Der CKY-Parser

Vorlesung “Computerlinguistische Techniken”
Alexander Koller

3. November 2014

Übersicht

- Laufzeit von rekursionsbasierten Parsern
- Asymptotische Laufzeiten
- Der CKY-Parser
- Komplexität des CKY-Algorithmus
- Implementierung in Python

Shift-Reduce-Parsing

- Shift-Regel:
  $(a \cdot w, s) \rightarrow (w, s \cdot a)$

- Reduce-Regel:
  $(w, s \cdot w') \rightarrow (w, s \cdot A)$ falls $A \rightarrow w'$ in P

- Start: (w, ε)

- Wende Regeln nichtdeterministisch an,
  bis Zustand (ε, S) erreicht ist

Shift-Reduce: Ein Problem

- Es gibt Strings, die der SR-Parser nur sehr ineffizient erkennt.

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

b b ... b c ?

- Alle Sätze dieser Form sind zwar in der Sprache, aber Parser führt trotzdem fast ganze Zeit erfolglose Berechnungen durch.

Shift-Reduce: Ein Problem

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

b b b c

Shift-Reduce: Ein Problem

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

b b b c

$\rightarrow^*$ C C C T $\rightarrow^*$ T ✘

Shift-Reduce: Ein Problem

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

b b b c

$\rightarrow^*$ C C C T $\rightarrow^*$ T ✘

$\rightarrow^*$ C C B T ✘

Shift-Reduce: Ein Problem

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

b b b c

$\rightarrow^*$ C C C T $\rightarrow^*$ T ✗

$\rightarrow^*$ C C B T ✗

$\rightarrow^*$ C B C T $\rightarrow^*$ C B T ✗

Shift-Reduce: Ein Problem

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

b b b c

$\rightarrow^*$	C	C	C	T	$\rightarrow^*$	T ✗
$\rightarrow^*$	C	C	B	T ✗		
$\rightarrow^*$	C	B	C	T	$\rightarrow^*$	C B T ✗
$\rightarrow^*$	C	B	B	T ✗		

etc.

Shift-Reduce: Ein Problem

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

b b b c

$\rightarrow^*$	C	C	C	T	$\rightarrow^*$ T	✗
$\rightarrow^*$	C	C	B	T	✗	
$\rightarrow^*$	C	B	C	T	$\rightarrow^*$ C B T	✗
$\rightarrow^*$	C	B	B	T	✗	

etc.

Shift-Reduce: Ein Problem

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

b b b c

$\rightarrow^*$	C	C	C	T	$\rightarrow^*$	T ✗
$\rightarrow^*$	C	C	B	T ✗		
$\rightarrow^*$	C	B	C	T	$\rightarrow^*$	C B T ✗
$\rightarrow^*$	C	B	B	T ✗		

etc.

SR-Erkenner muss für String der Länge n bis zu 2^n Kombinationen durchprobieren.

Shift-Reduce: Ein Problem

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

b b b c

$\rightarrow^*$	C	C	C	T	$\rightarrow^*$ T ✗
$\rightarrow^*$	C	C	B	T ✗	
$\rightarrow^*$	C	B	C	T	$\rightarrow^*$ C B T ✗
$\rightarrow^*$	C	B	B	T ✗	

etc.

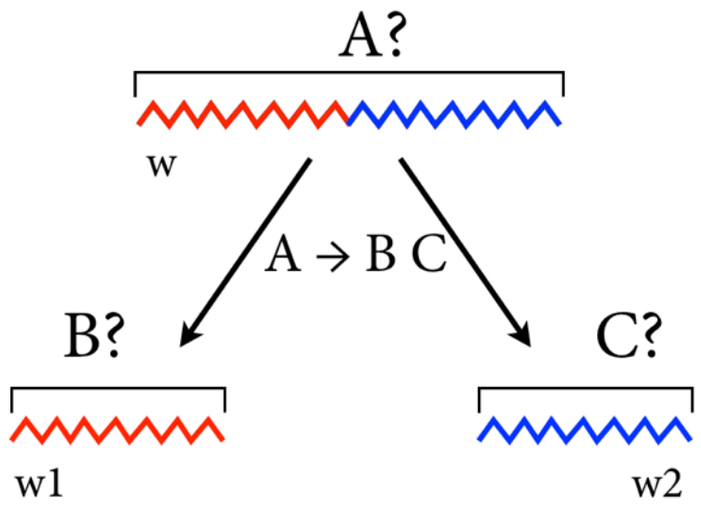
SR-Erkenner muss für String der Länge n bis zu 2^n Kombinationen durchprobieren.

Probleme und Algorithmen

Problem

Wortproblem(G,w) = 1
gdw w ∈ L(G)

Algorithmus




```
s(W)  :- append(W1,W2,W), np(W1), vp(W2).
vp(W) :- append(W1,W2,W), v(W1), np(W2).
np(W) :- append(W1,W2,W), det(W1), n(W2).

np([hans]).
n([kaesebroetchen]).

det([ein]).

v([isst]).
```

Programm

Laufzeit

- Laufzeit eines Algorithmus auf einer Eingabe ist Anzahl der Berechnungsschritte.

- Um Algorithmen zu vergleichen, interessiert man sich für Laufzeit
  - als Funktion der Größe der Eingabe
  - für den worst case (= die schwersten Eingaben)
  - asymptotisch (= ohne konstante Faktoren)

Ein Beispiel

- Wie lange dauert es, zu erkennen, ob ein Wert x ein Element einer Liste L ist?

```
x = 5
L = [1,2,3,7]

for i in range(len(L)):
    if x == L[i]:
        print "yes"
```

x	len(L)	Schleifendurchläufe
1	4	1
2	4	2
7	4	4
5	4	4
5	5	5
5	6	6
5	7	7

- Wie lange dauert ein Schleifendurchlauf?

Analyse

- Wichtige Eckdaten der Laufzeit:
  - Eingabegröße len(L): Länge der Liste.
  - Im worst-case (Element steht am Schluss) brauchen wir len(L) Durchläufe der Schleife.
  - Details der Zeit pro Schleifendurchlauf sind uns egal.

- Uns reicht es zu sagen, dass Laufzeit *linear* in der Eingabegröße wächst.
  - Abstraktion über Implementierungsdetails
  - reicht für asymptotischen Vergleich von Laufzeitklassen

O-Notation

- Asymptotische Laufzeit eines Algorithmus: Abstrahiert über Implementierungsdetails.

- Seien f, g Funktionen. Definition:

  f = O(g) gdw.
  ex. c, n_0 mit $f(n) \leq c \cdot g(n)$ f.a. $n \geq n_0$

- Man nimmt normalerweise kleinstes g, für das f = O(g).

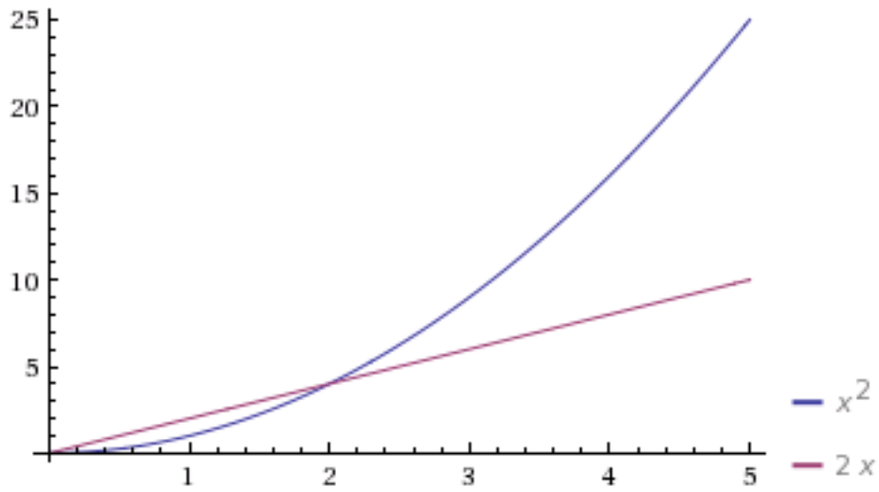
Zurück zu den Listen

$f = O(g)$ gdw.
ex. c, n_0 mit $f(n) \leq c \cdot g(n)$ f.a. $n \geq n_0$

- In unserem Listen-Element-Programm ist:
  - f = Laufzeit
  - n = Länge der Liste
  - c = Laufzeit eines Schleifendurchlaufs
  - $g(n) = n$
  - $n_0 = 0$; damit ist $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$, also $f = O(n)$.

Hierarchie von Laufzeitklassen

- Für alle c, c’ wird ab einer bestimmten Stelle
  $c \cdot n \leq c' \cdot n^2$

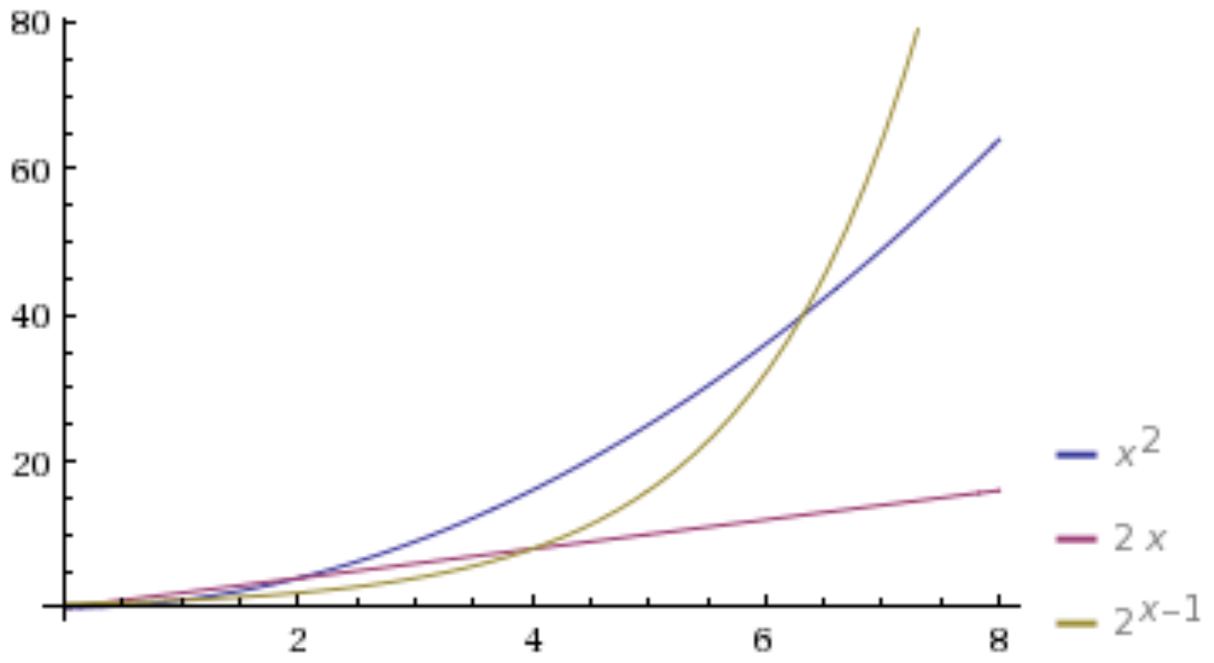


- Für große n also kleine Polynome schneller:
  - $O(n)$ linear < $O(n^2)$ quadratisch
    (auch für n + 5, 100 * n - 27 usw.)
  - $O(n^2)$ quadratisch < $O(n^3)$ kubisch
  - etc.

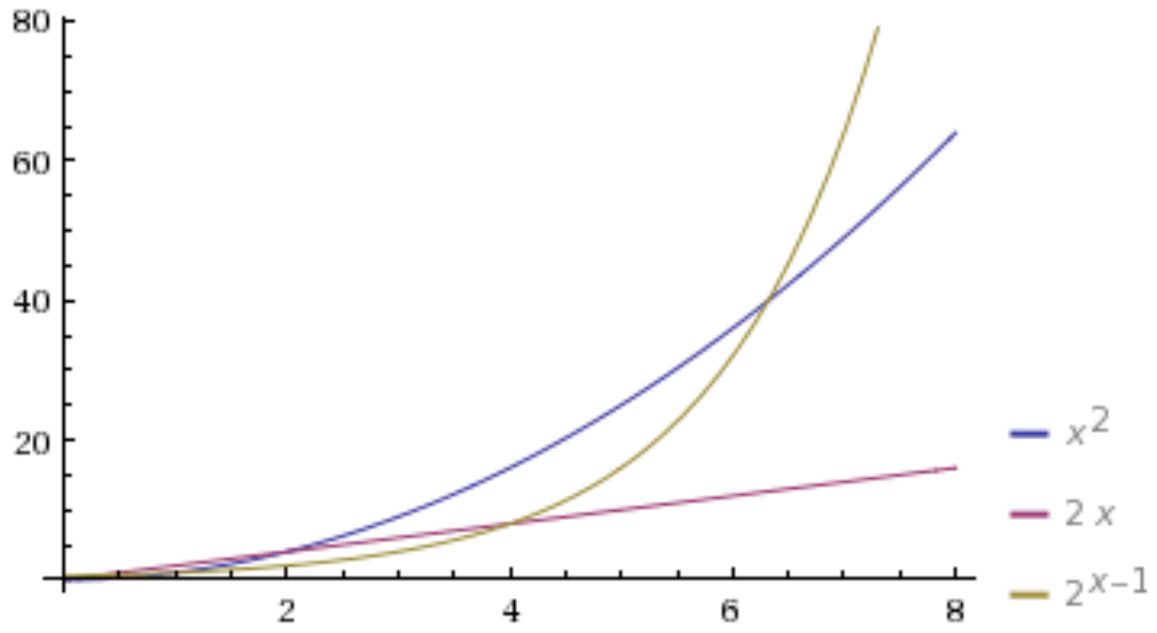
Exponentielle Laufzeit

- Worst-Case-Laufzeit von Shift-Reduce:
  2^n Berechnungsschritte (n ist Länge des Strings).

- Exponentialfunktion wächst schneller als jedes Polynom: Es gibt kein k, so dass $2^n = O(n^k)$.



Polynomiell vs. exponentiell



- Man unterscheidet deshalb oft zwischen polynomiell und exponentiell.
  - Faustregel: exponentiell = langsam.

- Gibt es einen polynomiellen Algorithmus für das Wortproblem von kontextfreien Grammatiken?

Was ist das Problem?

- Warum braucht der SR-Parser exponentielle Laufzeit?

- Zwischenergebnisse werden mehrfach berechnet. Versuchen wir das zu vermeiden.

	b	b	b	c
$\rightarrow^*$	C	C	C	T
$\rightarrow^*$	C	C	B	T
$\rightarrow^*$	C	B	C	T
$\rightarrow^*$	C	B	B	T

Auswege

- Für top-down: *Memoisierung*. Speichere frühere berechnete Zwischenergebnisse in Tabelle und schlage sie beim zweiten Aufruf nach.

- Für bottom-up: *Dynamisches Programmieren* (auch bekannt als *Chart-Parsing*). Algorithmus arbeitet direkt auf einer Tabelle (der Chart).

Der CKY-Parser

- Einfachster Chartparser für kfGs in CNF.

- Erfunden in den 1960ern von
  Cocke, Younger, Kasami;
  heißt manchmal auch CYK-Parser.

- Bottom-up; berechnet Aussagen der Form
  "$A \Rightarrow^* w_i \ldots w_{k-1}$?".

