Bottom-Up- und Top-Down-Parsing

Vorlesung “Computerlinguistische Techniken”

Alexander Koller

20. Oktober 2014

Kontextfreie Grammatiken

T = {Hans, isst, Käsebrot, ein}
N = {S, NP, VP, V, N, Det}; Startsymbol: S

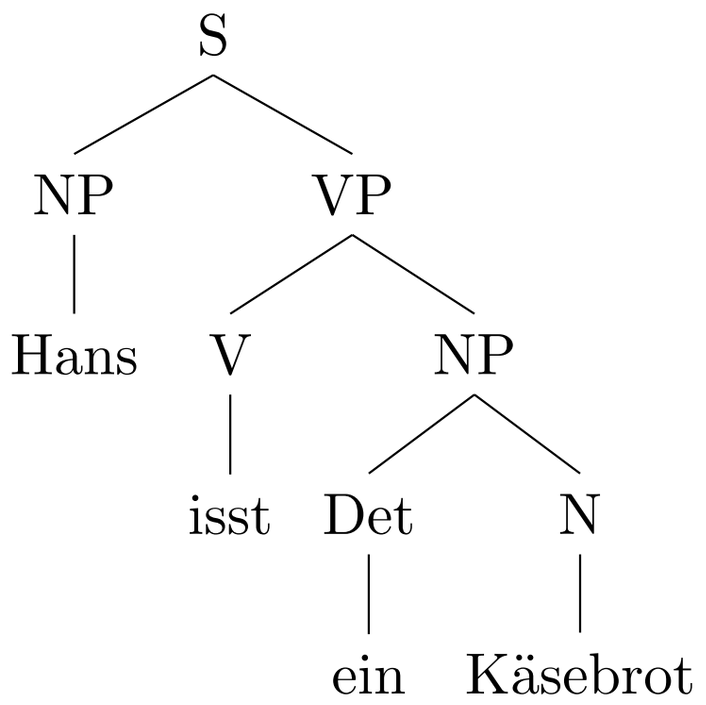
Produktionsregeln:

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

Ableitung

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

Parsebaum



Parsing

- Gegeben seien eine kfG G und ein Wort w.

- Das *Wortproblem* ist die Frage, ob $w \in L(G)$ ist. Wortproblem wird von *Erkenner* gelöst.

- Das *Parsingproblem* ist das Problem, alle Parsebäume von w bzgl. G zu bestimmen. Parsingproblem wird von *Parser* gelöst.

- Wie löst man das Wort- und das Parsingproblem so systematisch, dass man es implementieren kann?

Recursive-Descent-Parsing

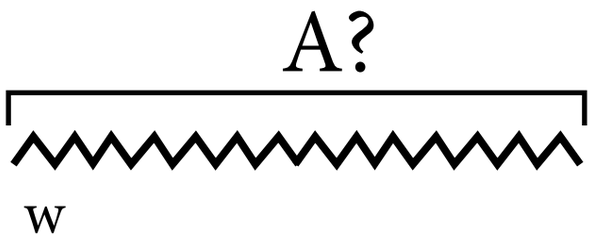
- Ein erster Ansatz für Parsing: Recursive Descent.
- Wir lösen hier nur das Wortproblem; das Parsingproblem lösen Sie in der Übung.
- Wir nehmen an, dass kfGs in Chomsky-Normalform (CNF) sind: Jede Regel ist von der Form $A \to B\ C$ oder $A \to a$.

Recursive-Descent-Parsing

- Rekursiver Algorithmus, der für A und w die Frage "A $\Rightarrow^*$ w?" entscheidet.

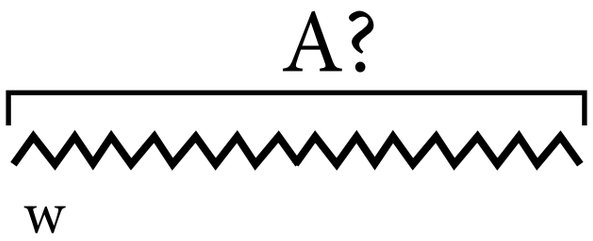
Recursive-Descent-Parsing

- Rekursiver Algorithmus, der für A und w die Frage "A $\Rightarrow^*$ w?" entscheidet.



Recursive-Descent-Parsing

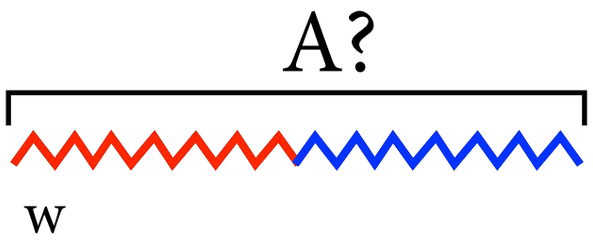
- Rekursiver Algorithmus, der für A und w die Frage "A $\Rightarrow^*$ w?" entscheidet.



$A \rightarrow B\ C$

Recursive-Descent-Parsing

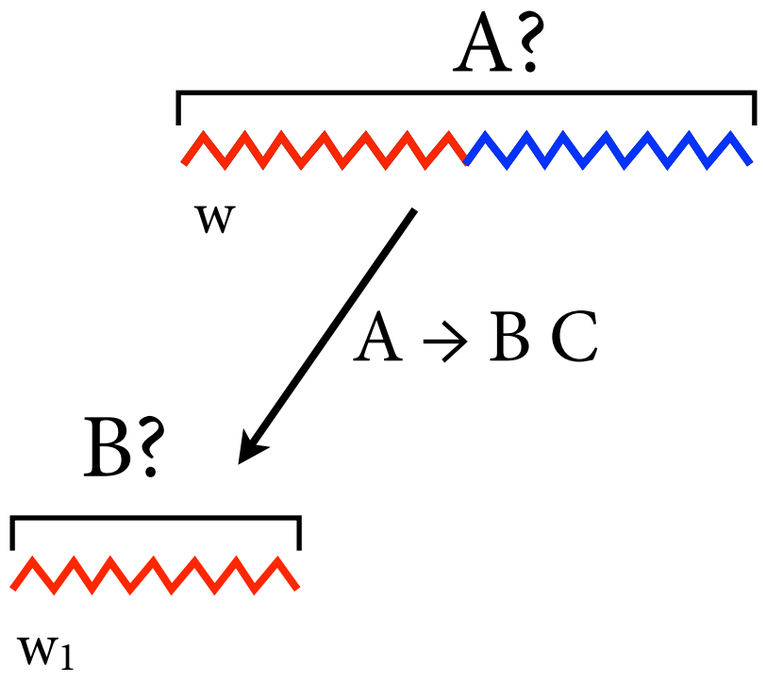
- Rekursiver Algorithmus, der für A und w die Frage "A $\Rightarrow^*$ w?" entscheidet.



$A \rightarrow B\ C$

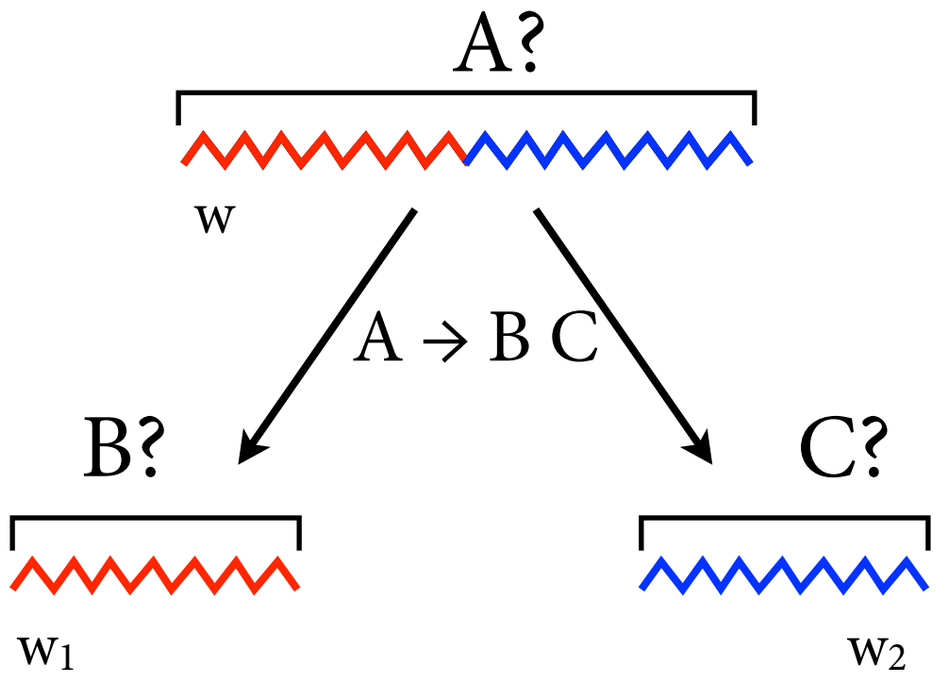
Recursive-Descent-Parsing

- Rekursiver Algorithmus, der für A und w die Frage "A $\Rightarrow^*$ w?" entscheidet.



