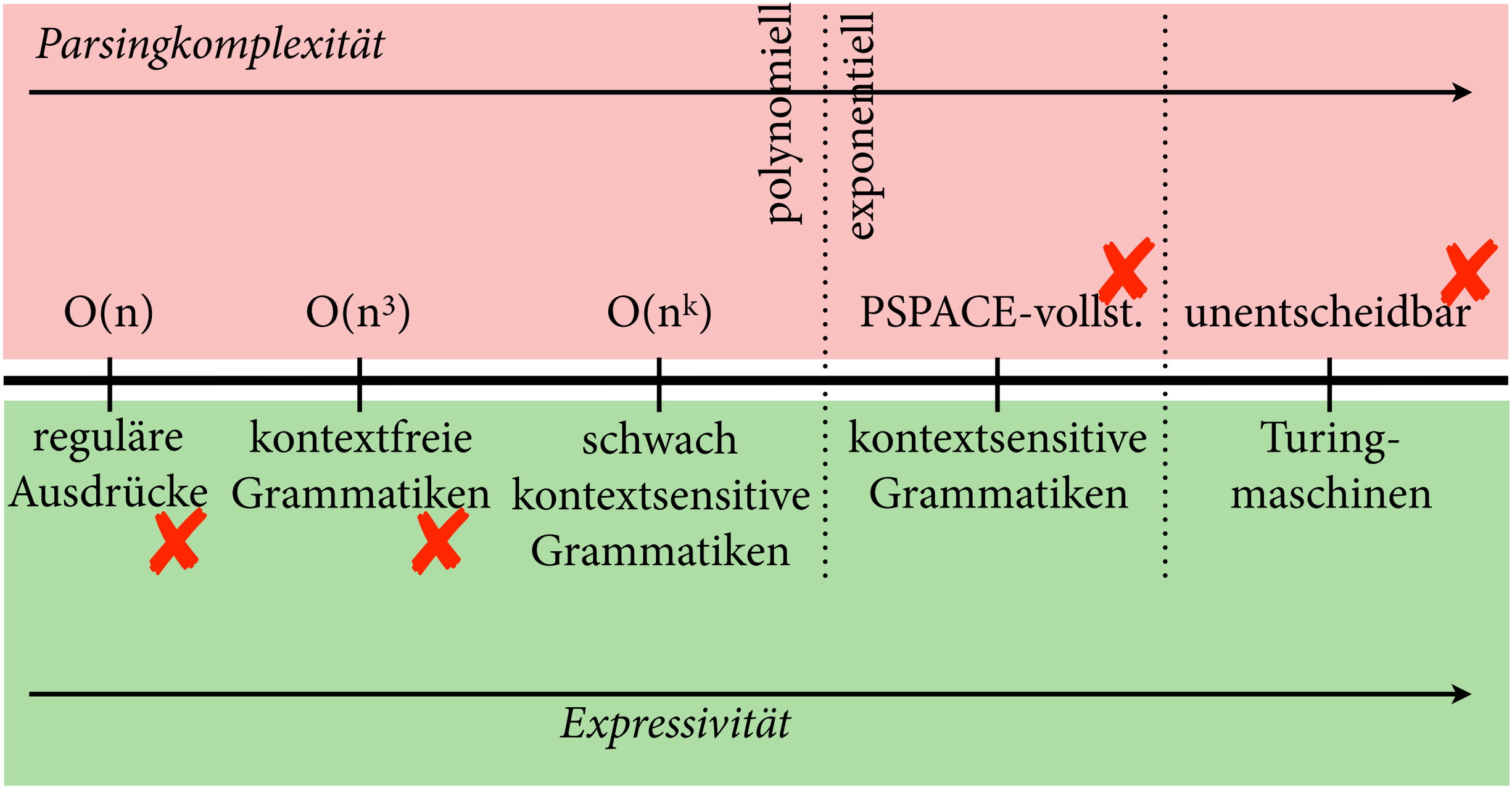
TAG:
Parsing und Expressivität

Vorlesung “Grammatikformalismen”

Alexander Koller

2. Mai 2017

Komplexität vs. Expressivität

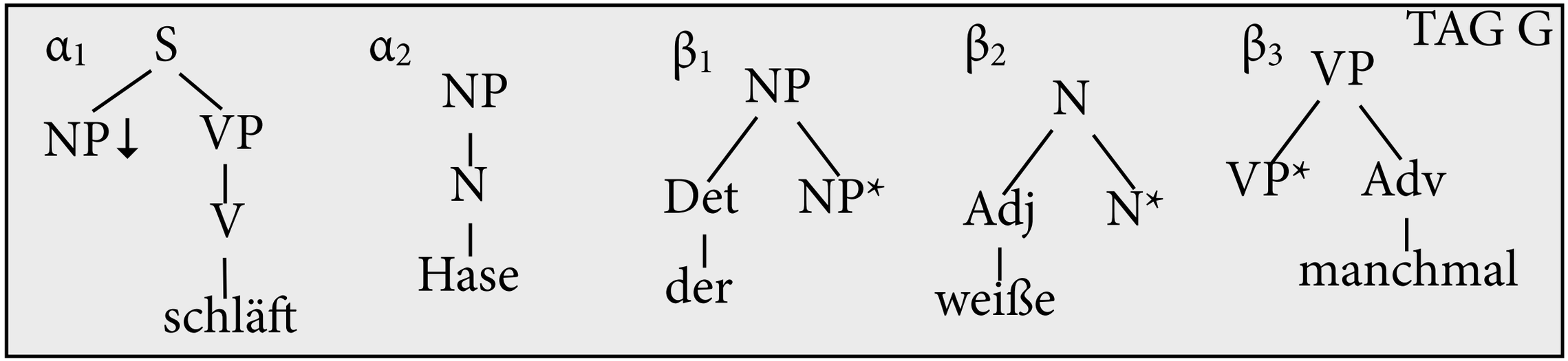


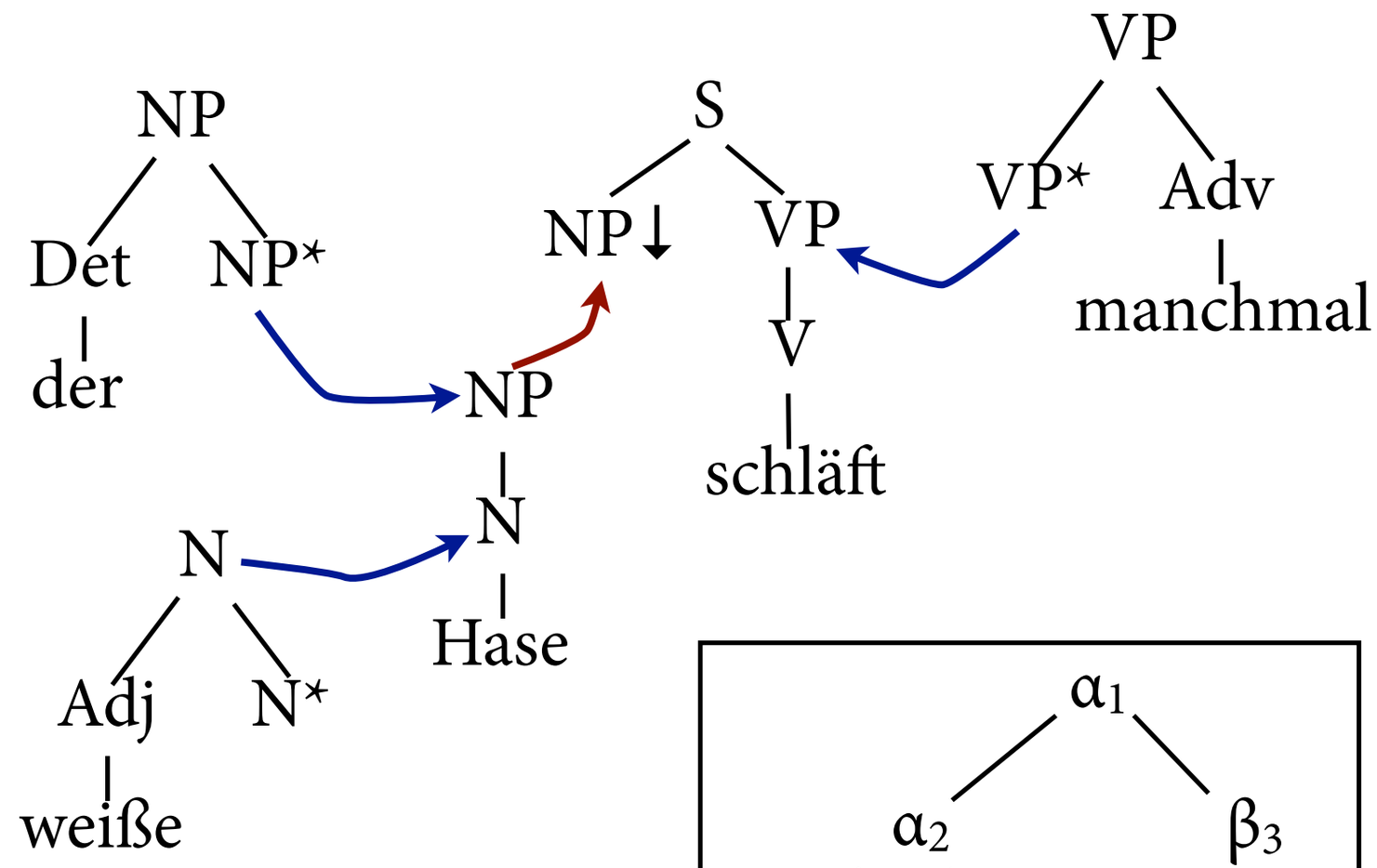
Baumadjunktionsgrammatiken

- *Baumadjunktionsgrammatik* (tree adjoining grammar, TAG): endliche Menge von Elementarbäumen, und zwar
  - *Initialbäumen:* Elementarbäume wie in TSG
  - *Auxiliarbäume:* Elementarbäume, in denen genau ein Blatt ein Fußknoten ist

- TAG-Ableitungen: in jedem Schritt
  - Substitution eines Initialbaums, oder
  - Adjunktion eines Auxiliarbaums

Ein Beispiel





abgeleiteter Baum

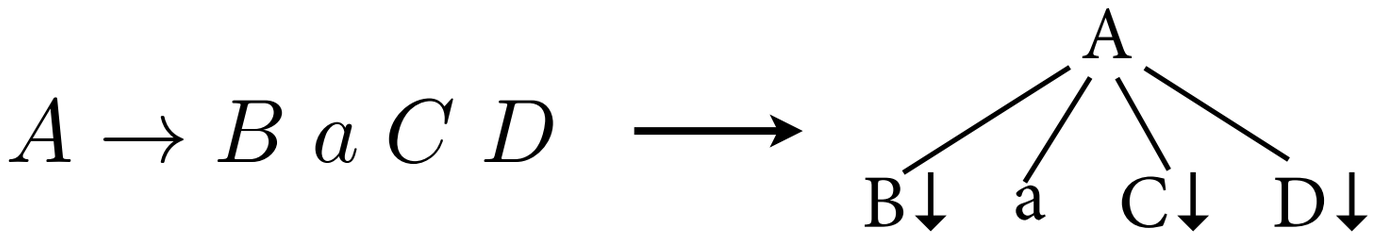
Ableitungsbaum

Übersicht

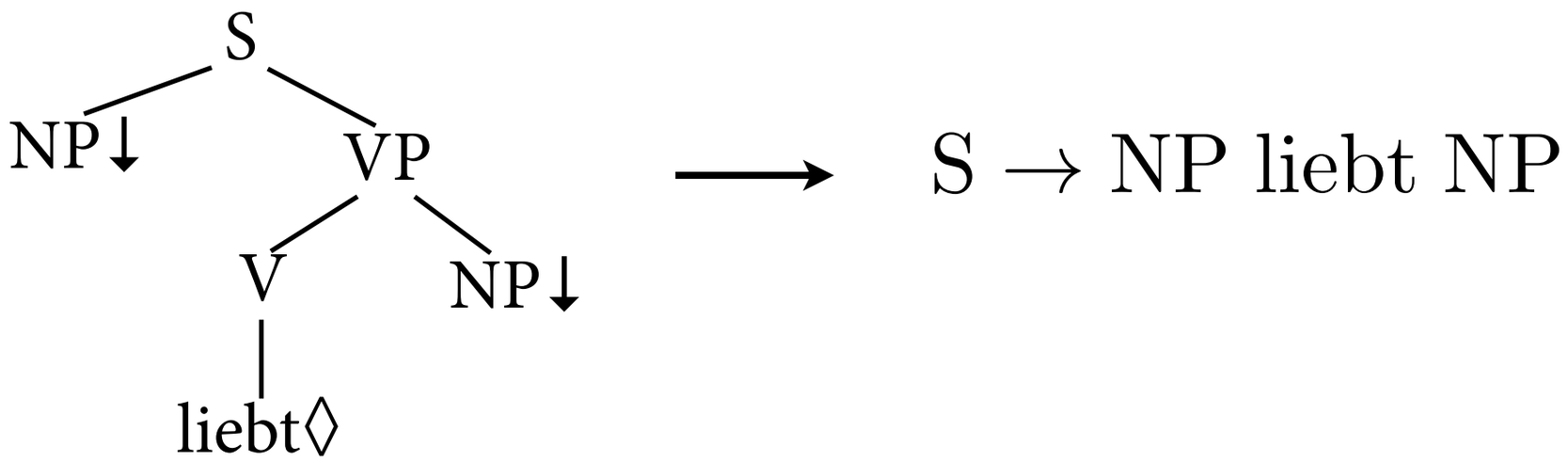
- Expressivität von TAG
- Parsing

Expressivität von TAG

- Jede kfG kann man in stark äquivalente TSG übersetzen (nicht unbedingt lexikalisiert).



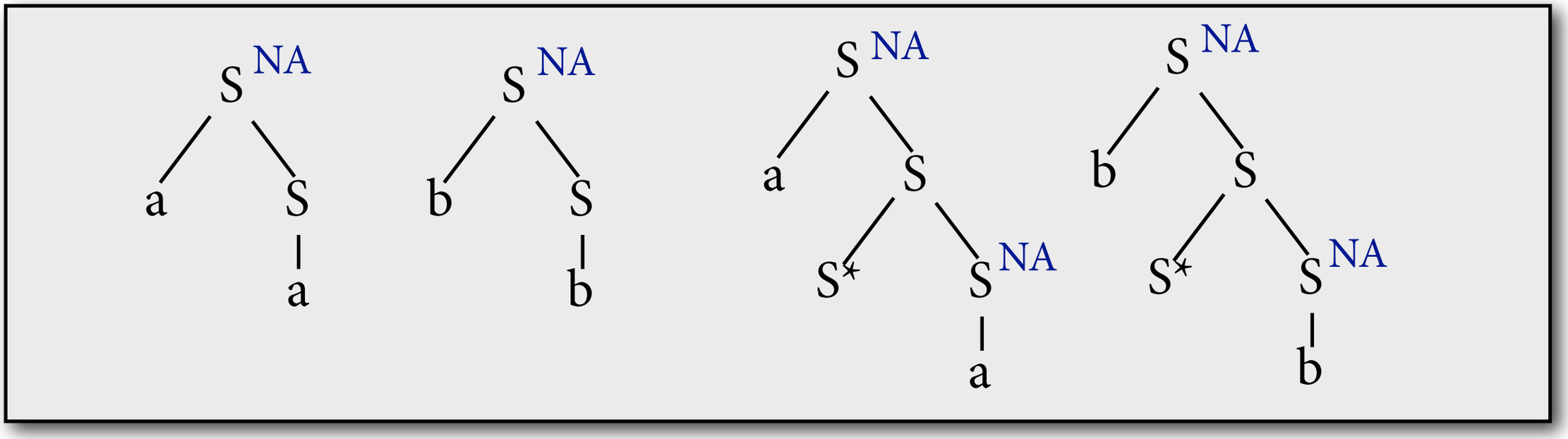
- Jede TSG kann man in schwach äquivalente kfG übersetzen.



- Das heißt: TSG ist schwach kontextfrei.

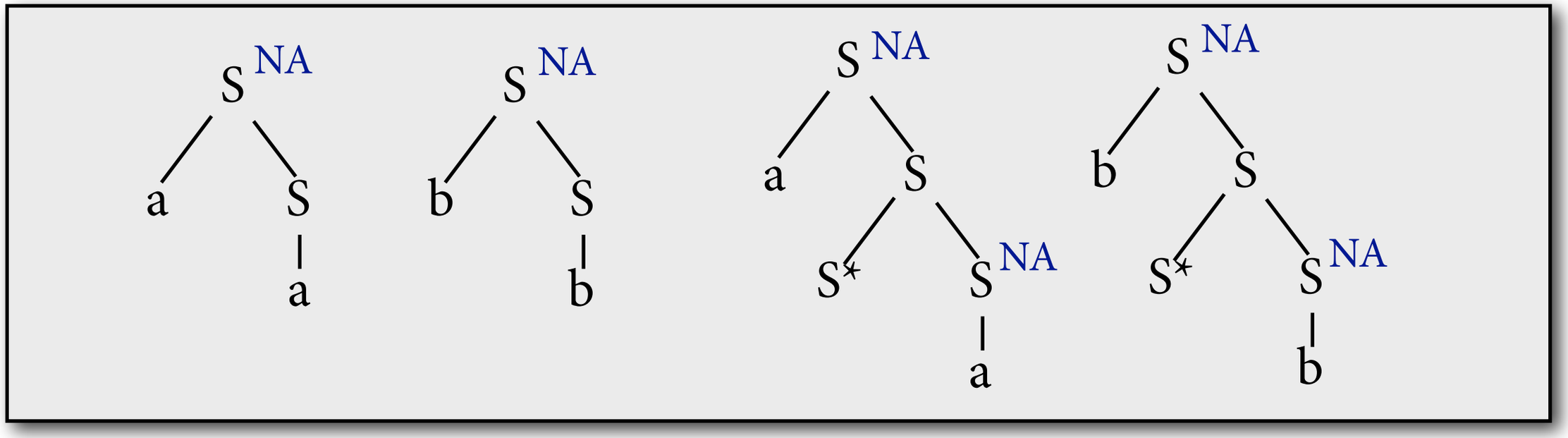
TAG für die Copy-Sprache

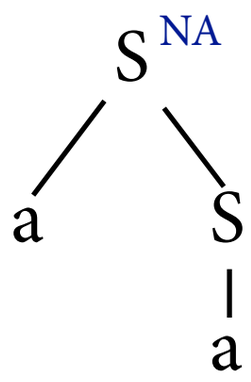
$$\mathrm{COPY} = \{\, ww \mid w \in \Sigma^* \,\}$$



TAG für die Copy-Sprache

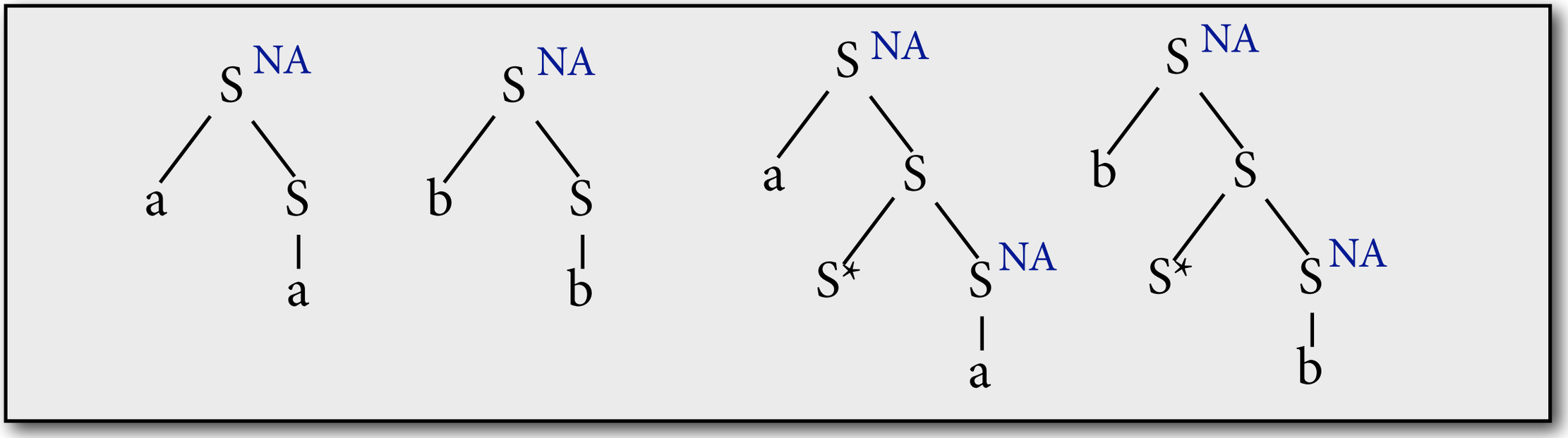
$$\text{COPY} = \{ ww \mid w \in \Sigma^* \}$$