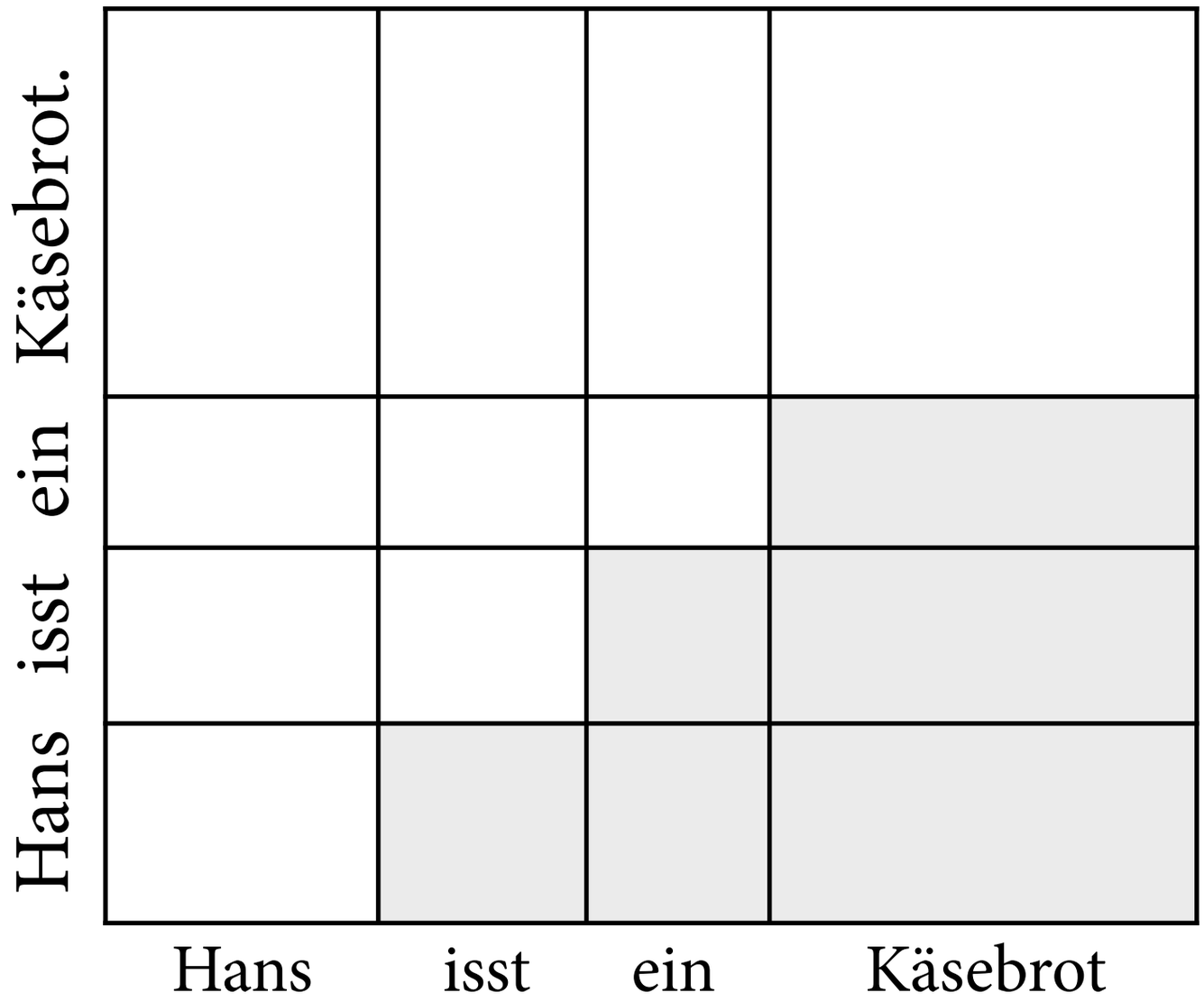
Der CKY-Parser

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

Hans isst ein Käsebrot

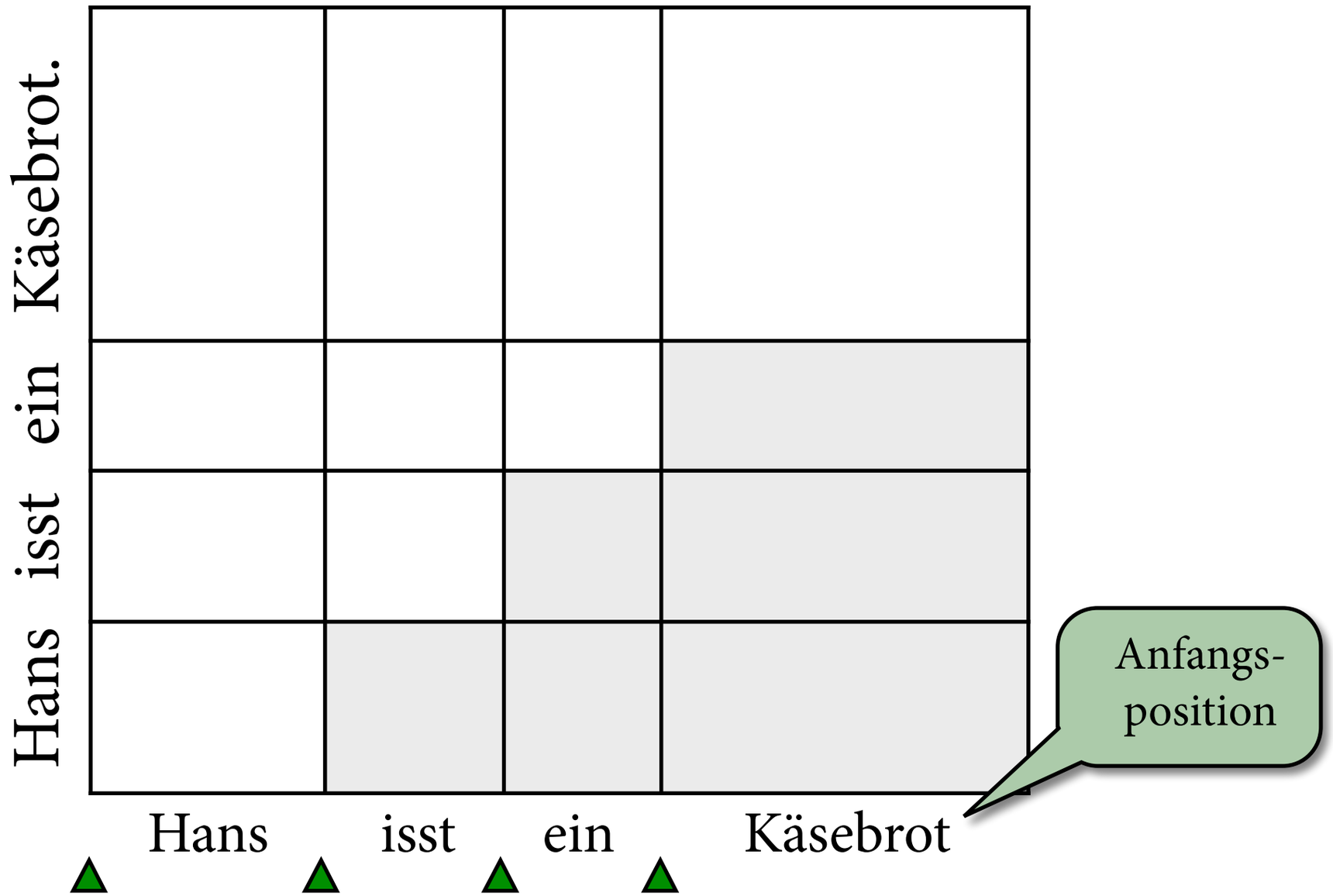
Der CKY-Parser

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



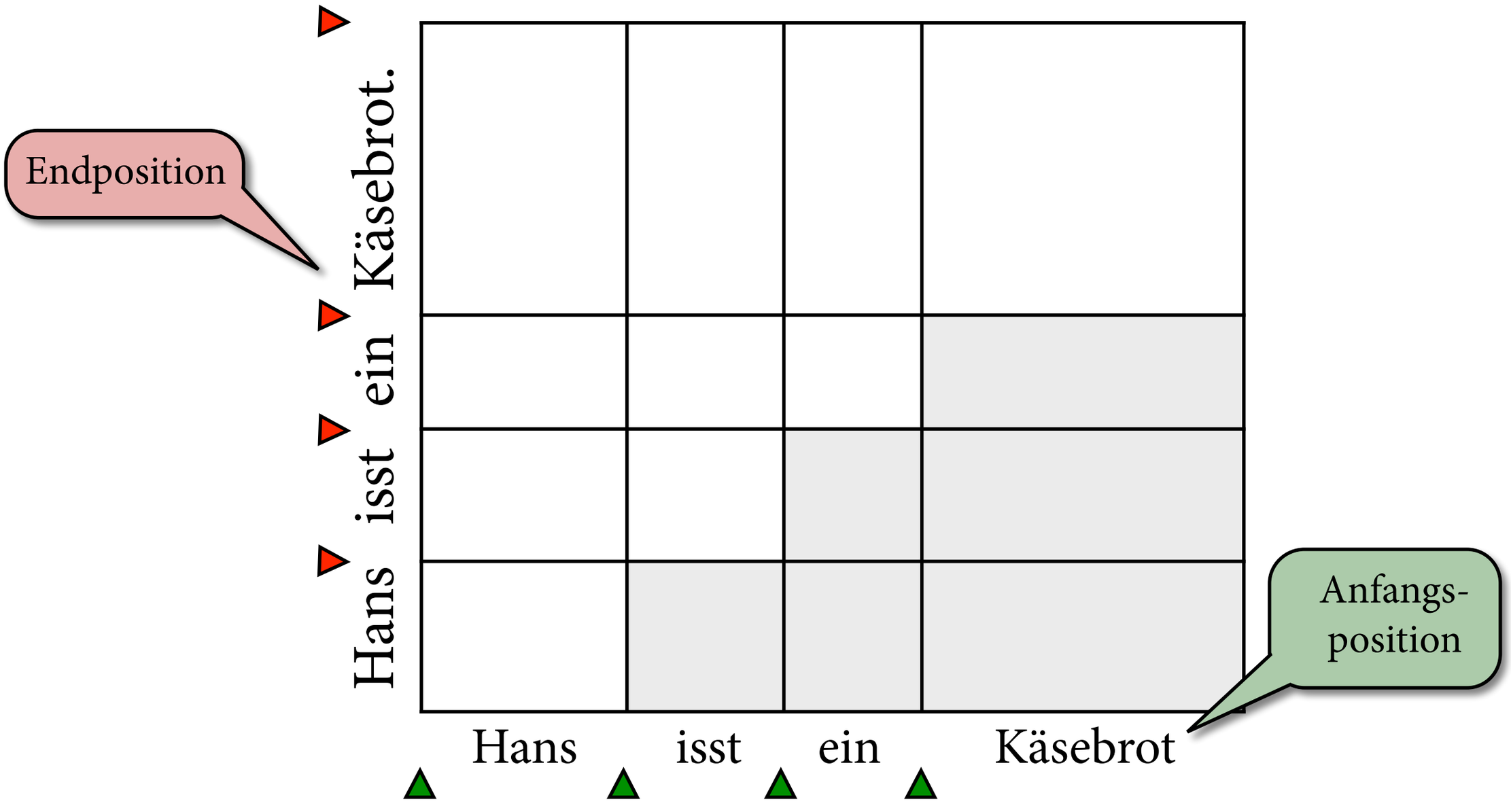
Der CKY-Parser

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



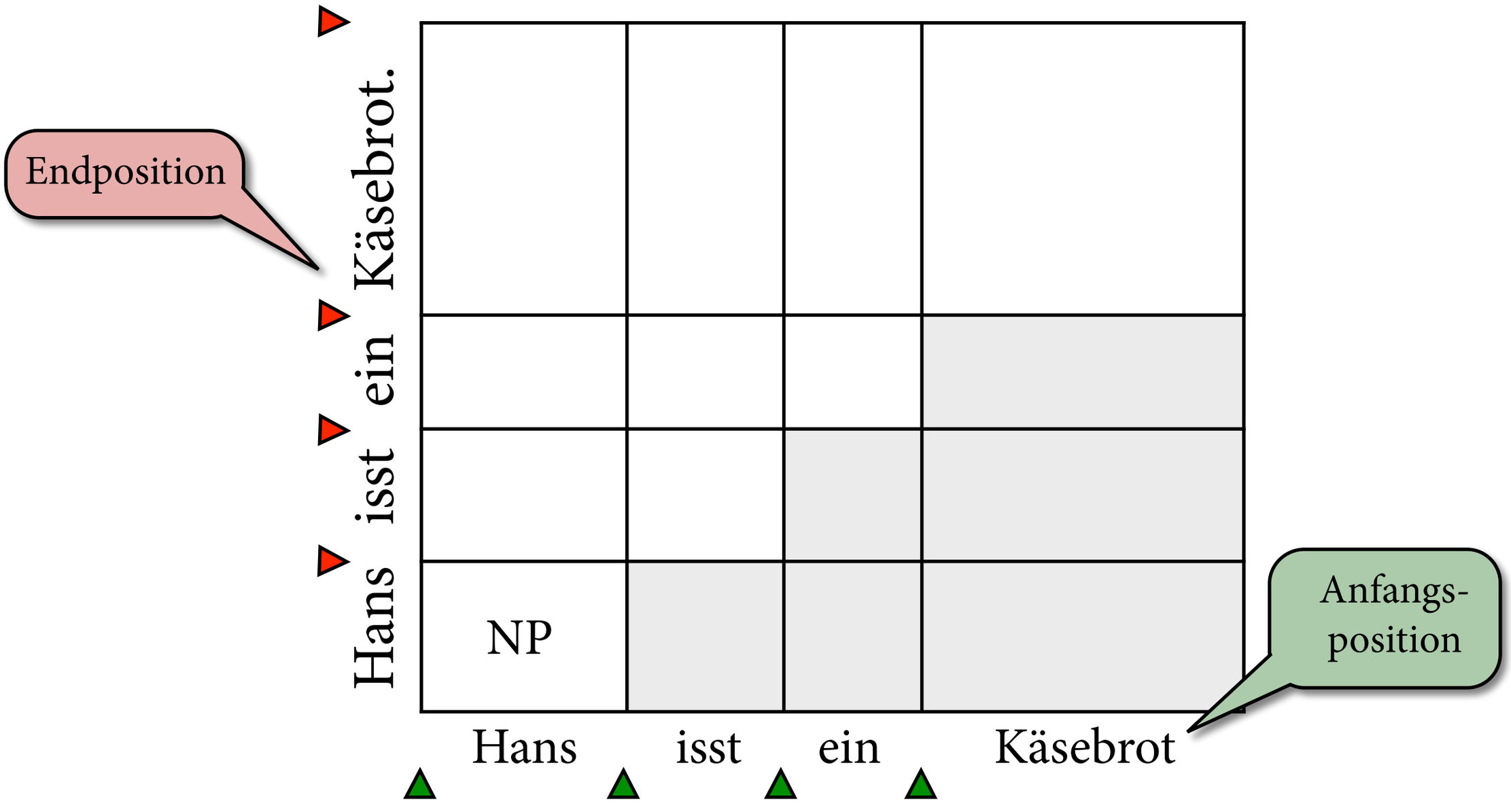
Der CKY-Parser

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



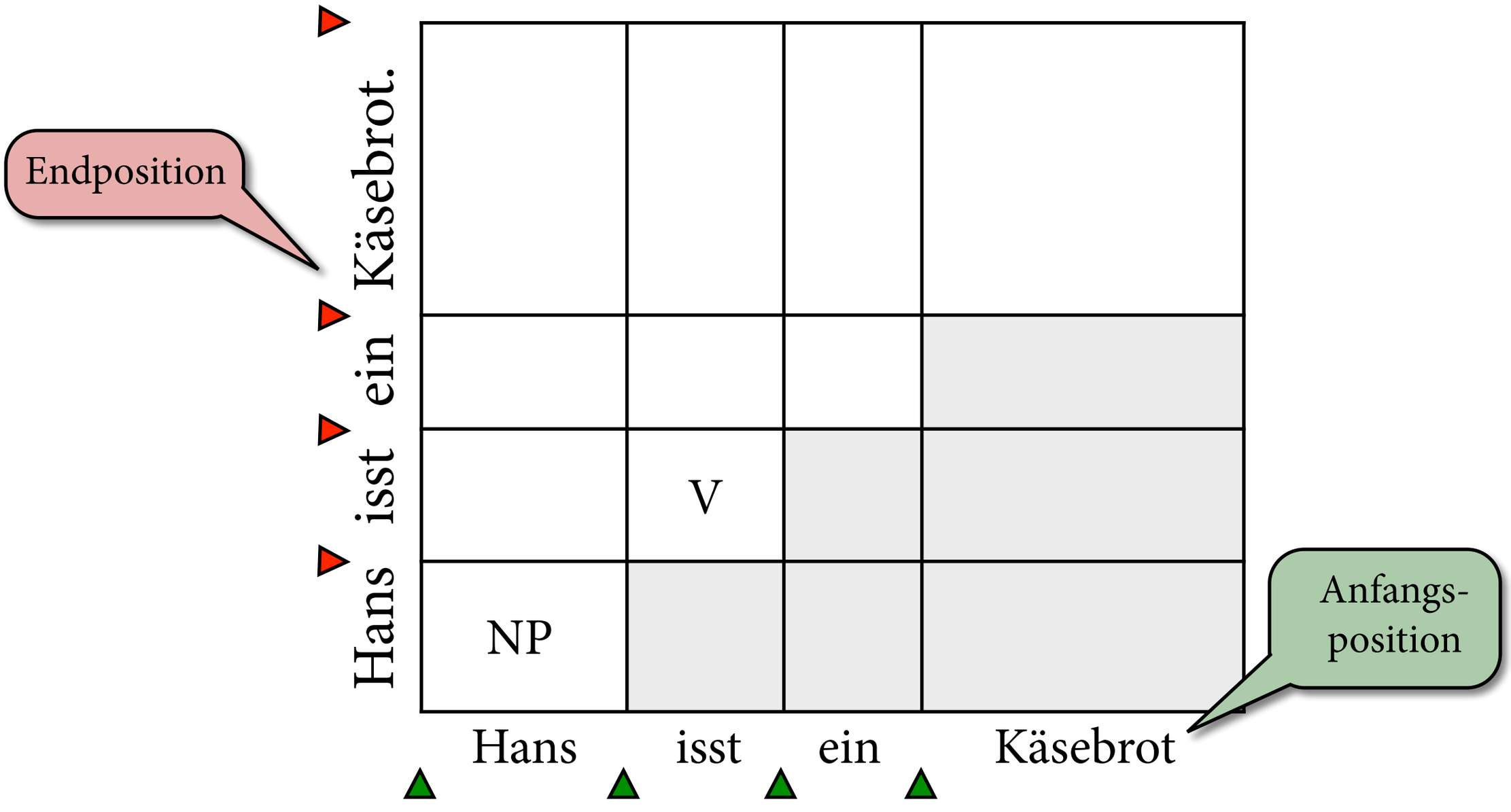
Der CKY-Parser

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



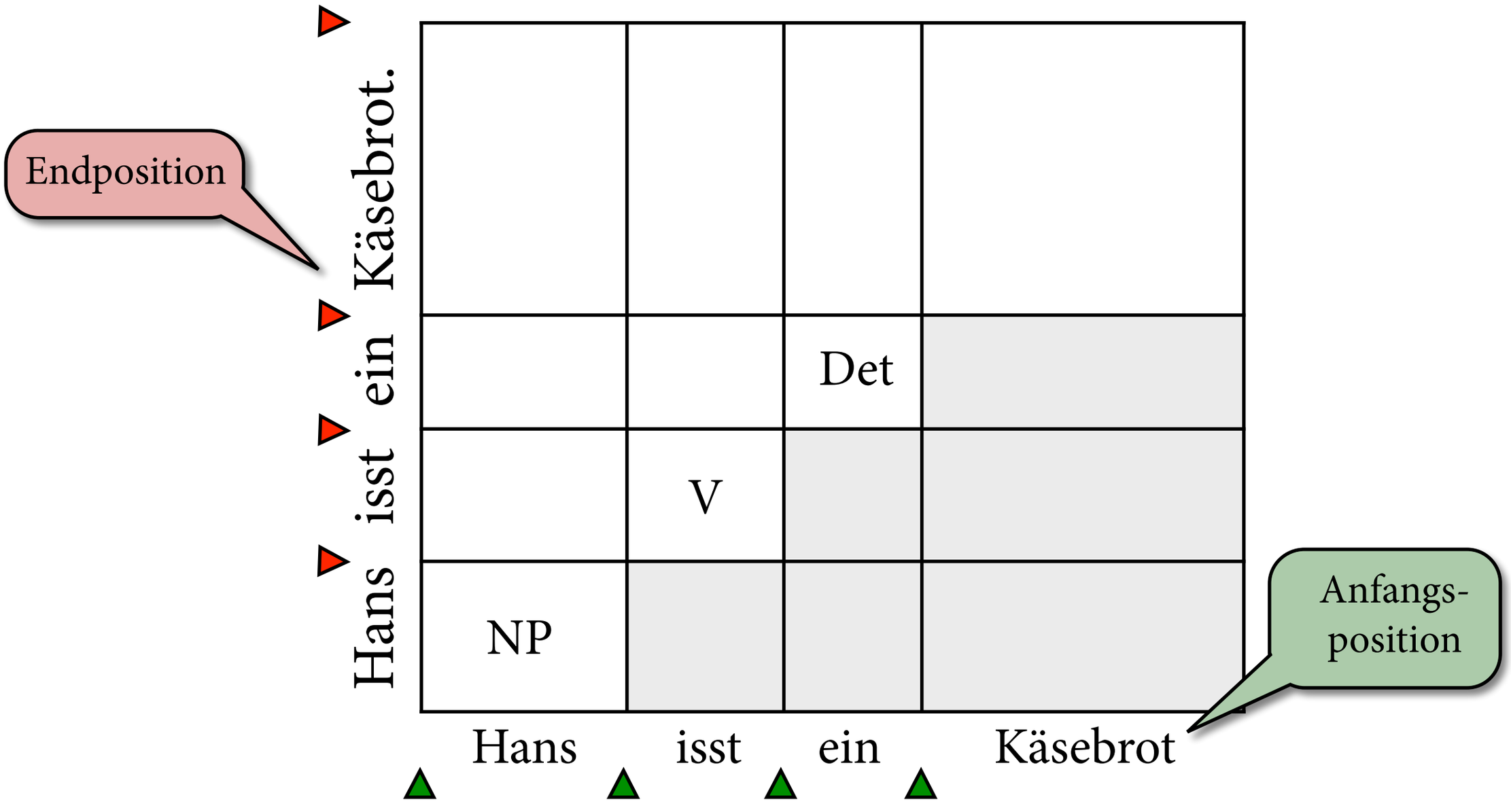
Der CKY-Parser

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



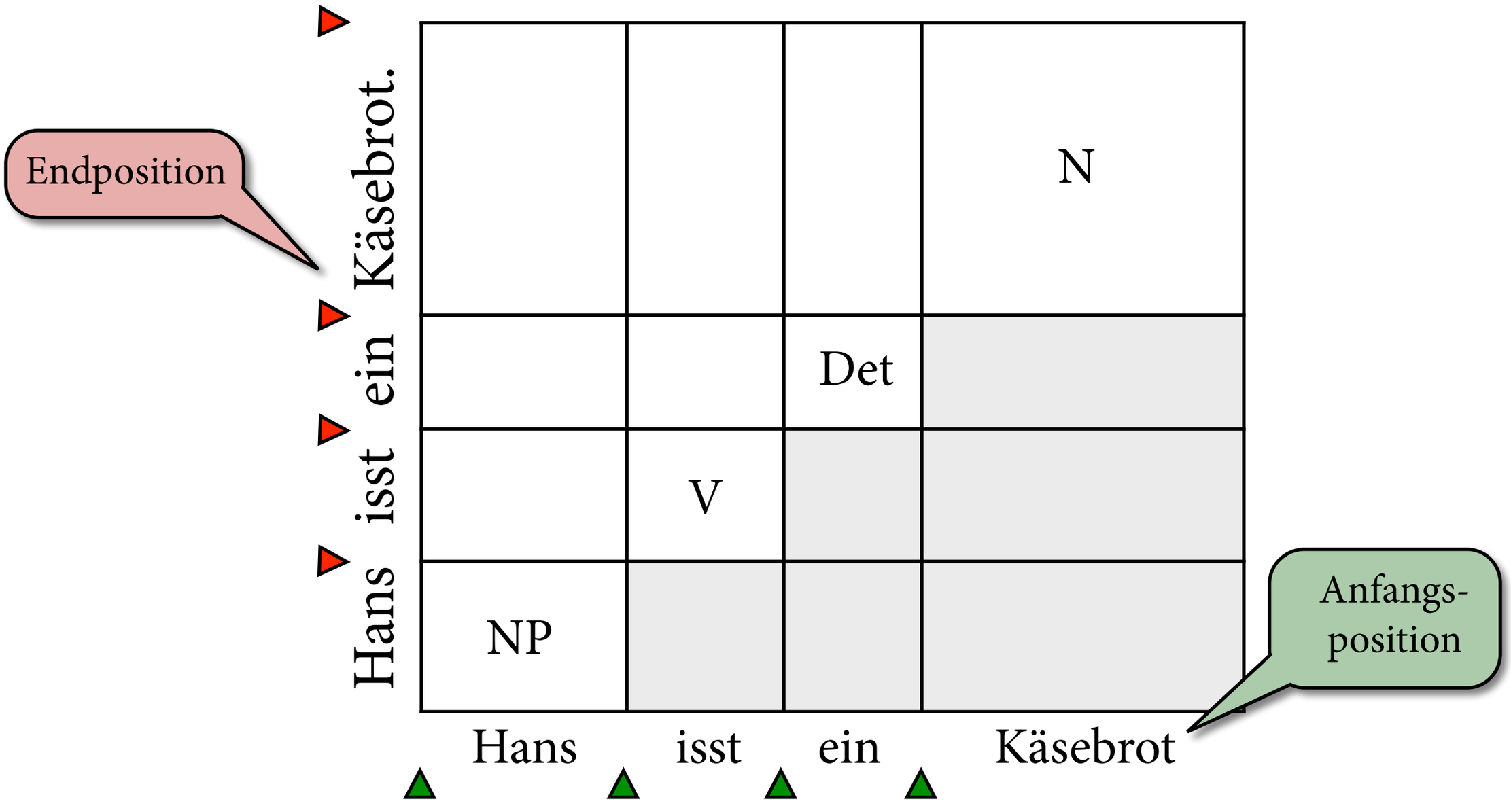
Der CKY-Parser

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



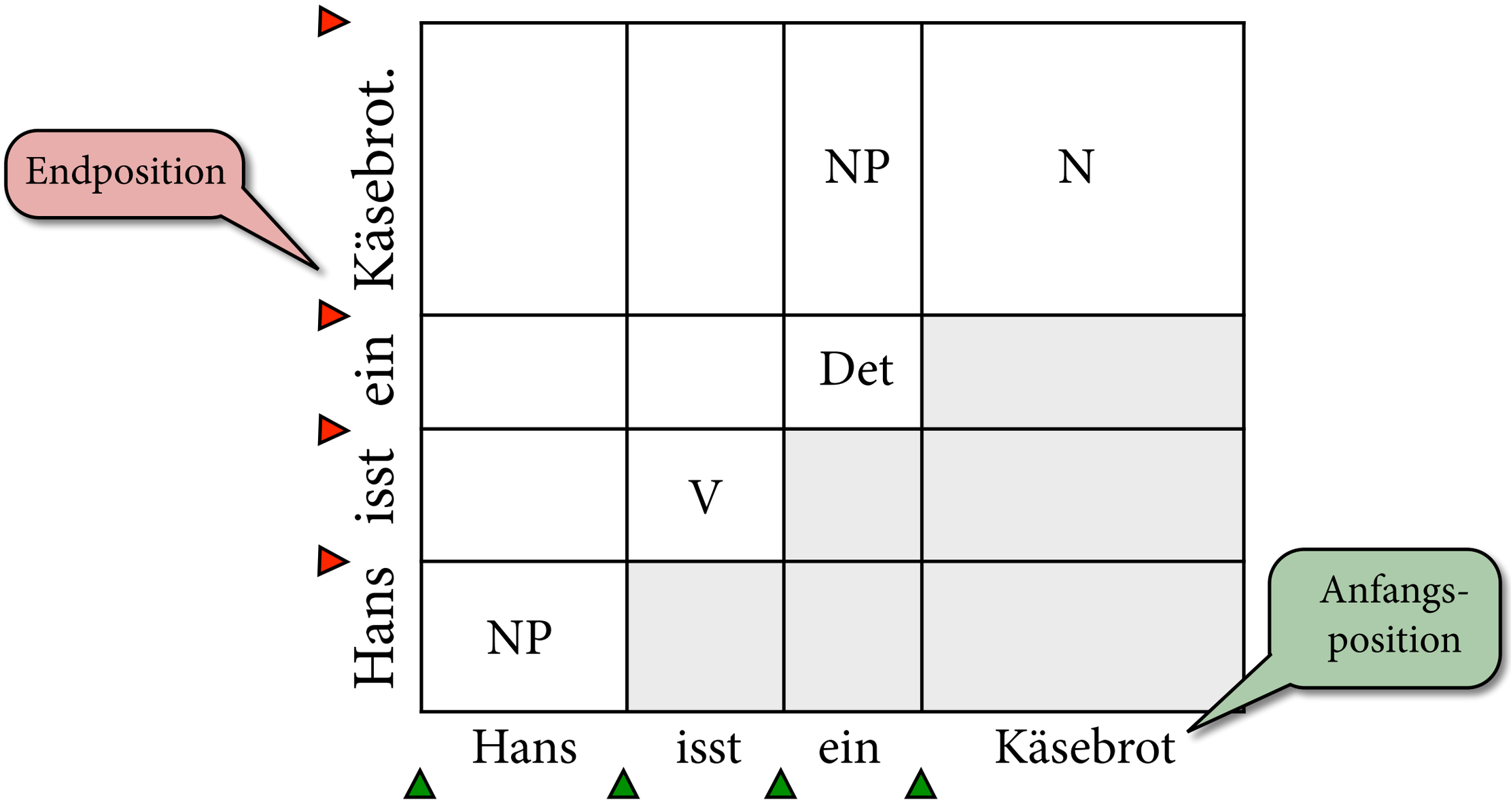
Der CKY-Parser

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



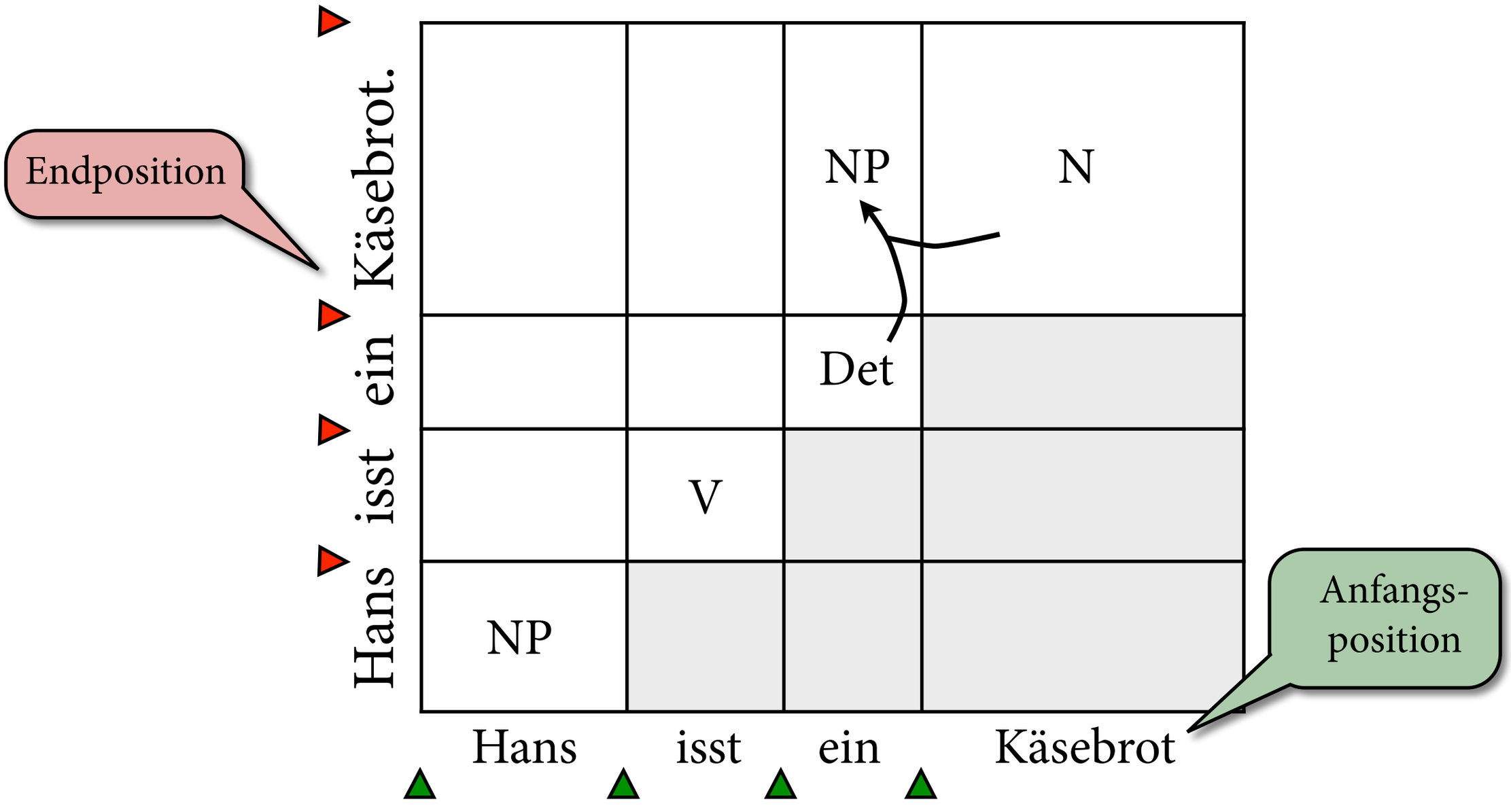
Der CKY-Parser

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



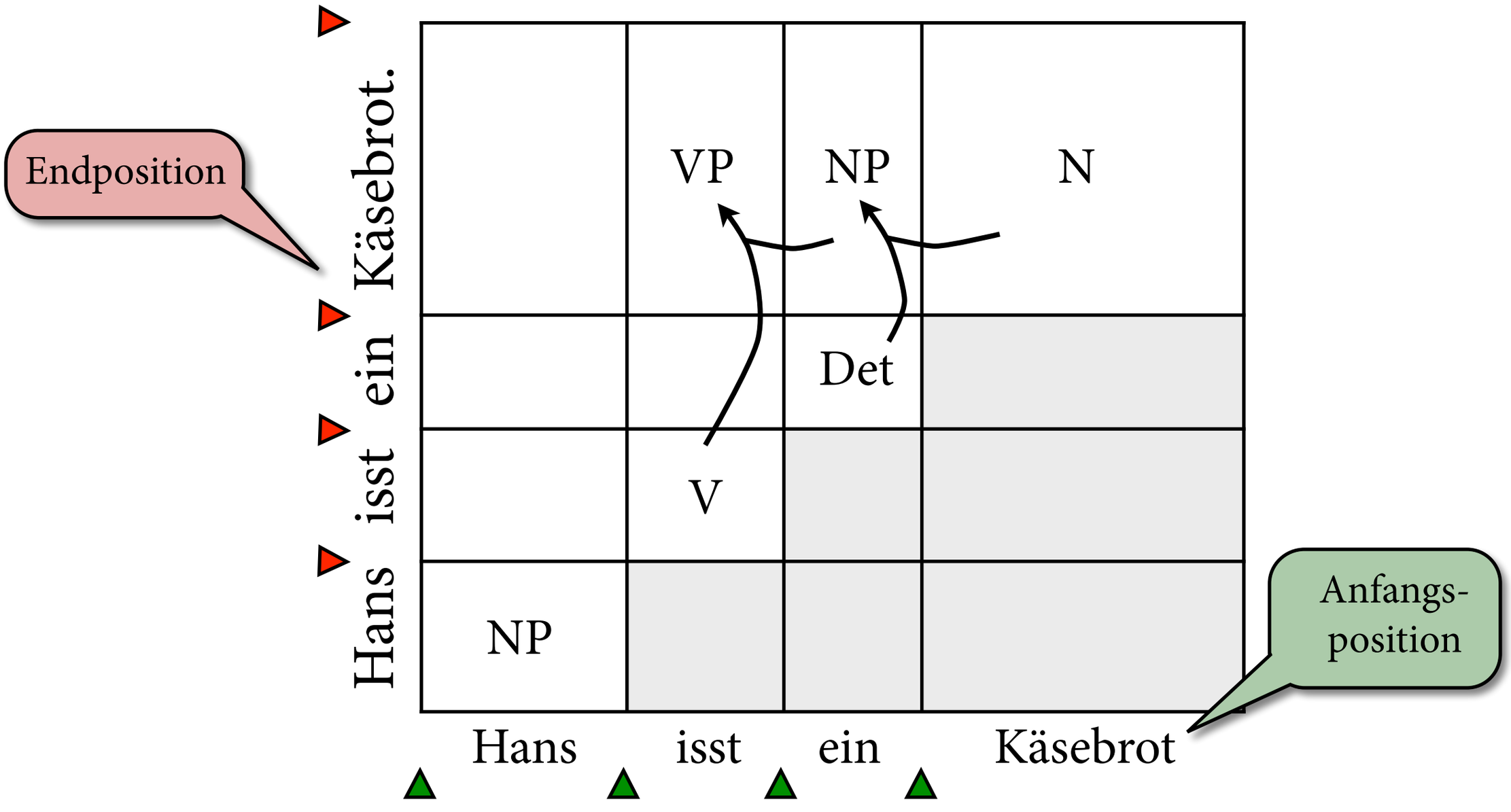