Recursive-Descent-Parsing

- Rekursiver Algorithmus, der für A und w die Frage "A $\Rightarrow^*$ w?" entscheidet.



Recursive-Descent-Parsing

- Der Algorithmus sieht so aus:
  - Wir wollen wissen, ob $A \Rightarrow^* w$ ist.
  - Wenn $w \in T$ und es eine Regel $A \to w$ gibt, gib "ja" zurück, sonst "nein".
  - Rate eine Regel $A \to B\ C$, die wir anwenden wollen.
  - Rate eine Zerlegung $w = w_1\ w_2$.
  - Wenn sowohl $B \Rightarrow^* w_1$ als auch $C \Rightarrow^* w_2$ "ja" gesagt haben, gib "ja" zurück, sonst "nein".

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

$$S \Rightarrow^* a\ b\ c\ ?$$

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

$$S \Rightarrow^* a\ b\ c\ ?$$

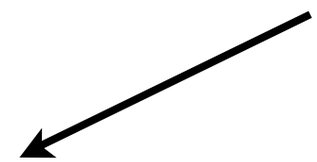
(versuche abc = a· bc, $S \rightarrow A\ B$)

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \rightarrow A\ B$)



$A \Rightarrow^* a\ ?$

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

$$S \Rightarrow^* a\ b\ c\ ?$$

(versuche abc = a· bc, $S \rightarrow A\ B$)

RD-Parsing: Ein Beispiel

$S \to A\ B$	$A \to a$	$B \to b$
$S \to A\ C$	$C \to B\ C$	$C \to c$

$$S \Rightarrow^* a\ b\ c\ ?$$

(versuche abc = a· bc, $S \to A\ B$)

$A \Rightarrow^* a\ ?$

(ja, da $A \to a$)

$B \Rightarrow^* b\ c\ ?$

RD-Parsing: Ein Beispiel

$S \to A\ B$	$A \to a$	$B \to b$
$S \to A\ C$	$C \to B\ C$	$C \to c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \to A\ B$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \to a$)

$B \Rightarrow^* b\ c\ ?$

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \rightarrow A\ B$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \rightarrow a$)

$B \Rightarrow^* b\ c\ ?$

(nein: es gibt keine binären Regeln für B)

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \rightarrow A\ B$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \rightarrow a$)

$B \Rightarrow^* b\ c\ ?$ ✗

(nein: es gibt keine binären Regeln für B)

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

$S \Rightarrow^* a\ b\ c\ ?$ ✗

(versuche abc = a· bc, $S \rightarrow A\ B$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \rightarrow a$)

$B \Rightarrow^* b\ c\ ?$ ✗

(nein: es gibt keine binären Regeln für B)

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

$$S \Rightarrow^* a\ b\ c\ ?$$

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

$$S \Rightarrow^* a\ b\ c\ ?$$

(versuche abc = a· bc, $S \rightarrow A\ C$)

RD-Parsing: Ein Beispiel

$S \to A\ B$	$A \to a$	$B \to b$
$S \to A\ C$	$C \to B\ C$	$C \to c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \to A\ C$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \to a$)

$C \Rightarrow^* b\ c\ ?$

RD-Parsing: Ein Beispiel

$S \to A\ B$	$A \to a$	$B \to b$
$S \to A\ C$	$C \to B\ C$	$C \to c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \to A\ C$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \to a$)

$C \Rightarrow^* b\ c\ ?$

(versuche bc = b· c, $C \to B\ C$)

RD-Parsing: Ein Beispiel

$S \to A\ B$	$A \to a$	$B \to b$
$S \to A\ C$	$C \to B\ C$	$C \to c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \to A\ C$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \to a$)

$C \Rightarrow^* b\ c\ ?$

(versuche bc = b· c, $C \to B\ C$)

$B \Rightarrow^* b\ ?$

$C \Rightarrow^* c\ ?$

RD-Parsing: Ein Beispiel

$S \rightarrow A\ B$	$A \rightarrow a$	$B \rightarrow b$
$S \rightarrow A\ C$	$C \rightarrow B\ C$	$C \rightarrow c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \rightarrow A\ C$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \rightarrow a$)

$C \Rightarrow^* b\ c\ ?$

(versuche bc = b· c, $C \rightarrow B\ C$)

$B \Rightarrow^* b\ ?$

(ja, da $B \rightarrow b$)

$C \Rightarrow^* c\ ?$

RD-Parsing: Ein Beispiel

$S \to A\ B$	$A \to a$	$B \to b$
$S \to A\ C$	$C \to B\ C$	$C \to c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \to A\ C$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \to a$)

$C \Rightarrow^* b\ c\ ?$

(versuche bc = b· c, $C \to B\ C$)

$B \Rightarrow^* b\ ?$ ✓

(ja, da $B \to b$)

$C \Rightarrow^* c\ ?$

RD-Parsing: Ein Beispiel

$S \to A\ B$	$A \to a$	$B \to b$
$S \to A\ C$	$C \to B\ C$	$C \to c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \to A\ C$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \to a$)

$C \Rightarrow^* b\ c\ ?$

(versuche bc = b· c, $C \to B\ C$)

$B \Rightarrow^* b\ ?$ ✓

(ja, da $B \to b$)

$C \Rightarrow^* c\ ?$

(ja, da $C \to c$)

RD-Parsing: Ein Beispiel

$S \to A\ B$	$A \to a$	$B \to b$
$S \to A\ C$	$C \to B\ C$	$C \to c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \to A\ C$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \to a$)

$C \Rightarrow^* b\ c\ ?$

(versuche bc = b· c, $C \to B\ C$)

$B \Rightarrow^* b\ ?$ ✓

(ja, da $B \to b$)

$C \Rightarrow^* c\ ?$ ✓

(ja, da $C \to c$)

RD-Parsing: Ein Beispiel

$S \to A\ B$	$A \to a$	$B \to b$
$S \to A\ C$	$C \to B\ C$	$C \to c$

$S \Rightarrow^* a\ b\ c\ ?$

(versuche abc = a· bc, $S \to A\ C$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \to a$)

$C \Rightarrow^* b\ c\ ?$ ✓

(versuche bc = b· c, $C \to B\ C$)

$B \Rightarrow^* b\ ?$ ✓

(ja, da $B \to b$)

$C \Rightarrow^* c\ ?$ ✓

(ja, da $C \to c$)

RD-Parsing: Ein Beispiel

$S \to A\ B$	$A \to a$	$B \to b$
$S \to A\ C$	$C \to B\ C$	$C \to c$

$S \Rightarrow^* a\ b\ c\ ?$ ✓

(versuche abc = a· bc, $S \to A\ C$)

$A \Rightarrow^* a\ ?$ ✓

(ja, da $A \to a$)

$C \Rightarrow^* b\ c\ ?$ ✓

(versuche bc = b· c, $C \to B\ C$)

$B \Rightarrow^* b\ ?$ ✓

(ja, da $B \to b$)

$C \Rightarrow^* c\ ?$ ✓

(ja, da $C \to c$)

Zwei Beobachtungen

- Der RD-Algorithmus ist rekursiv:
  Um $A \Rightarrow^* w$ zu testen, überprüfen wir
  $B \Rightarrow^* w_1$ und $C \Rightarrow^* w_2$.

- Der RD-Algorithmus ist nichtdeterministisch: Er "rät" Regeln und Stringzerlegungen.
  - Implementierung auf echtem Computer muss deshalb alle Möglichkeiten der Reihe nach durchprobieren.

Implementierung in Prolog

```
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]).
```

Implementierung in Prolog

$S \Rightarrow^* w$

```
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]).
```

Implementierung in Prolog

$S \Rightarrow^* w$

$NP \Rightarrow^* w1$

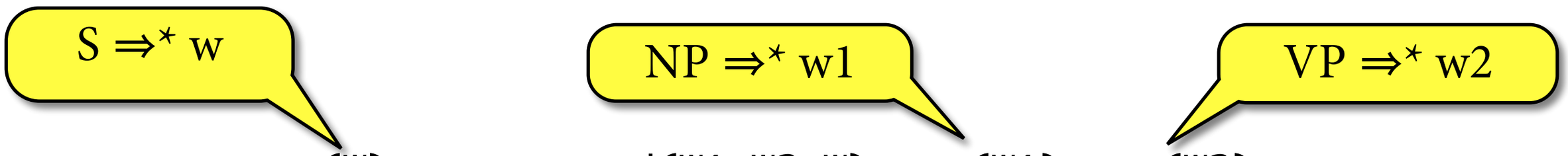
```
s(W)  :- append(W1,W2,W), np(W1), vp(W2).
vp(W) :- append(W1,W2,W), v(W1), np(W2).
np(W) :- append(W1,W2,W), det(W1), n(W2).

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

det([ein]).

v([isst]).
```

Implementierung in Prolog




```
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]).
```

Implementierung in Prolog

S ⇒* w

NP ⇒* w1

VP ⇒* w2

```
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]).
```

NP ⇒* hans !