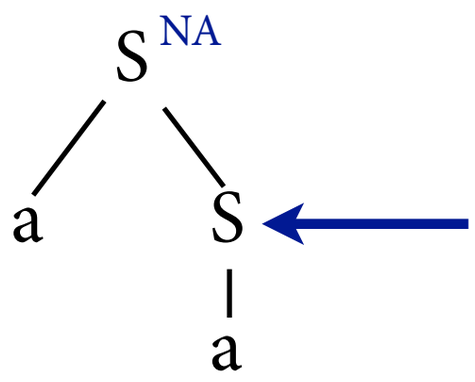




TAG für die Copy-Sprache

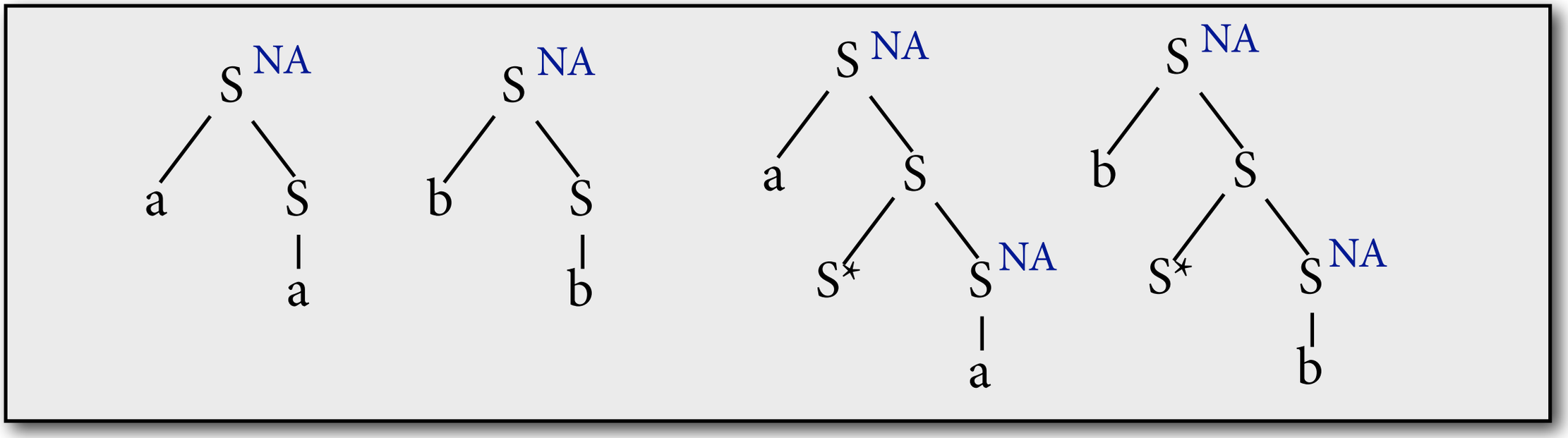
$$\text{COPY} = \{ ww \mid w \in \Sigma^* \}$$

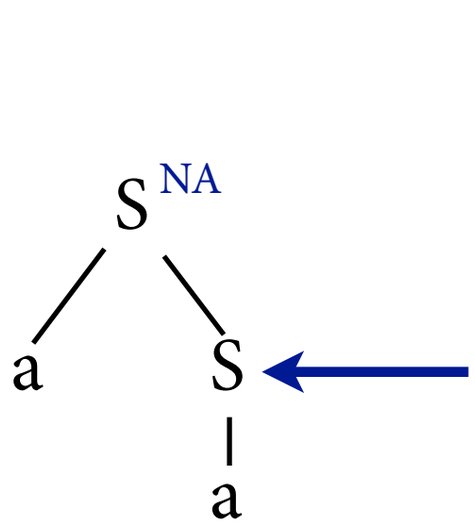


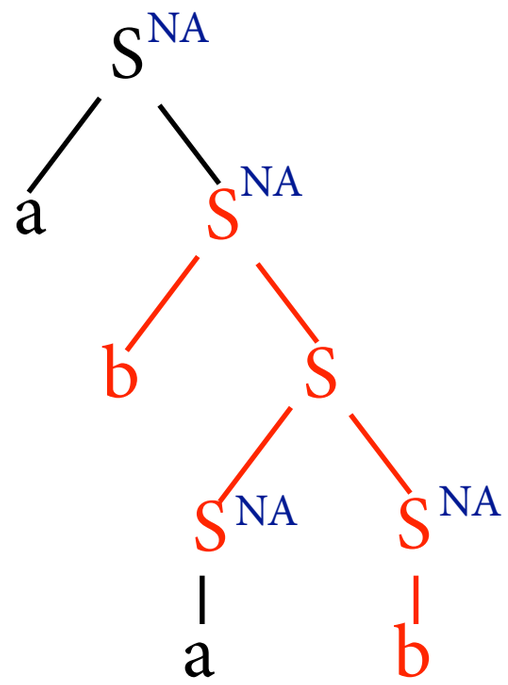


TAG für die Copy-Sprache

$$\mathrm{COPY} = \{\, ww \mid w \in \Sigma^* \,\}$$

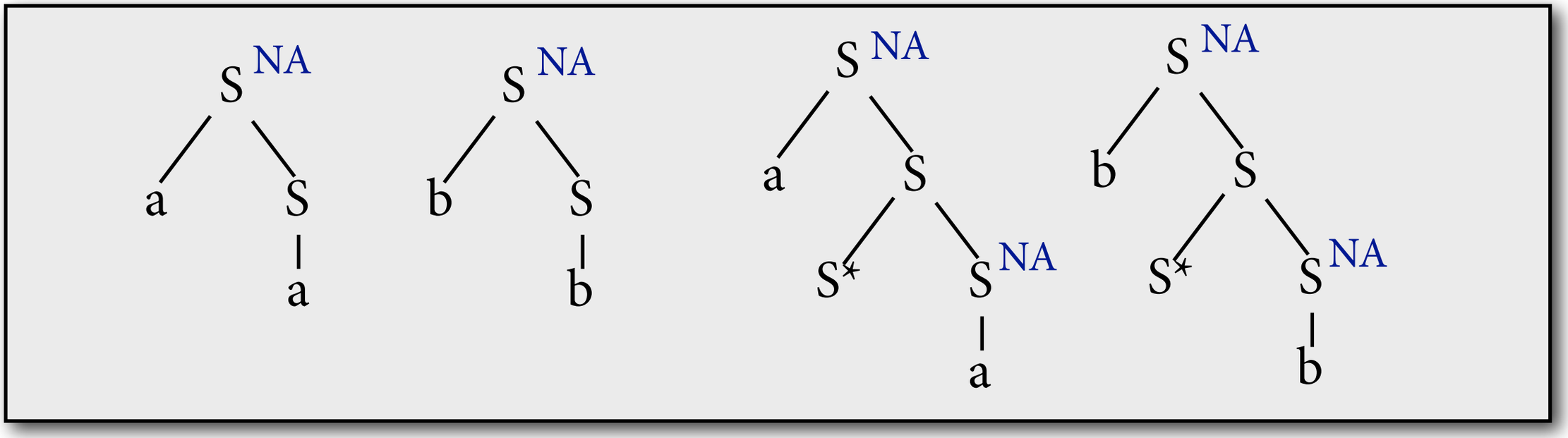


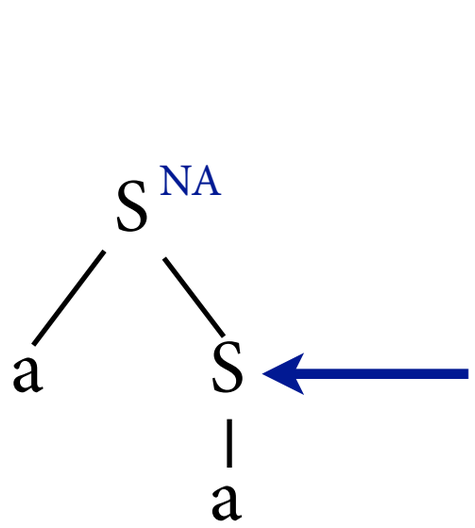


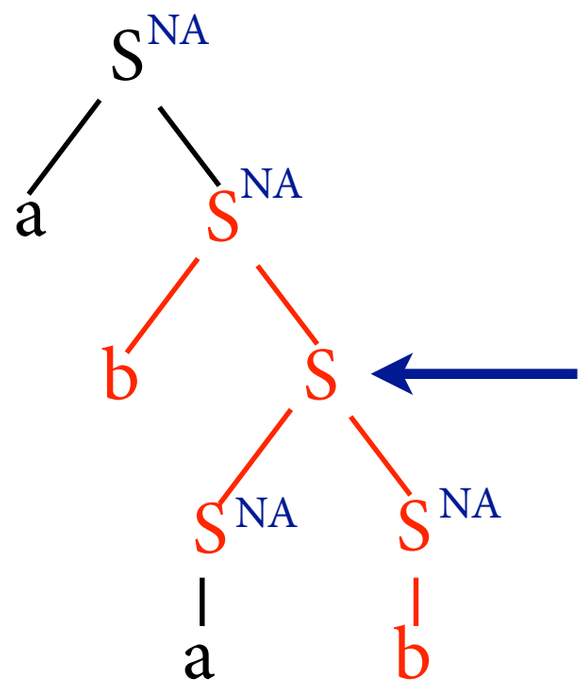


TAG für die Copy-Sprache

$$\mathrm{COPY} = \{\, ww \mid w \in \Sigma^* \,\}$$

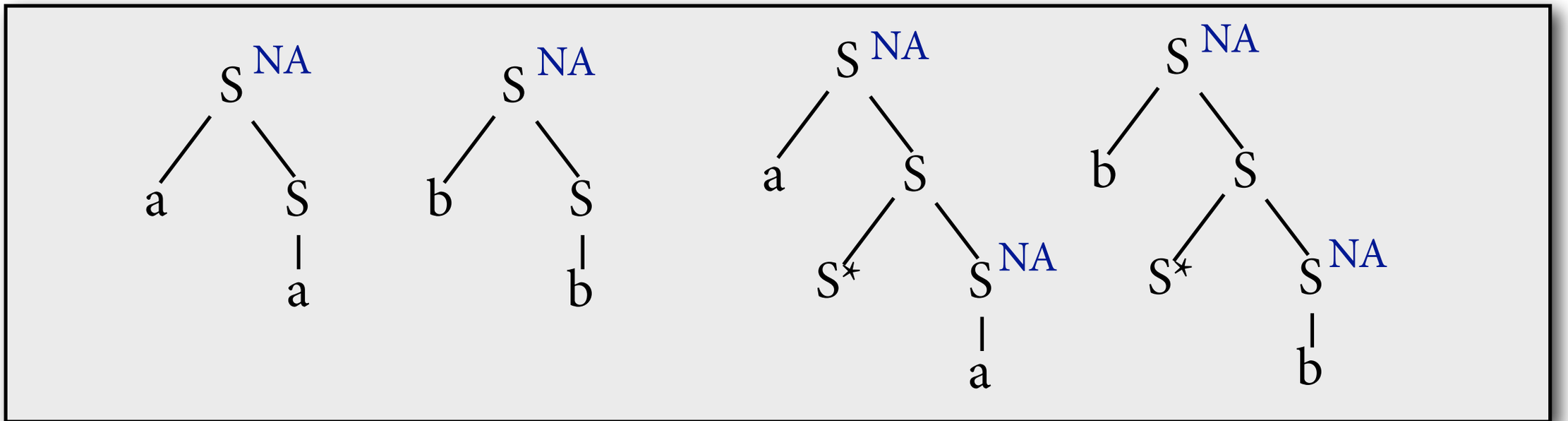


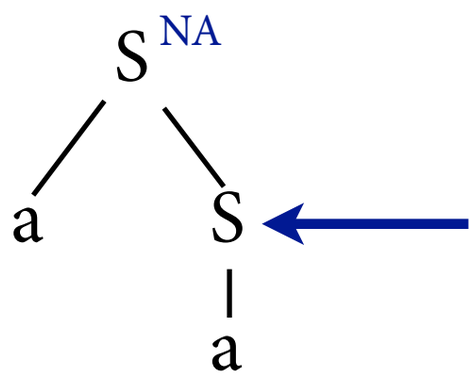


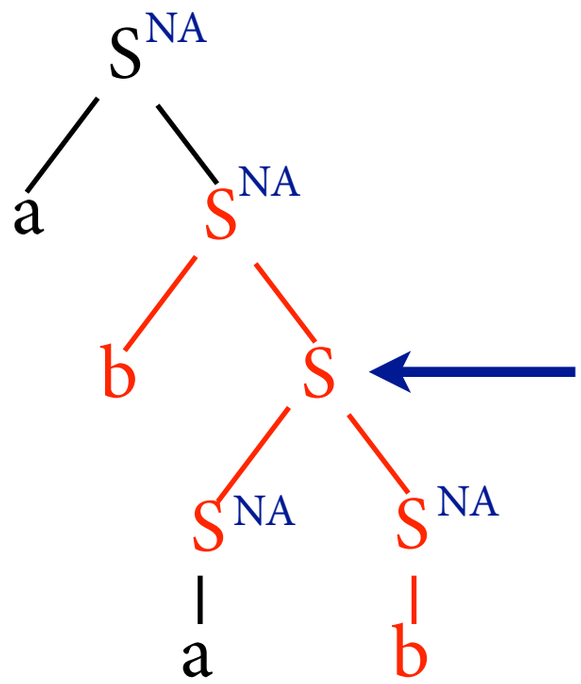


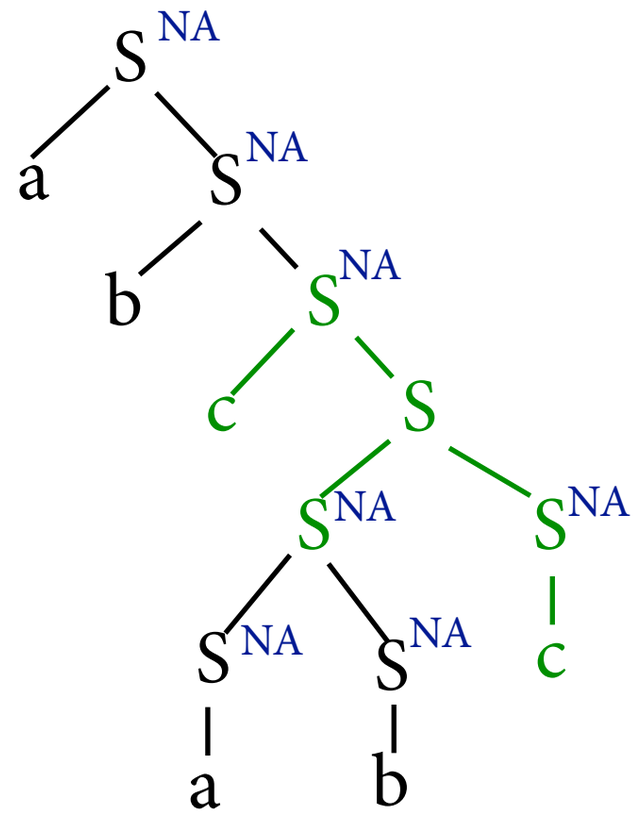
TAG für die Copy-Sprache

$$\mathrm{COPY} = \{\, ww \mid w \in \Sigma^* \,\}$$









KfG für COUNT(2)

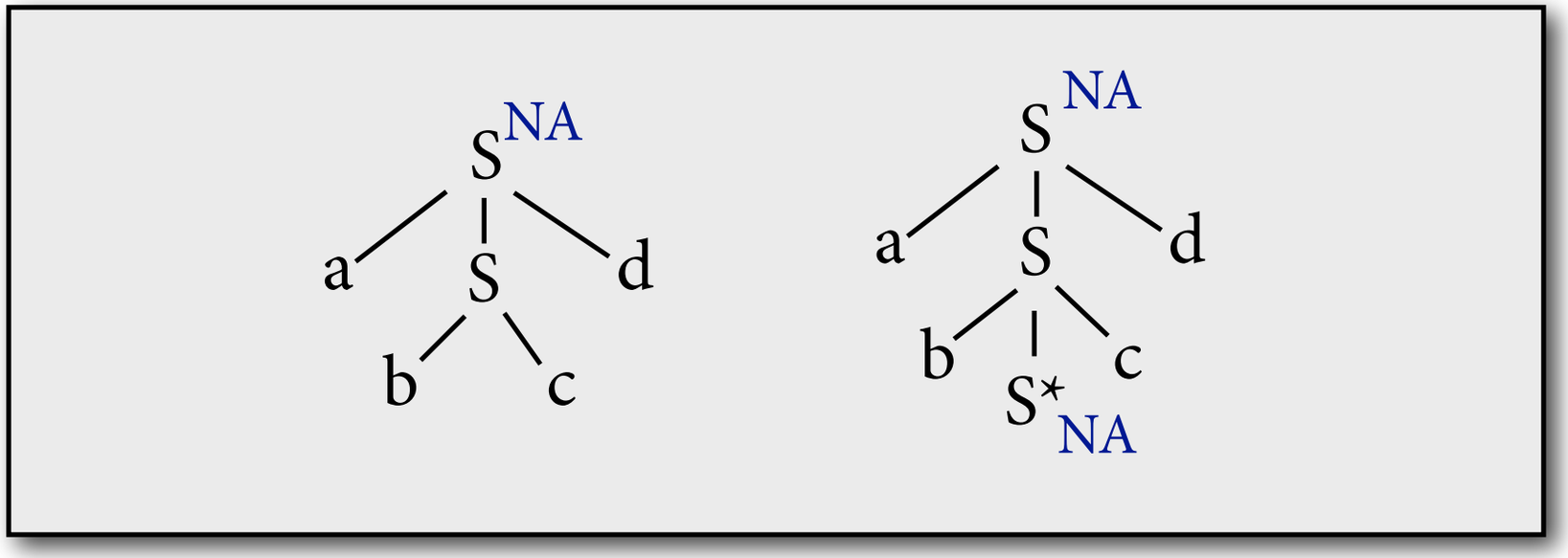
$$\text{COUNT}(2) = \{ a^n b^n \mid n \geq 1 \}$$

$$S \rightarrow a\ S\ b \mid a\ b$$

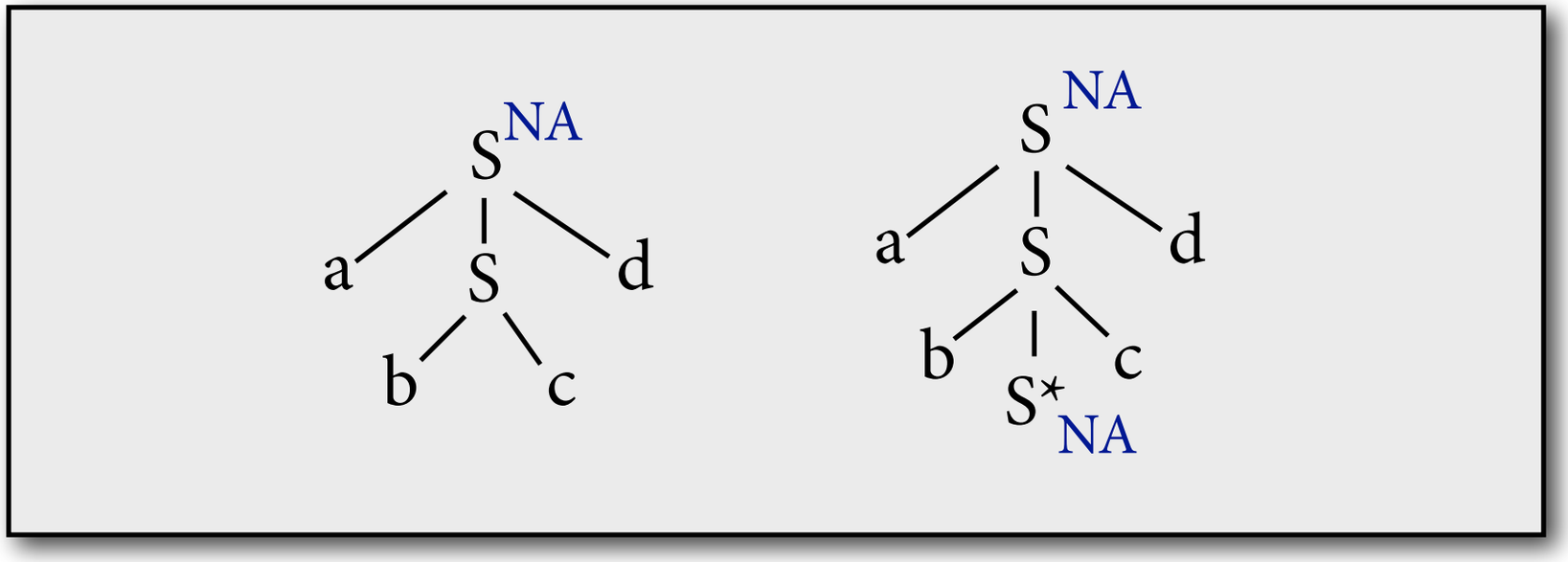
- KfGs können an einer einzigen Stelle den String “aufpumpen”. Die neuen Terminale müssen alle beieinander stehen.