Der CKY-Parser

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



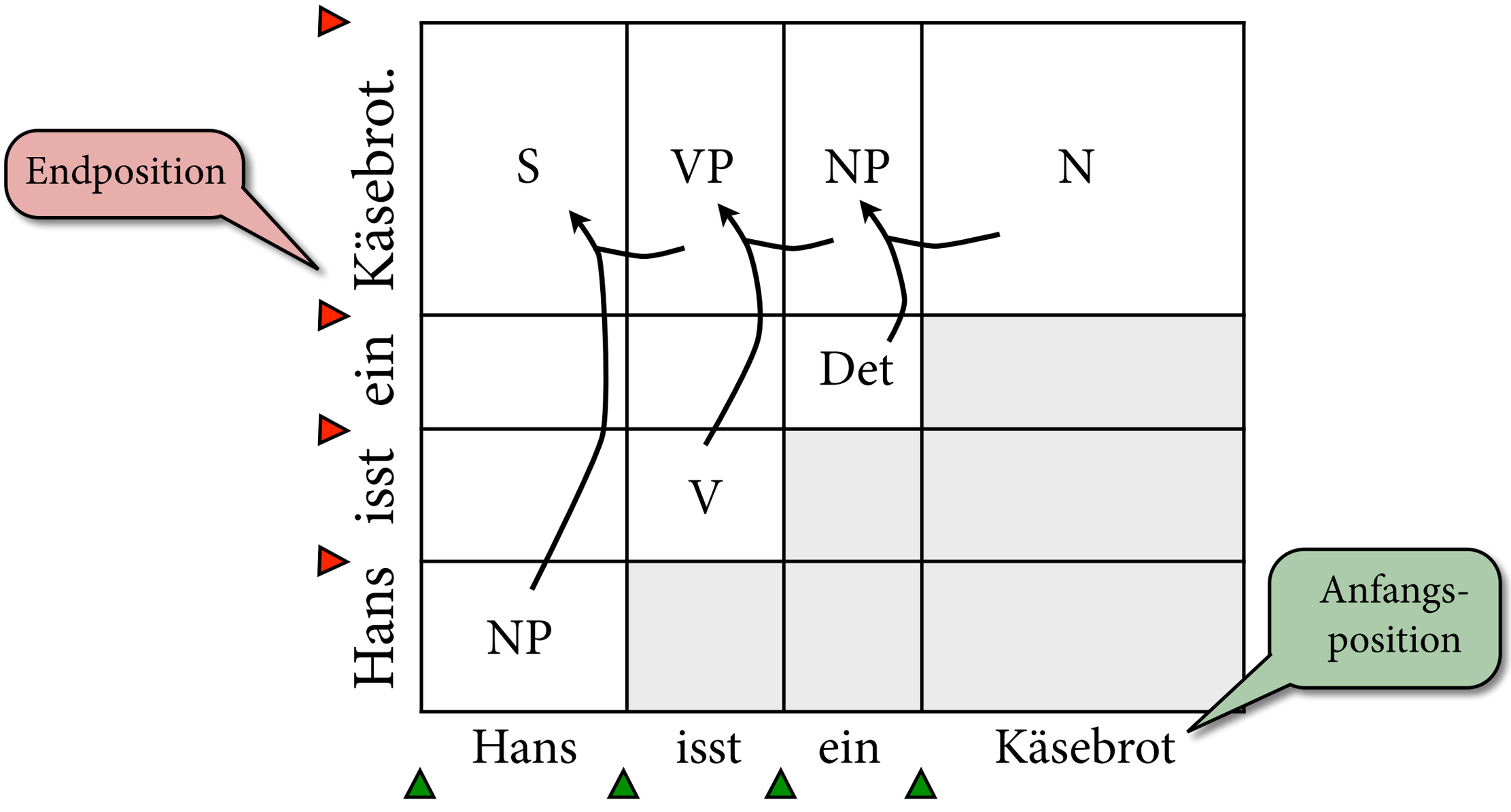
Der CKY-Parser

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



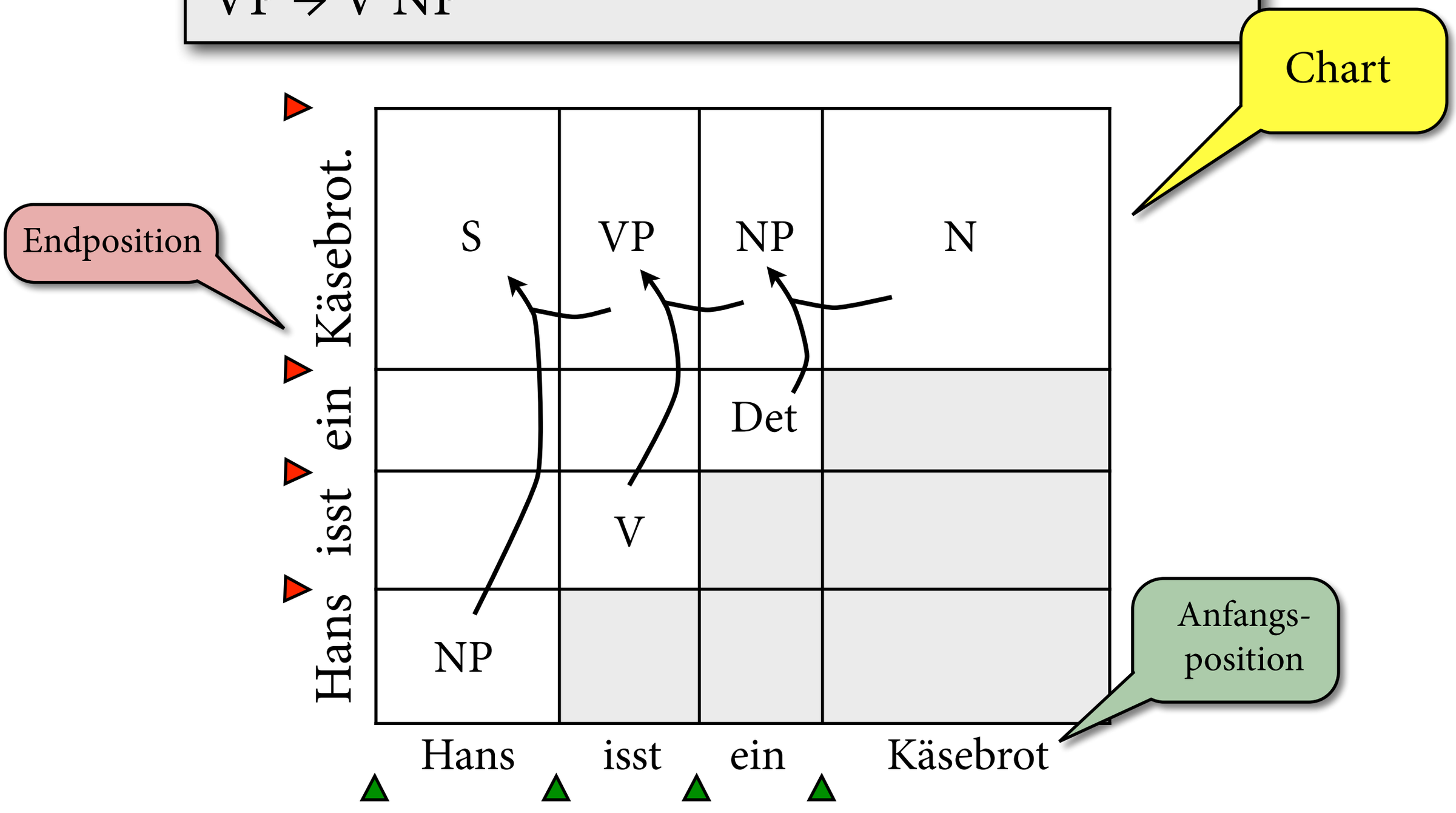
Der CKY-Parser

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



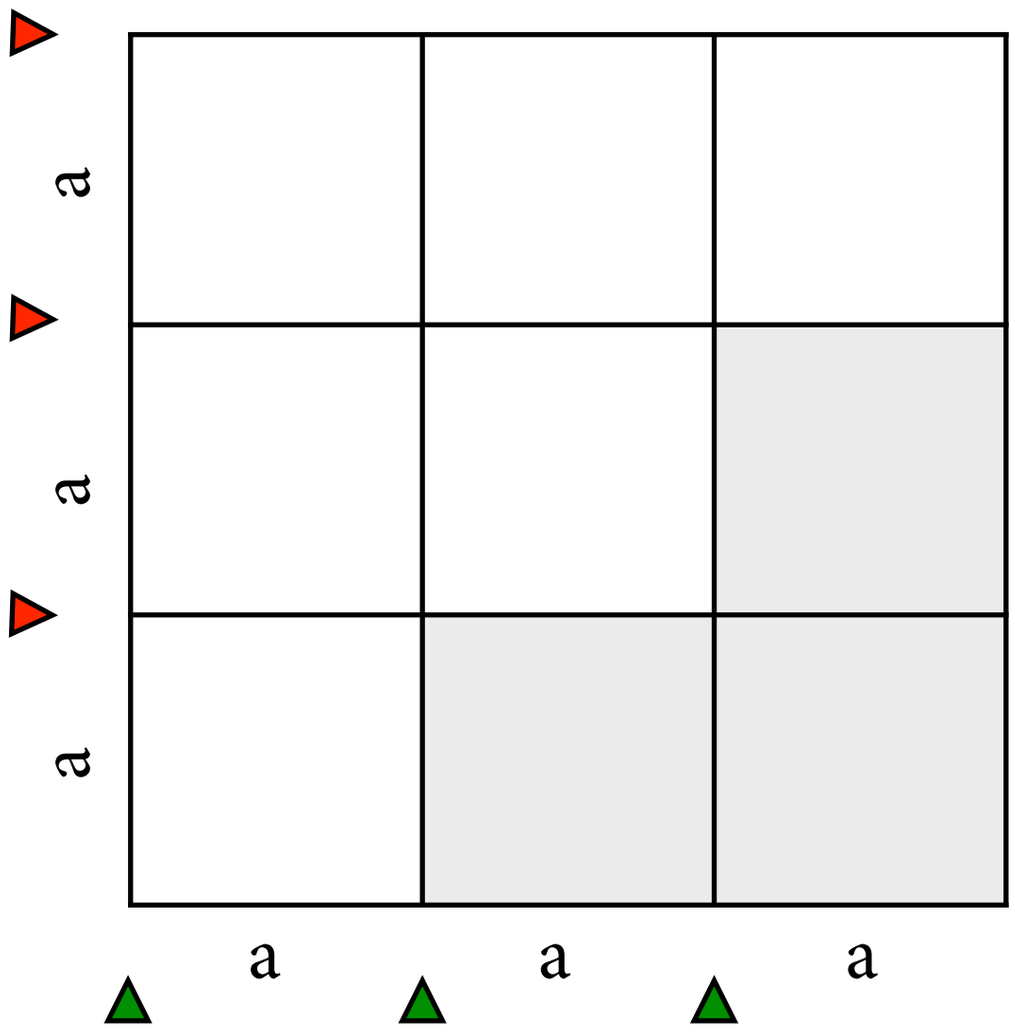
Der CKY-Parser

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



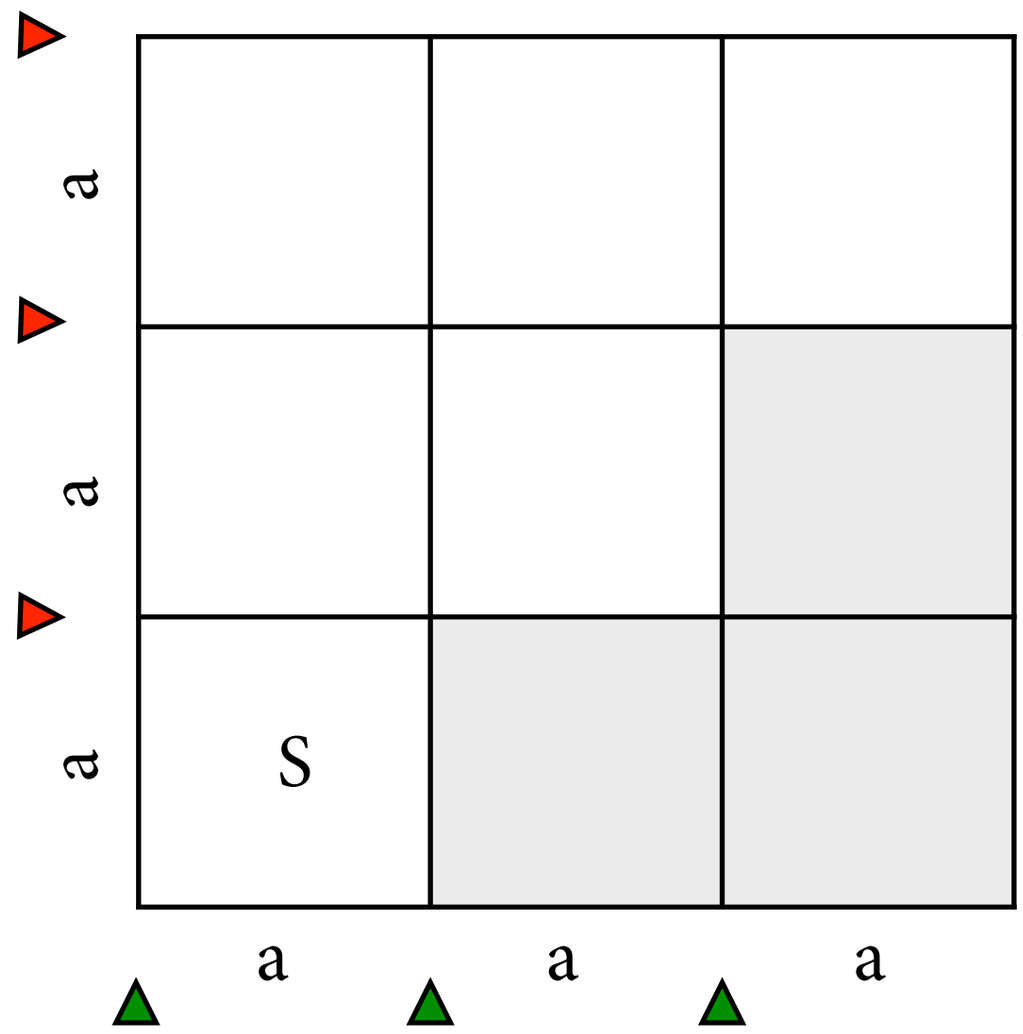
Der CKY-Parser

$S \rightarrow S\ S$ $S \rightarrow a$



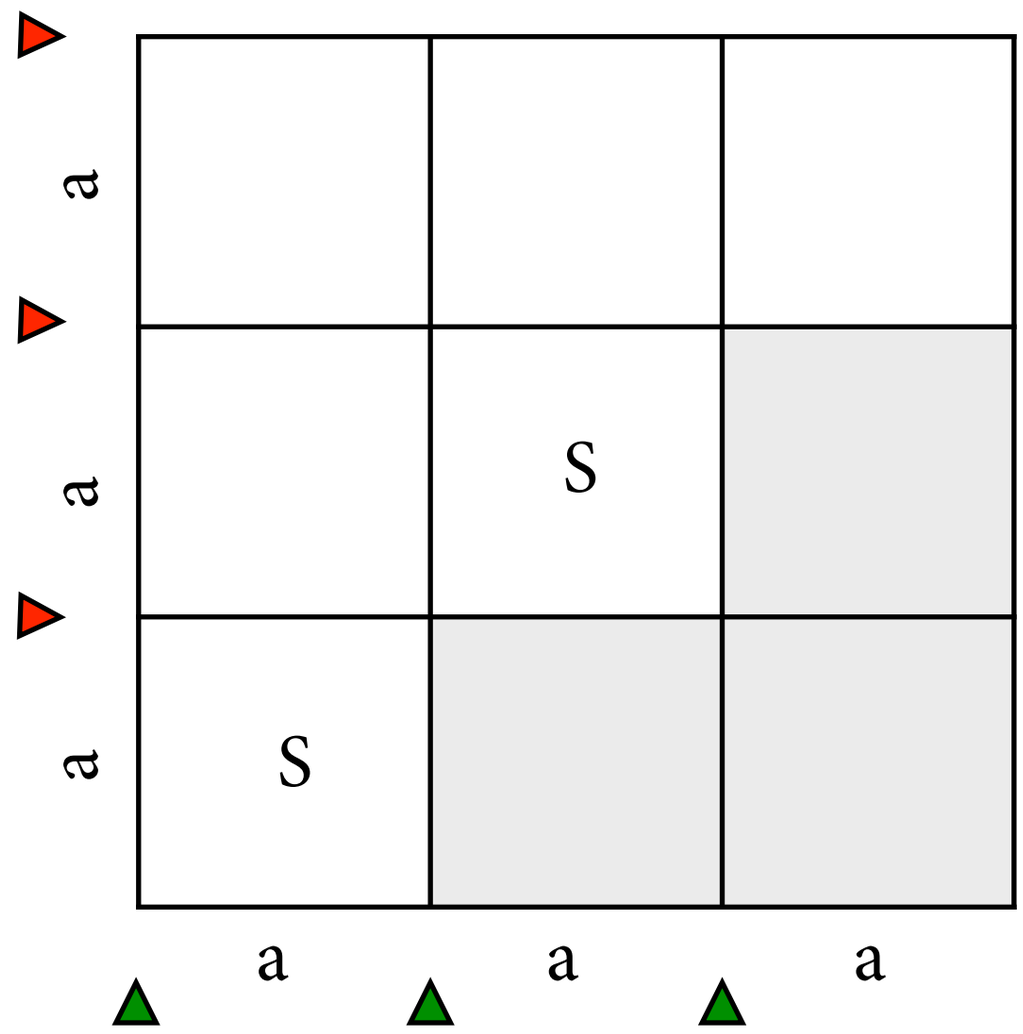
Der CKY-Parser

$S \rightarrow S\ S$ $S \rightarrow a$



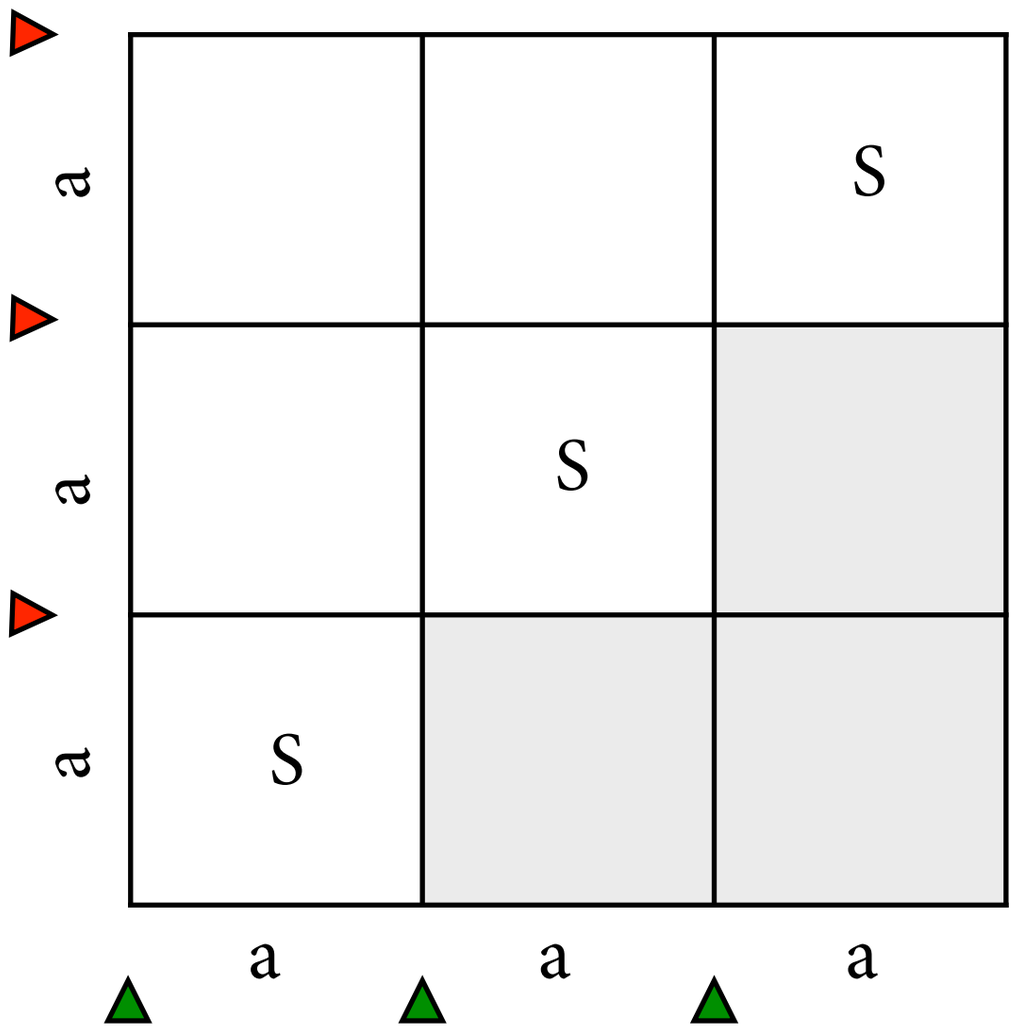
Der CKY-Parser

$S \rightarrow S\ S$ $S \rightarrow a$



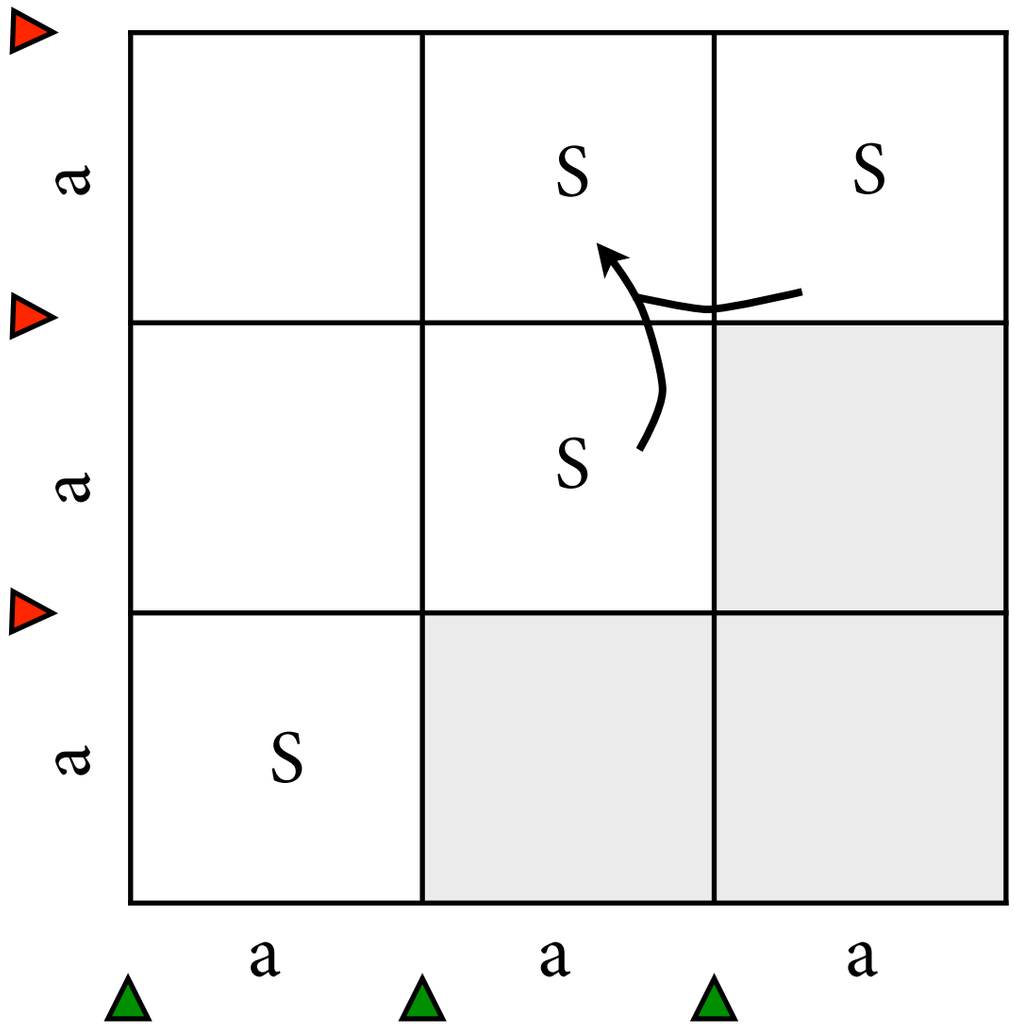
Der CKY-Parser

$S \rightarrow S\ S$ $S \rightarrow a$



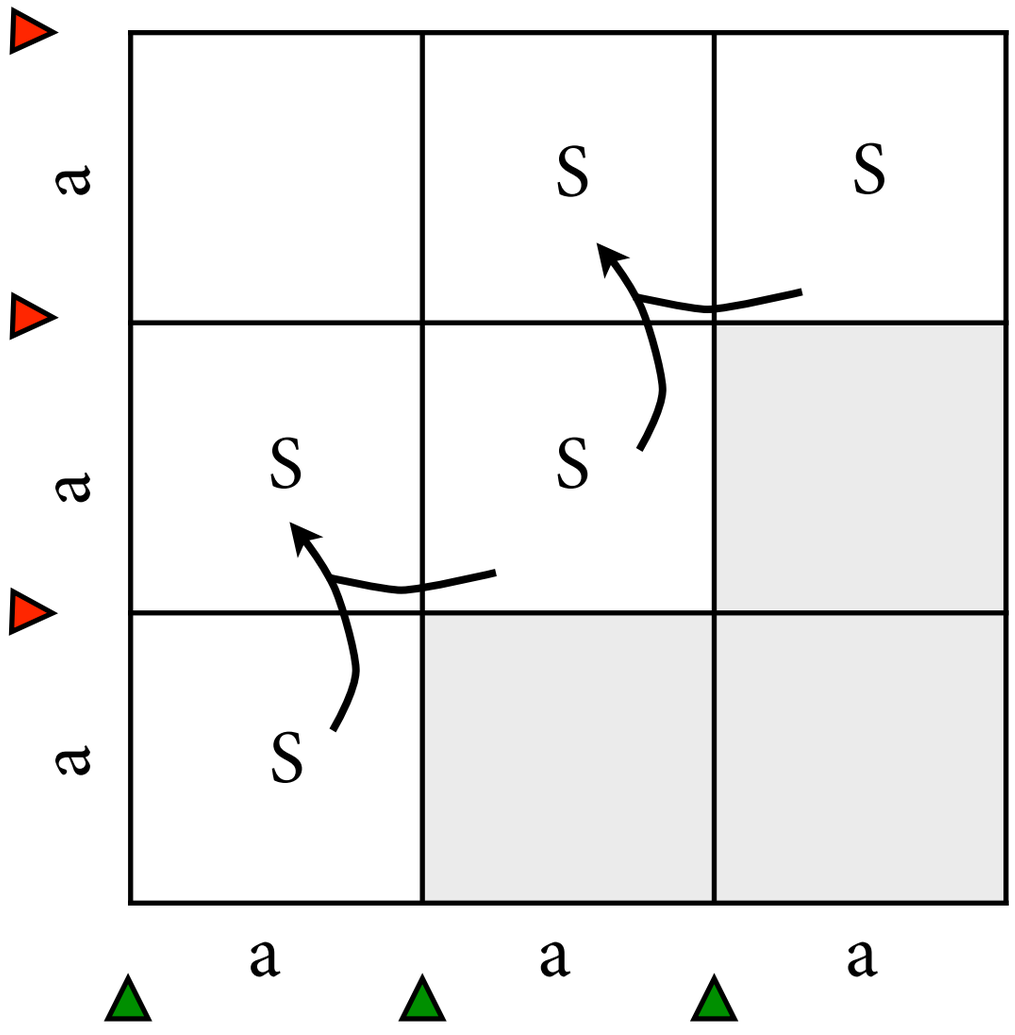
Der CKY-Parser

$S \rightarrow S\ S$ $S \rightarrow a$



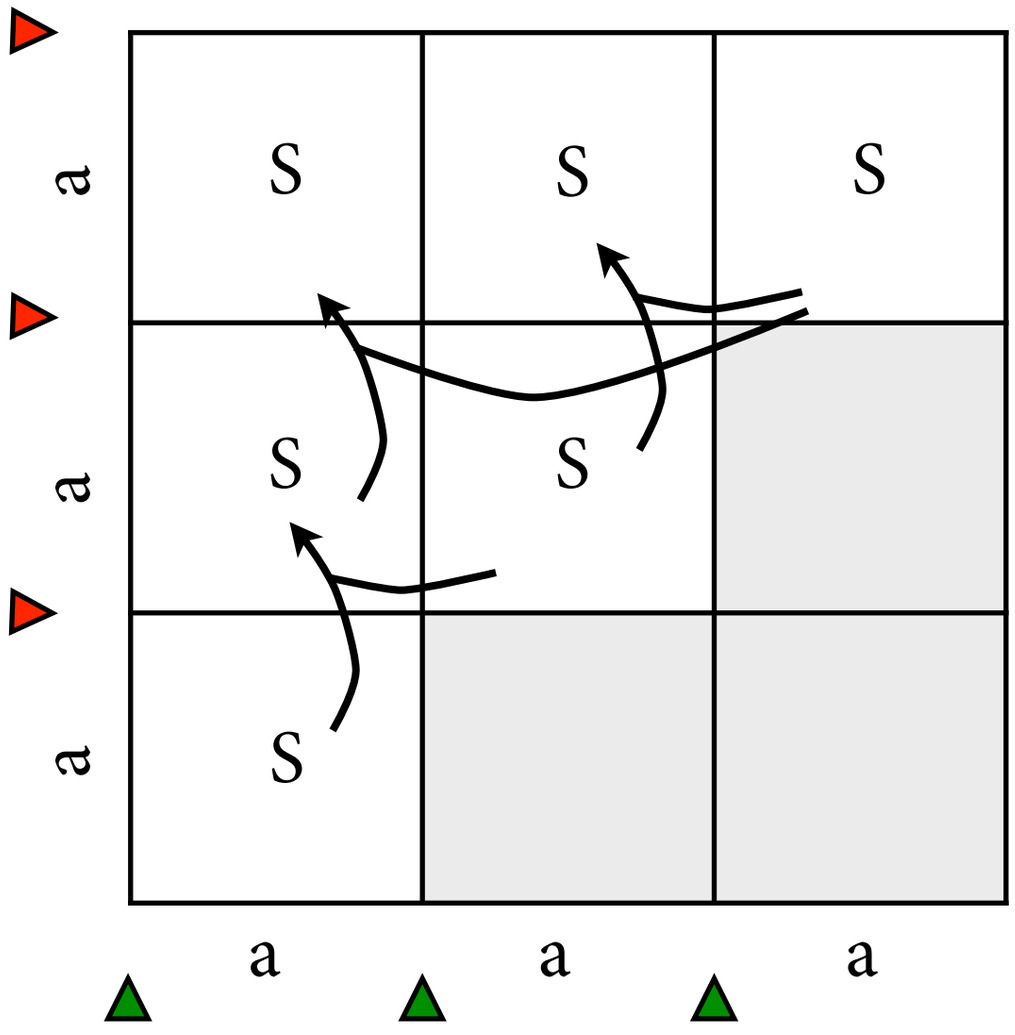
Der CKY-Parser

$S \rightarrow S\ S$ $S \rightarrow a$



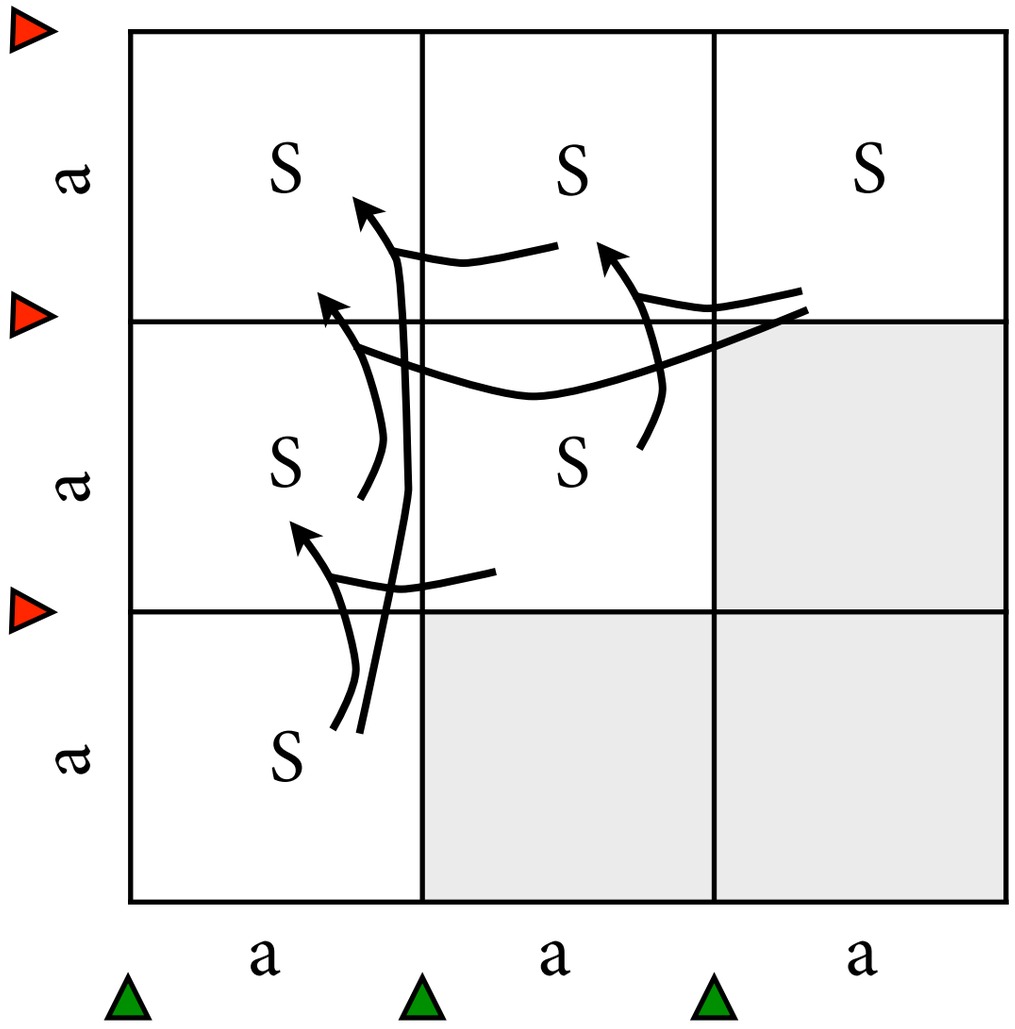
Der CKY-Parser

$S \rightarrow S\ S$ $S \rightarrow a$