Korrektheit und Vollständigkeit

- Warum sollen wir glauben, dass RD-Erkenner für alle Eingaben das richtige Ergebnis liefert?

- Wir müssen beweisen:
  - *Korrektheit:* RD-Erkenner behauptet $w \in L(G)$ nur dann, wenn es auch wahr ist.
  - *Vollständigkeit:* Wenn $w \in L(G)$ ist, dann behauptet das der RD-Erkenner auch.

Probleme und Algorithmen

- *Problem*: Funktion von irgendwelchen Eingaben zu {0,1}.
  - Wortproblem von kfGs: gegeben Eingabe (G,w),
    gib 1 aus gdw w in L(G), sonst 0.

- *Algorithmus*: abstrakte Berechnungsvorschrift, die das Problem löst.
  - RD-Parser ist Algorithmus für das Wortproblem.

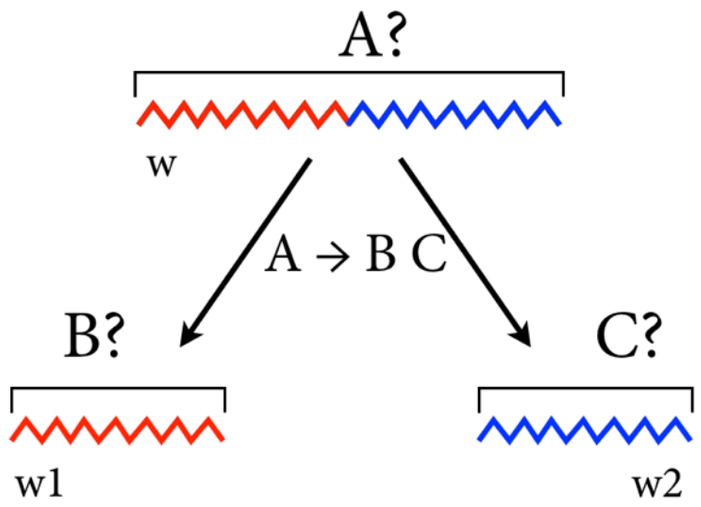
- Algorithmen kann man in Programmiersprachen *implementieren*.

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

Beweistechnik

- Korrektheit und Vollständigkeit von Parsingalgo. beweist man typischerweise mit *vollständiger Induktion:*
  - Korrektheit: … über Länge der Berechnung des Algorithmus
  - Vollständigkeit: … über Länge der Ableitung.

- Vollständige Induktion:
  - Zeige, dass Aussage für n = 0 (oder n = 1) wahr ist.
  - Zeige: Wenn Aussage für beliebiges n wahr ist,
    dann ist sie auch für n+1 wahr.
    (alternativ: sei n beliebig, und nimm an, dass Aussage
    für alle $k \leq n$ wahr ist),
  - Daraus folgt dann: Aussage für alle $n \geq 0$ (bzw. $n \geq 1$) wahr.

Korrektheit des RD-Erkenners

- Korrektheit: Wenn RD-Erkenner $w \in L(G)$ behauptet, dann ist es auch wahr.

- Zeige stärkere Aussage: Wenn RD-Erkenner $A \Rightarrow^* w$ behauptet, ist das wahr. (Daraus folgt Korrektheit.)

- Induktion über die Länge n der Berechnung (= Anzahl der rekursiven Aufrufe):
  - Fall n = 1: Algo. hat ohne rekursiven Aufruf direkt “ja” gesagt. Dann war das, weil es Regel $A \to w$ gibt, also ist $A \Rightarrow^* w$.

Korrektheit des RD-Erkenners

- Zeige stärkere Aussage: Wenn RD-Erkenner $A \Rightarrow^* w$ behauptet, ist das wahr. (Daraus folgt Korrektheit.)
  - Fall n → n+1: Algo. hat "ja" gesagt und dabei im ersten Schritt $w = w_1\, w_2$ zerlegt und Regel $A \to B\ C$ angewendet, und rekursive Aufrufe $B \Rightarrow^* w_1$ und $C \Rightarrow^* w_2$ waren beide "ja".
  - Rekursive Berechnungen waren beide kürzer: $k_B, k_C \leq n$. Wegen Induktionsannahme gilt deshalb:
    $B \Rightarrow^* w_1$ und $C \Rightarrow^* w_2$ sind beide wahr.
  - Deshalb ist auch $A \Rightarrow^* w$ wahr, nämlich wie folgt:

    $A \Rightarrow B\ C \Rightarrow^* w_1\ C \Rightarrow^* w_1\ w_2 = w.$

Vollständigkeit des RD-Erkenners

- Vollständigkeit: Wenn $w \in L(G)$ wahr ist, dann behauptet es der RD-Erkenner auch.

- Zeige stärkere Aussage: Wenn $A \Rightarrow^* w$ wahr ist, dann sagt RD-Erkenner "ja". (Daraus folgt Vollständigkeit.)

- Induktion über die Länge n der Ableitung $A \Rightarrow^* w$:

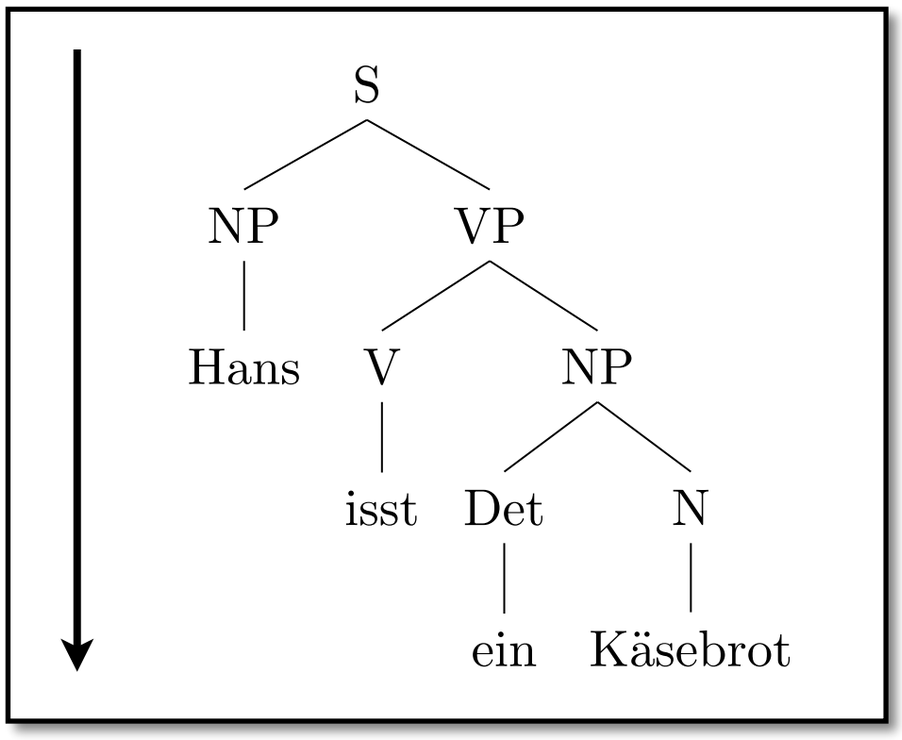
  - Fall $n = 1$: $A \Rightarrow w$, es gibt also Regel $A \to w$.
    RD-Erkenner sagt ohne rekursiven Aufruf "ja".

Vollständigkeit des RD-Erkenners

- Vollständigkeit: Wenn $w \in L(G)$ wahr ist, dann behauptet es der RD-Erkenner auch.
  - Fall $n \to n + 1$: Es gilt $A \Rightarrow^* w$. Diese Ableitung beginnt mit einem ersten Schritt, $A \Rightarrow B\ C \Rightarrow^* w_1\ C \Rightarrow^* w_1\ w_2 = w$.
  - RD-Erkenner überprüft alle Zerlegungen von w, also insbesondere auch $A \to B\ C$ und $w = w_1\ w_2$.
  - Die Ableitungen $B \Rightarrow^* w_1$ und $C \Rightarrow^* w_2$ haben beide weniger als n Schritte. Nach Induktionsannahme sagt RD-Erkenner für beide rekursive Aufrufe deshalb “ja”.
  - Daher sagt RD-Erkenner auch für $A \Rightarrow^* w$ “ja”.