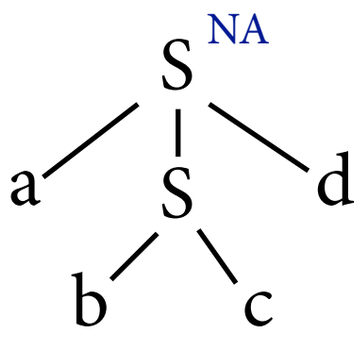
- TAG: Mit Adjunktion den *Baum* aufpumpen. Neue Blätter müssen im String nicht nahe beieinander stehen.

TAG für COUNT(4)

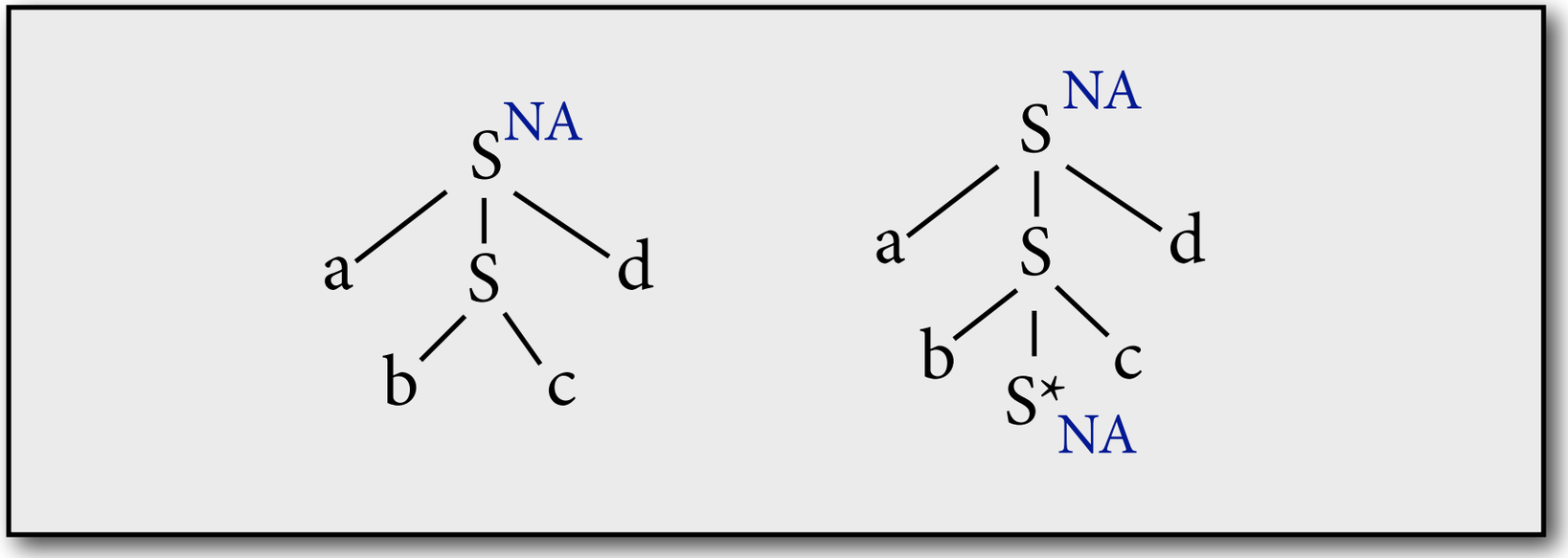


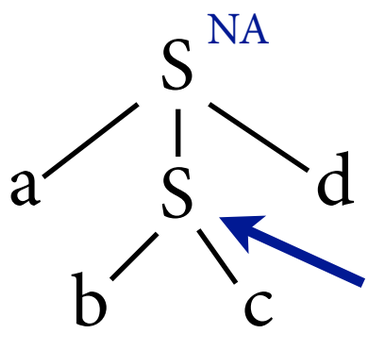
TAG für COUNT(4)



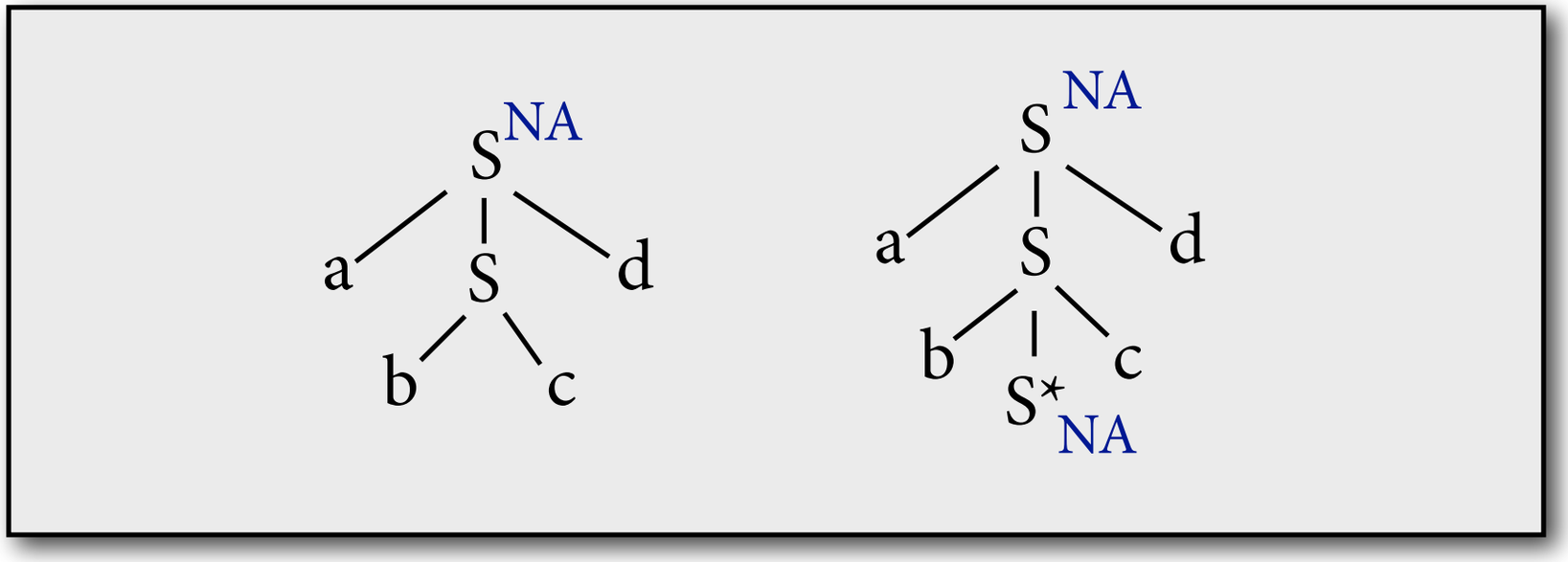


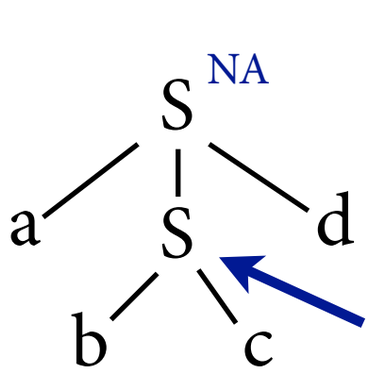
TAG für COUNT(4)

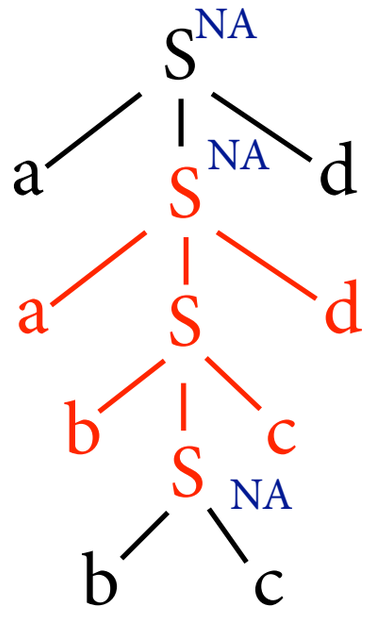




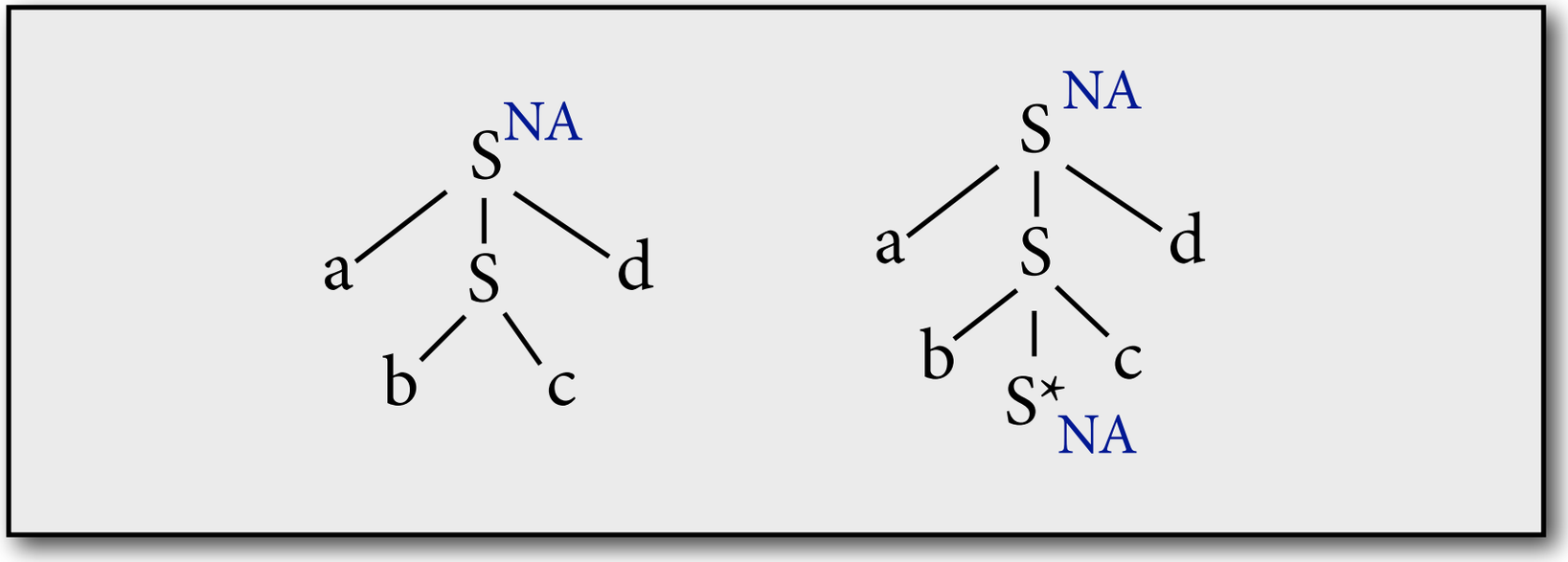
TAG für COUNT(4)

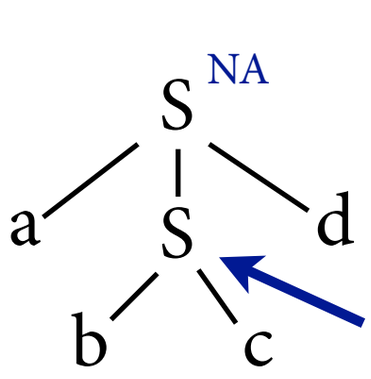


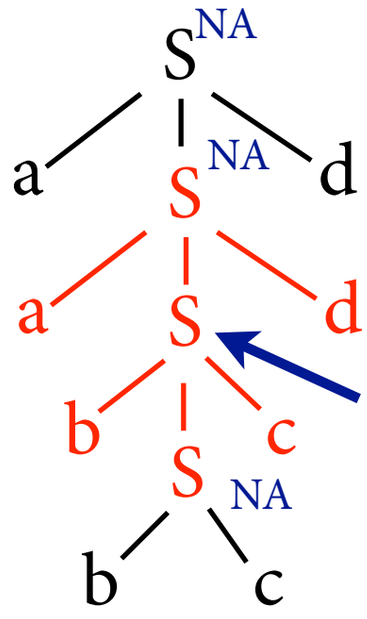




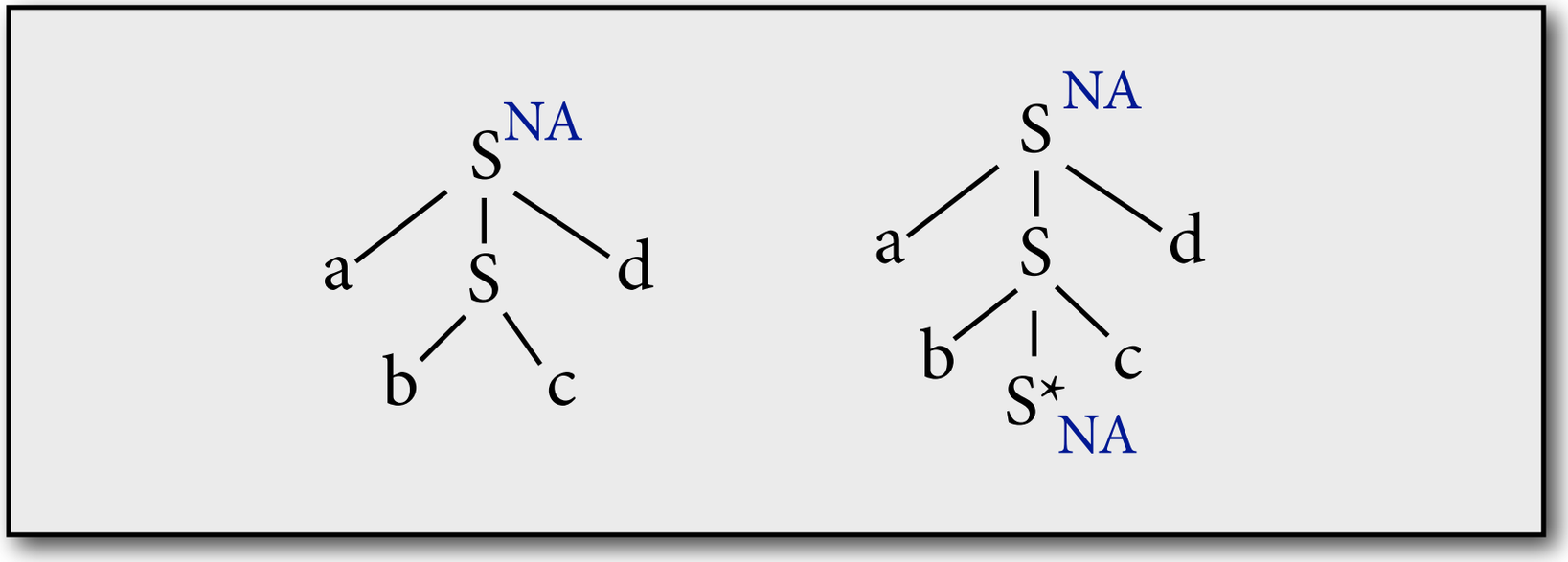
TAG für COUNT(4)

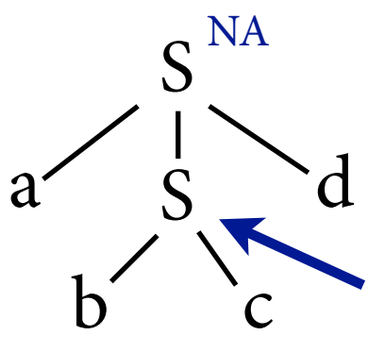


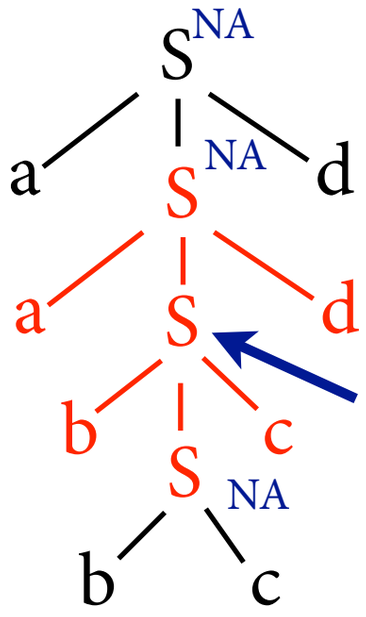


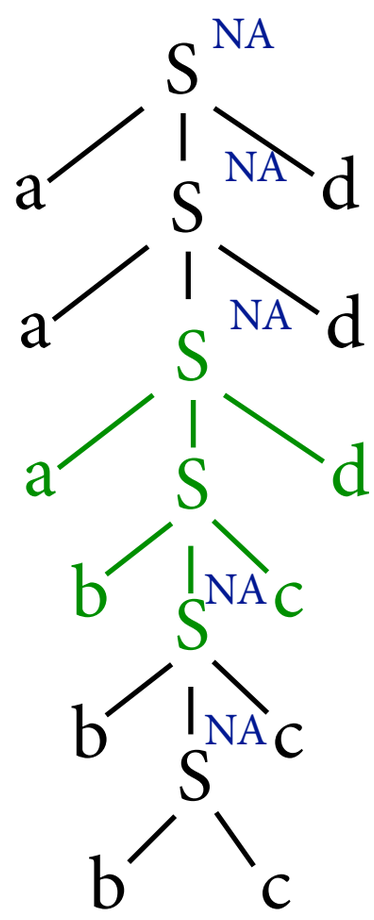


TAG für COUNT(4)









Expressivität von TAG

- TAG kann mehr Sprachen beschreiben als kfG, insbesondere:
  - Fernabhängigkeiten (≈ COUNT(4))
  - Cross-serial dependencies (≈ COPY)
- TAG kann aber nicht alle kontextsensitiven Sprachen beschreiben.
  - Z.B. nicht COUNT(5); kann man mit TAG-Variante des Pumping-Lemmas zeigen.
  - Idee: TAG kann *zwei* Stellen im String gleichzeitig aufpumpen, aber nicht drei.

