Programmierkurs Python II

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

Sommersemester 2013

Übersicht

- Wiederholung: Elementare Parsing-Strategien
- CYK-Parser:
  - Parsing als dynamische Programmierung
  - Charts als kompakte Repräsentatation von Teilergebnissen
  - Der Algorithmus: Erkenner & Parser
- Earley-Parser
  - ...kennt Ihr schon. :)

Probleme

- Die elementaren Parsing-Strategien (top-down, bottom-up) sind nicht auf allgemeine Grammatiken anwendbar
  - keine Tilgungsregeln, keine zyklischen Kettenregeln (BU)
  - keine (links-) rekursiven 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*

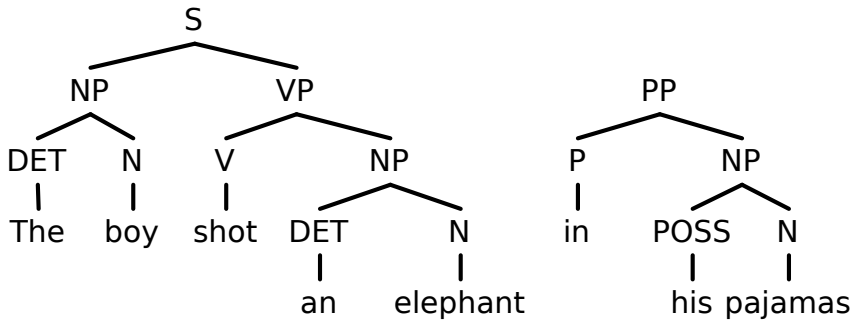
The boy shot an elephant in …

⟨[], [the boy shot an elephant in his pajamas]⟩

⇒* ⟨[NP VP], [in his pajamas]⟩

⇒ ⟨[S], [in his pajamas]⟩

⇒* ⟨[S PP], []⟩ ⇒ Backtracking

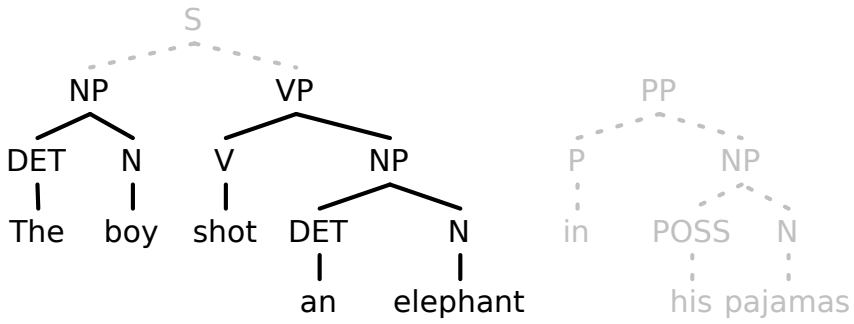


(Shift-Reduce)

The boy shot an elephant in …

⟨[], [the boy shot an elephant in his pajamas]⟩

⇒* ⟨[NP VP], [in his pajamas]⟩



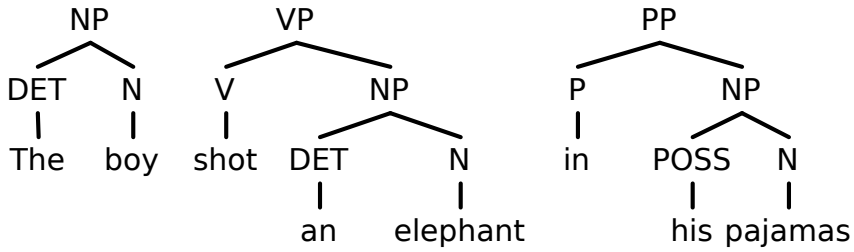
(Shift-Reduce)