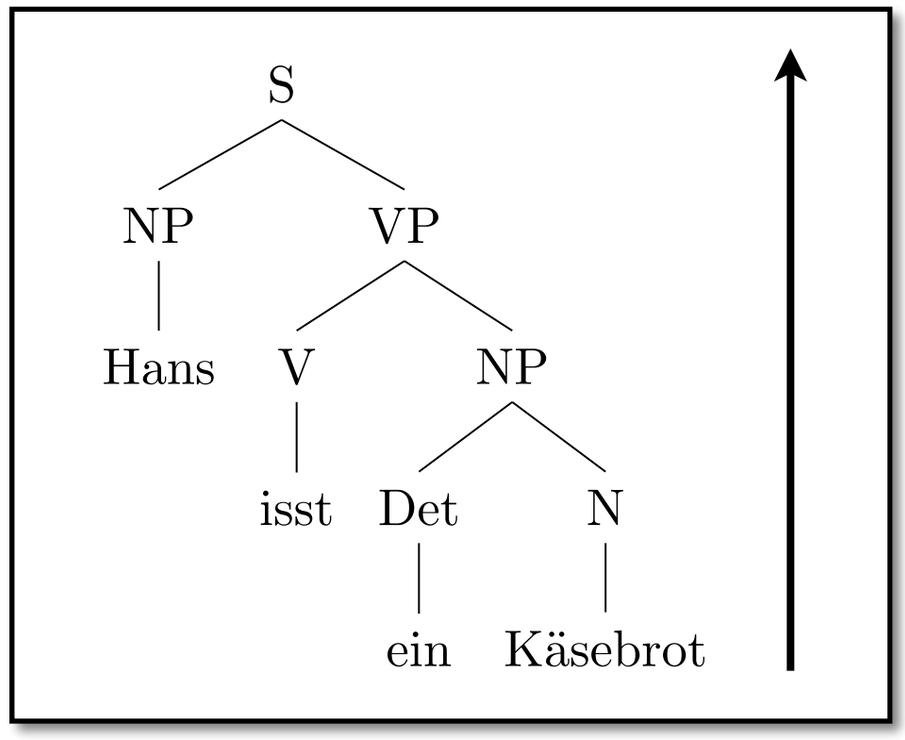
Top-Down vs. Bottom-Up

- Parser kann den Parsebaum top-down oder bottom-up zu berechnen versuchen.



top-down

(z.B. Recursive Descent)



bottom-up

Problematik von Top-Down

- Der RD-Parser muss Zerlegung und Regel raten. Dieses Raten ist weitgehend blind.

$A \rightarrow A\ A$	$A \rightarrow A\ B$	$A \rightarrow a$
$B \rightarrow B\ B$	$B \rightarrow B\ A$	$B \rightarrow b$

a b a b ?

- Parser kann viel Zeit damit verschwenden, aussichtslose Alternativen durchzuprobieren.

Bottom-Up-Parsing

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

Hans isst ein Käsebrot.

Bottom-Up-Parsing

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

Bottom-Up-Parsing

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

NP	V		
Hans	isst	ein	Käsebrot.

Bottom-Up-Parsing

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

NP	V	Det	
|	|	|	
Hans	isst	ein	Käsebrot.

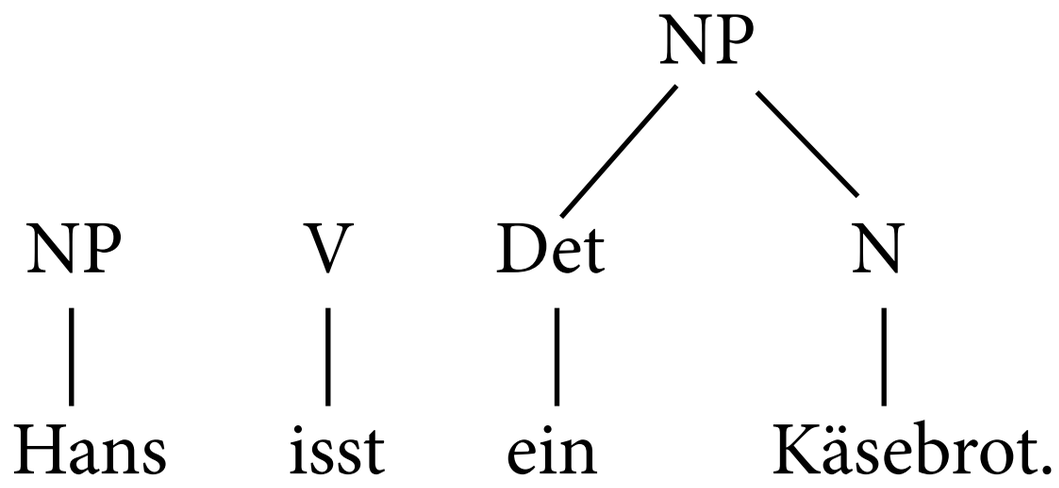
Bottom-Up-Parsing

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

NP	V	Det	N
Hans	isst	ein	Käsebrot.

