Programmierkurs Python II

Michaela Regneri & Stefan Thater
FR 4.7 Allgemeine Linguistik (Computerlinguistik)
Universität des Saarlandes

Sommersemester 2010

Übersicht

- Kurze Wiederholung:
  - Probleme mit elementaren Parsing-Strategien
- Der CYK-Algorithmus:
  - Parsing als dynamische Programmierung
  - Charts als kompakte Repräsentatation von Teilergebnissen
  - Der Algorithmus: Erkenner, Parser
- Earleys Algorithmus

Probleme mit elementaren Parsing-Strategien

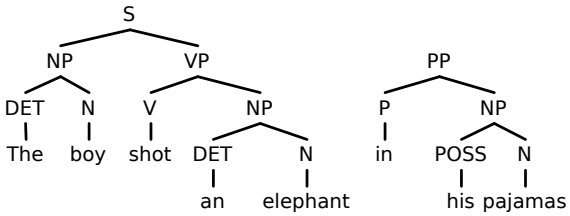
- Nicht auf allgemeine Grammatiken anwendbar
  - keine Tilgungsregeln, keine zyklischen Kettenregeln (BU)
  - keine linksrekursiven Regeln (TD)
- Lokale Ambiguität ⇒ Suche & Backtracking
  - Identische Teilergebnisse werden u.U. mehrfach berechnet
  - ⇒ Laufzeit exponentiell in der Eingabelänge (worst case)
- Lokale Ambiguität: Welche Regel bzw. Operation muss angewendet werden?

Beispielgrammatik

S → NP VP	DET → *an*
NP → DET N	DET → *the*
NP → POSS N	POSS → his
NP → NP PP	N → *elephant*
PP → P NP	N → *pajamas*
VP → V NP	N → *boy*
VP → VP PP	V → *shot*
	P → *in*

The boy shot an elephant in …

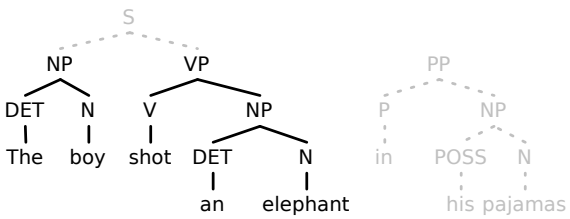
⟨[], [the boy shot an elephant in his pajamas]⟩
⇒* ⟨[NP VP], [in his pajamas]⟩
⇒ ⟨[S], [in his pajamas]⟩
⇒* ⟨[S PP], []⟩ ⇒ Backtracking



(Shift-Reduce)

The boy shot an elephant in …

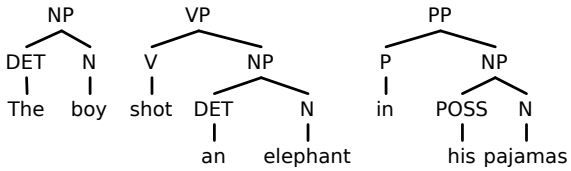
⟨[], [the boy shot an elephant in his pajamas]⟩
⇒* ⟨[NP VP], [in his pajamas]⟩



(Shift-Reduce)

The boy shot an elephant in …

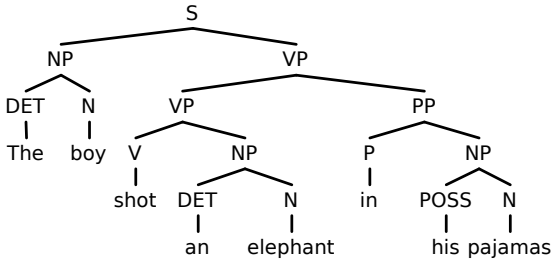
⟨[], [the boy shot an elephant in his pajamas]⟩
⇒* ⟨[NP VP], [in his pajamas]⟩
⇒* ⟨[NP VP PP], []⟩



(Shift-Reduce)

The boy shot an elephant in …

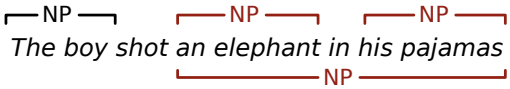
⟨[], [the boy shot an elephant in his pajamas]⟩
⇒* ⟨[NP VP], [in his pajamas]⟩
⇒* ⟨[NP VP PP], []⟩
⇒* ⟨[NP VP], []⟩
⇒* ⟨[S], []⟩



(Shift-Reduce)

Dynamische Programmierung

- Kontextfreie Grammatiken: die Anwendbarkeit einer Regel in einer Ableitung ist unabhängig vom Kontext.



- Chart-Parsing: Speichere bereits analysierte Teilergebnisse (Konstituenten) in einer „Chart.“
- Charts sind kompakte Repräsentation aller (lokal) möglichen Teilkonstituenten der Eingabekette.