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 …

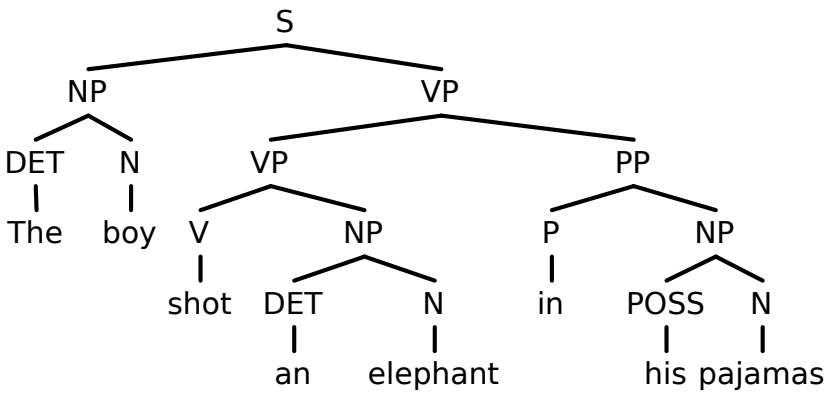
⟨[], [the boy shot an elephant in his pajamas]⟩

⇒* ⟨[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 (z.B. Earley-Algorithmus).
- Verschiedene Chart-Parser:
  - CYK, Earley, …

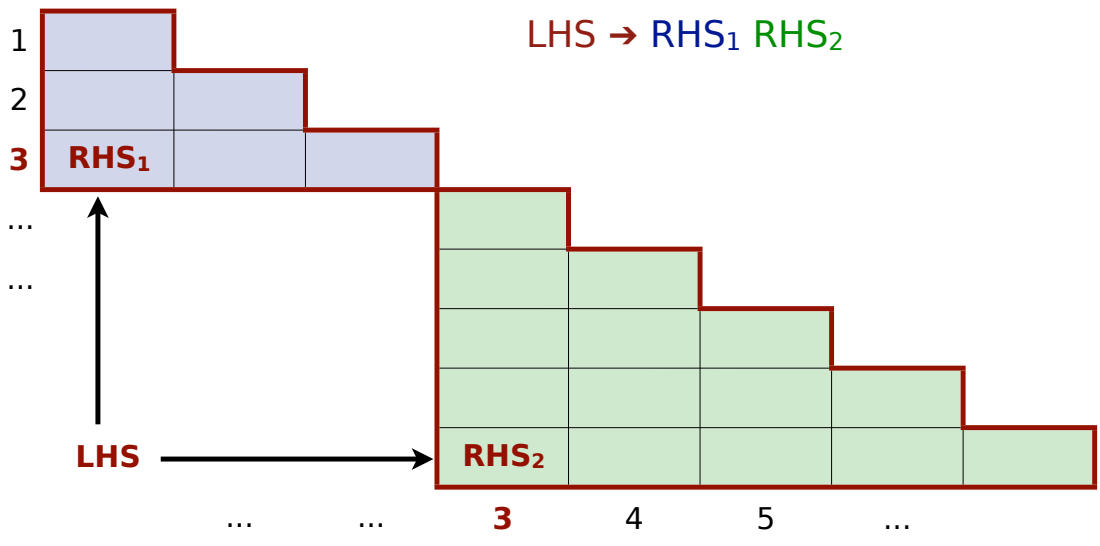
Charts

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

$A \in T[i, j]$ gdw. $A \Rightarrow^* w_{i+1} \ldots w_j$

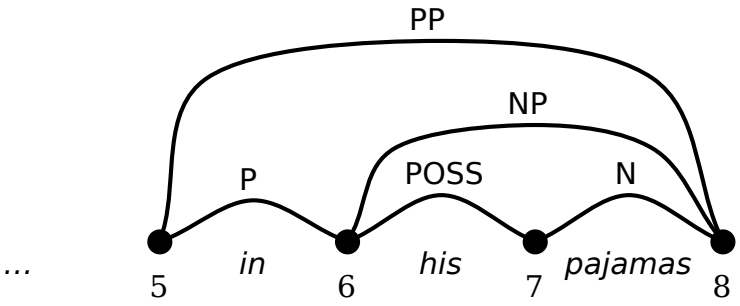
0 *The* 1 boy 2 *shot* 3 *an* 4 *elephant* 5 *in* 6 *his* 7 *pajamas* 8