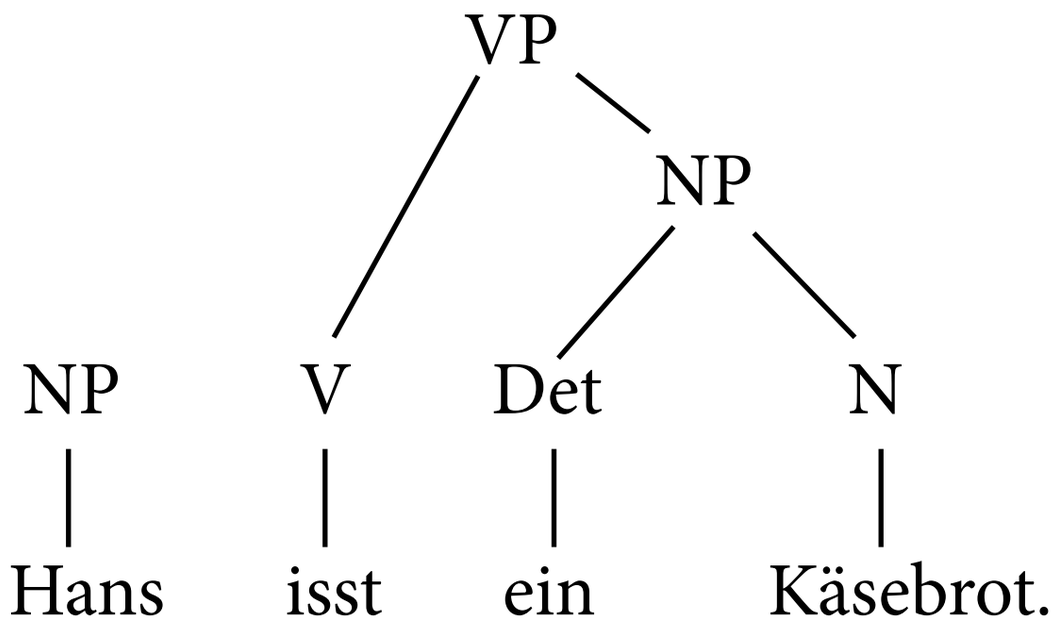
Bottom-Up-Parsing

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



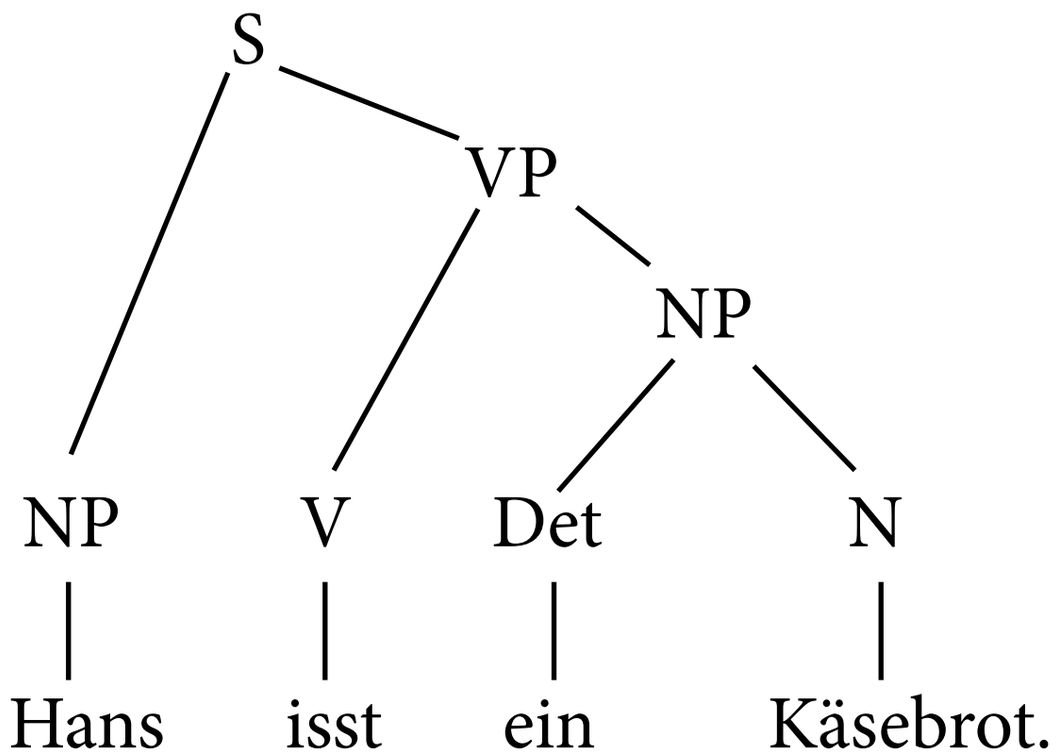
Bottom-Up-Parsing

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



Bottom-Up-Parsing

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



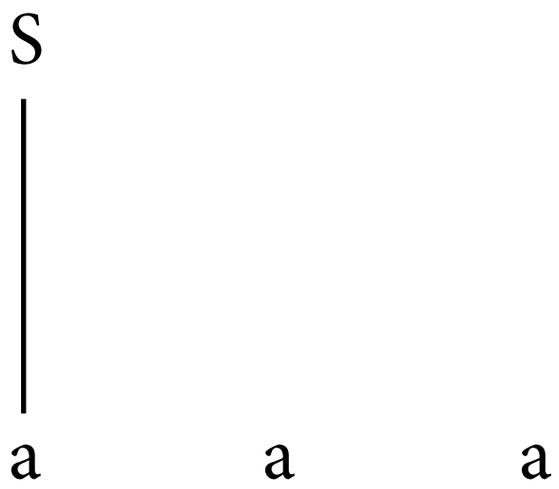
Bottom-up-Parsing

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

a a a

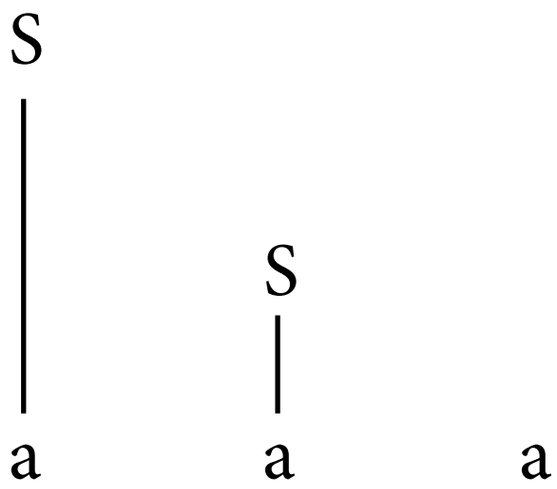
Bottom-up-Parsing

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



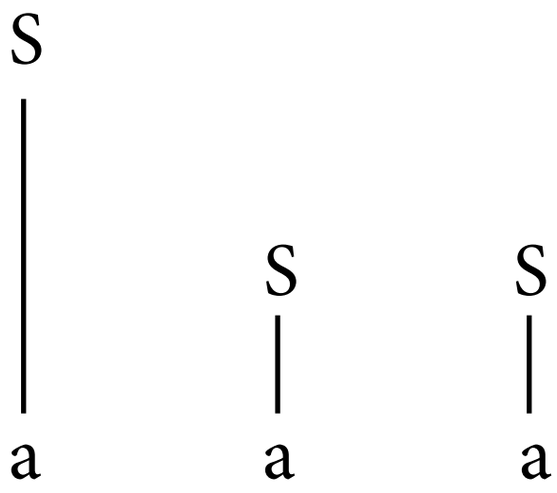
Bottom-up-Parsing

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



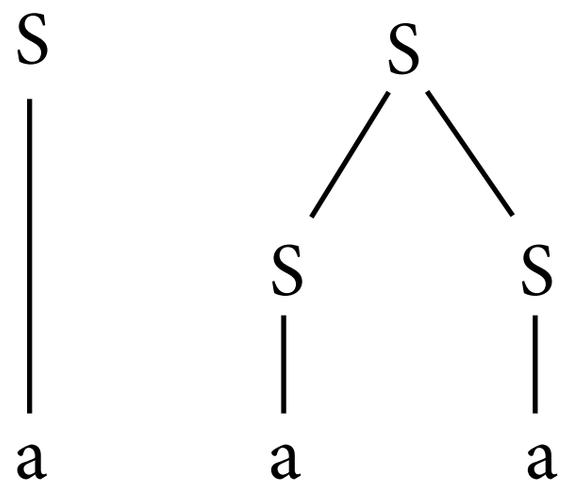
Bottom-up-Parsing

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



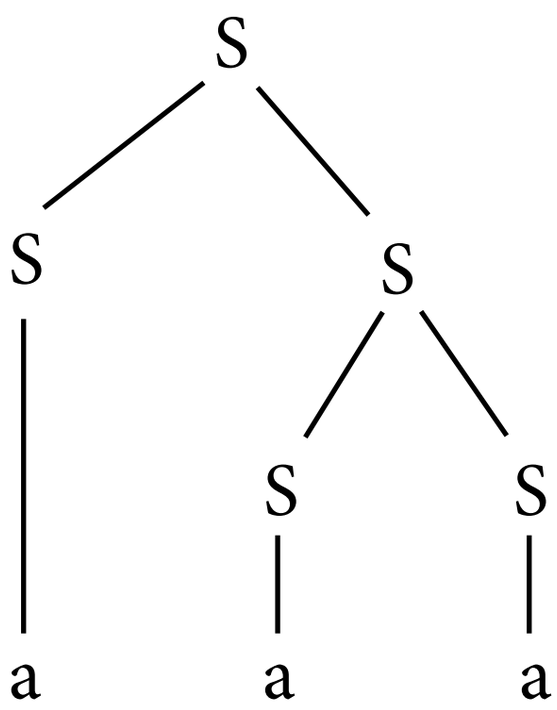