Chart-Parsing

- Chart-Parsing: speichere bereits analysierte Teilergebnisse (Konstituenten) in einer „Chart.“
  - Charts aka. „well-formed substring table“
- Charts können enthalten:
  - Konstituenten, die bereits gefunden wurden
  - Hypothesen darüber, welche Konstituenten gefunden werden können (⇒ Earley).
- Verschiedene Chart-Parser:
  - CYK, Earley, Bottom-up chart parser, …

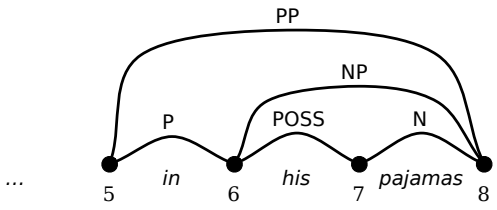
Charts als Matrix

1	DET							
2	NP	N						
3	∅	∅	V					
4	∅	∅	∅	DET				
5	S	∅	VP	NP	N			
6	∅	∅	∅	∅	∅	P		
7	∅	∅	∅	∅	∅	∅	POSS	
8	S	∅	VP	NP	∅	PP	NP	N
	0	1	2	3	4	5	6	7
	0 *The*	1 boy	2 *shot*	3 *an*	4 *elephant*	5 *in*	6 *his*	7 *pajamas* 8

A ∈ T[i, j] gdw. A ⇒* w_{i+1} … w_j

Charts als Graph

- Ecken repräsentieren Positionen in der Eingabekette
- Kanten zwischen zwei Knoten repräsentieren Teilketten der Eingabe
  - Kante $n_i \rightarrow n_j$ gdw. A ⇒* w_{i+1}… w_j



CYK Algorithmus

- CYK (Cocke, Younger, Kasami) ist ein einfacher, chart-basierter bottom-up Parser.
- Die Grammatik muss in Chomsky-Normalform vorliegen:
  - $A \to w$ (w Terminalsymbol)
  - $A \to B\ C$ (B und C Nichtterminalsymbole)
  - $S \to \varepsilon$ (S Startsymbol, nur wenn $\varepsilon \in L$)
- Anmerkung: hier nehmen wir an, dass $\varepsilon \notin L$, die Grammatik enthält also keine Regel $S \to \varepsilon$

CYK (Erkenner)

```
CYK(G, w₁ ... wₙ):
  for i in 1 ... n:
    T[i-1, i] = { A | A → wᵢ ∈ R }
    for j in i - 2 ... 0:
      T[j, i] = ∅
      for k in j + 1 ... i - 1:
        T[j, i] = T[j, i] ∪
                { A | A → B C, B ∈ T[j,k], C ∈ T[k, i] }
  return S ∈ T[0, n]
```


Eigenschaften

- Korrekt:
  Wenn $S \in T[0, n]$, dann $S \Rightarrow^* w_1 \ldots w_n$
- Vollständig:
  Wenn $S \Rightarrow^* w_1 \ldots w_n$, dann $S \in T[0, n]$
- Laufzeit:
  Polynomiell in der Eingabelänge: $O(n^3)$

Variante (Kettenregeln)

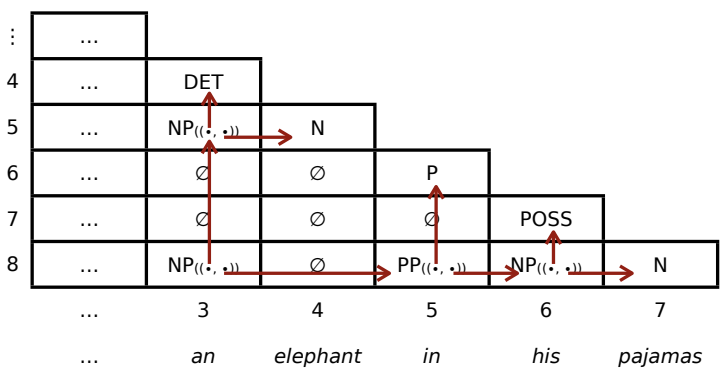
Closure(A) = { B | B ⇒* A }

```
CYK(G, w1 ... wn):
  for i in 1 ... n:
    T[i-1, i] = { Closure(A) | A → wi ∈ R }
    for j in i - 2 ... 0:
      T[j, i] = ∅
      for k in j + 1 ... i - 1:
        T[j, i] = T[j, i] ∪
          { Closure(A) |A → B C, B ∈ T[j,k], C ∈ T[k,i]}
  return S ∈ T[0, n]
```


Erkenner → Parser

- Speichere zu jeder Kategorie A in der Chart eine Liste von Listen mit Verweisen auf Einträge in der Chart, die verwendet wurden, um A abzuleiten.
- Liste von Listen nötig, da Teilketten mehrere Ableitungsbäume haben können