Parsing von TAG

- Wenn TAG expressiver ist als kfG, kann man dann noch effizient parsen?

- Antwort: es geht polynomiell. Hier: CKY-Parser.

- CKY-Parser verlangt *binäre* TAG-Grammatiken: Jeder Knoten in jedem Elementarbaum hat höchstens zwei Kinder.

CKY-Parser für kfG

- Parsing für kfGs in CNF: CKY-Algorithmus (Cocke-Younger-Kasami).

- Grundidee: Leite *Items* der Form [A,i,k] ab.
  Bedeutet: Kann Substring $w_i \ldots w_{k-1}$ aus A ableiten.

- String in Sprache = CKY leitet [S, 1, n+1] ab.

$$\frac{A \to w_i \text{ in Grammatik}}{[A, i, i+1]} \qquad \frac{[B, i, j] \quad [C, j, k] \quad A \to B\ C \text{ in Grammatik}}{[A, i, k]}$$

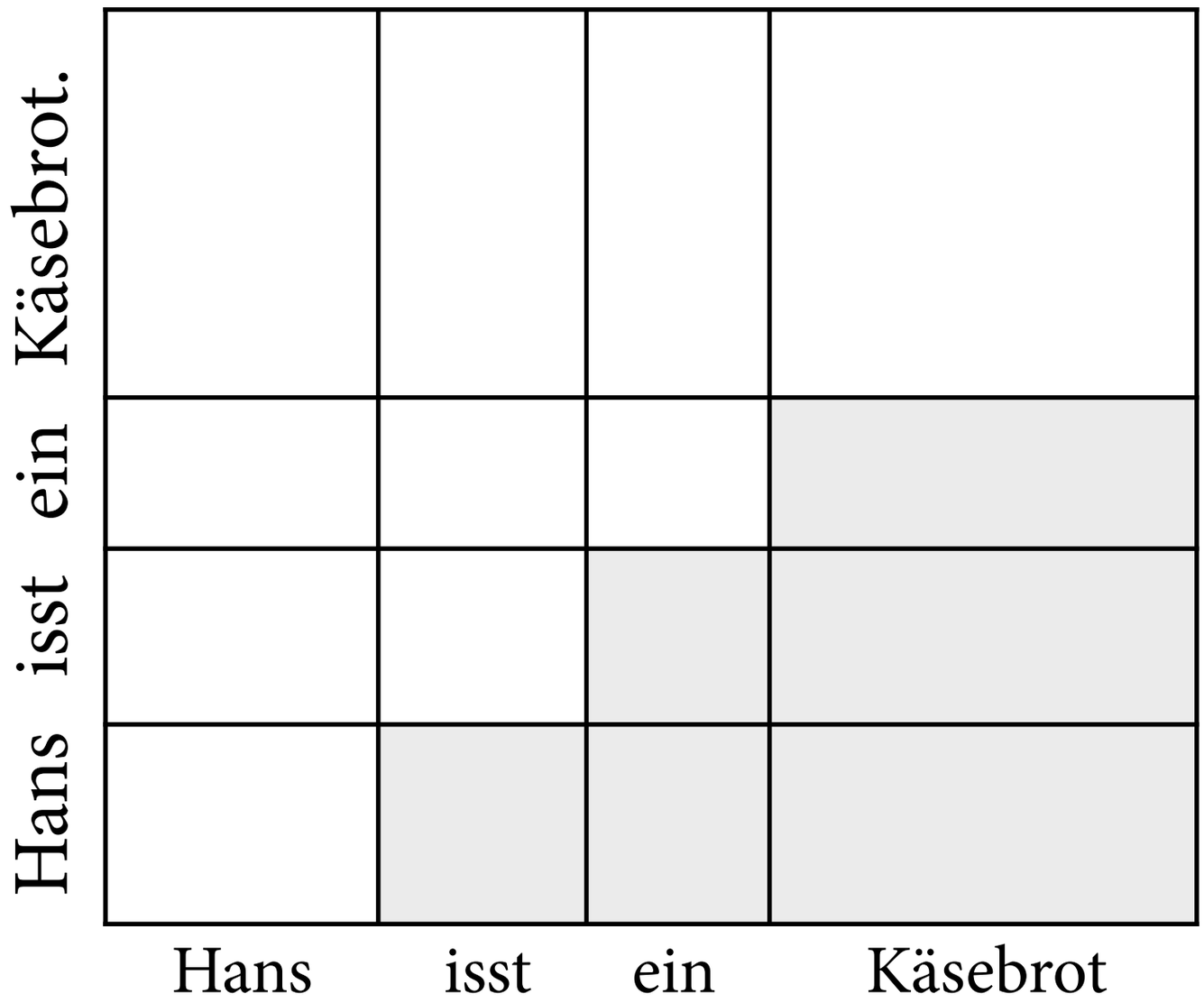
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		

Hans isst ein Käsebrot

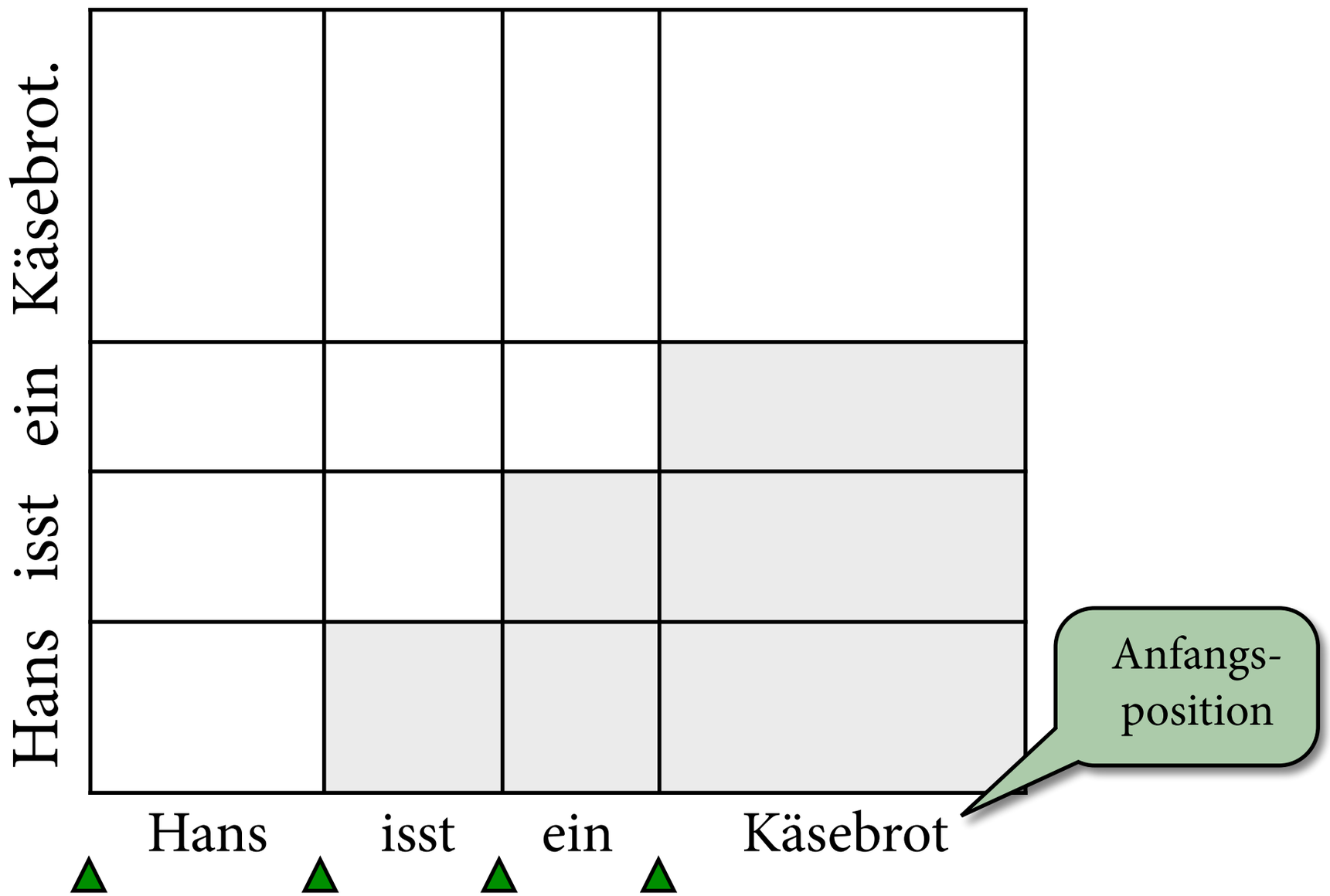
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



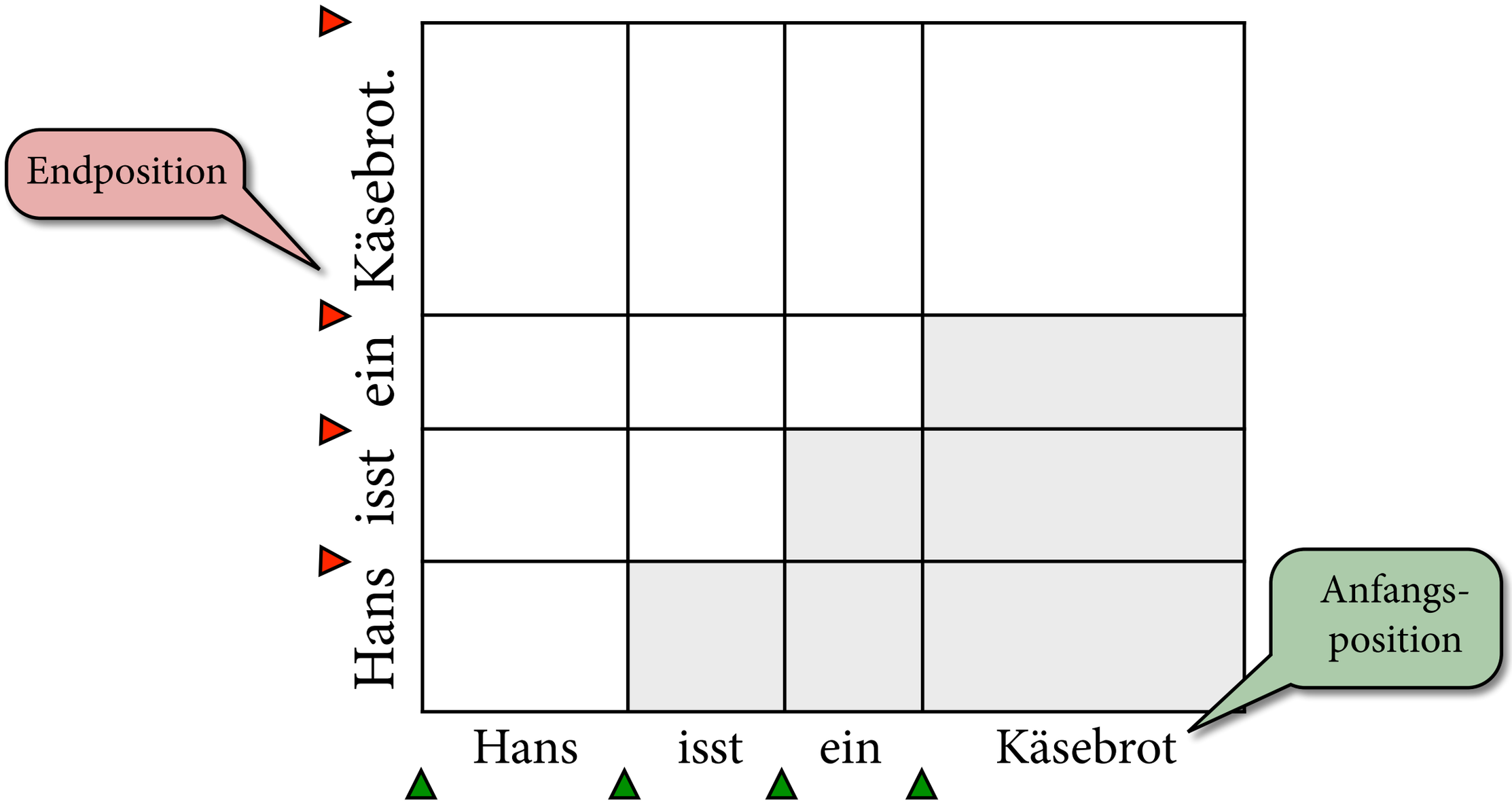
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



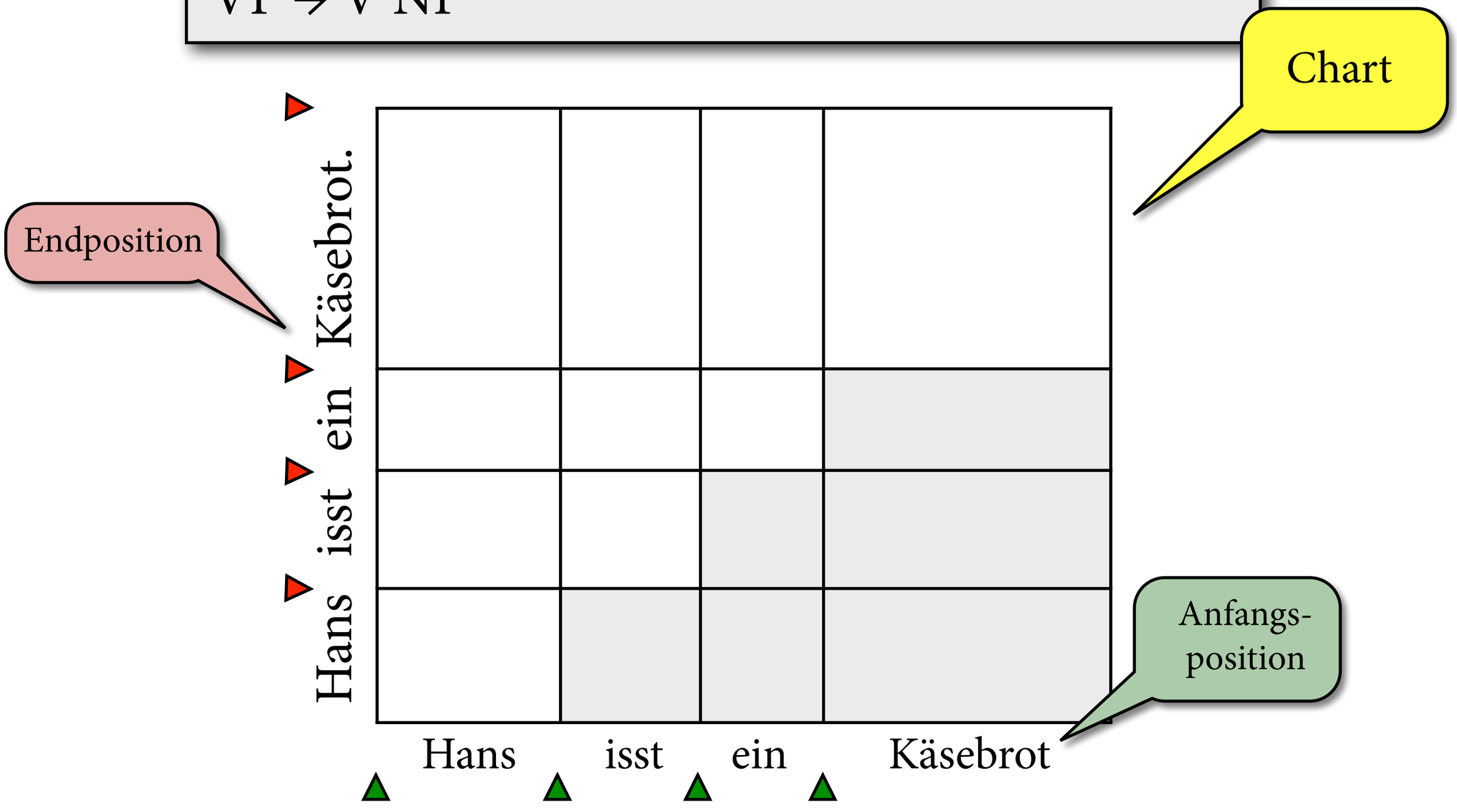
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



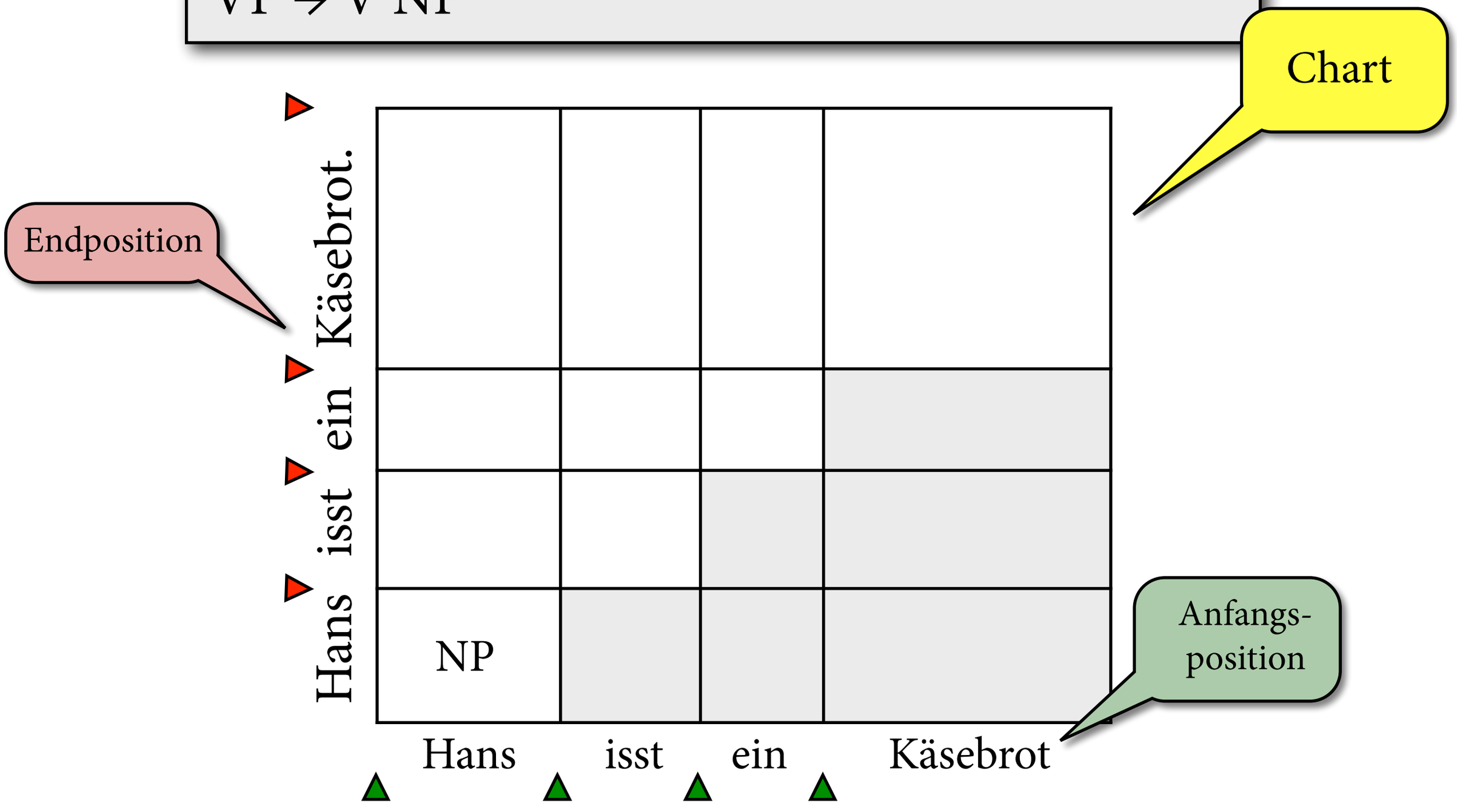
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



