S [ NP V 'das' * Huhn mit dem Fernglas]
  R [ NP V Det * Huhn mit dem Fernglas]
  S [ NP V Det 'Huhn' * mit dem Fernglas]
  R [ NP V Det N * mit dem Fernglas]
  R [ NP V NP * mit dem Fernglas]
  R [ NP VP * mit dem Fernglas]
  R [ S * mit dem Fernglas]
  S [ S 'mit' * dem Fernglas]
  R [ S P * dem Fernglas]
  S [ S P 'dem' * Fernglas]
  R [ S P Det * Fernglas]
  S [ S P Det 'Fernglas' * ]
  R [ S P Det N * ]
  R [ S P NP * ]
  R [ S PP * ]
```

Shift-Reduce Parser

S → NP VP
VP → VP PP
VP → V NP PP
#VP → V NP
NP → Det N
NP → NP PP
NP → Pron
PP → P NP

Pron → Er
V → sieht
N → Huhn
N → Fernglas
Det → das
Det → dem
P → mit

Beispielsatz: „Er sieht das Huhn mit dem Fernglas“

Der Parser findet den Parse-Baum mit VP-Attachment nur dann, wenn wir die Regel VP -> V NP PP
als einzige VP-Regel definieren.

Der Parse-Baum mit NP-Attachment wird nicht gefunden!

Shift-Reduce Parser

S → NP VP
#VP → VP PP
#VP → V NP PP
VP → V NP
NP → Det N PP
#NP → NP PP
NP → Pron
PP → P Det N

Pron → Er
V → sieht
N → Huhn
N → Fernglas
Det → das
Det → dem
P → mit

Beispielsatz: „Er sieht das Huhn mit dem Fernglas“

Der Parser findet den Parse-Baum mit NP-Attachment nur dann, wenn wir die Regel NP -> Det N PP
als einzige NP-Regel (neben NP -> Pron) definieren und gleichzeitig die PP-Regel entsprechend anpassen.
Außerdem darf es keine weitere VP-Regel neben VP -> V NP geben.

Der Parse-Baum mit VP-Attachment wird nicht gefunden!

Fazit: Der Parser kann mit einer Grammatik für Sätze mit PP-Attachment-Ambiguität sehr schlecht umgehen.

Shift-Reduce Parser

Beispielsatz: „the old man the boat“

S → NP VP
VP → V NP
NP → Det N
NP → Det ADJP N
ADJP → ADJ

Det → the
ADJ → old
N → man
N → boat
N → old
V → man

Der Parser findet keine Ableitung.

Tracing-Output von NLTK:

```
Parsing 'the old man the boat'
    [ * the old man the boat]
  S [ 'the' * old man the boat]
  R [ Det * old man the boat]
  S [ Det 'old' * man the boat]
  R [ Det ADJ * man the boat]
  R [ Det ADJP * man the boat]
  S [ Det ADJP 'man' * the boat]
  R [ Det ADJP N * the boat]
  R [ NP * the boat]
  S [ NP 'the' * boat]
  R [ NP Det * boat]
  S [ NP Det 'boat' * ]
  R [ NP Det N * ]
  R [ NP NP * ]
```

Shift-Reduce Parser

Beispielsatz: „the old man the boat“

S → NP VP
VP → V NP
NP → Det N
NP → Det ADJP N
ADJP → ADJ

Det → the
N → old
V → man
ADJ → old
N → man
N → boat

Wenn wir die Reihenfolge der lexikalischen Regeln entsprechend ändern, findet der Parser eine Ableitung.

Tracing-Output von NLTK:

```
Parsing 'the old man the boat'
    [ * the old man the boat]
  S [ 'the' * old man the boat]
  R [ Det * old man the boat]
  S [ Det 'old' * man the boat]
  R [ Det N * man the boat]
  R [ NP * man the boat]
  S [ NP 'man' * the boat]
  R [ NP V * the boat]
  S [ NP V 'the' * boat]
  R [ NP V Det * boat]
  S [ NP V Det 'boat' * ]
  R [ NP V Det N * ]
  R [ NP V NP * ]
  R [ NP VP * ]
  R [ S * ]
```

Shift-Reduce Parser

2 Operationen: Shift & Reduce

Verwendet einen Stack (Stapel) und verschiebt eingelesene Wörter auf den Stapel, um sie auf passende Produktionsregeln zurückzuführen.

Effizienter als ein Top-Down Parser, da er vom Eingabesatz ausgeht.

Probleme:

- kann Teilstrukturen erzeugen, die zu keinem Ergebnis führen
- benötigt daher Backtracking (ist aber nicht immer implementiert)
- Probleme bei PP-Attachment-Ambiguität
- Probleme mit temporaler und lexikalischer Ambiguität, wenn kein Backtracking verwendet wird. Der Eingabesatz wird dann evtl. nicht erkannt, obwohl er ableitbar ist.

Earley Parser

3 Operationen: Scan, Predict & Complete

- Vermeidet doppelte Berechnungen durch dynamisches Programmieren
- Zwischenergebnisse werden in einem Chart gespeichert: Chart Parser

Earley Parser

S → NP VP
VP → VP PP
VP → V NP
NP → Det N
NP → NP PP
NP → Pron
PP → P NP

Pron → Er
V → sieht
N → Huhn
N → Fernglas
Det → das
Det → dem
P → mit

Beispielsatz: „Er sieht das Huhn mit dem Fernglas“

Der Parser findet beide Parse-Bäume für den ambigen Satz.

Auszug aus dem Tracing-Output von NLTK:

```
Top Down Init Rule:
|>    .    .    .    .    .    .    .| [0:0] S  -> * NP VP

* Processing queue: 0

Predictor Rule:
|>    .    .    .    .    .    .    .| [0:0] NP -> * NP PP
|>    .    .    .    .    .    .    .| [0:0] NP -> * Det N
|>    .    .    .    .    .    .    .| [0:0] NP -> * Pron
Predictor Rule:
|>    .    .    .    .    .    .    .| [0:0] Pron -> * 'er'

* Processing queue: 1

Scanner Rule:
|[----]    .    .    .    .    .    .| [0:1] Pron -> 'er' *
Completer Rule:
|[----]    .    .    .    .    .    .| [0:1] NP -> Pron *
Completer Rule:
|[----->    .    .    .    .    .    .| [0:1] S  -> NP * VP
|[----->    .    .    .    .    .    .| [0:1] NP -> NP * PP
Predictor Rule:
|.    >    .    .    .    .    .    .| [1:1] PP -> * P NP
Predictor Rule:
|.    >    .    .    .    .    .    .| [1:1] VP -> * VP PP
|.    >    .    .    .    .    .    .| [1:1] VP -> * V NP
Predictor Rule:
|.    >    .    .    .    .    .    .| [1:1] V  -> * 'sieht'
```

Manuelles Parsen mit Earley

0	Er 1	sieht 2	das 3	Huhn 4	mit 5	dem 6	Fernglas 7
S -> • NP VP NP -> • Det N NP -> • NP PP NP -> • Pron Det -> • das Det -> • dem Pron -> • Er	Pron -> Er • NP -> Pron • NP -> NP • PP S -> NP • VP			S -> NP VP •			S -> NP VP •
SCAN PREDICT COMPLETE	PP -> • P NP P -> • mit VP -> • VP PP VP -> • V NP V -> • sieht	V -> sieht • VP -> V • NP		VP -> V NP • VP -> VP • PP			VP -> VP PP • VP -> V NP • VP -> VP • PP
		NP -> • Det N NP -> • NP PP NP -> • Pron Det -> • das Det -> • dem Pron -> • Er	Det -> das • NP -> Det • N	NP -> Det N • NP -> NP • PP			NP -> NP PP • NP -> NP • PP
			N -> • Huhn N -> • Fernglas	N -> Huhn •			
				PP -> • P NP P -> • mit	P -> mit • PP -> P • NP		PP -> P NP •
					NP -> • Det N NP -> • NP PP NP -> • Pron Det -> • das Det -> • dem Pron -> • Er	Det -> dem • NP -> Det • N	NP -> Det N • NP -> NP • PP
						N -> • Huhn N -> • Fernglas	N -> Fernglas •
							PP -> • P NP P -> • mit

S → NP VP
VP → VP PP
VP → V NP
NP → Det N
NP → NP PP
NP → Pron
PP → P NP

Pron → Er
V → sieht
N → Huhn
N → Fernglas
Det → das
Det → dem
P → mit

Earley Parser

3 Operationen: Scan, Predict & Complete

- Vermeidet doppelte Berechnungen durch dynamisches Programmieren
- Zwischenergebnisse werden in einem Chart gespeichert: Chart Parser

Vorteile und Nachteile:

- komplizierter als Recursive Descent und Shift-Reduce
- dafür aber viel schneller
- benötigt kein Backtracking und erzeugt keine unnötigen Teilstrukturen

Fazit:

Der Earley Parser (oder generell ein Chart Parser) ist am besten für unsere Zwecke geeignet, denn er kann am besten mit Rekursion und Ambiguität umgehen.

NLKT-Parser

Recursive Decent:

```
parser = nltk.RecursiveDescentParser(grammar, trace=0)
```

Shift-Reduce:

```
parser = nltk.ShiftReduceParser(grammar, trace=0)
```

Earley:

```
parser = nltk.EarleyChartParser(grammar, trace=0)
```

Tracing-Output:

Defaultwert = 0, d.h. kein Tracing Output
Je höher der Wert, desto ausführlicher das Tracing Output.

Nachzulesen bei NLTK-Doku, z.B. für Recursive Decent Parser:
https://www.nltk.org/_modules/nltk/parse/recursivedescent.html