Bottom-up-Parsing

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



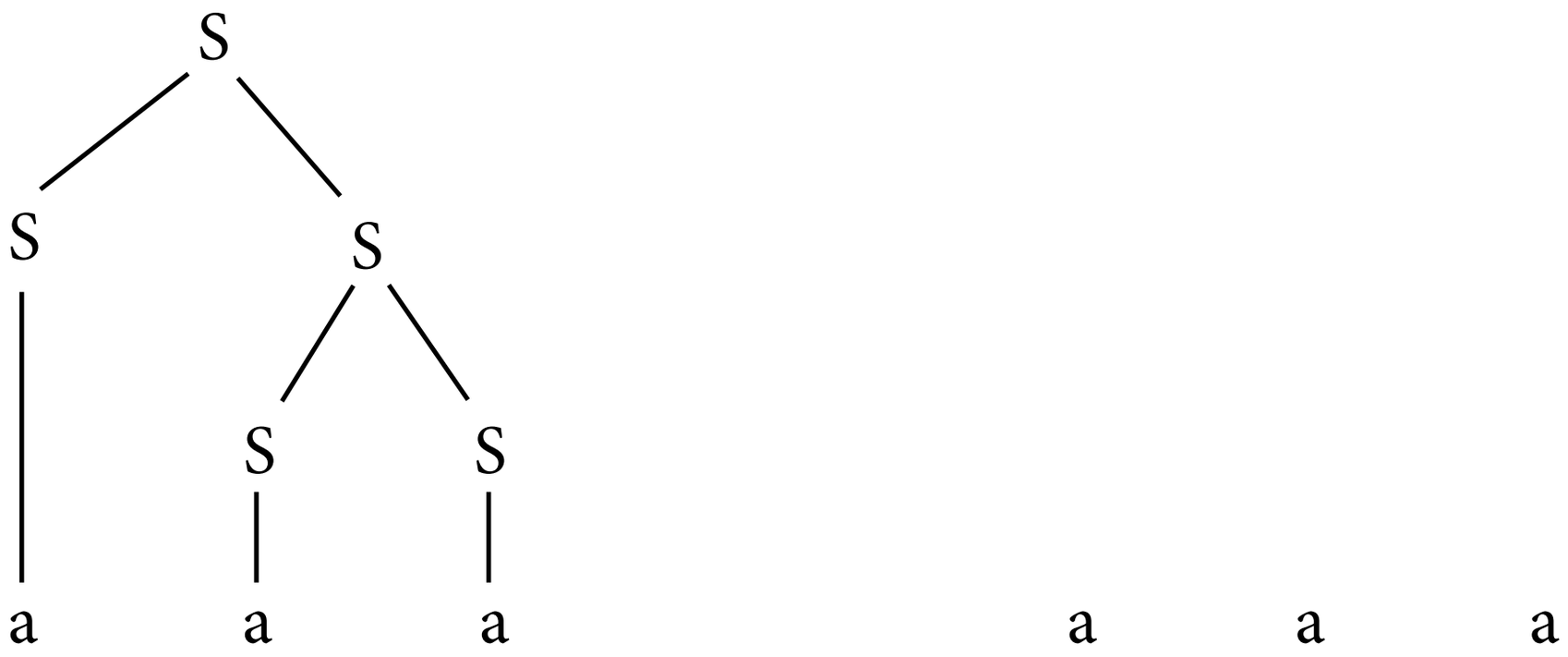
Bottom-up-Parsing

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



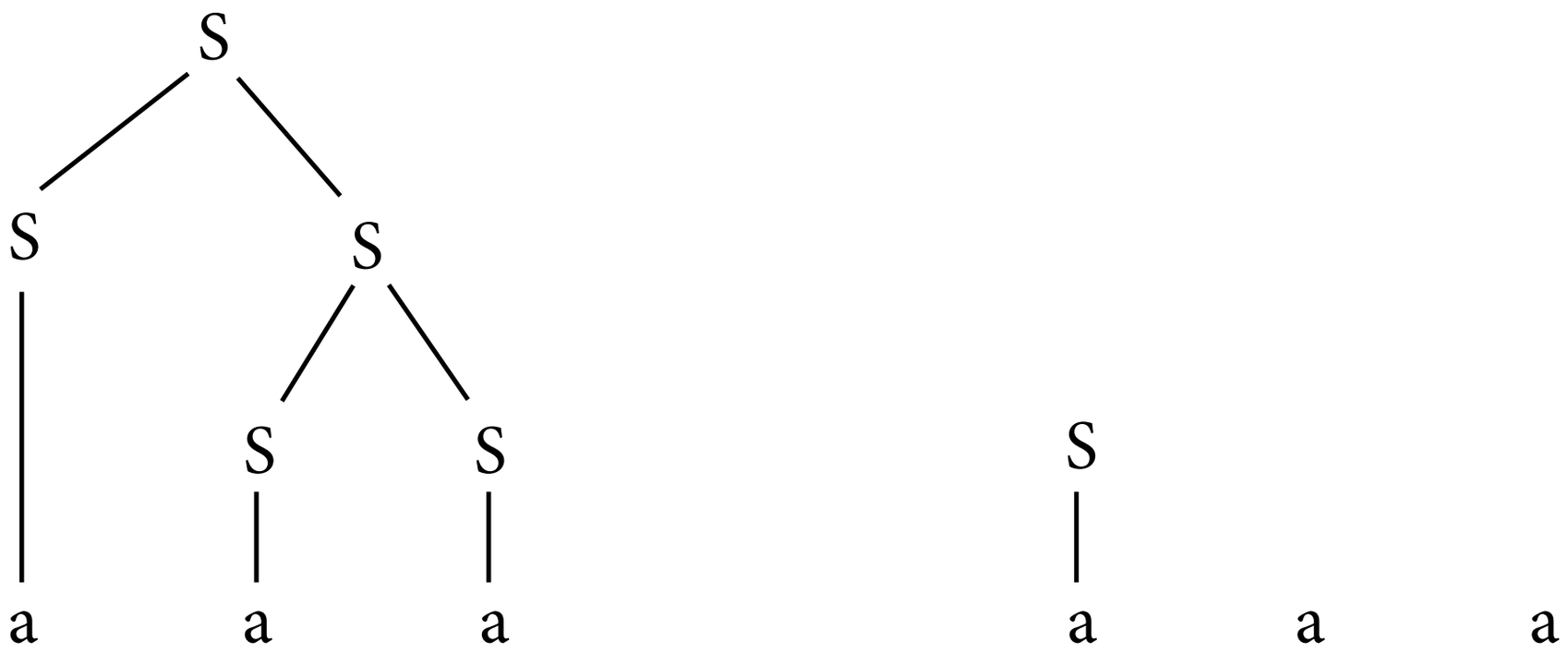
Bottom-up-Parsing

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



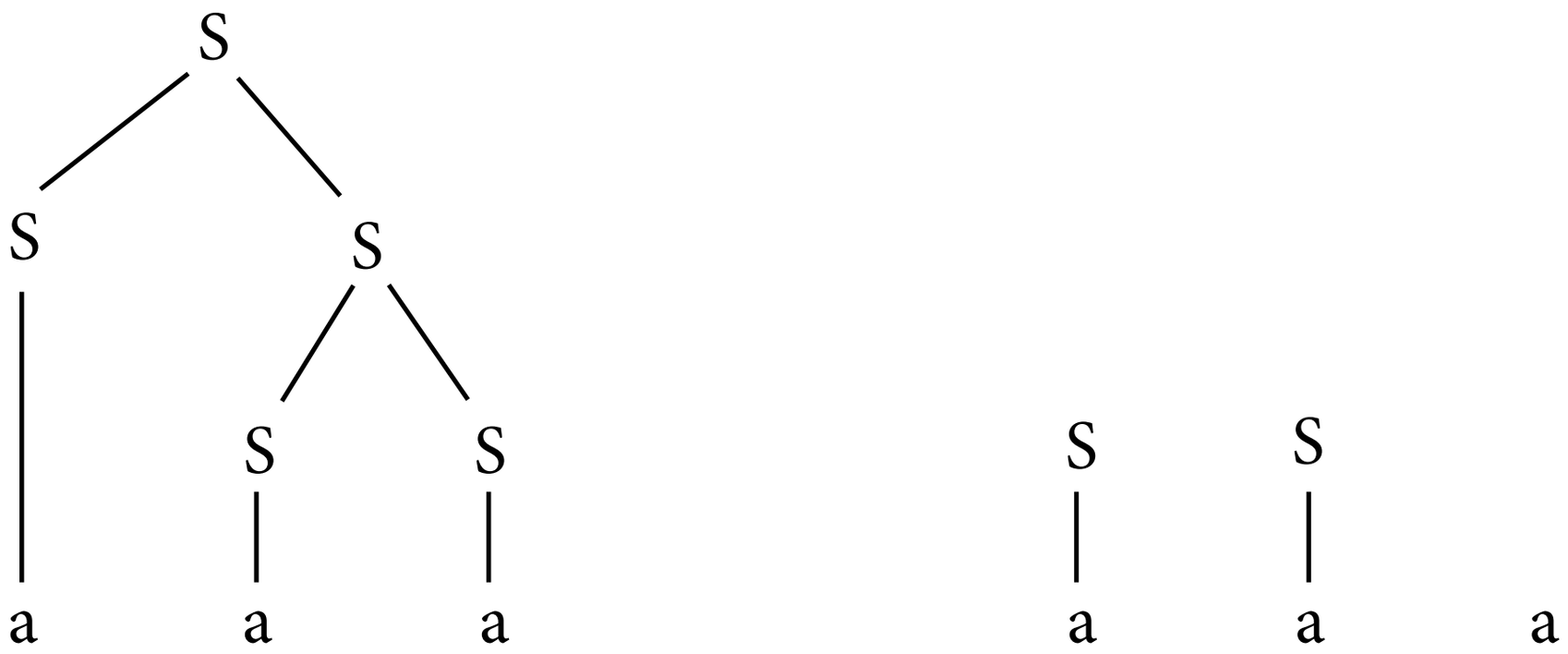
Bottom-up-Parsing

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



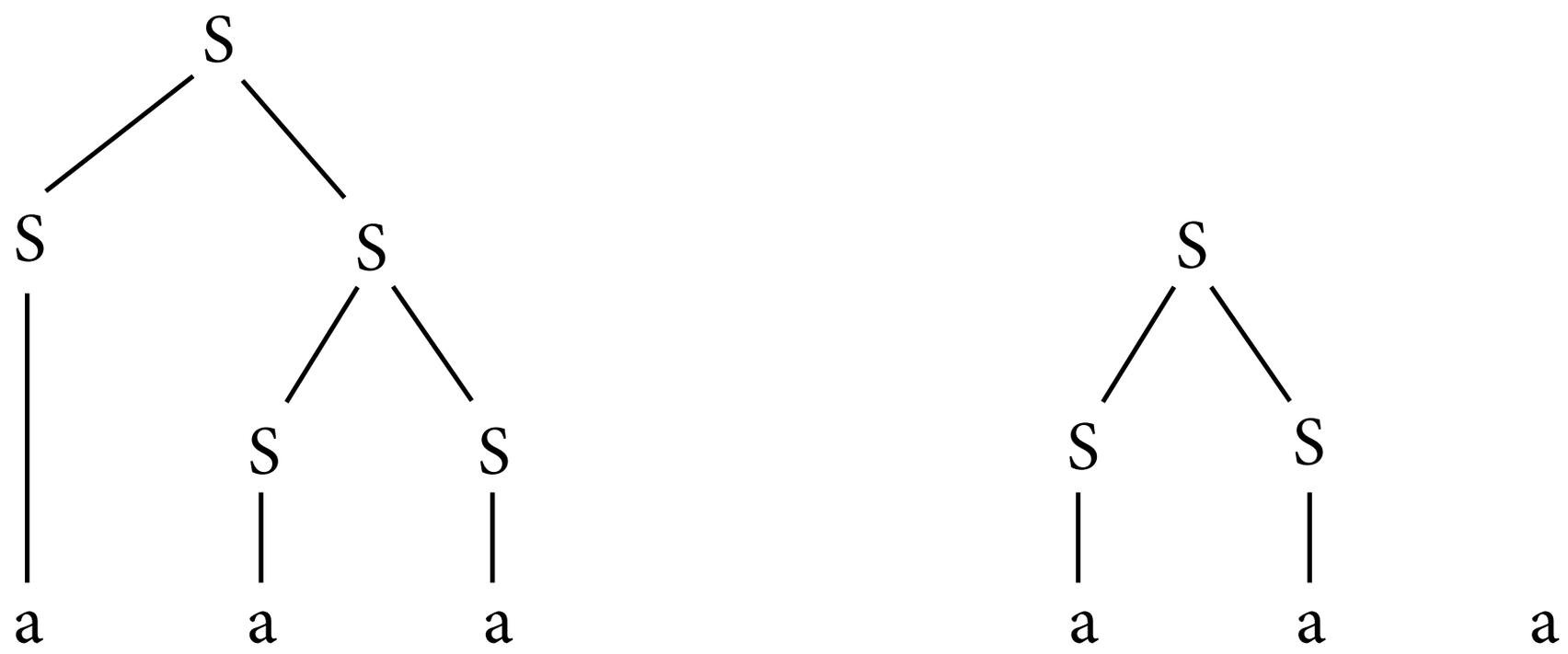
Bottom-up-Parsing

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



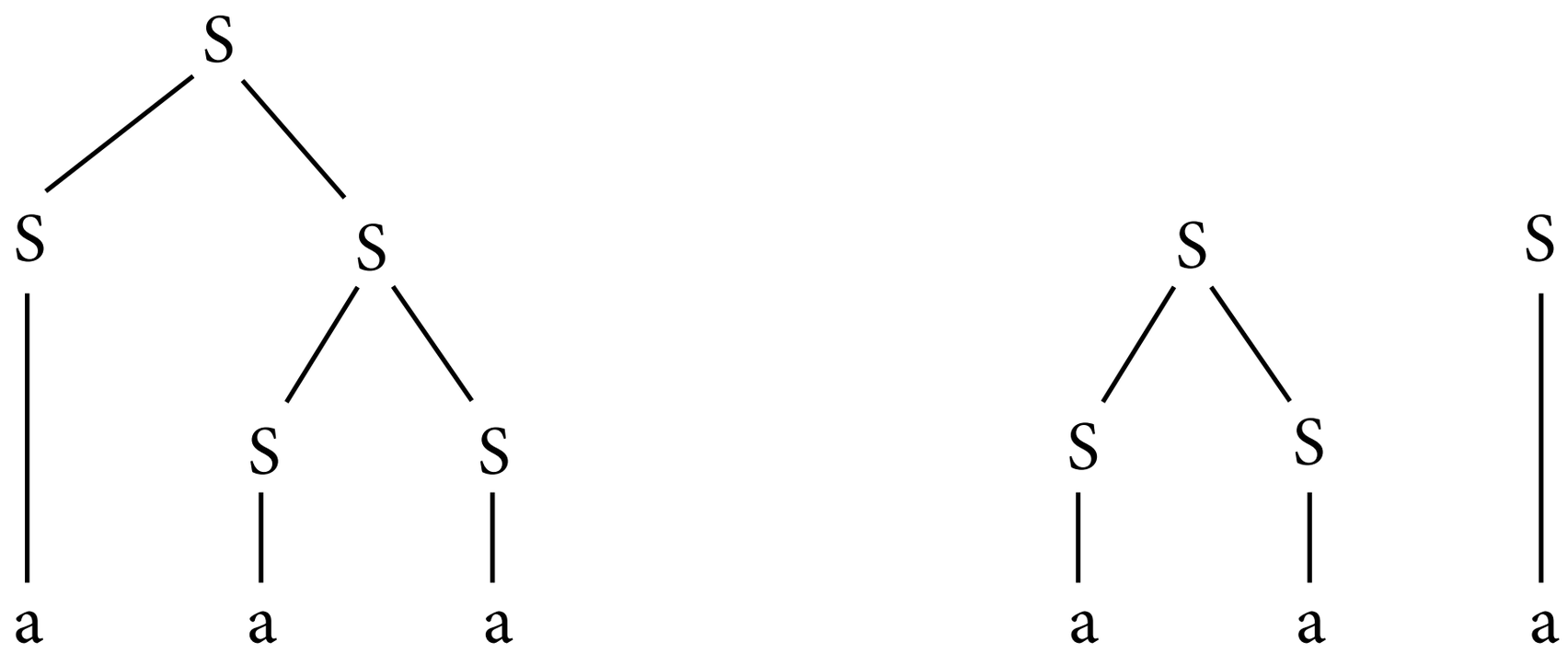
Bottom-up-Parsing

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



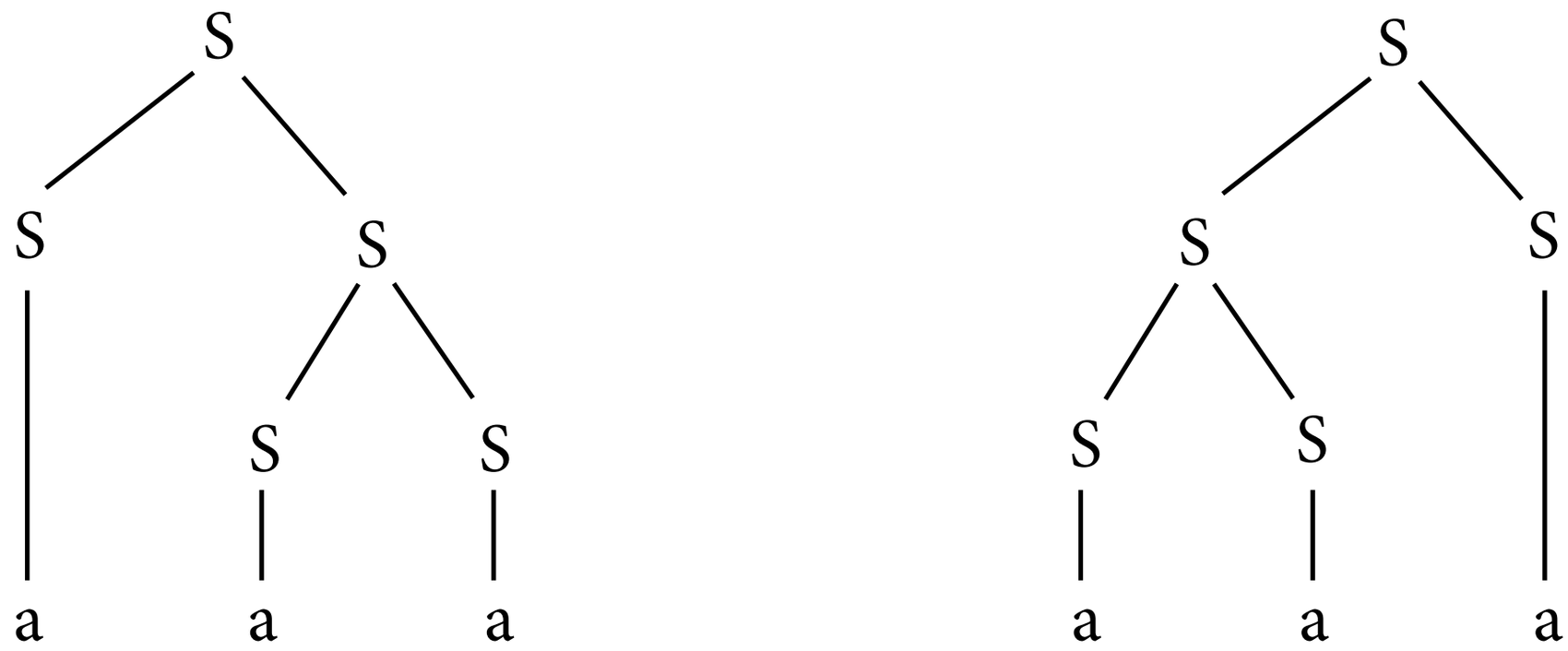
Bottom-up-Parsing

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



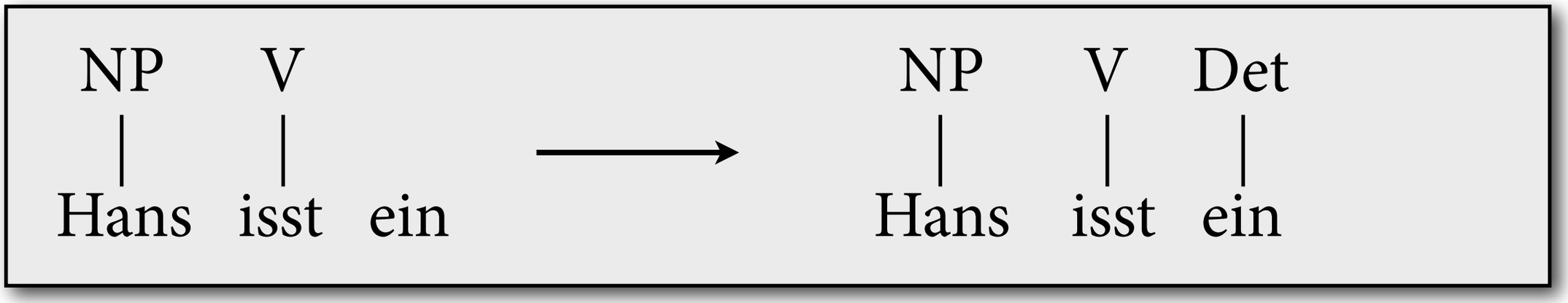
Bottom-up-Parsing