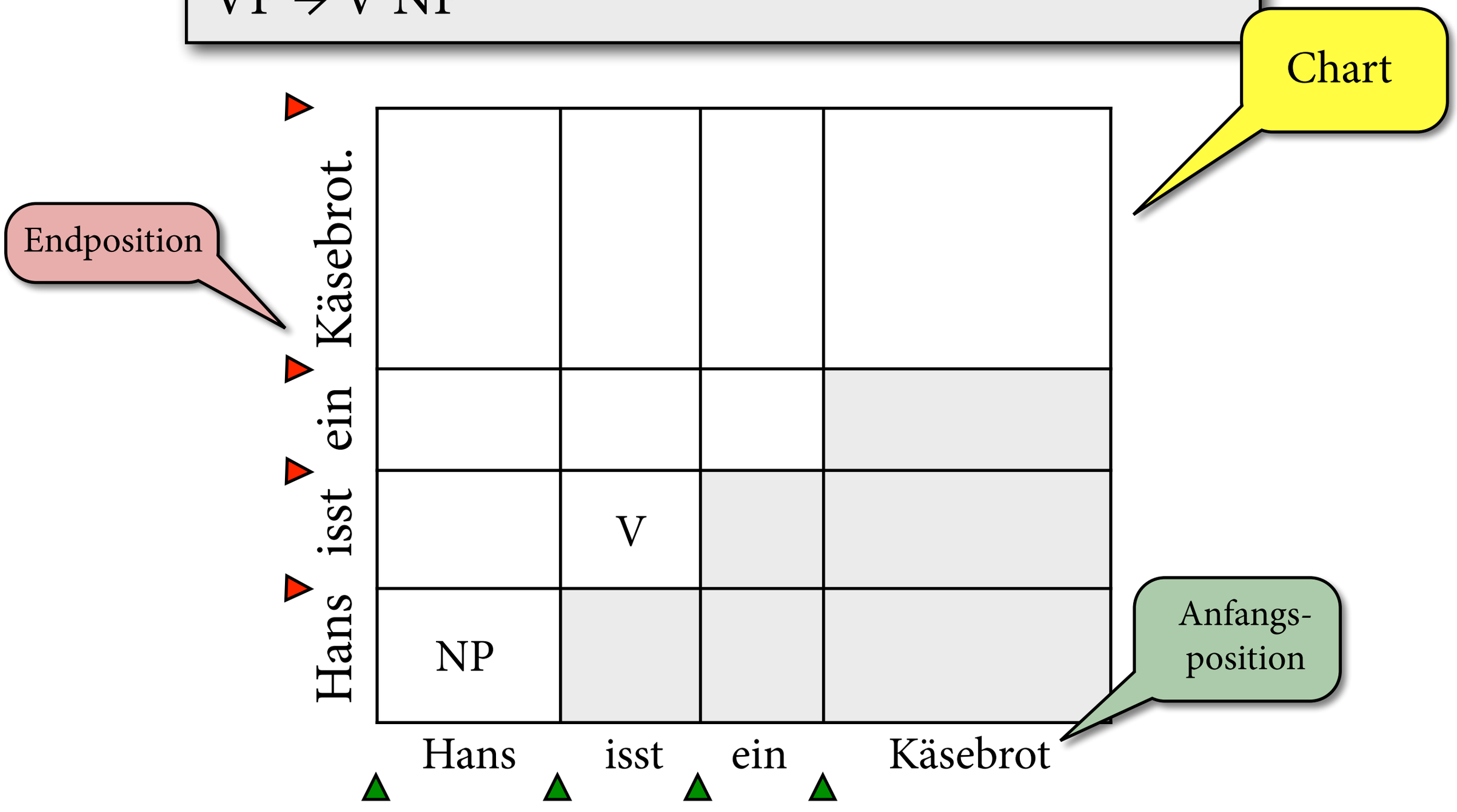
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



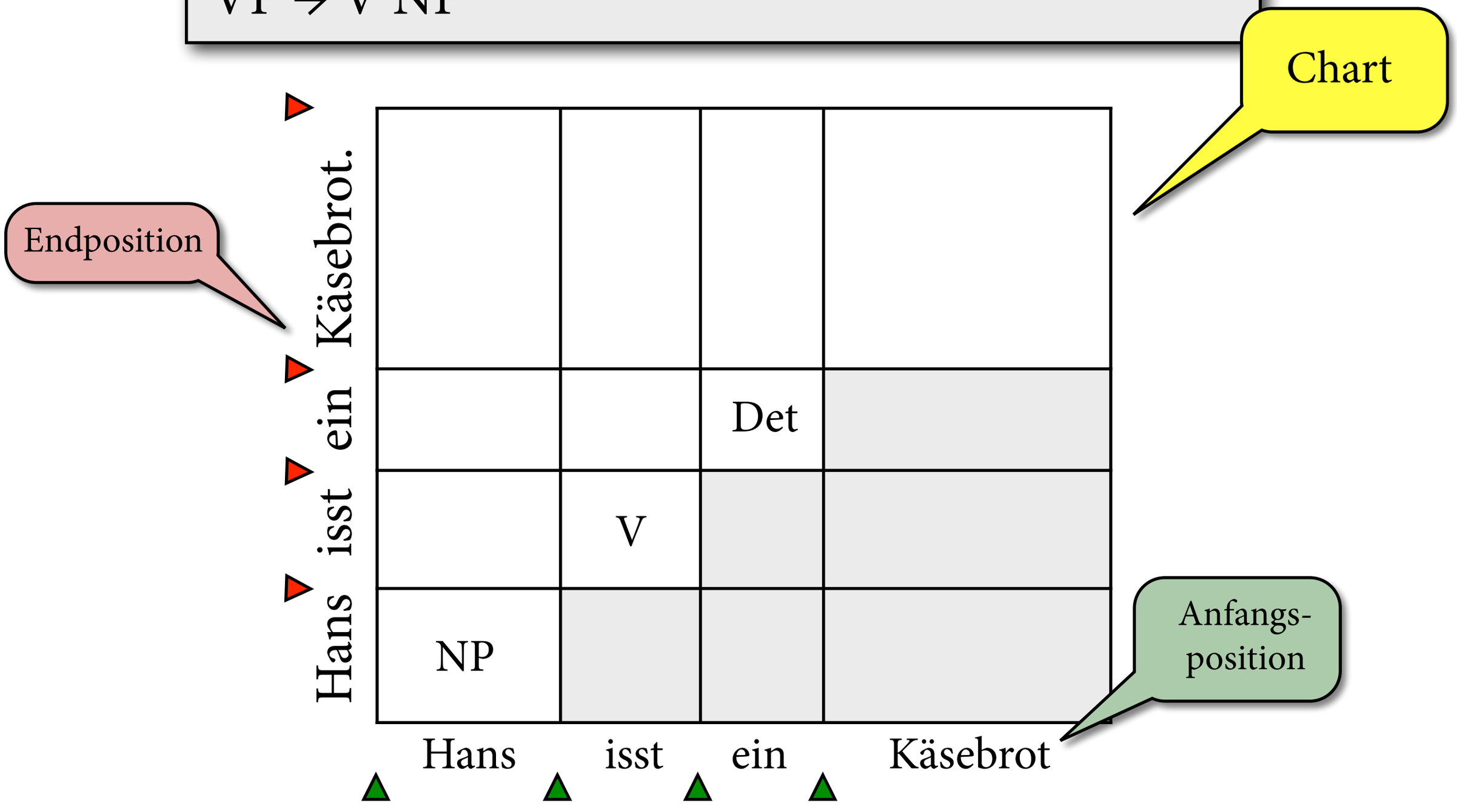
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



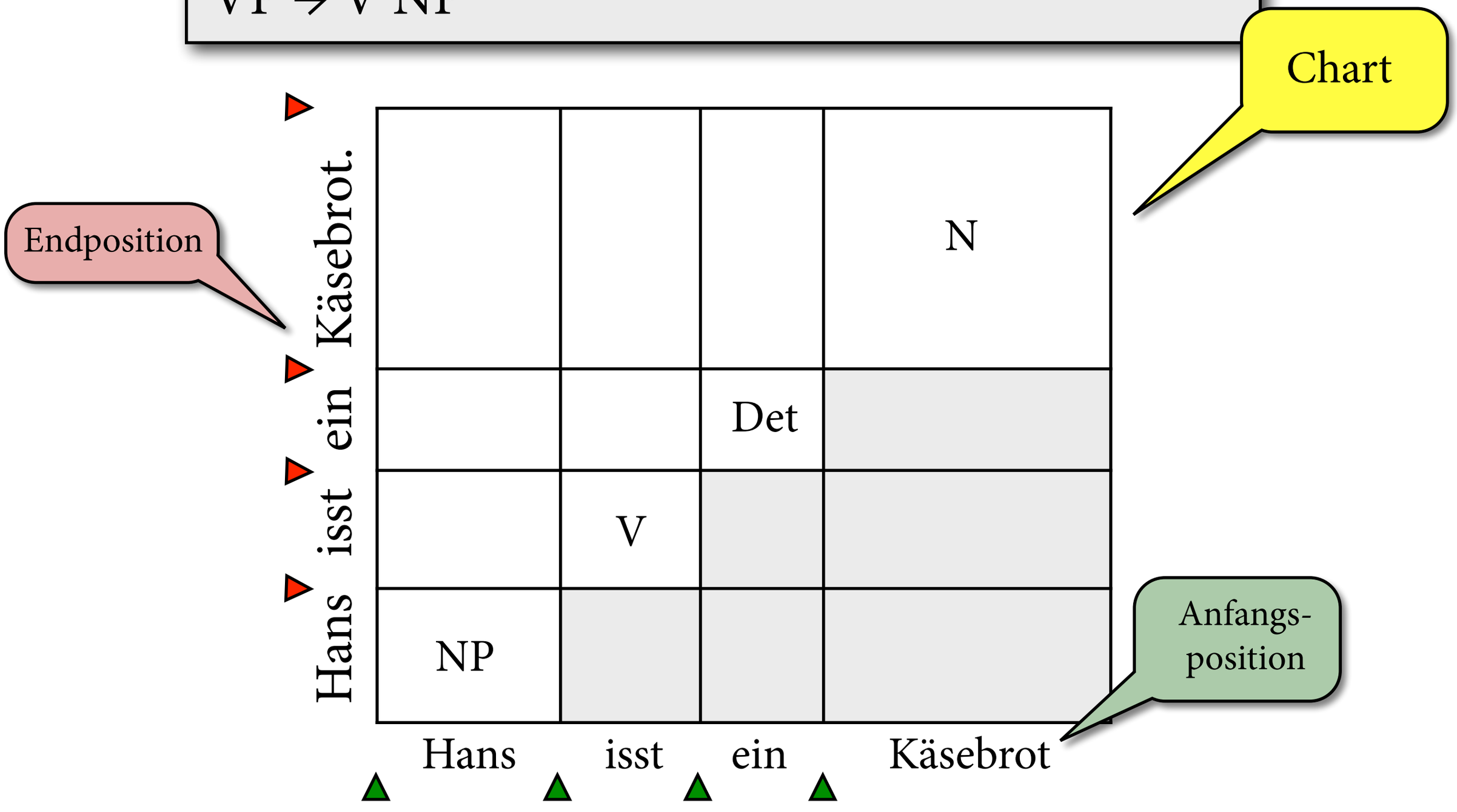
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



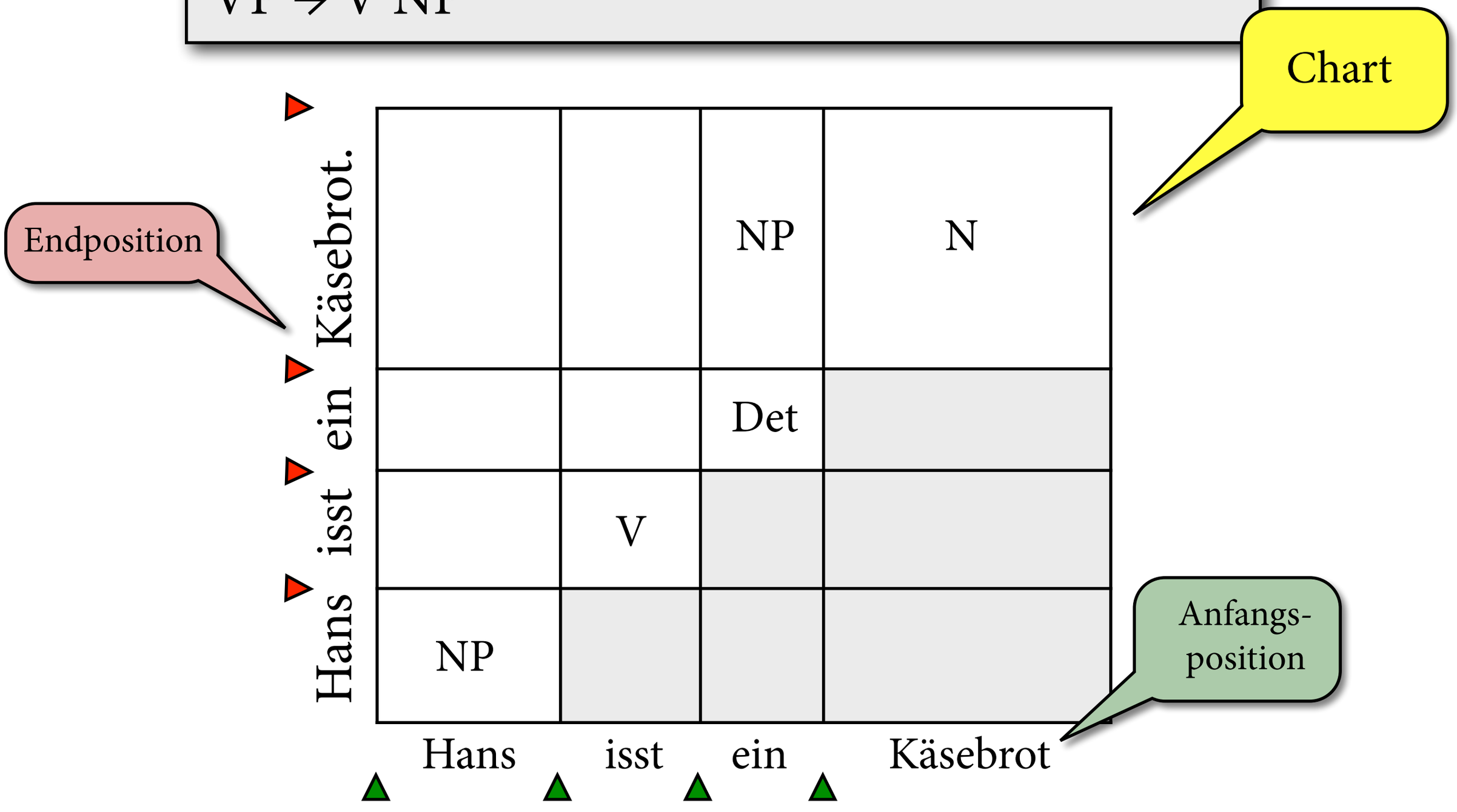
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



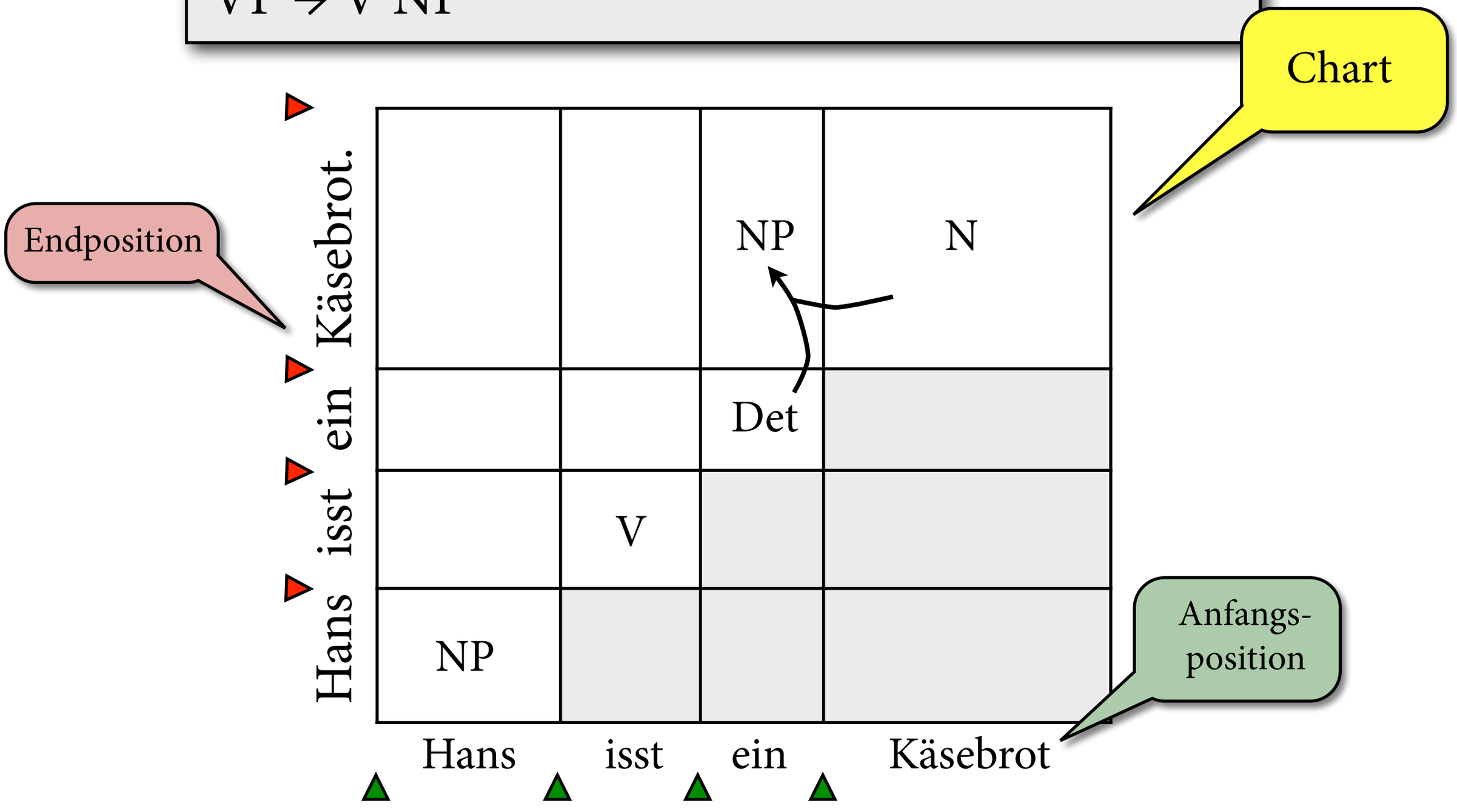
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