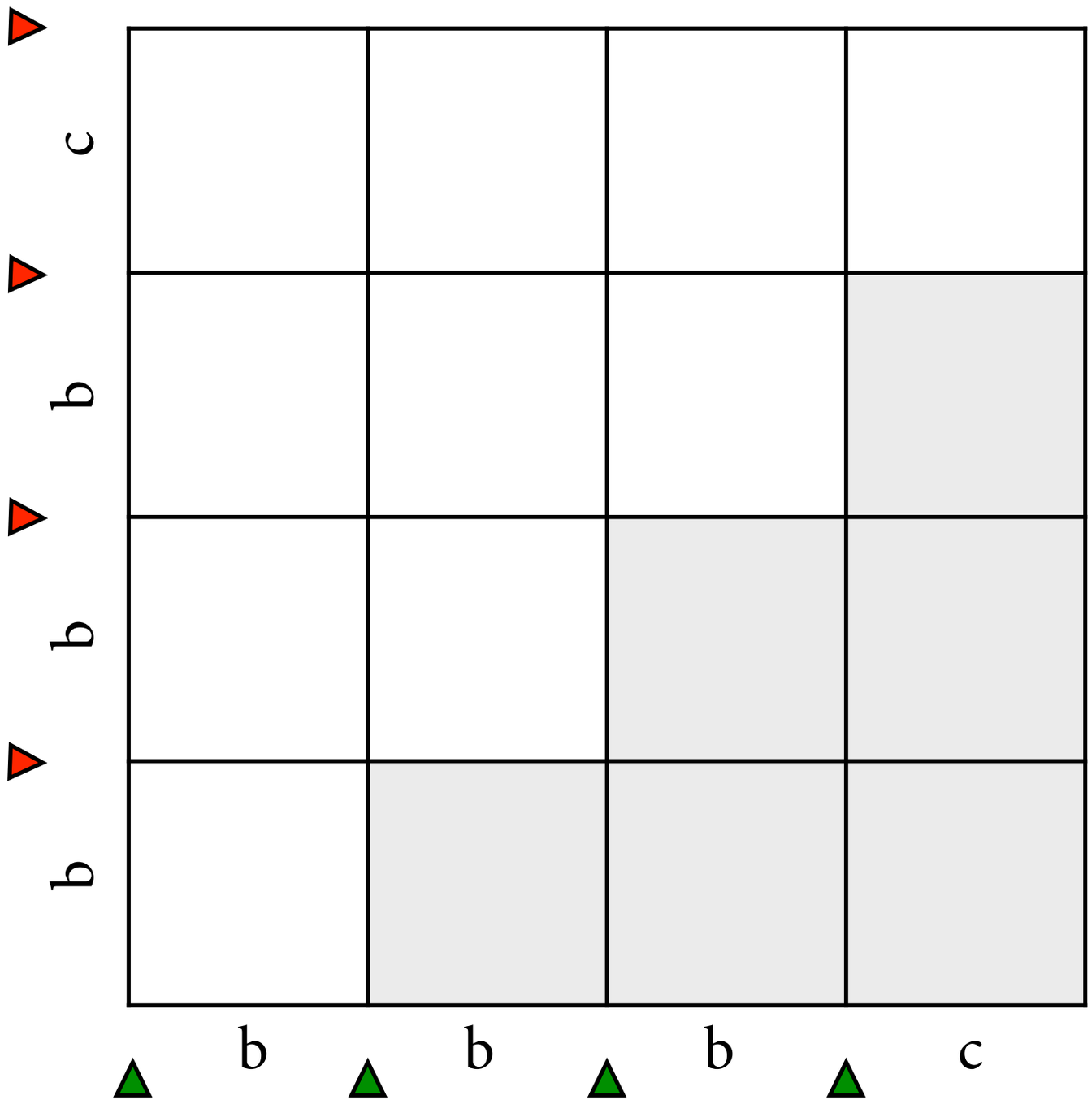
Der CKY-Parser

$S \rightarrow S\ S$ $S \rightarrow a$



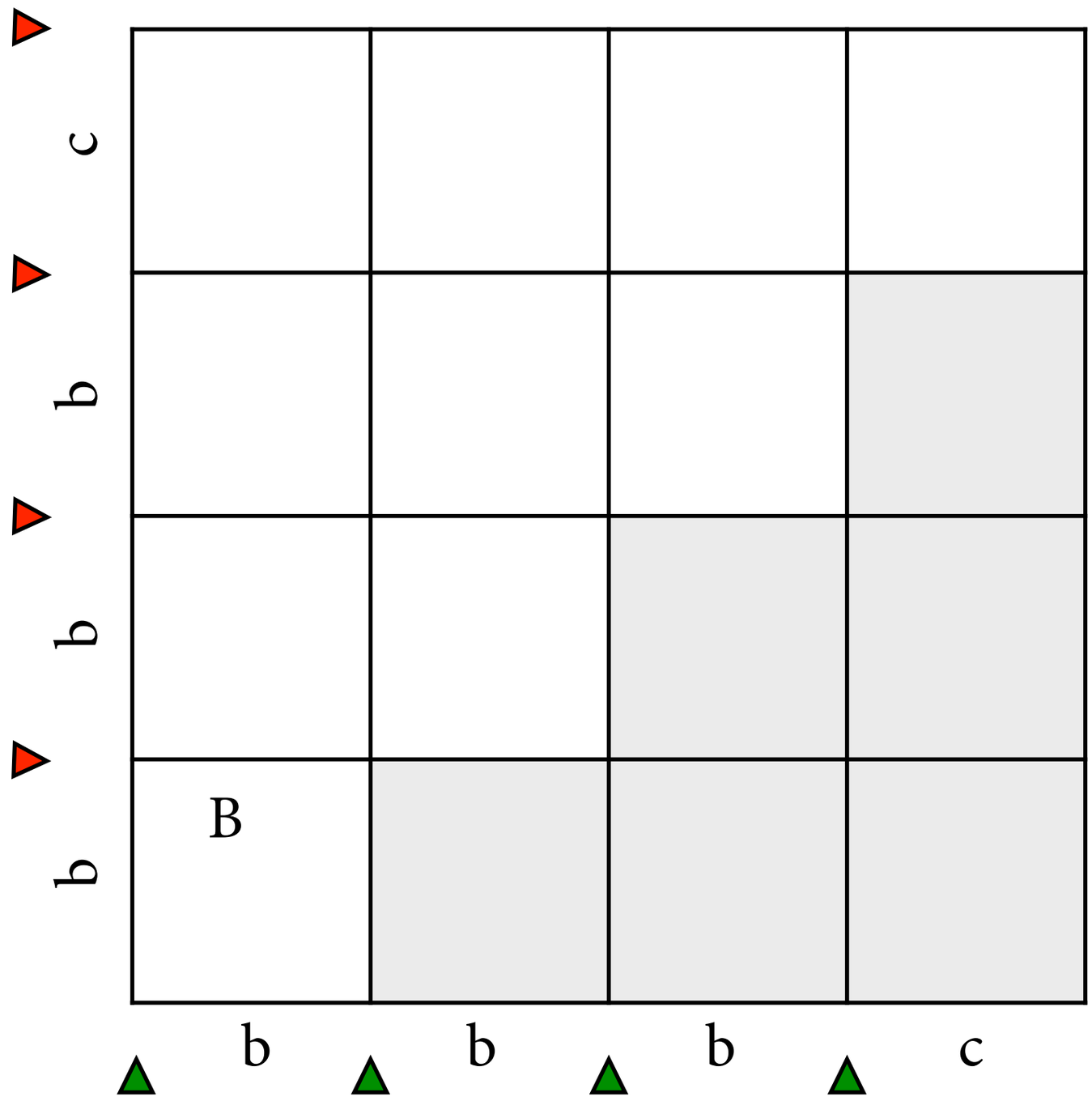
Der CKY-Parser

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$



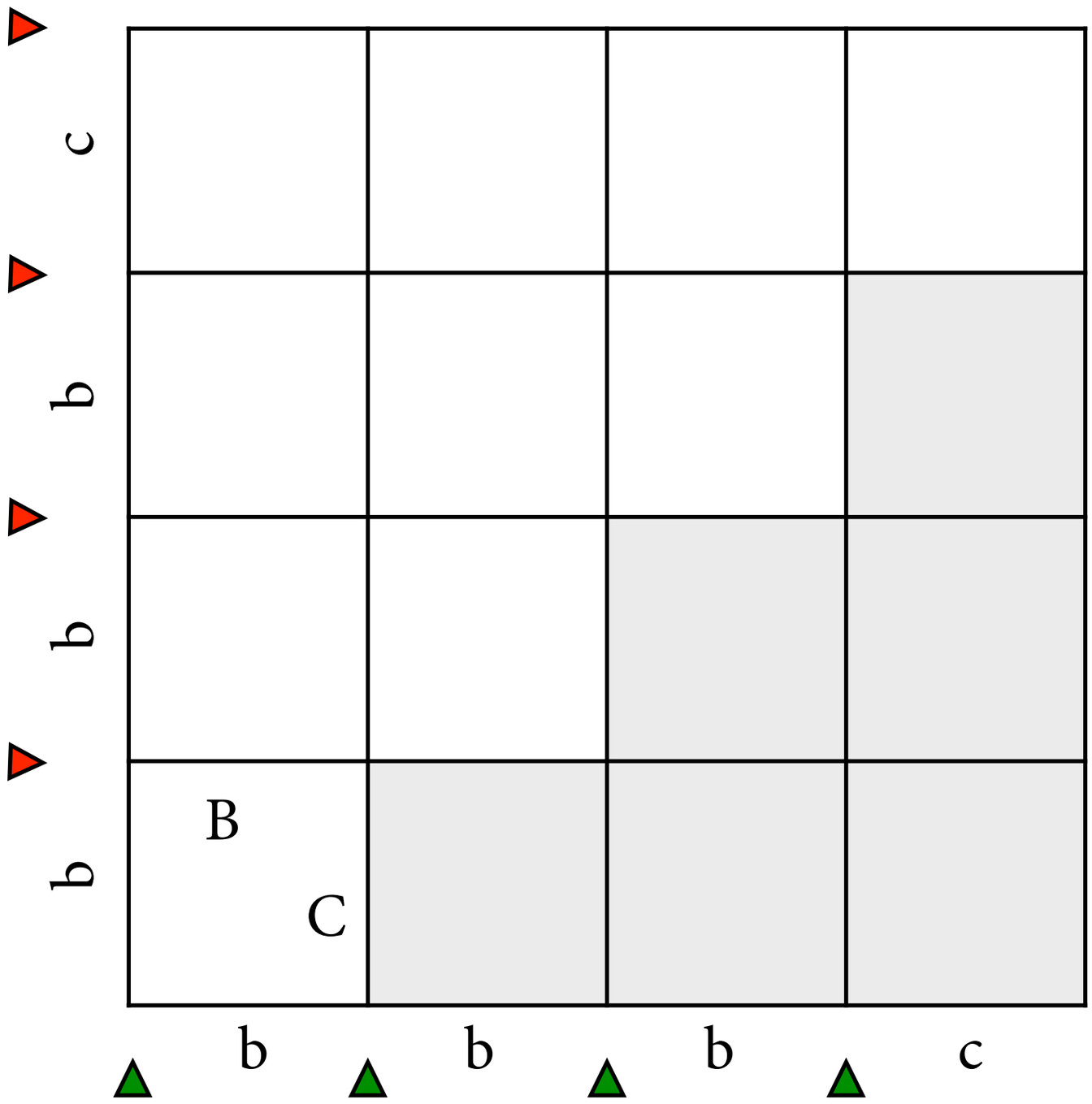
Der CKY-Parser

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$



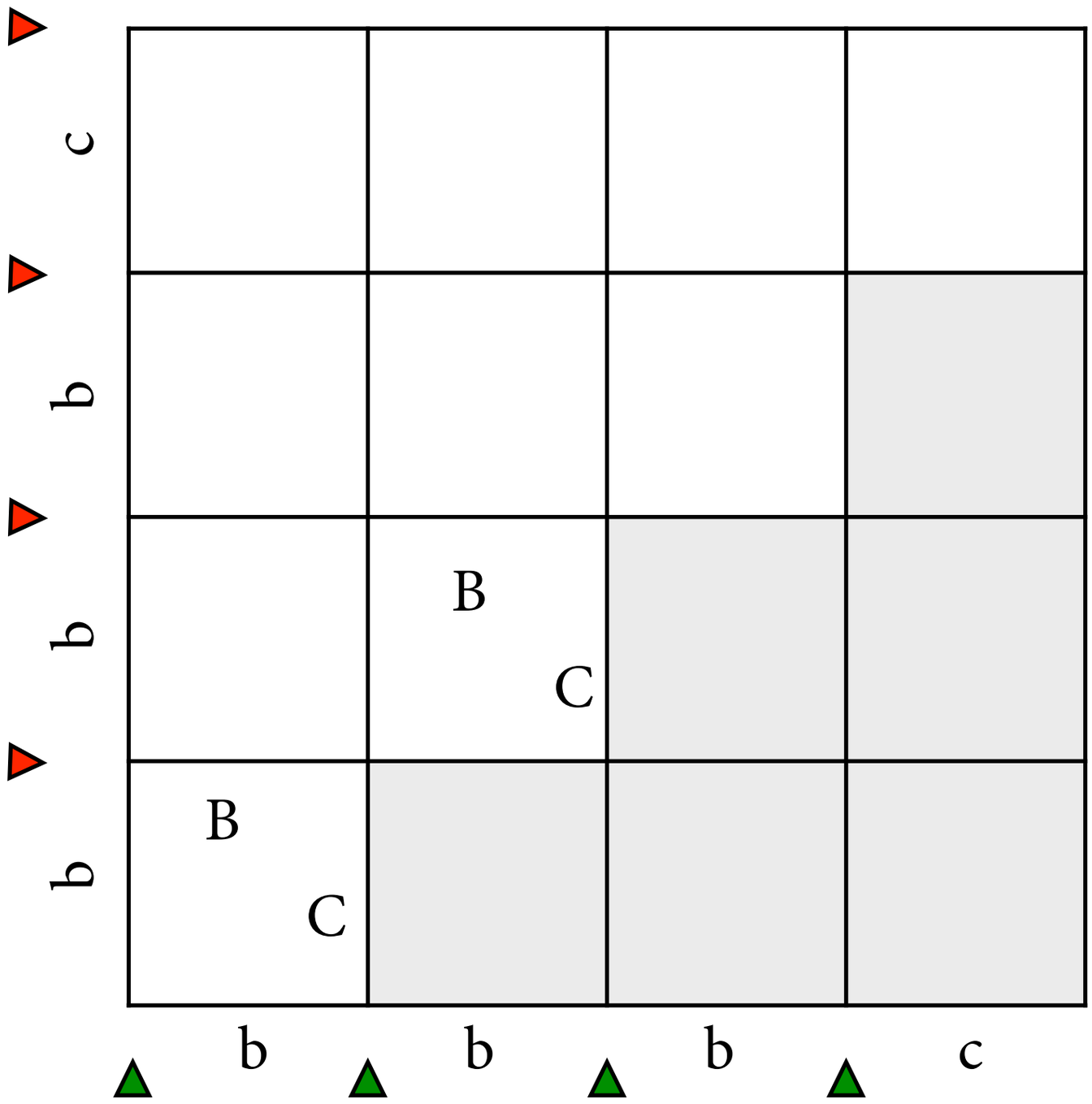
Der CKY-Parser

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$



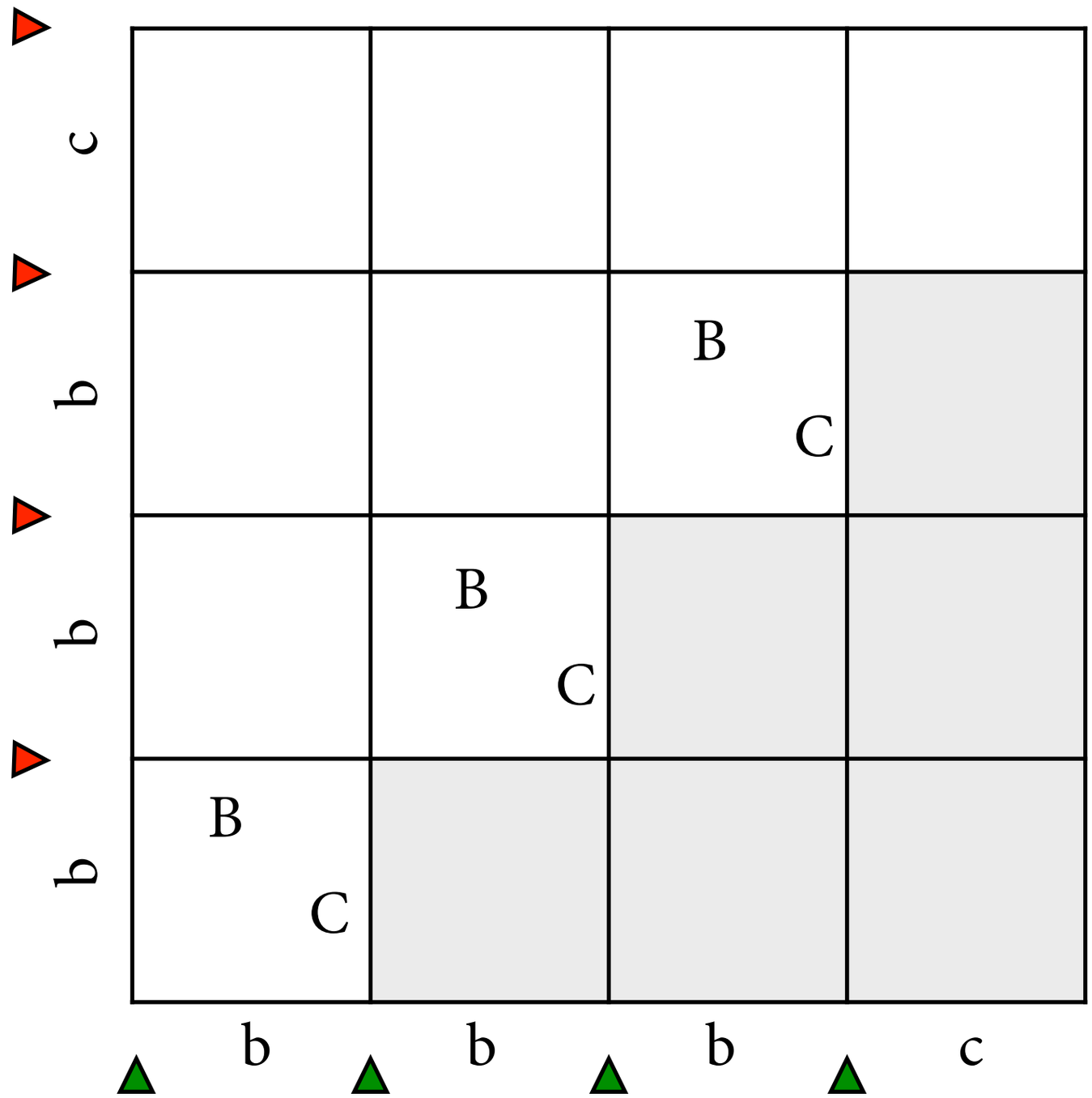
Der CKY-Parser

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$



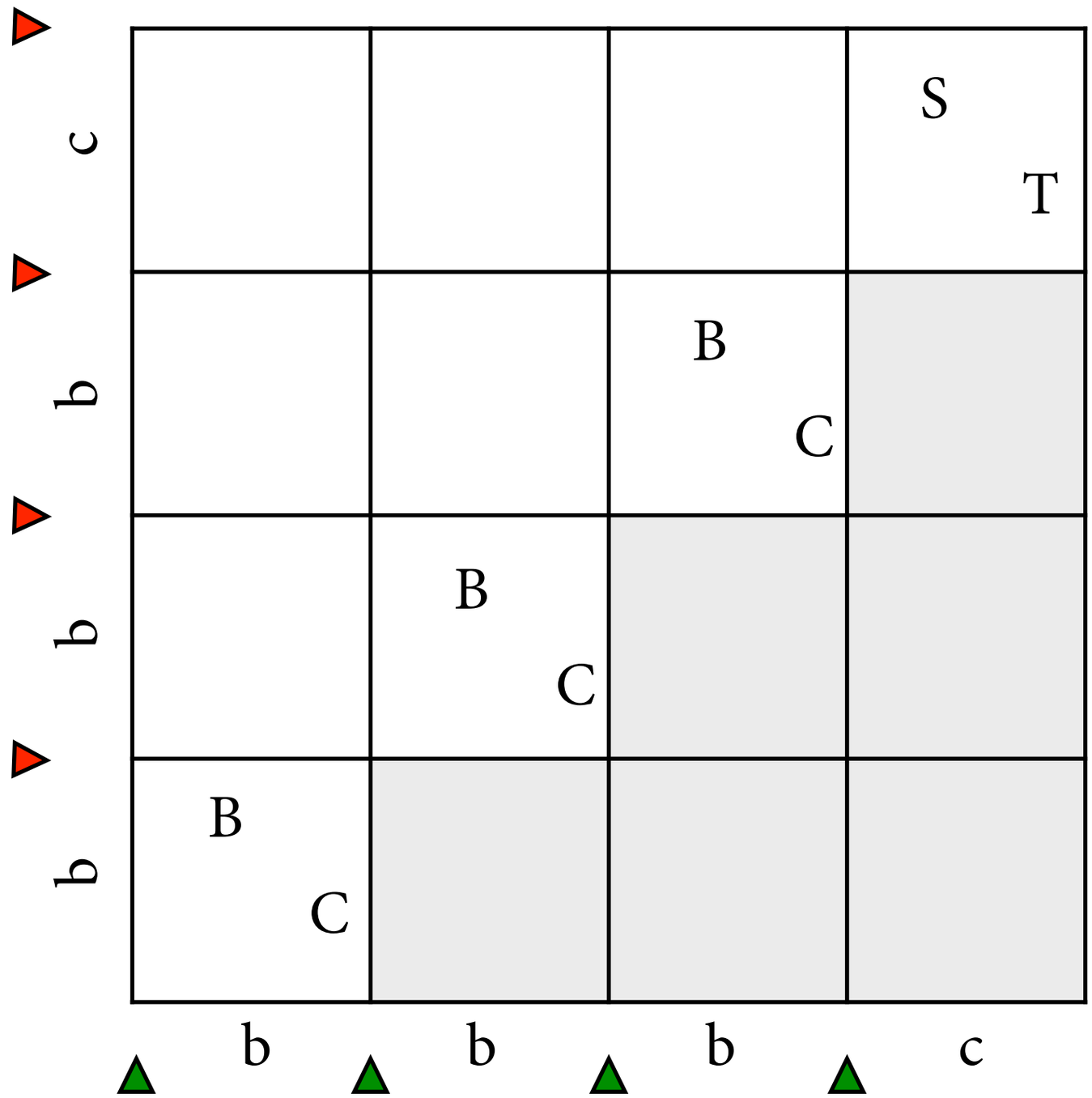
Der CKY-Parser

S → B S	B → b	S → c
T → C T	C → b	T → c



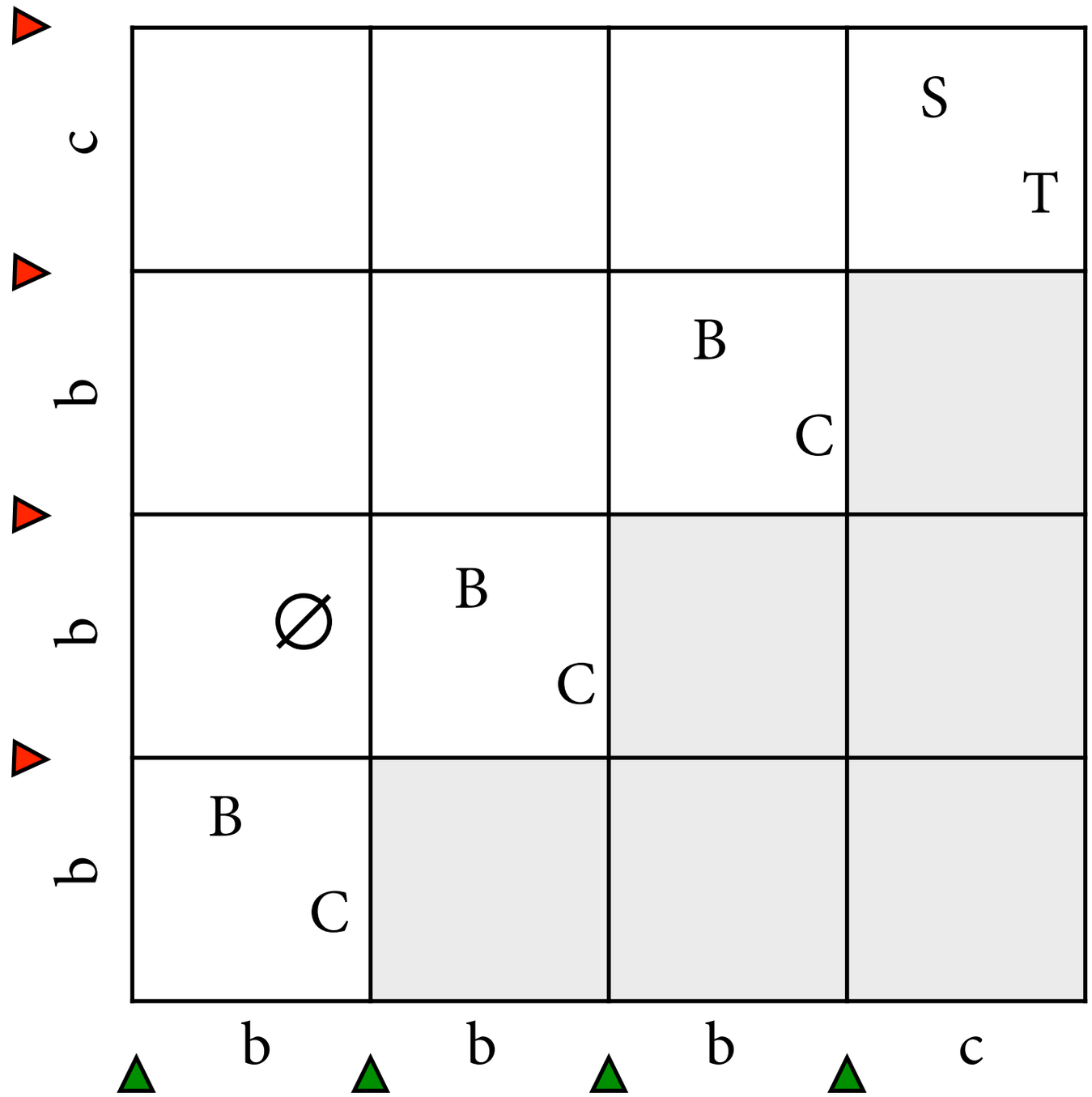
Der CKY-Parser

S → B S	B → b	S → c
T → C T	C → b	T → c



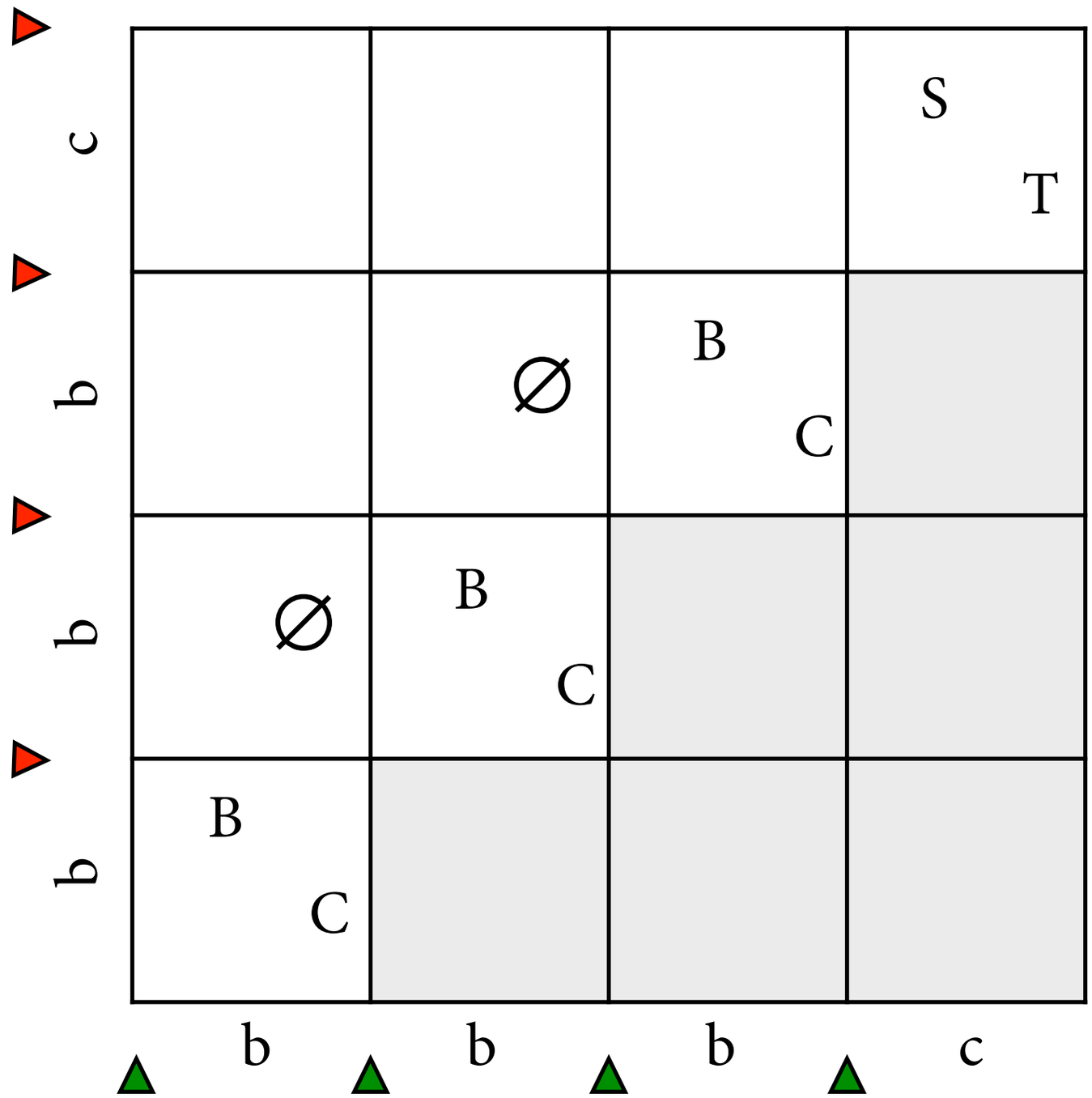
Der CKY-Parser

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$



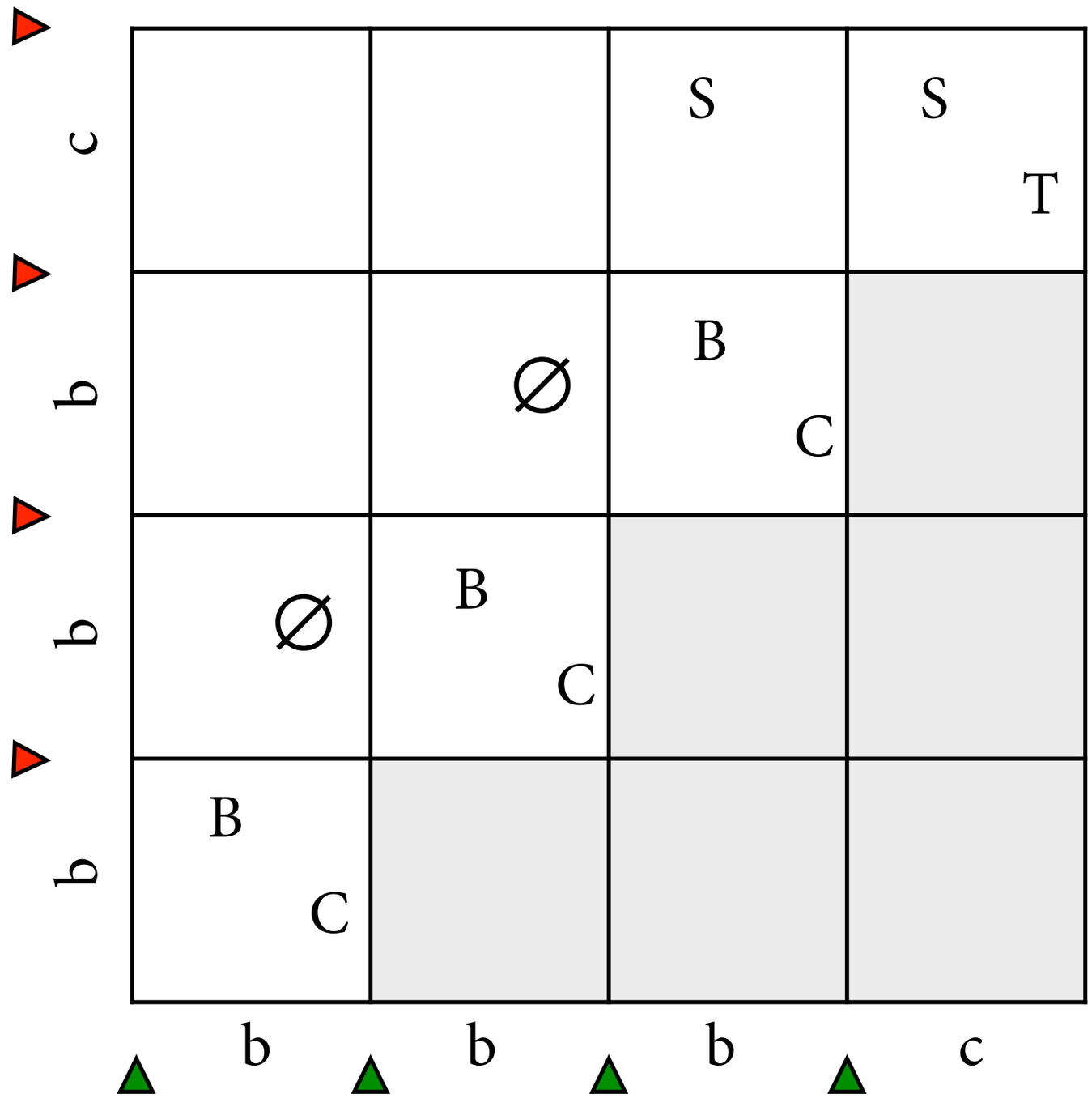
Der CKY-Parser

S → B S	B → b	S → c
T → C T	C → b	T → c



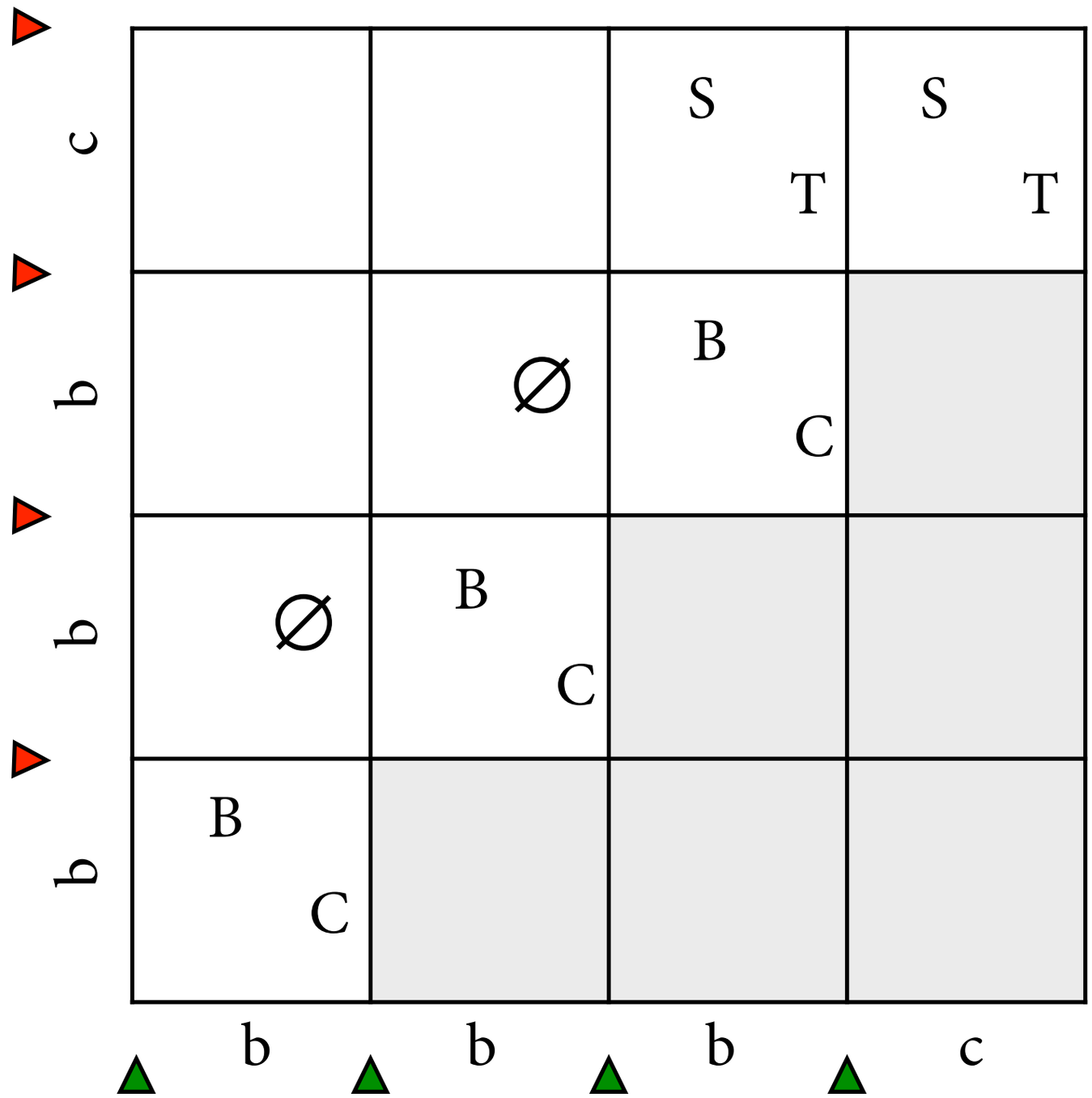
Der CKY-Parser

S → B S	B → b	S → c
T → C T	C → b	T → c