Charts



Charts als Graph

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



CYK Algorithmus

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

Grundlegende Idee

- Wenn
  - $B \Rightarrow^* w_i \ldots w_{j-1}$
  - $C \Rightarrow^* w_j \ldots w_{k-1}$
  - $A \rightarrow B\ C$
- Dann
  - $A \Rightarrow^* w_i \ldots w_{k-1}$

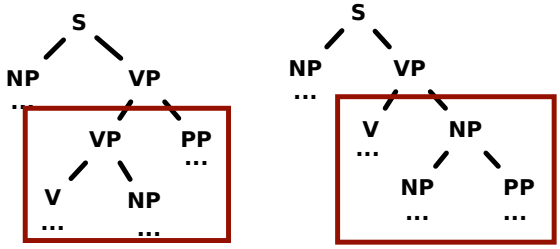
CYK (Erkenner, Pseudo-code)

```
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] }
  if S ∈ T[0, n] then return True else return False
```

Erkenner → Parser

- Speichere zu jedem Nichtterminal NT in der Chart die “spans”, die verwendet werden können, um NT abzuleiten
- Liste von Listen nötig, da Teilketten mehrere Ableitungsbäume haben können (Ambiguität)

CYK (Parser)



…	…						
3	…	V					
4	…	∅	DET				
5	…	VP [(2,3-4)]	NP [(3,4)]	N			
6	…	∅	∅	∅	P		
7	…	∅	∅	∅	∅	POSS	
8	…	VP [(2-4,5-7) (2, 3-7)]	NP [(3-4,5-7)]	∅	PP [(5,6-7)]	NP [(6,7)]	N
	…	2	3	4	5	6	7
	boy	*shot*	*an*	*elephant*	*in*	*his*	*pajamas*

Chomsky Normalform

- Zu jeder kontextfreien Grammatik G gibt es eine äquivalente kontextfreie Grammatik G’ in Chomsky Normalform.
- Algorithmus:
  - Eliminierung von ε-Regeln
  - Eliminierung “zu kurzer” Regeln (“Kettenregeln”)
  - Eliminierung zu langer Regeln (mehr als 2 NTe)

Eliminierung “zu langer” Regeln

```
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⟩
```

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)

- Ein paar konkrete Zahlen zur Grammatikgröße

	# of Symbols	# of Rules
Original	72	14,971
Right	10,654	25,553
Left	12,944	27,843
Head	11,798	26,697
Compact	3,644	18,543
Ours	8,407	23,306

Table 3: Grammar size of different binarizations

Song &al. (2008)

- Ein paar konkrete Laufzeitresultate:

Binarization	Constituents	Time (s)
Right	241,924,229	5,747
Left	193,238,759	3,474
Head	166,425,179	3,837
Compact	94,257,478	2,302
Ours	**52,206,466**	**2,182**

Table 4: Performance on test set

- Das Test-Set umfasst 90% der Sätze im Wall Street Journal mit maximal 40 Worten.

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

Earley-Algorithmus

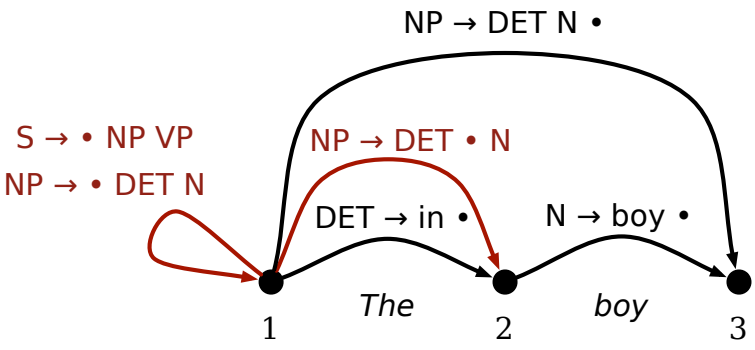
- Ein effizienter „aktiver“ Chart-Parser
  - Zeikomplexität $O(n^3)$ in der Eingabelänge n
- Für beliebige kontextfreie Grammatiken geeignet
  - Tilgungsregeln
  - Zyklische Kettenregeln
  - Links- und rechtsrekursive Regeln
- Mischung aus bottom-up und top-down
  - Analyserichtung links → rechts