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

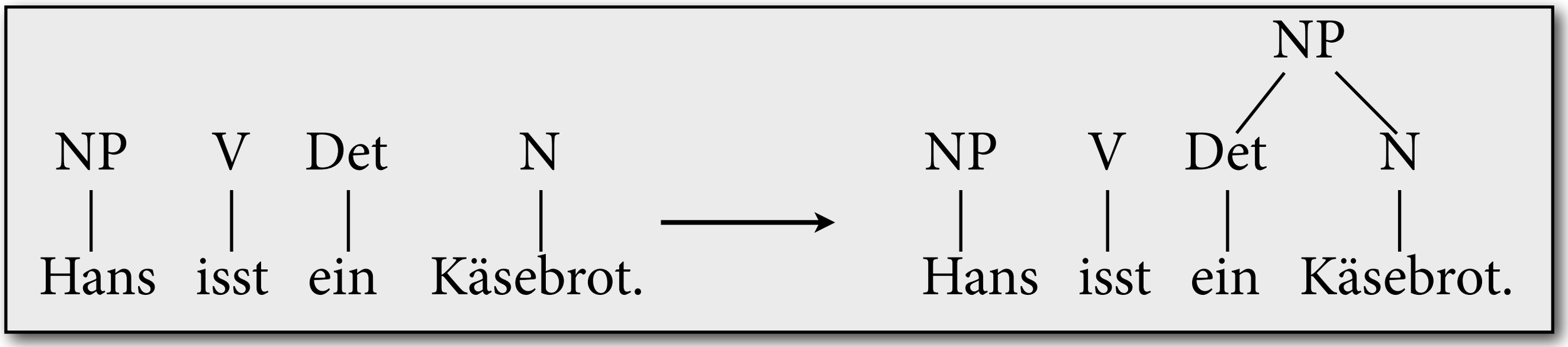


Shift-Reduce-Parsing

Shift: nächstes Terminalsymbol lesen



Reduce: Regel auf letzte offene Symbole anwenden

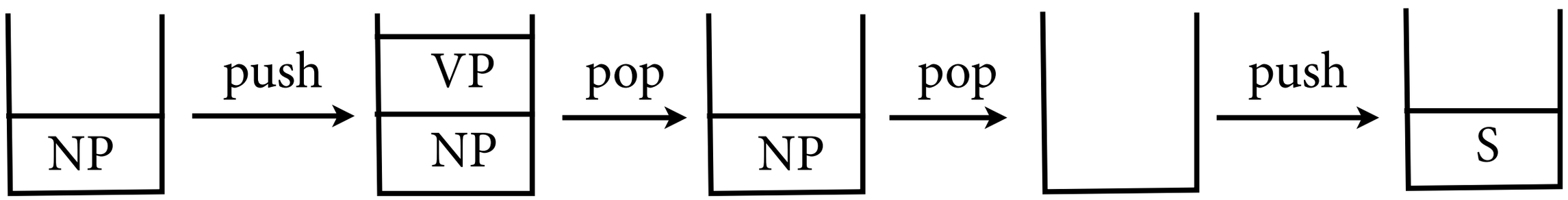


Shift-Reduce-Parsing

- Wir verwenden als Datenstruktur einen Stack von Terminal- und Nichtterminalsymbolen.

- Ein Stack ist eine Liste, in der ich nur an einem Ende (“oben”) lesen und schreiben kann.

- Unser Stack enthält die unverarbeiteten Terminal- und Nichtterminalsymbole.
  Schreibweise: in Stack s_1 s_2 s_3 ist s_3 “oben”.



Shift-Reduce-Parsing

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

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

- Start: (ε, w)

- Wende Regeln nichtdeterministisch an. Algorithmus sagt “ja”, wenn er Konfiguration (S, ε) erreicht (d.h. erreichen kann).