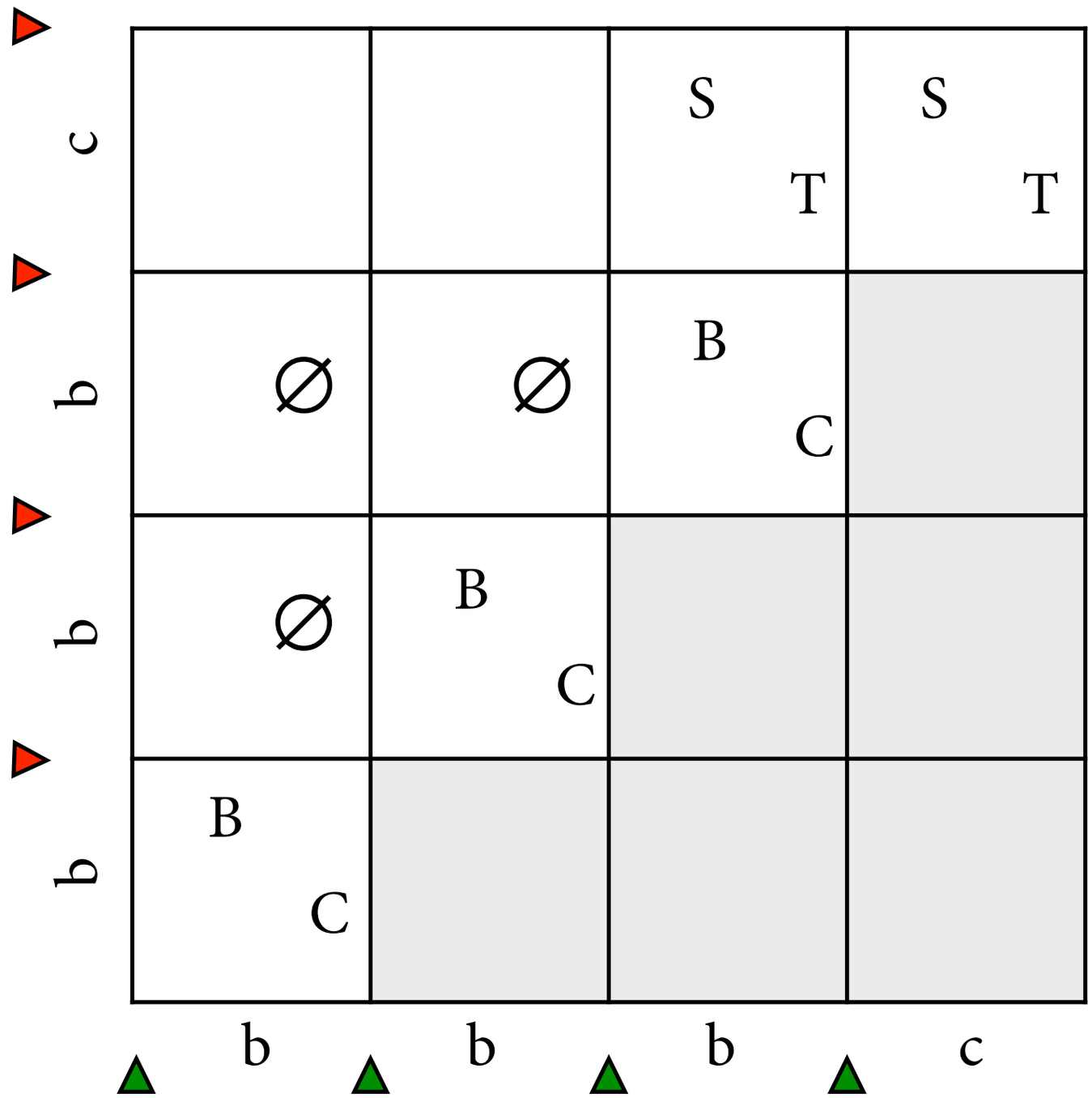
Der CKY-Parser

S → B S	B → b	S → c
T → C T	C → b	T → c



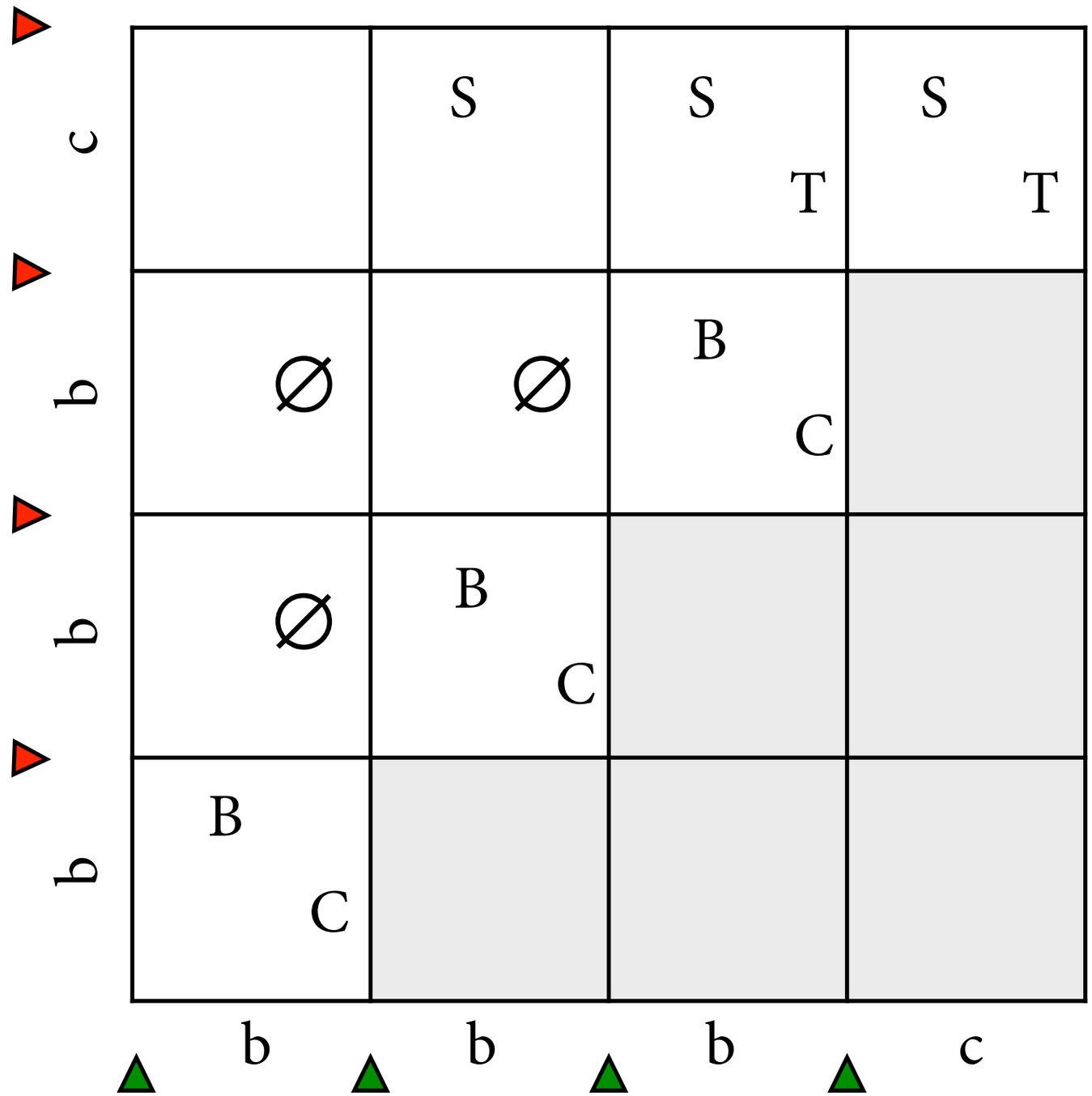
Der CKY-Parser

S → B S	B → b	S → c
T → C T	C → b	T → c



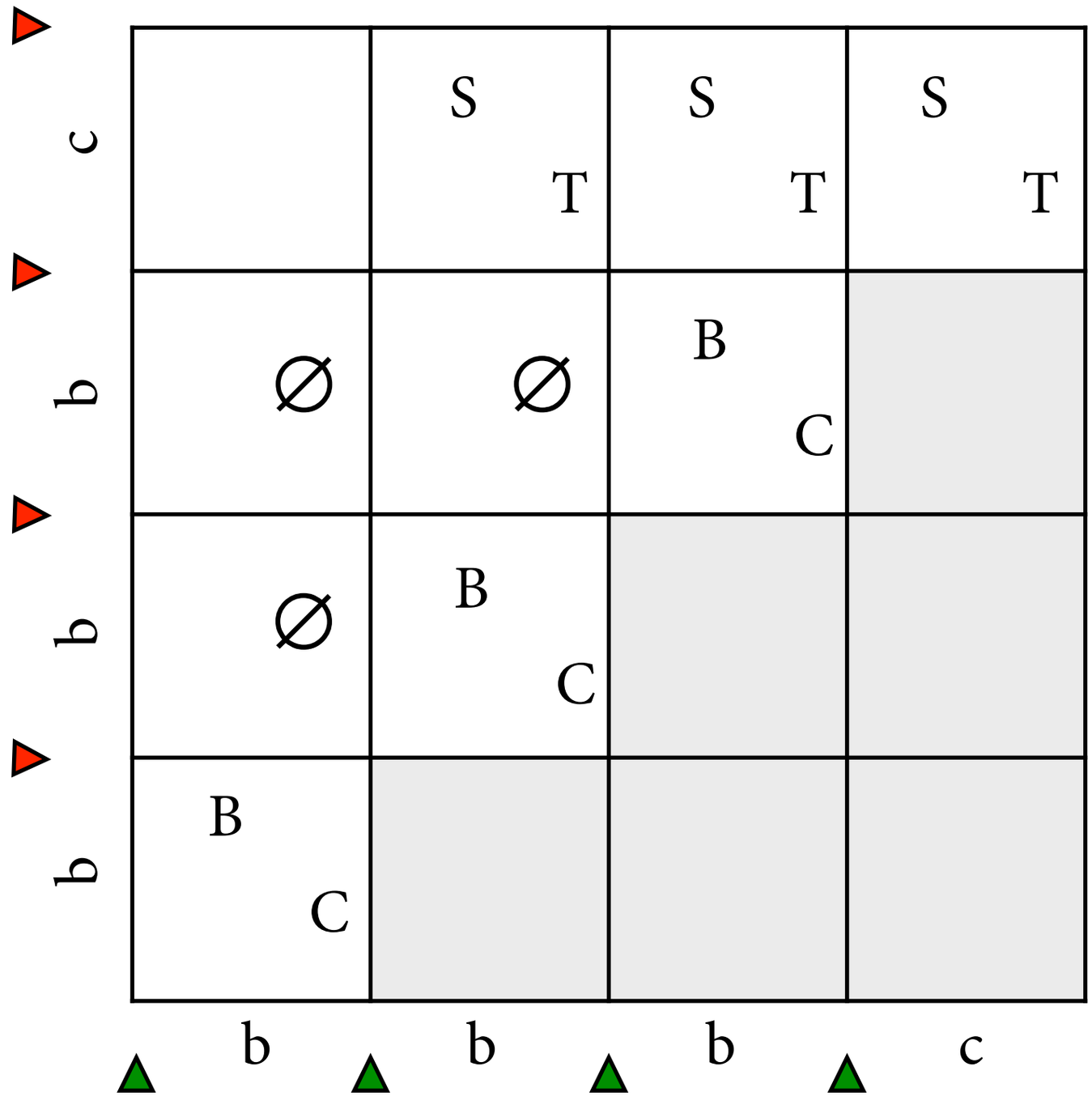
Der CKY-Parser

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$



Der CKY-Parser

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$



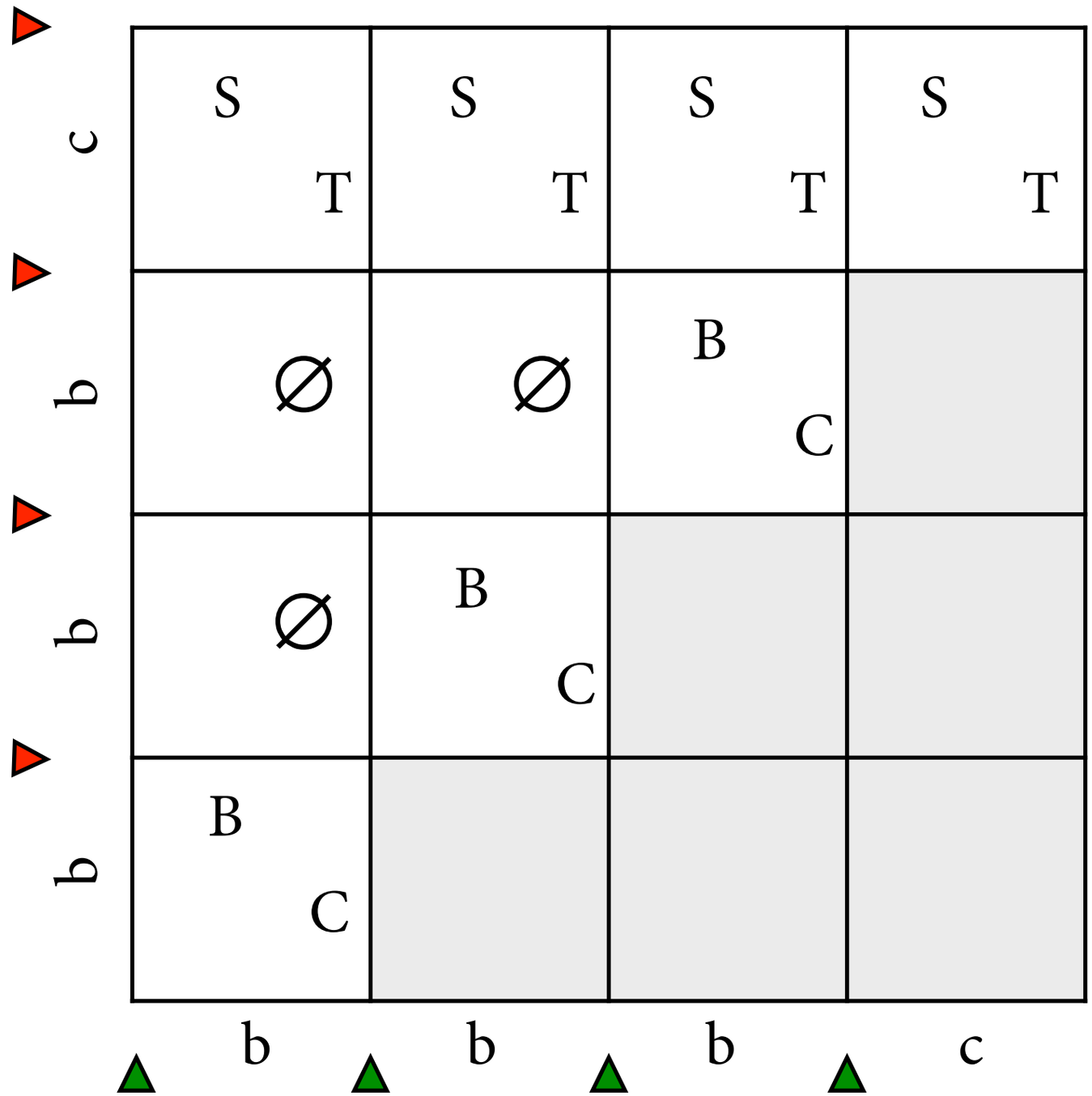
Der CKY-Parser

S → B S	B → b	S → c
T → C T	C → b	T → c

c	S	S T	S T	S T
b	∅	∅	B C	
b	∅	B C		
b	B C			
	b	b	b	c

Der CKY-Parser

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$



CKY-Parser: Algorithmus

Datenstruktur: Ch(i,k) enthält Menge aller Nichtterminale A mit $A \Rightarrow^* w_i \ldots w_{k-1}$ (anfangs überall leer).

für alle i von 1 bis n:
    für alle Produktionen $A \rightarrow w_i$:
        füge A zu Ch(i, i+1) hinzu

für alle b von 2 bis n:
    für alle i von 1 bis n-b+1:
        für alle k von 1 bis b-1:
            für alle $B \in Ch(i, i+k)$ und $C \in Ch(i+k,i+b)$:
                für alle Produktionen $A \rightarrow B\ C$:
                    füge A zu Ch(i,i+b) hinzu

CKY-Parser

- Erkenner: Wort in Sprache gdw am Schluss das Startsymbol in Ch(1,n+1) steht.

- Parser: Muss sich für jeden Eintrag in der Tabelle merken, wie man ihn aus kleineren Einträgen bauen kann.

Laufzeit

- Jede Menge Ch(i,k) kann höchstens so viele Elemente haben, wie es Nichtterminale gibt (= konstant in der Eingabelänge).

- Es bleiben drei Schleifen (b, i, k) über die Eingabelänge n.

- Deshalb terminiert CKY-Algorithmus unbedingt nach $O(n^3)$ Schritten: polynomieller Erkenner!
  - mal Faktor für die Größe der Grammatik (hängt aber nicht von der Eingabelänge ab)

Korrektheit

- Zu zeigen: Wenn Parser A zu Ch(i,k) hinzufügt, dann gilt $A \Rightarrow^* w_i \ldots w_{k-1}$.

- Vollständige Induktion über k-i:
  - k-i = 1: Folgt aus Regeln für Terminalsymbole.
  - CKY-Parser fügt A zu Ch(i,k) nur dann hinzu, wenn es Regel $A \rightarrow B\ C$ und $i < j < k$ gibt mit $B \in Ch(i,j)$ und $C \in Ch(j,k)$.
  - Wenn Aussage also für Ch(i,j) und Ch(j,k) gilt, dann ist $A \Rightarrow B\ C \Rightarrow^* w_i \ldots w_{j-1}\ C \Rightarrow^* w_i \ldots w_{j-1}\ w_j \ldots w_{k-1}$