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

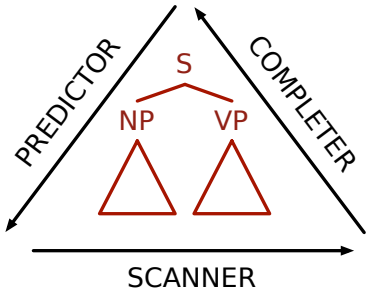
Die Chart

- **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

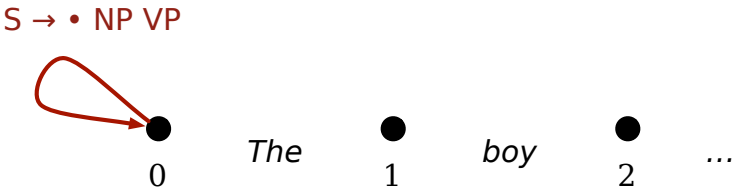
- **Drei Prozeduren**
  - Predictor
  - Scanner
  - Completer



- 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

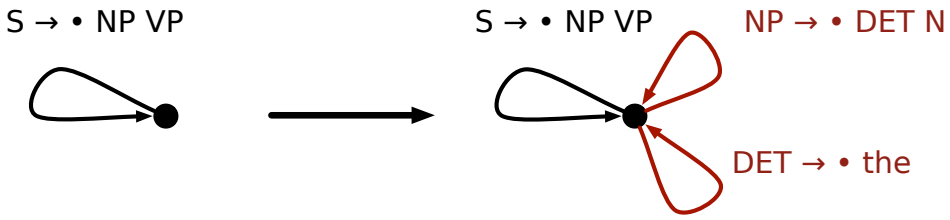
Initialisierung

- Grammatik $G = \langle V, \Sigma, R, S \rangle$
- Eingabe $w_1 \ldots w_n$
- Für jede Regel $S \rightarrow \alpha$
  - Füge $\langle S \rightarrow \bullet\ \alpha, 0, 0 \rangle$ zur Chart hinzu



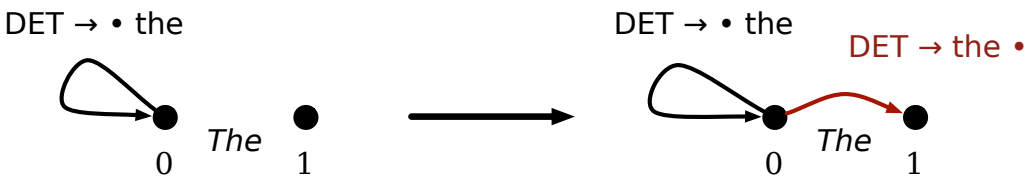
Predictor

- 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⟩



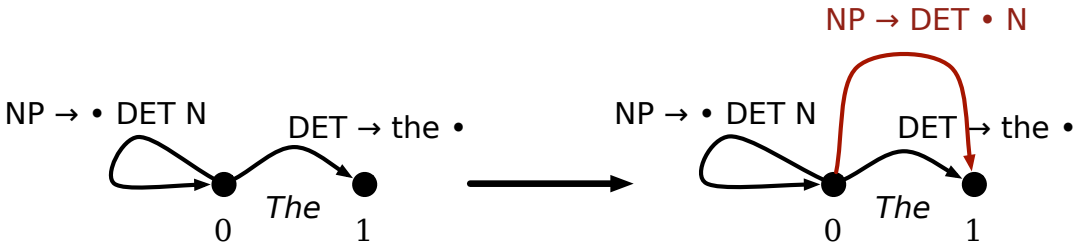
Scanner

- Der Scanner konsumiert Terminalsymbole aus der Eingabekette.
- Für neue Kanten ⟨A → α • w_{j+1} β, i, j⟩
  - Neue Kante ⟨A → α w_{j+1} • β, i, j + 1⟩



Completer

- Der Completer vervollständigt aktive Kanten durch Kombination mit inaktiven Kanten (bottom-up).
- Für jede neue Kante ⟨A → α •, j, k⟩
  - Für jede alte Kante ⟨B → γ • A β, i, j⟩
    - Neue Kante: ⟨B → γ A • β, i, k⟩