CYK (Parser)



Grammatik ⇒ CNF

```
Linksbinarisierung(G):
   while G enthält Regel A → A1 A2 A3 ... Ak, k ≥ 3
      entferne die Regel aus G
      neue Regel: ⟨A1, ...,Ak-1⟩ → A1 ... Ak-1
      neue Regel: A → ⟨A1, ...,Ak-1⟩ Ak

Rechtsbinarisierung(G):
   while G enthält Regel A → A1 A2 A3 ... Ak, k ≥ 3
      entferne die Regel aus G
      neue Regel: ⟨A2, ...,Ak⟩ → A2 ... Ak
      neue Regel: A → A1 ⟨A2, ...,Ak⟩
```


Grammatik ⇒ CNF

- Anmerkung: mit „CNF“ sind hier Grammatiken mit maximal zwei Nichtterminalen auf der rechten Regelseite gemeint (Kettenregeln sind also erlaubt).
  - A → w
  - A → B
  - A → B C

Implementierungsvarianten

- T[i,j] = T[i,j] ∪ { A | A → B C, B ∈ T[i,k], C ∈ T[k,j] }
  - ⇒ kann verschieden implementiert werden
- Variante 1
  - Iteriere über alle Regeln A → B C
  - Prüfe, ob B ∈ T[i,k] und C ∈ T[k,j]
- Variante 2
  - Iteriere über alle B ∈ T[i,k]
  - Iteriere über alle Regeln A → B C
  - Prüfe, ob C ∈ T[k, j]

Implementierungsvarianten

- T[i,j] = T[i,j] ∪ { A | A → B C, B ∈ T[i,k], C ∈ T[k,j] }
  - ⇒ kann verschieden implementiert werden
- Variante 3
  - Iteriere über alle C ∈ T[k,j]
  - Iteriere über alle Regeln A → B C
  - Prüfe, ob A ∈ T[i,k]
- Variante 4
  - Iteriere über alle B ∈ T[i,k] und C ∈ T[k,j]
  - Prüfe, ob es Regel A → B C gibt

Song &al. (2008)

- Experimente mit CYK & Wall Street Journal
- Laufzeit abhängig von Binarisierungsmethode
  - Rechtsbinarisierung ⇒ Variante 3 am effizientesten
  - Linksbinarisierung ⇒ Variante 2 am effizientesten

CYK

- CYK ist eingeschränkt auf Grammatiken in CNF
  - keine echte Einschränkung, Grammatiken können in CNF überführt werden
- CYK ist ein Bottom-Up-Parser und erzeugt unter Umständen viele nicht benötigte Konstituenten
- *Mary gave the man a book*
  - N → man
  - V → man ⇒ CYK leitet für „man a book“ eine VP ab

Earleys Algorithmus

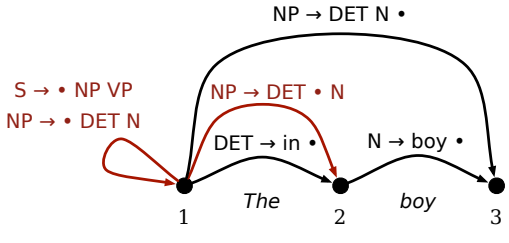
- Ein effizienter „aktiver“ Chart-Parser
  - Zeikomplexität $O(n^3)$ in der Eingabelänge n
- Für beliebige kontextfreie Grammatiken geeignet
  - Tilgungsregeln
  - Zyklische Kettenregeln
  - Links- und rechtsrekursive Regeln
- Mischung aus bottom-up und top-down
  - Analyserichtung links → rechts

Earleys Algorithmus

- Passive Chart-Parser speichern nur vollständige Konstitutenten als Teilergebnis in der Chart.
- Aktive Chart-Parser speichern auch Hypothesen.
- A → **B_1 … B_n** • **C_{n+1} … C_k**
  - **B_1 … B_2** wurden bereits gefunden
  - **C_{n+1} … C_k** müssen noch gefunden werden

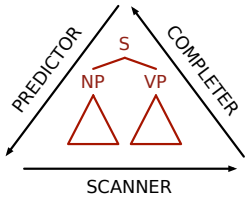
Die Chart

- **Ecken** („Knoten“) entsprechen Positionen in der Eingabe
- **Kanten** entsprechen gefundenen Teilkonstituenten. Auch unvollständige Konstituenten werden gespeichert.
  - Aktive Kanten ⟨A → α • B β, i, j⟩
  - Passive Kanten ⟨A → α •, i, j⟩