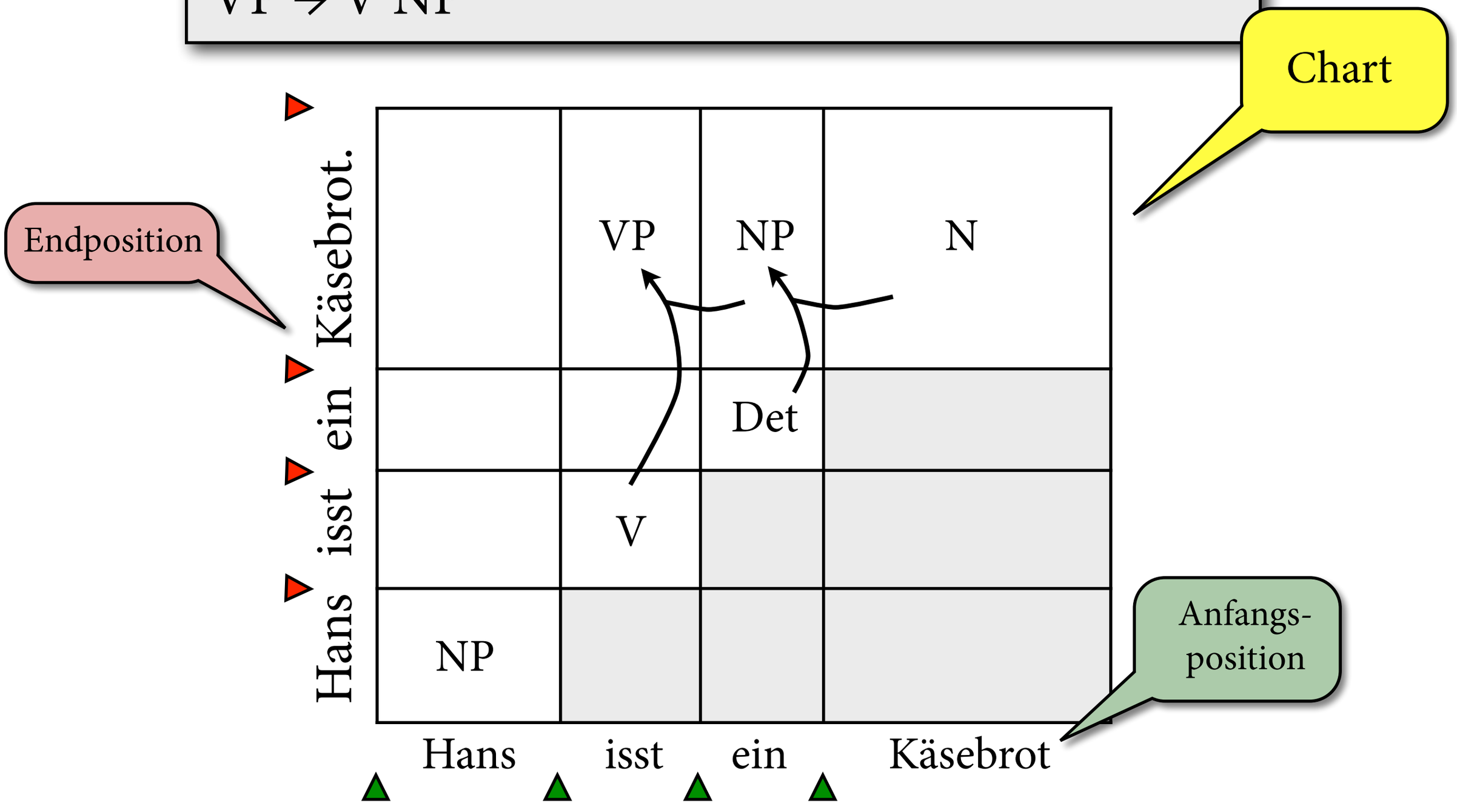
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



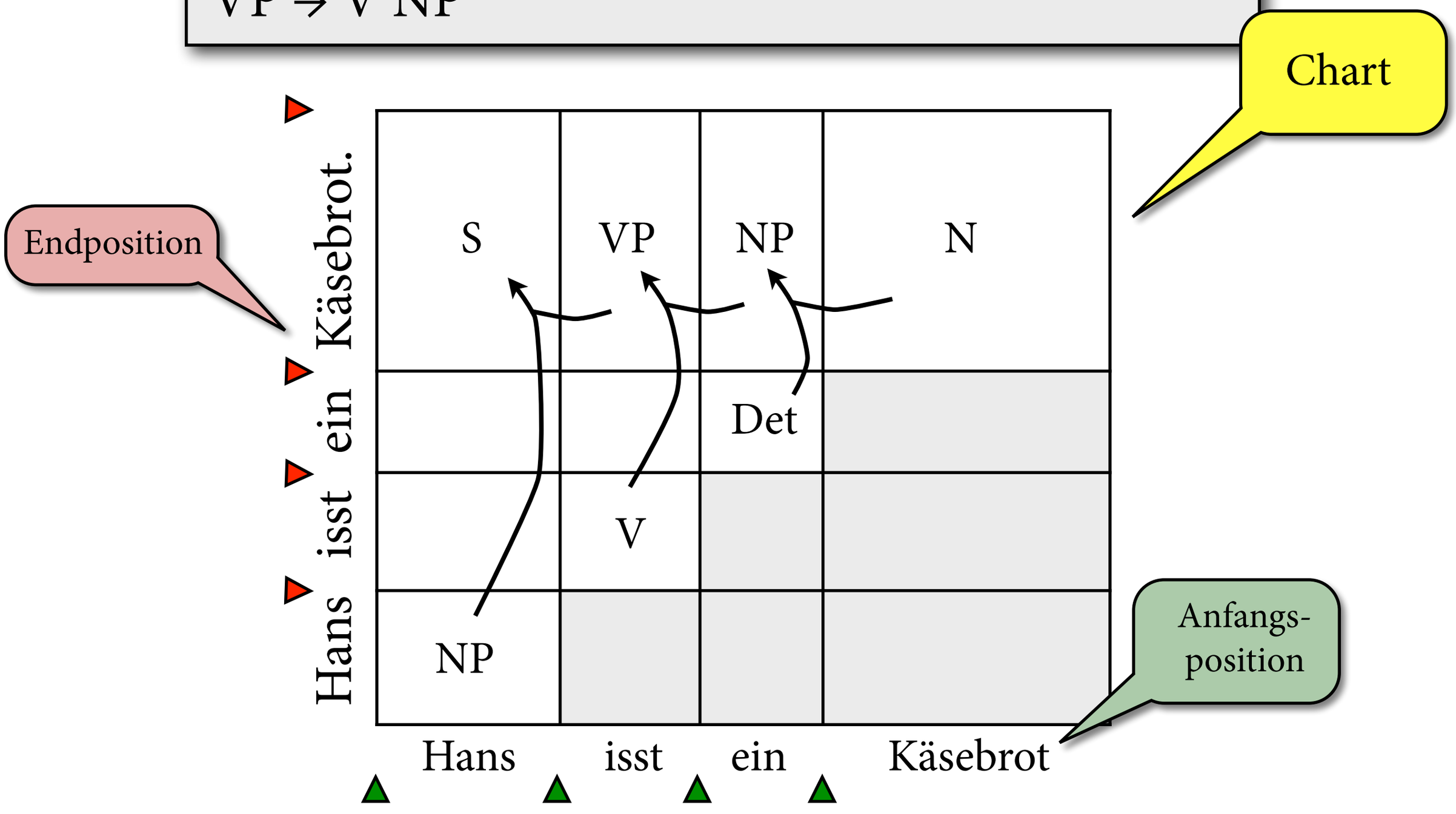
Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



Der CKY-Parser

S → NP VP	V → isst	Det → ein
NP → Det N	NP → Hans	N → Käsebrot
VP → V NP		



Laufzeit von CKY

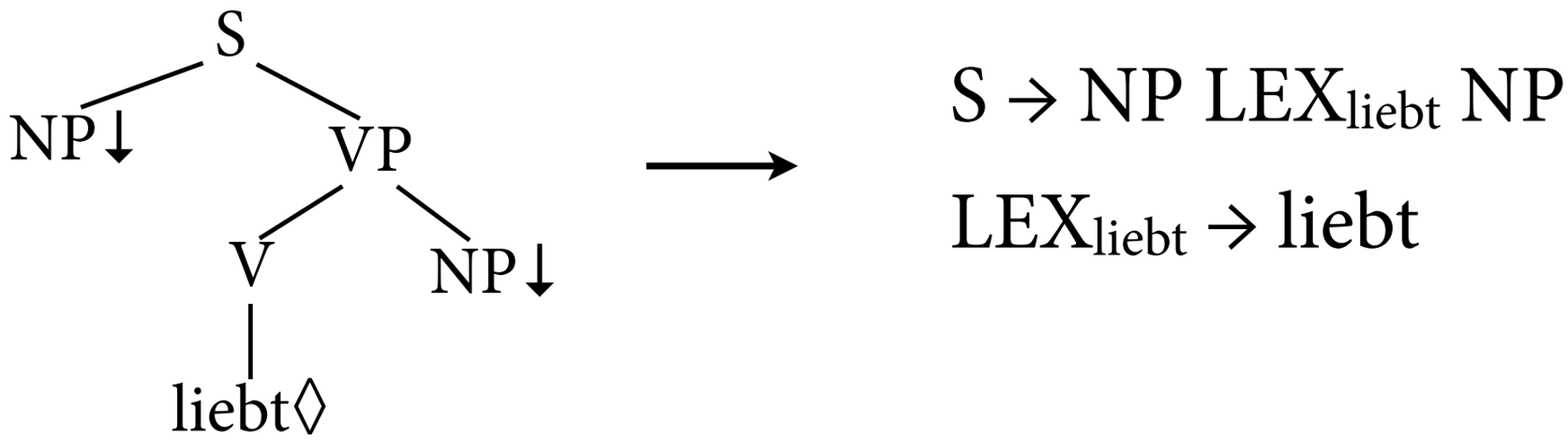
- Faustregel für Laufzeit von Parsing-Schema: wenn Regeln max. k unabhängige Variablen für Stringpositionen haben, ist Laufzeit $O(n^k)$.

- Laufzeit von CKY ist also $O(n^3)$.

$$\frac{A \to w_i \text{ in Grammatik}}{[A, i, i+1]} \qquad \frac{[B, i, j] \quad [C, j, k] \quad A \to B\ C \text{ in Grammatik}}{[A, i, k]}$$

Parsing von TSG: 1. Ansatz

- Konvertiere jeden Elementarbaum als Ganzes in schwach äquivalente kfG-Regel:



- Verwende normale CKY-Regel für nicht-binäre kfGen.

$$\frac{[B_1, i_1, i_2] \ldots [B_r, i_r, i_{r+1}] \quad A \rightarrow B_1 \ldots B_r}{[A, i_1, i_{r+1}]}$$

Komplexität

$$\frac{[B_1, i_1, i_2] \ldots [B_r, i_r, i_{r+1}] \quad A \to B_1 \ldots B_r}{[A, i_1, i_{r+1}]}$$

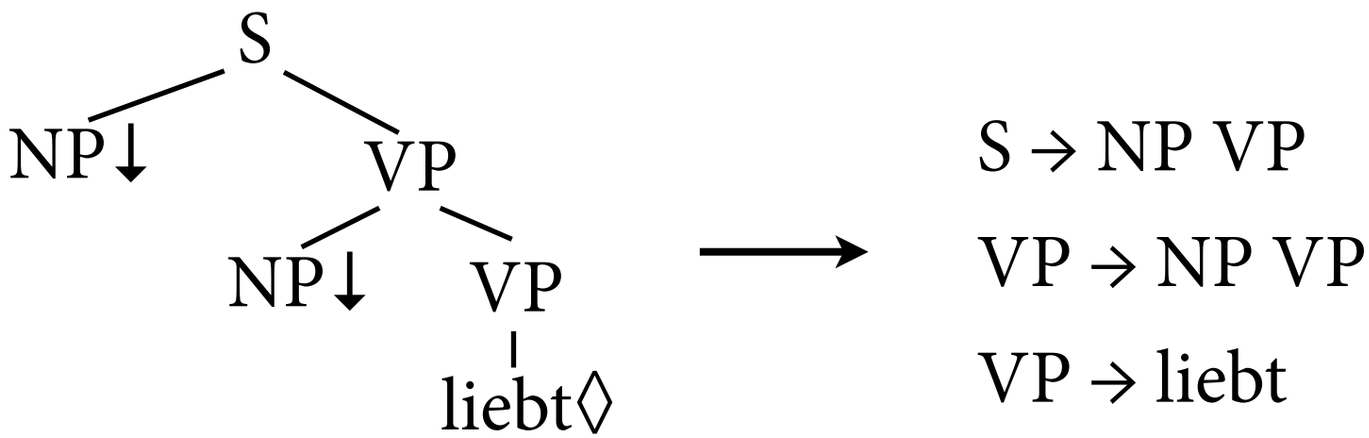
- Elementarbaum mit r Blättern: Regel enthält r+1 Variablen für Stringpositionen.
- Daher ist Laufzeit $O(n^{r+1})$.
- Das ist zu langsam.

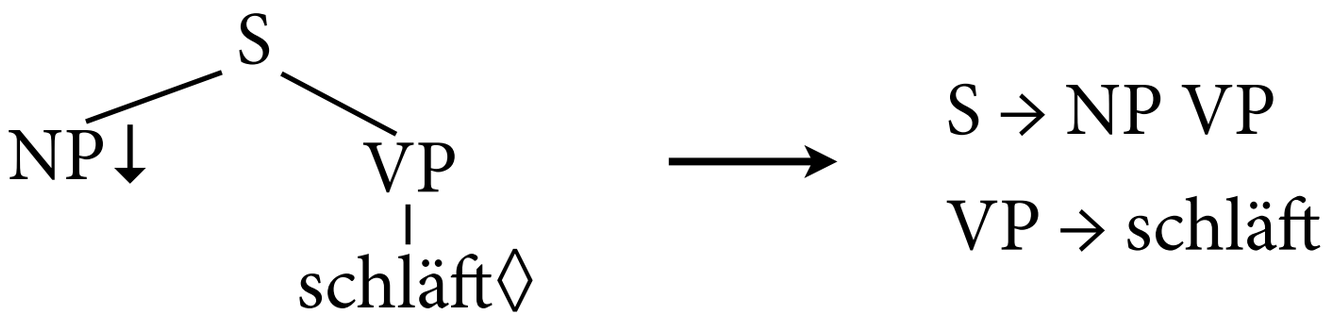
TSG-Parsing, 2. Ansatz

- Grundidee aus CNF-Transformation für kfGen: *Binarisierung*
  - r-stellige Regel in r-1 zweistellige Regeln aufbrechen
  - damit sinkt Rang r auf 2 ⇒ Laufzeit $O(n^{r+1}) = O(n^3)$
- Für TSG mit binären Elementarbäumen bietet es sich an, entlang der Baumstruktur aufzubrechen.

Gut aufpassen

- Die naive Lösung klappt nicht:

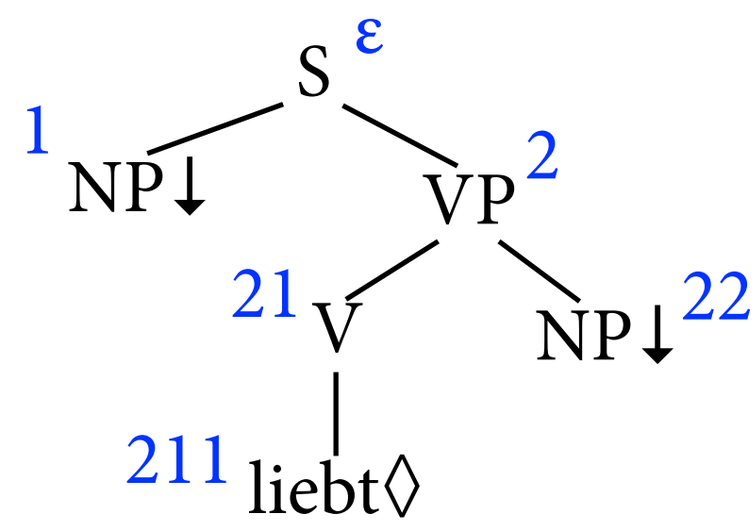




- Man könnte dann Teile verschiedener Elementarbäume miteinander kombinieren.

Pfade in Bäumen

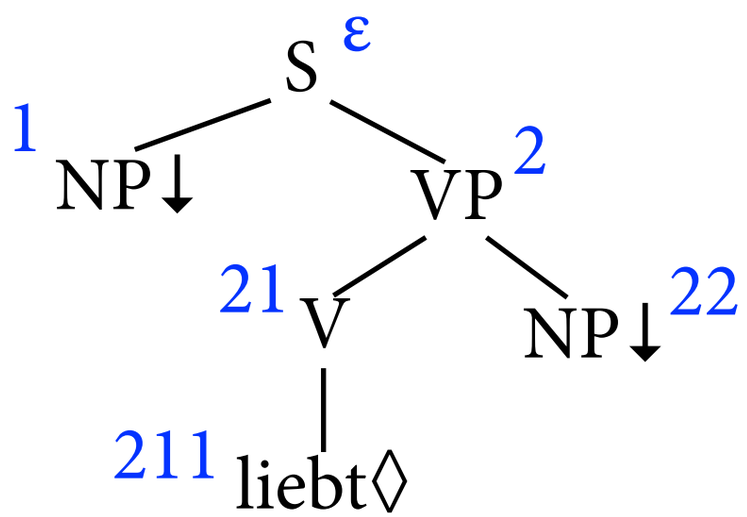
- Man kann jeden Knoten in einem Baum mit einem Wort aus **N*** beschreiben, das sagt, wie man von der Wurzel aus hinkommt.



- Diese Wörter heißen manchmal auch *Gorn-Adressen*.

Buchführen über Bäume

Lösung: eigene NT-Symbole für jeden Elementarbaum.



$$\mathrm{S} \rightarrow \mathrm{NP}_1^{\alpha_1}\ \mathrm{VP}_2^{\alpha_1}$$

$$\mathrm{VP}_2^{\alpha_1} \rightarrow \mathrm{V}_{21}^{\alpha_1}\ \mathrm{NP}_{22}^{\alpha_1}$$

$$\mathrm{V}_{21}^{\alpha_1} \rightarrow \text{liebt}$$

$$\mathrm{NP}_1^{\alpha_1} \rightarrow \mathrm{NP} \qquad \mathrm{NP}_{22}^{\alpha_1} \rightarrow \mathrm{NP}$$

Bsp. CKY-Regelinstanz:

$$\frac{[\mathrm{V}_{21}^{\alpha_1}, i, j] \quad [\mathrm{NP}_{22}^{\alpha_1}, j, k]}{[\mathrm{VP}_2^{\alpha_1}, i, k]}$$

Allgemein:

$$\frac{[\mathrm{B}_{1\,\pi 1}^{\alpha}, i_1, i_2] \quad \ldots \quad [\mathrm{B}_{\mathrm{n}\,\pi r}^{\alpha}, i_r, i_{r+1}]}{[\mathrm{A}_{\pi}^{\alpha}, i_1, i_{r+1}]}$$

CKY-Parser für TSG

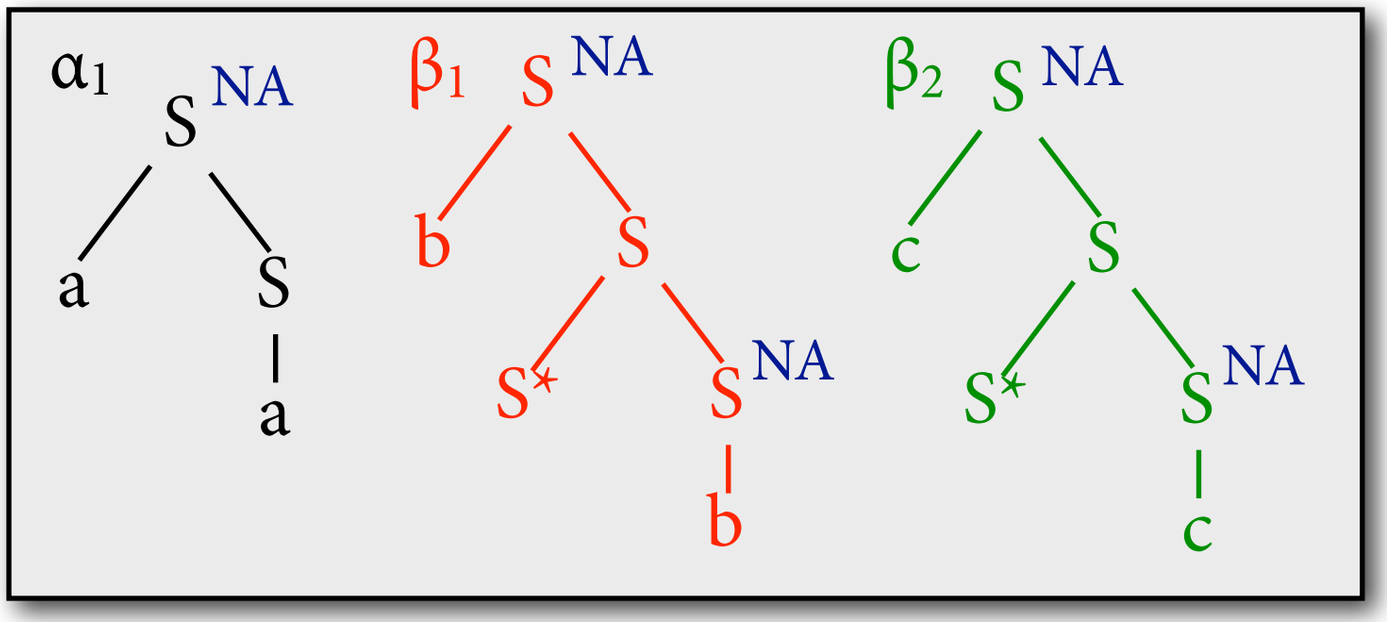
- Statt in Nichtterminalen kann man sich die Knoten auch direkt in den Parse-Items merken.