Übersicht

- **Scan**
  - ⟨A → α • w_{j+1} β, i, j⟩ ⇒ ⟨A → α w_{j+1} • β, i, j + 1⟩
- **Predict**
  - ⟨A → α • B β, i, j⟩, B → γ ⇒ ⟨B → • γ, j, j⟩
- **Complete**
  - ⟨A → α • B β, i, j⟩, ⟨B → γ •, j, k⟩ ⇒ ⟨A → α B • β, i, k⟩

Eine Chart

der Hund bellt

0		1		2		3	
START → • S S → • NP VP NP→ • D N	0,0 0,0 0,0	D → *der* • NP→ D • N	0,1 0,1	N → *Hund* • NP→ D N • S → NP • VP VP→ • V	1,2 0,2 0,2 2,2	V → *bellt* • VP→ V • S → NP VP •	2,3 2,3 0,3

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⟩

Implementierung

- Kanten ⟨A → α • B β, i, j⟩ stellen wir dar als (A, [B, …], i, j), d.h. α wird nicht explizit repräsentiert.
- Der Algorithmus (Skizze)
  - Auf einer Agenda (Liste) werden durch Anwendung von Scan, Predict, Complete erzeugte Kanten (zwischen-) gespeichert
    - Die Agenda enthält initial die Kante (START, [S], 0, 0)
  - In jedem Schritt entfernen wir eine Kante von der Agenda, wenden Scan, Predict oder Complete an und speichern die Kante abschließend in der Chart (Liste)
  - Durch Anwendung von Scan, Predict, Complete erzeugte Kanten werden auf der Agenda gespeichert

Implementierung

```
rules = [ ('S', ['NP', 'VP']),
          ('NP', ['DET', 'N']),
          ('VP', ['V', 'NP']),
          ('VP', ['V']),
          ('DET', ['der']),
          ('DET', ['das']),
          ('N', ['Student']),
          ('N', ['Buch']),
          ('V', ['liest']),
          ('V', ['arbeitet'])
        ]
```

Implementierung

```
def earley(start, rules, words):
    length = len(words)
    chart = []
    agenda = [('START', [start], 0, 0)]
    while agenda:
        (lhs, rhs, i, j) = agenda.pop()
        # scan, predict, complete (nächste Folie)
        chart.append((lhs, rhs, i, j))
    return ('START', [], 0, length) in chart
```


Implementierung

```
def earley(start, rules, words):
   ...
   while agenda:
      (lhs, rhs, i, j) = agenda.pop()
      if rhs:
         if j < length and rhs[0] == words[j]: # scan
            agenda.append((lhs, rhs[1:], i, j + 1))
         else: # predict
            for (_lhs, _rhs) in rules:
               if rhs[0] == _lhs:
                  agenda.append((_lhs, _rhs, j, j))
      else:
         ...
```

Implementierung

```
def earley(start, rules, words):
    ...
    while agenda:
        (lhs, rhs, i, j) = agenda.pop()
        if rhs:
            ...
        else: # complete
            for (_lhs, _rhs, _i, k) in chart:
                if k == i and _rhs and _rhs[0] == lhs:
                    agenda.append((_lhs, _rhs[1:], _i, j))
        chart.append((lhs, rhs, i, j))
    return ('START', [], 0, length) in chart
```


Earley-Parser

- Um aus der Chart die Parsebäume zu konstruieren, braucht der Algorithmus mehr Informationen
- Lösung: Der *Completer* notiert, welches komplettierte Item jeweils zum Verschieben des Punktes geführt hat

Hund bellt

[...]

2				3			
I04 I05 I06 I07	N → *Hund* • NP→ D N • S → NP • VP VP→ • V	1,2 0,2 0,2 2,2	 I04 I05	I08 I09 I10 I11	V → *bellt* • VP→ V • S → NP VP •	2,3 2,3 0,3	 I08 I09

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 $\langle A \rightarrow \alpha \bullet B\ \beta, i, j \rangle$
    - Neue Kante $\langle A \rightarrow \alpha\ B \bullet \beta, i, j + 1 \rangle$ wenn $B \rightarrow w_{i+1}$