Earleys Algorithmus

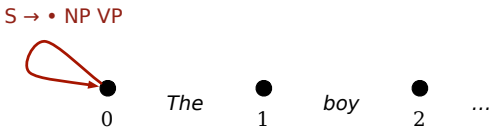
- **Drei Prozeduren**
  - Predictor
  - Scanner
  - Completer



- Die drei Prozeduren werden für jeden Knoten in der Chart von links nach rechts aufgerufen.
- ⇒ jeweils solange, bis keine neuen Kanten zur Chart hinzugefügt werden

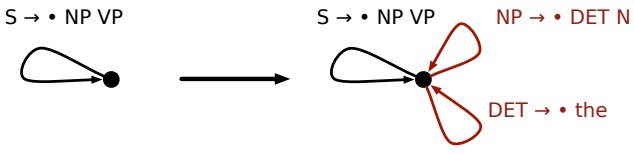
Initialisierung

- Grammatik G = ⟨V, Σ, R, S⟩
- Eingabe $w_1 \ldots w_n$
- Für jede Regel S → α
  - Füge ⟨S → • α, 0, 0⟩ zur Chart hinzu



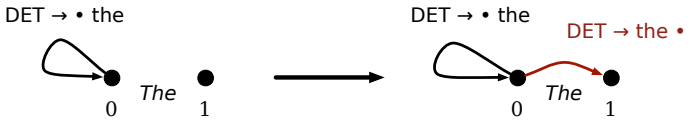
Predictor

- Der Predictor sagt vorraus, welche Grammatikregeln zum Ziel führen könnten (top-down).
- Für neue Kanten ⟨A → α • B β, i, j⟩
  - Für alle Regeln B → γ ∈ Regeln
    - Neue Kante: ⟨B → • γ, j, j⟩



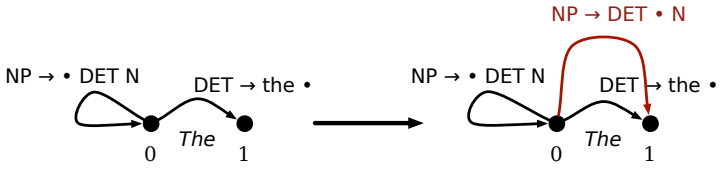
Scanner

- Der Scanner konsumiert Terminalsymbole aus der Eingabekette.
- Für neue Kanten $\langle A \rightarrow \alpha \bullet w_{j+1}\ \beta, i, j\rangle$
  - Neue Kante $\langle A \rightarrow \alpha\ w_{j+1} \bullet \beta, i, j+1\rangle$



Completer

- Der Completer vervollständigt aktive Kanten durch Kombination mit inaktiven Kanten (bottom-up).
- Für jede neue Kante $\langle A \rightarrow \alpha \bullet, j, k\rangle$
  - Für jede alte Kante $\langle B \rightarrow \gamma \bullet A\ \beta, i, j\rangle$
    - Neue Kante: $\langle B \rightarrow \gamma\ A \bullet \beta, i, k\rangle$



Übersicht

- **Scan**
  - $\langle A \rightarrow \alpha \bullet w_{j+1}\ \beta, i, j\rangle \Rightarrow \langle A \rightarrow \alpha\ w_{j+1} \bullet \beta, i, j+1\rangle$
- **Predict**
  - $\langle A \rightarrow \alpha \bullet B\ \beta, i, j\rangle, B \rightarrow \gamma \Rightarrow \langle B \rightarrow \bullet\ \gamma, j, j\rangle$
- **Complete**
  - $\langle A \rightarrow \alpha \bullet B\ \beta, i, j\rangle, \langle B \rightarrow \gamma \bullet, j, k\rangle \Rightarrow \langle A \rightarrow \alpha\ B \bullet \beta, i, k\rangle$

Der Algorithmus

- Neue Kante ⟨START → • S, 0, 0⟩
- Für alle j ∈ 1, …, n
  - Für alle ⟨A → α • β, i, j⟩ ∈ Chart
    - Scan, wenn β mit einem Terminalsymbol beginnt
    - Predict, wenn β mit einem Nichtterminal beginnt
    - Complete, wenn β = ε
- Akzeptiere, wenn ⟨START → S •, 0, n⟩

Variante

- Im hier vorgestellten Verfahren wird nicht zwischen Lexikon und „eigentlichen“ Regeln unterschieden.
- Regeln wie „DET → the“ werden vom Predictor behandelt
  - ⇒ es können Terminalsymbole vorhergesagt werden, die gar nicht in der EIngabekette vorkommen (ineffizient)
- **Sinnvolle Variante:**
  - Regeln wie „DET → the“ vom Scanner verarbeiten lassen:
  - Für neue Kanten ⟨A → α • B β, i, j⟩
    - Neue Kante ⟨A → α B • β, i, j + 1⟩ wenn B → w_{i+1}