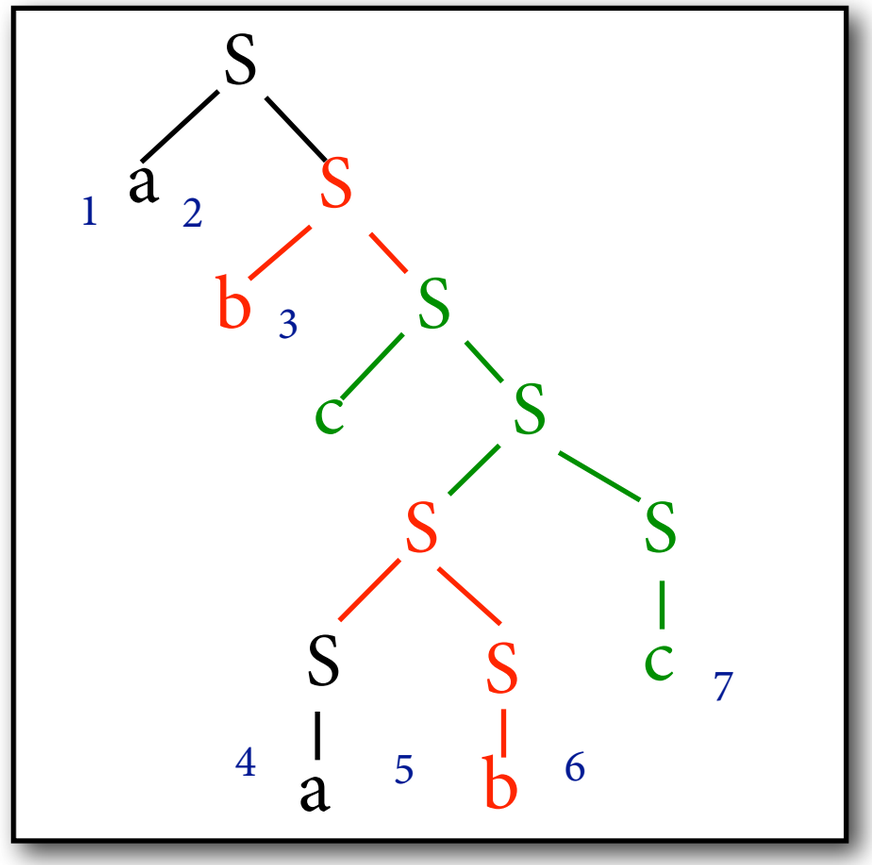
$$\frac{[B_1, i_1, i_2, \alpha, \pi 1] \ \ldots \ [B_r, i_r, i_{r+1}, \alpha, \pi r]}{[A, i_1, i_{r+1}, \alpha, \pi]} \ \text{conc11}$$

$$\frac{[A, i_1, i_2, \alpha', \epsilon] \quad \pi \text{ Substitionsknoten in } \alpha \text{ mit Label } A}{[A, i_1, i_2, \alpha, \pi]} \ \text{subst}$$

- Wir hatten $r \leq 2$ angenommen, daher Laufzeit: $O(n^3)$.

Jetzt zu TAG

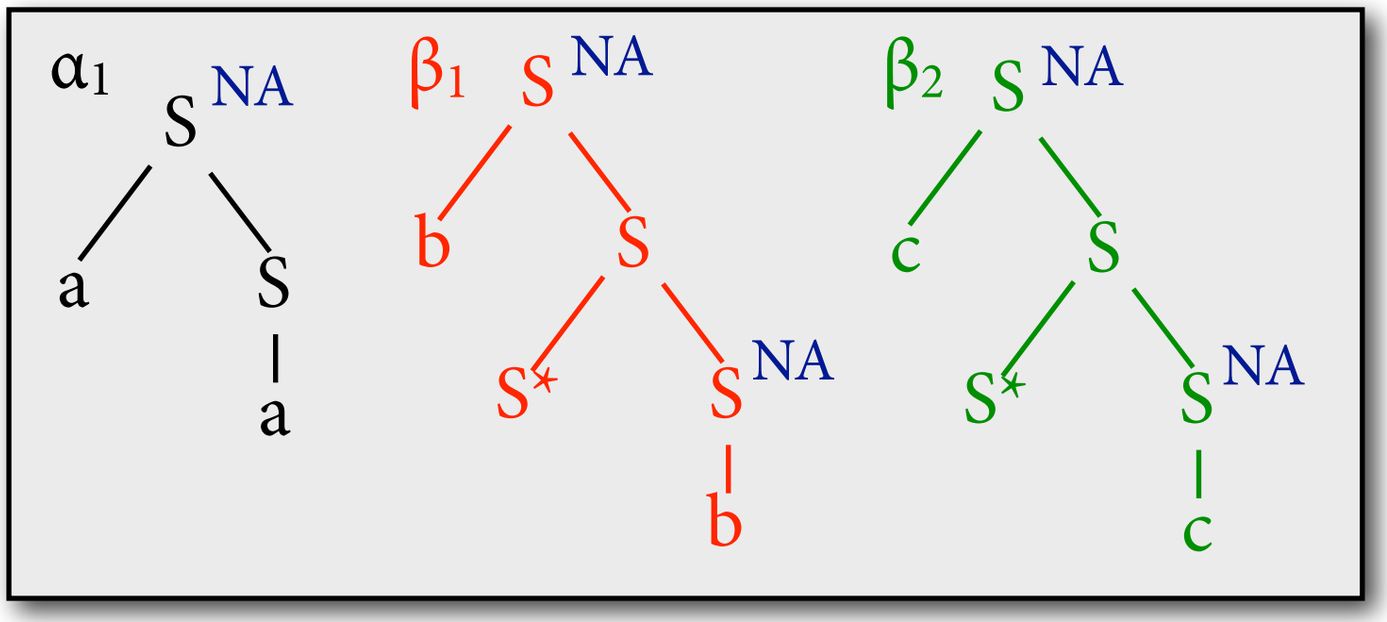


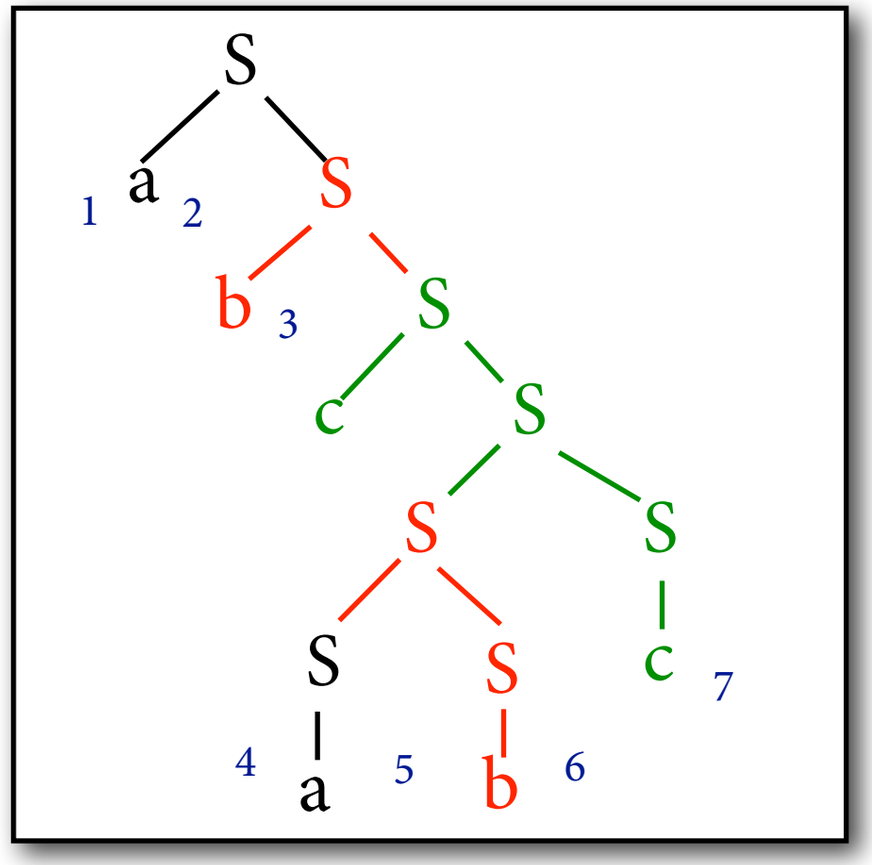


1. Versuch:

$$\frac{[S, 4, 5, \alpha_1, 2] \quad [S, 5, 6, \beta_1, 22]}{[S, 4, 6, \beta_1, 2]}$$

Jetzt zu TAG

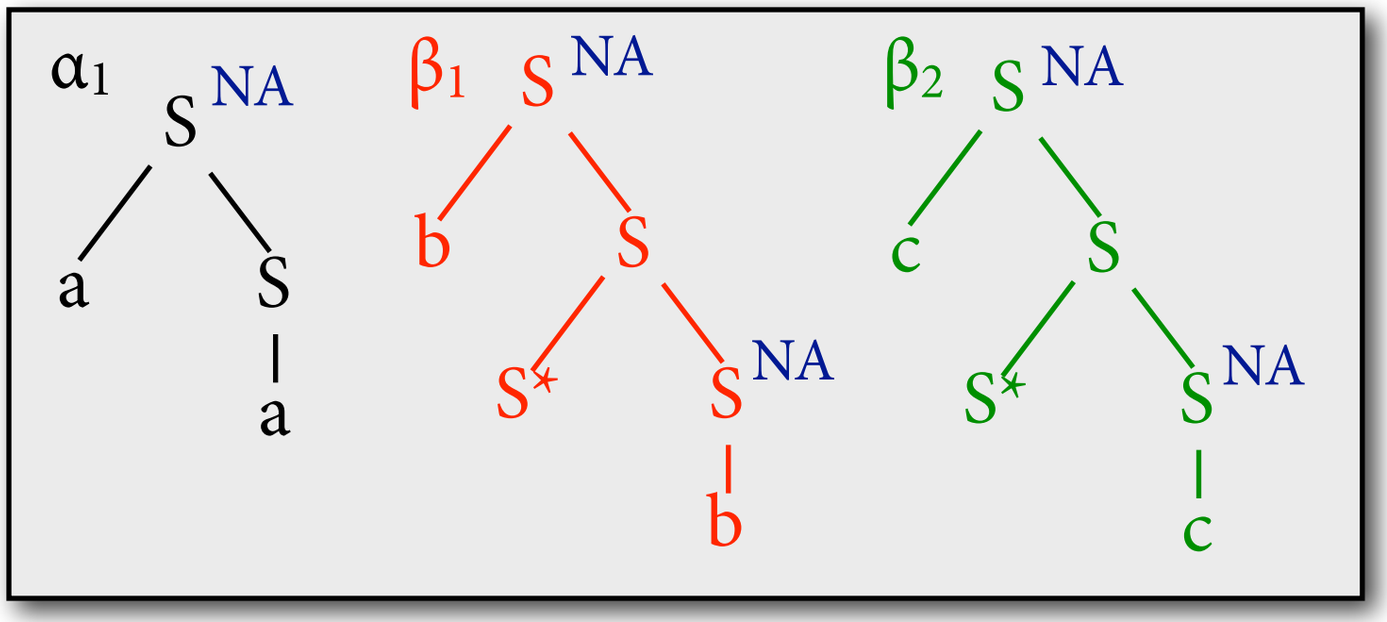


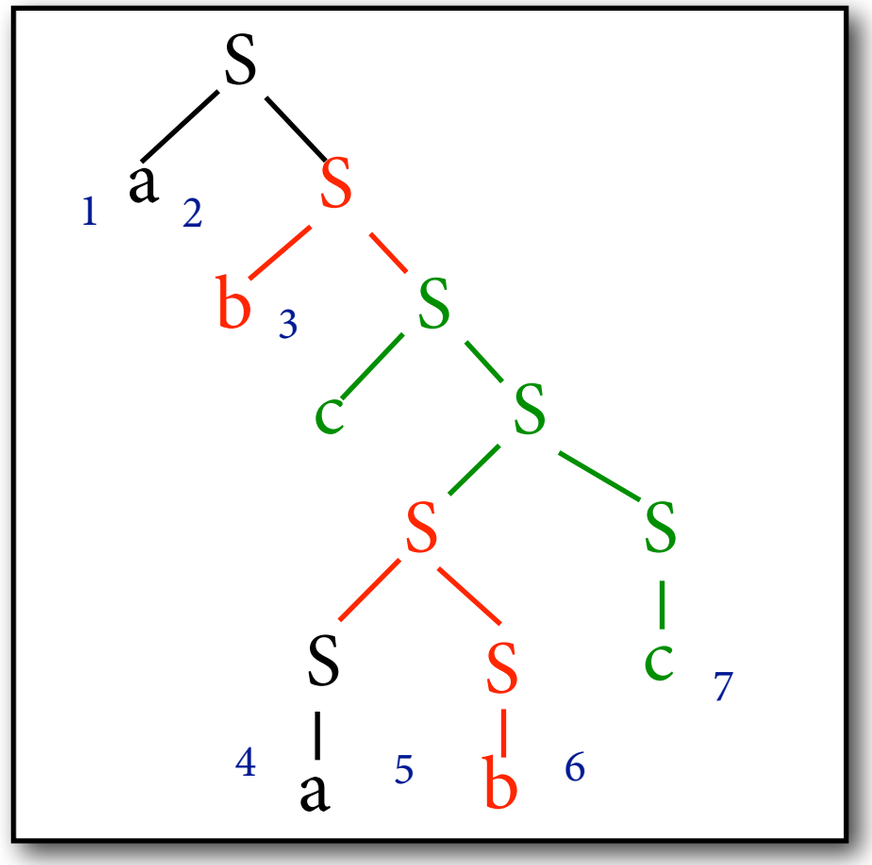


1. Versuch:

$$\frac{[S,4,5,\alpha_1,2] \quad [S,5,6,\beta_1,22]}{\begin{array}{c}[S,4,6,\beta_1,2] \\ \vdots \\ [S,2,7,\beta_1,\epsilon]\end{array}}$$

Jetzt zu TAG





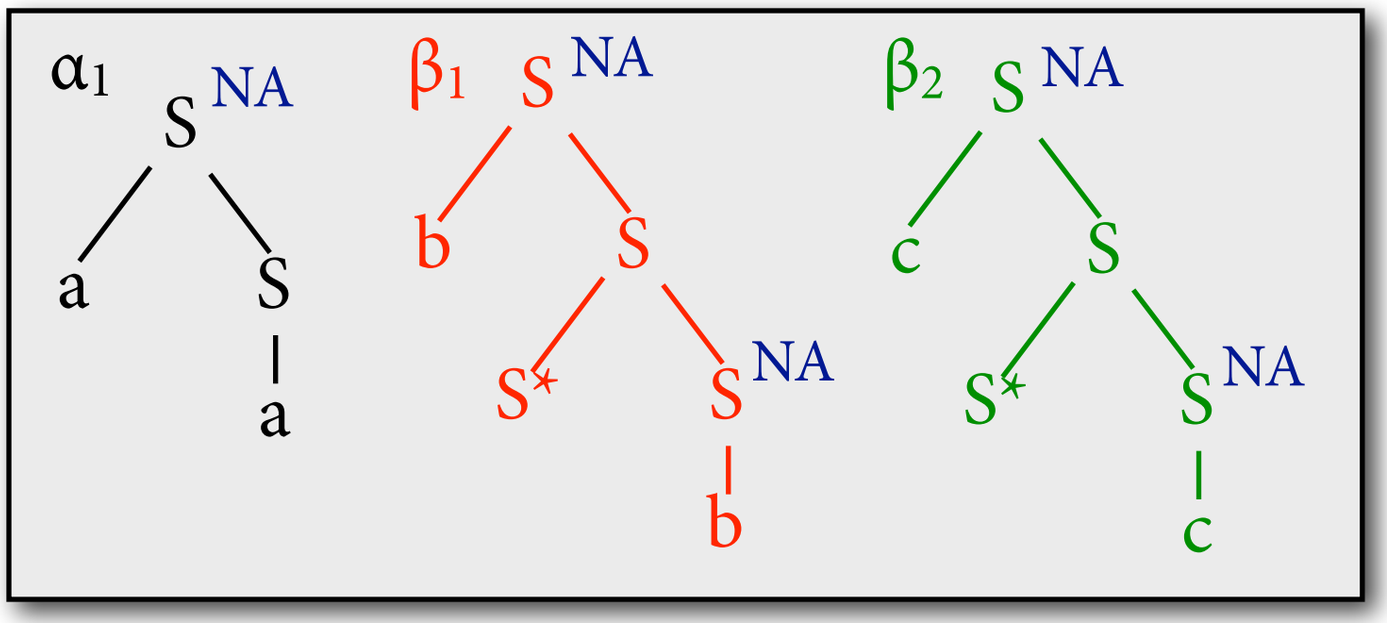
1. Versuch:

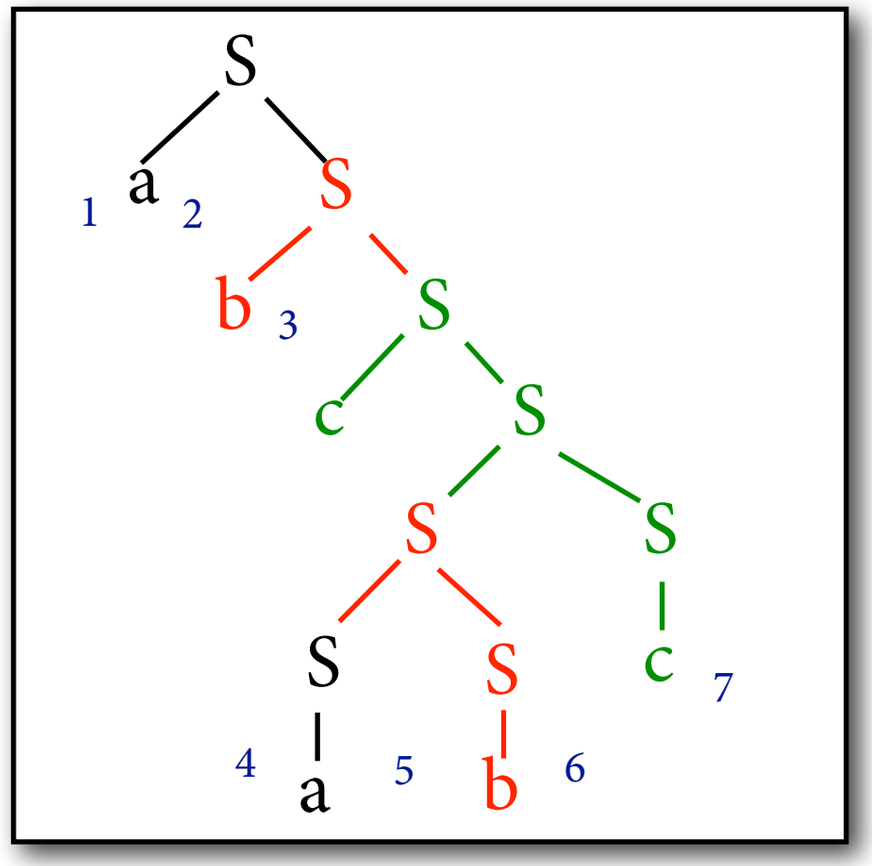
$$\dfrac{\dfrac{[S,4,5,\alpha_1,2] \quad [S,5,6,\beta_1,22]}{[S,4,6,\beta_1,2]}}{\vdots}$$

$$\dfrac{[S,2,7,\beta_1,\epsilon]}{\vdots}$$

$$\dfrac{[a,1,2,\alpha_1,1] \quad [S,2,7,\alpha_1,2]}{[S,1,7,\alpha_1,\epsilon]}$$

Jetzt zu TAG

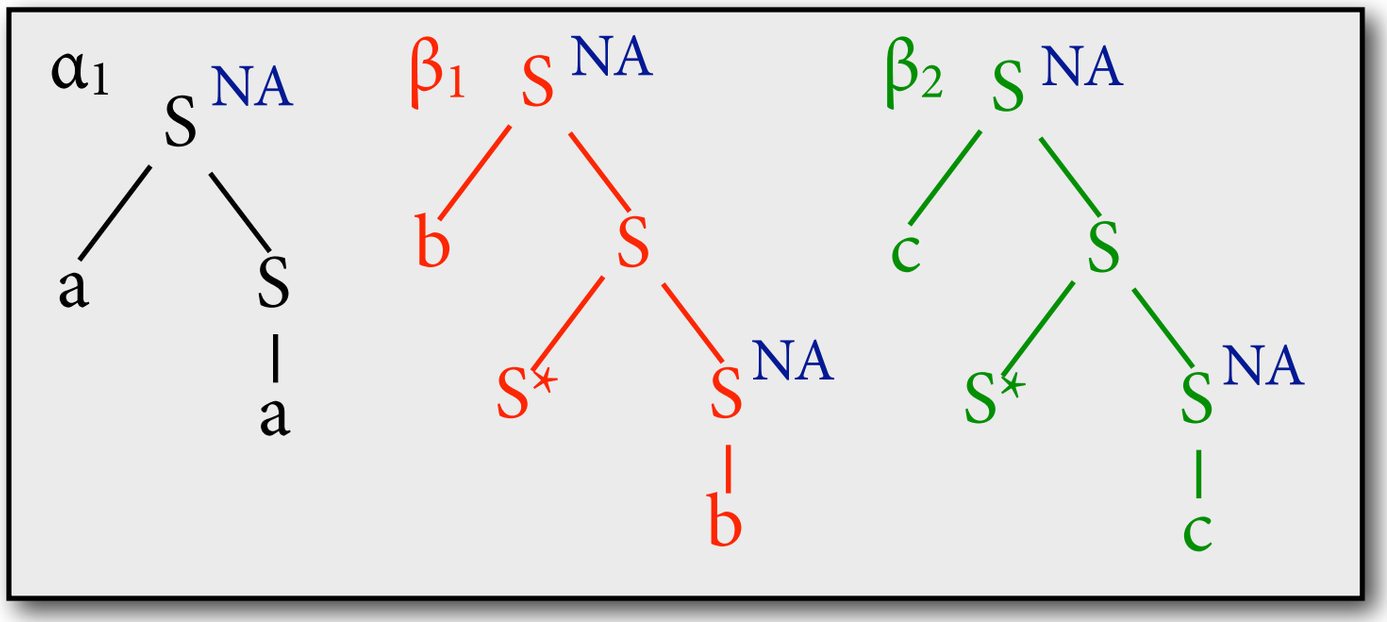


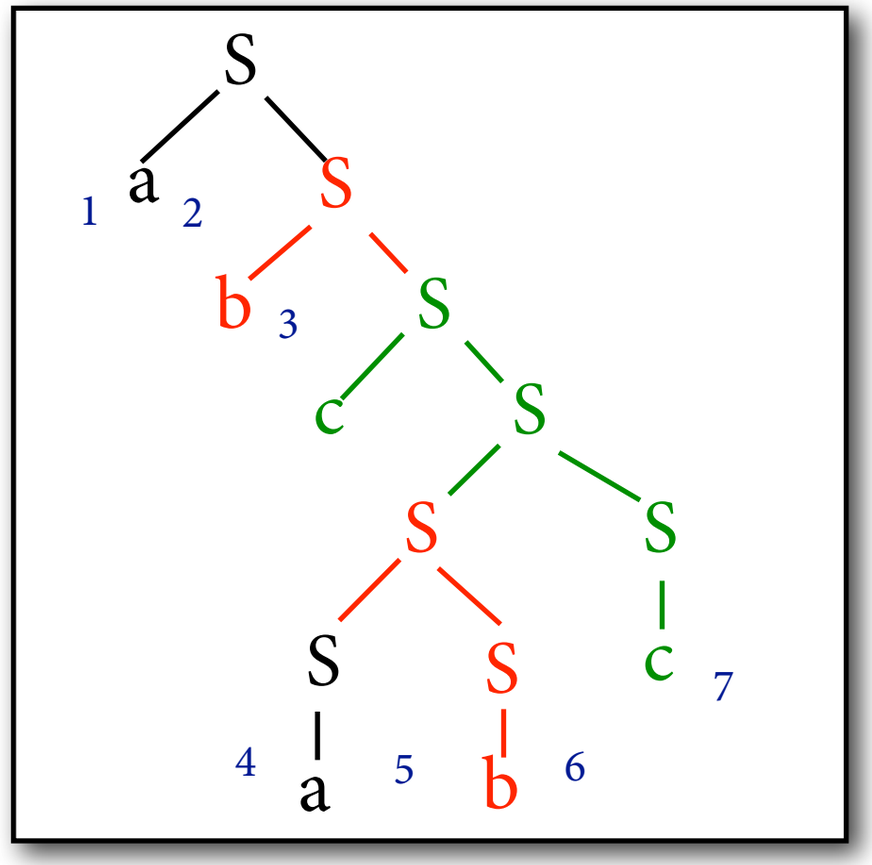


1. Versuch:

$$\dfrac{\dfrac{[a,1,2,\alpha_1,1] \quad [S,2,7,\alpha_1,2]}{[S,1,7,\alpha_1,\epsilon]} \quad \vdots ??? \quad \dfrac{[S,4,5,\alpha_1,2] \quad [S,5,6,\beta_1,22]}{\dfrac{[S,4,6,\beta_1,2]}{\vdots}}}{[S,2,7,\beta_1,\epsilon]}$$

Jetzt zu TAG





1. Versuch:

$$\dfrac{\dfrac{[S,4,5,\alpha_1,2] \quad [S,5,6,\beta_1,22]}{[S,4,6,\beta_1,2]}}{\vdots}$$

$$\vdots$$

$$\dfrac{[S,2,7,\beta_1,\epsilon]}{\vdots \; ???}$$

$$\dfrac{[a,1,2,\alpha_1,1] \quad [S,2,7,\alpha_1,2]}{[S,1,7,\alpha_1,\epsilon]}$$

Problem

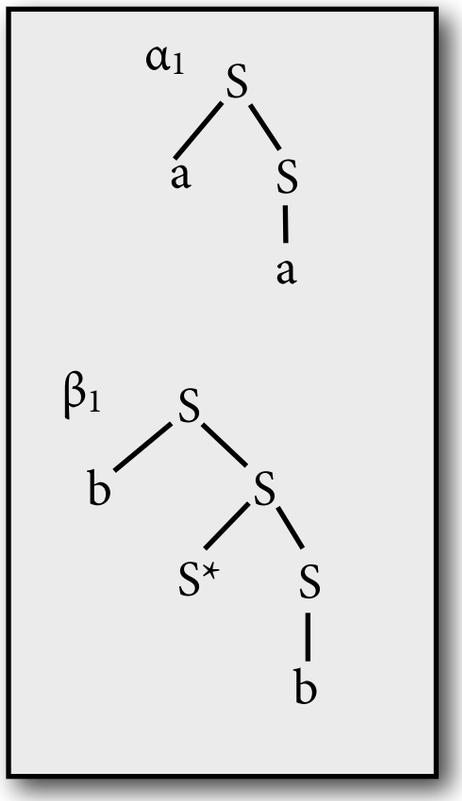
- Bei einer Adjunktion von β in α verarbeitet Parser erst mal β und “vergisst”, dass er noch den Rest von α verarbeiten muss.

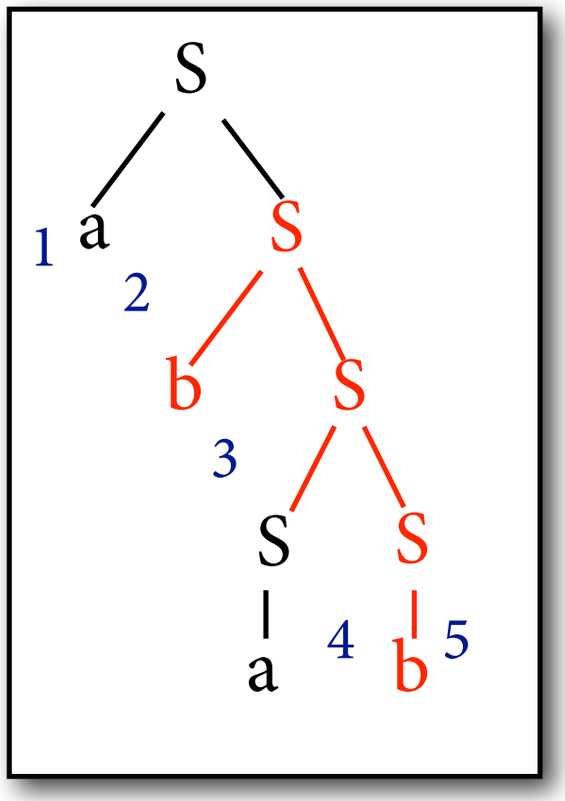
- Eine Lösung: Items enthalten Stack von unverarbeiteten Baumteilen.

- Führt aber zu exponentieller Laufzeit.
  - Zeige ich Ihnen in einer analogen Situation bei CCG.

CKY-Parser für TAG

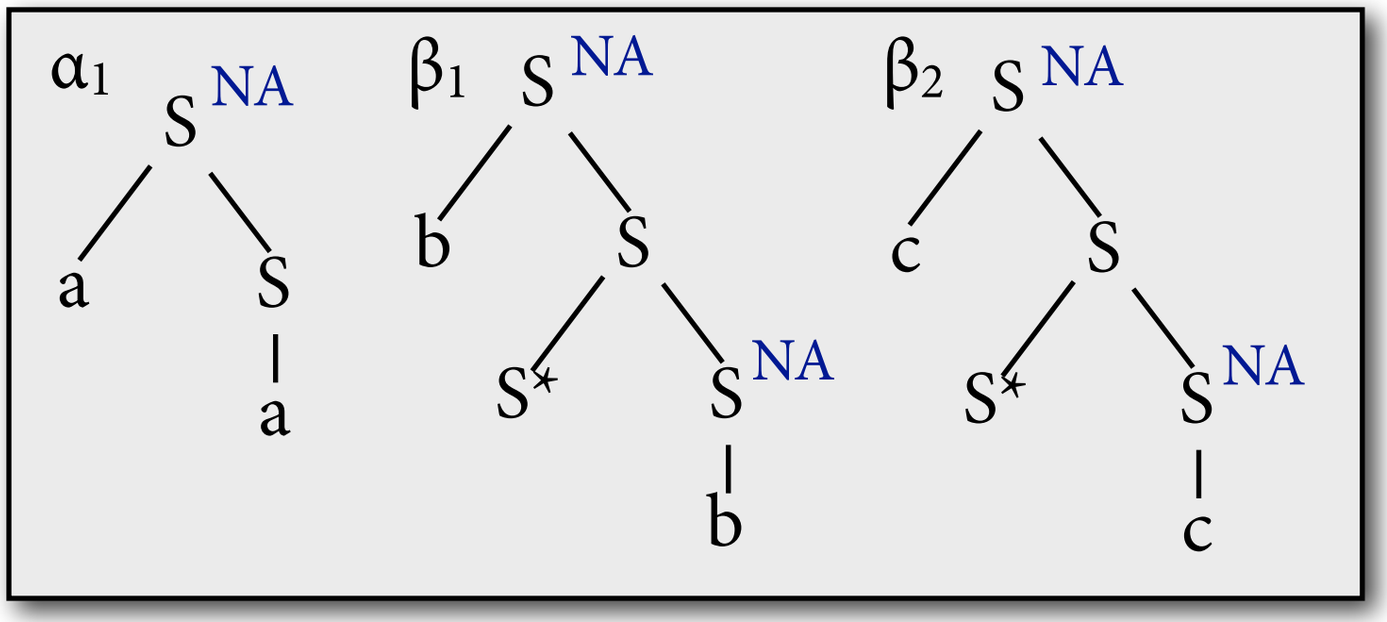
- Zwei Typen von Items:
  - [A,i,k,α,π]: Abgeleiteter Baum unter Knoten π in α deckt vollständigen String von i bis k ab (wie in TSG).
  - <A,i,j,k,l,β,π>: Abgeleiteter Baum unter π in β deckt String von i-j *sowie String von k-l* ab; dazwischen ist “Lücke”.

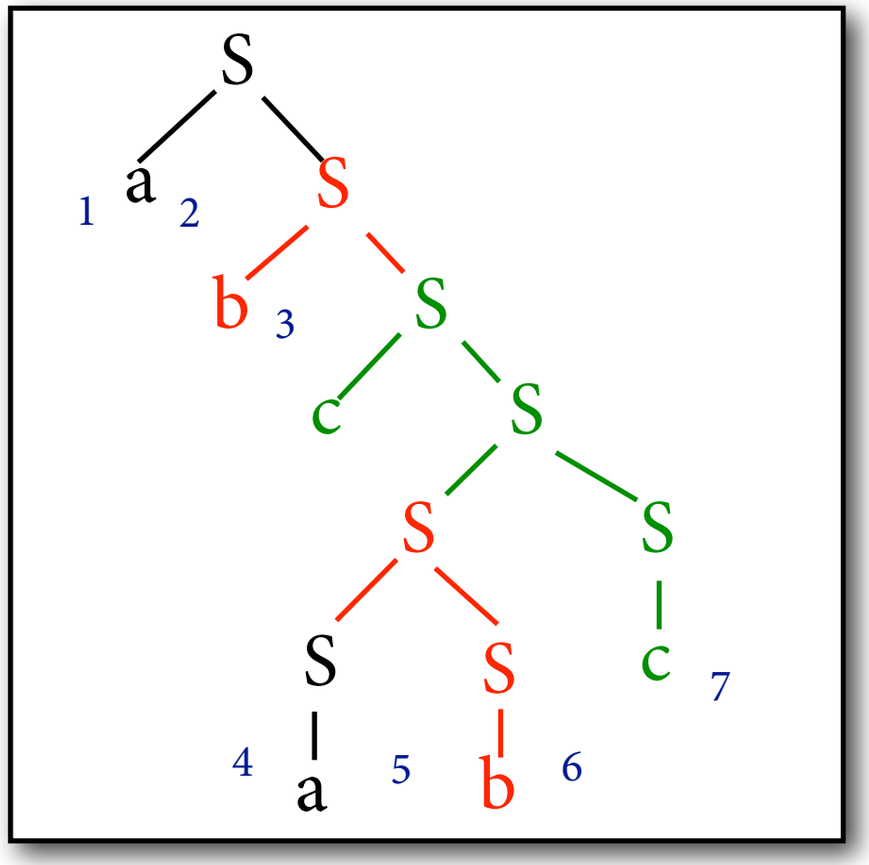




$$\frac{\langle S, 2, 3, 4, 5, \beta_1, \epsilon\rangle \quad [S, 3, 4, \alpha_1, 2]}{[S, 2, 5, \alpha_1, 2]}$$

Beispiel





$$\dfrac{\dfrac{[S,1\text{–}2,\alpha_1,1] \qquad \dfrac{\dfrac{[S,2\text{–}3,\beta_1,1] \qquad \dfrac{(\text{aus } \beta_2) \qquad \langle S,4\text{–}4,5\text{–}5,\beta_1,21\rangle \quad [S,5\text{–}6,\beta_1,22]}{\langle S,3\text{–}4,6\text{–}7,\beta_2,\epsilon\rangle}\, conc21}{\langle S,4\text{–}4,5\text{–}6,\beta_1,2\rangle}\, wrap22}{\langle S,3\text{–}4,5\text{–}7,\beta_1,2\rangle}\, conc12}{\langle S,2\text{–}4,5\text{–}7,\beta_1,\epsilon\rangle} \qquad [S,4\text{–}5,\alpha_1,2]}{[S,2\text{–}7,\alpha_1,2]}\, wrap21}{[S,1\text{–}7,\alpha_1,\epsilon]}\, conc11$$

TAG-Parser: Regeln

$$\frac{\langle A, i_1, i_2, i_3, i_4, \beta, \epsilon\rangle \quad [A, i_2, i_3, \alpha, \pi]}{[A, i_1, i_4, \alpha, \pi]} \text{ wrap21}$$

$$\frac{\text{Knoten } \pi \text{ von } \alpha \text{ ist Wort } w_i}{[w_i, i, i+1, \alpha, \pi]} \text{ lex}$$

$$\frac{\text{Knoten } \pi \text{ von } \beta \text{ ist Fußknoten } A*, \text{ und } i < k}{\langle A, i, i, k, k, \beta, \pi\rangle} \text{ foot}$$

NB: “foot” leitet Items für beliebige i, k ab.

TAG-Parser: Regeln

$$\frac{\langle B, i_1, i_2, i_3, i_4, \beta, \pi 1\rangle \quad [C, i_4, i_5, \beta, \pi 2]}{\langle A, i_1, i_2, i_3, i_5, \beta, \pi\rangle} \text{ conc21}$$

$$\frac{[B, i_1, i_2, \beta, \pi 1] \quad \langle C, i_2, i_3, i_4, i_5, \beta, \pi 2\rangle}{\langle A, i_1, i_3, i_4, i_5, \beta, \pi\rangle} \text{ conc12}$$

$$\frac{\langle A, i_1, i_2, i_5, i_6, \beta_1, \epsilon\rangle \quad \langle A, i_2, i_3, i_4, i_5, \beta_2, \pi\rangle}{\langle A, i_1, i_3, i_4, i_6, \beta_2, \pi\rangle} \text{ wrap22}$$

Diskussion

- Laufzeit: Teuerste Regel ist wrap22.
  Sechs unabhängige Stringpositionen, also $O(n^6)$.

- Kann Algorithmus auf TAG-Grammatiken mit mehr als zwei Kindern pro Knoten erweitern (analog Earley-Parser für kfGs).

- Algorithmus erlaubt mehrfache Adjunktion am gleichen Knoten. Kann man verbieten: Item merkt sich, ob schon adjungiert wurde.

Zusammenfassung

- Expressivität von TAG:
  kann COPY, COUNT(4), nicht mehr COUNT(5)

- Parsing von TAG: mit CKY-artigem Parser in $O(n^6)$.