Vollständigkeit

- Zu zeigen: Wenn $A \Rightarrow^* w_i \ldots w_{k-1}$, dann legt Parser irgendwann A nach Ch(i,k).

- Knackpunkt im Beweis: Zum Zeitpunkt der Berechnung von Ch(i,k) stehen alle Einträge, die kürzer als k-i sind, schon in der Chart.

- Das wird durch die Reihenfolge der Schleifen im CKY-Algorithmus garantiert.

Implementierung

- Idee der Datenstruktur:
  - chart[i][k] = Ch(i+1, k+1)
  - chart ist eine Liste von Zeilen der Charts
  - Jede Zeile ist eine Liste von Mengen von NT-Symbolen.

- Initialisierung:

```
chart = []

for in in range(n+1):
  row = []
  for j in range(n+1):
    row.append(set())
  chart.append(row)
```

Implementierung

```
for i in range(n):                                          # Terminalregeln
    for prod in grammar.productions(rhs=words[i]):
        chart[i][i+1].add(prod.lhs())



for width in range(2, n+1):                                 # Binaere Regeln
    for i in range(0, n-width+1):
        for j in range(1, width):
            nts1 = chart[i][i+j]
            nts2 = chart[i+j][i+width]
            for nt1 in nts1:
                productions = grammar.productions(rhs=nt1)
                for production in productions:
                    if production.rhs()[1] in nts2:
                        chart[i][i+width].add(production.lhs())
```

Implementierung: Parser

- Für einen *Parser* merkt man sich zu jedem Nichtterminal in einer Chartzelle, auf welche (mehreren) Weisen man es bauen kann.

- chart[i][k] ist jetzt Dictionary (statt Menge).
  - keys(chart[i][k]): Nichtterminale A, die den Teilstring w_i ... w_{k-1} abdecken können
  - Eintrag chart[i][k]["A"]: Liste von *Backpointern*, d.h. Tripeln (B, C, j), die angeben, dass A aus B von i bis j plus C von j bis k gebaut werden kann.

Zusammenfassung

- Rekursionsbasierte Parser (RD, SR) können exponentielle Laufzeit brauchen.
  - in der Praxis viel zu langsam

- Verschiedene Algorithmen für das gleiche Problem können verschiedene *asymptotische* Laufzeit haben.

- CKY ist polynomieller Erkenner.
  - verwendet *Chart*, um Zwischenergebnisse zu tabellieren.

- Laufzeit reduziert sich auf $O(n^3)$.