Shift-Reduce: Beispiel

Shift: $(s, a \cdot w) \rightarrow (s \cdot a, w)$
Reduce: $(s \cdot w', w) \rightarrow (s \cdot A, w)$

Hans isst ein Käsebrot.

Shift-Reduce: Beispiel

Shift: $(s, a \cdot w) \rightarrow (s \cdot a, w)$
Reduce: $(s \cdot w', w) \rightarrow (s \cdot A, w)$

Hans isst ein Käsebrot.

(ε, Hans isst ein K.)

Shift-Reduce: Beispiel

Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)

(ε, Hans isst ein K.) → (Hans, isst ein K.)

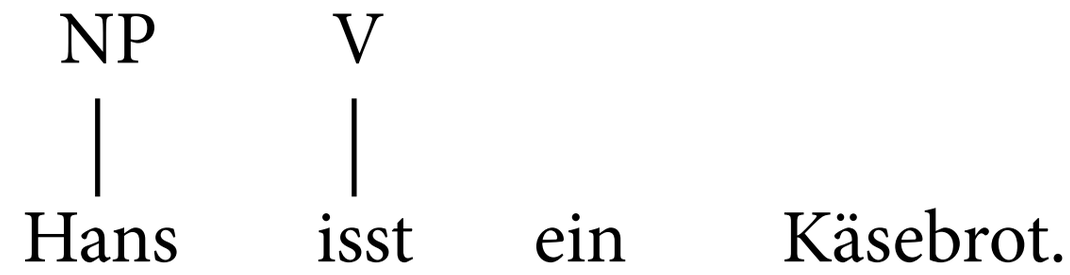
Shift-Reduce: Beispiel

Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)

(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)

Shift-Reduce: Beispiel

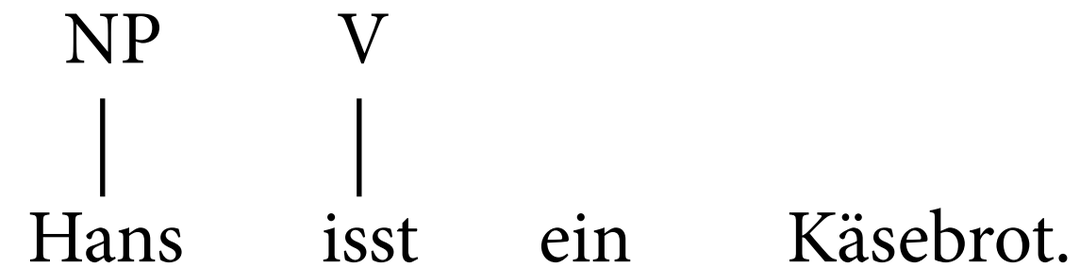
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.)

Shift-Reduce: Beispiel

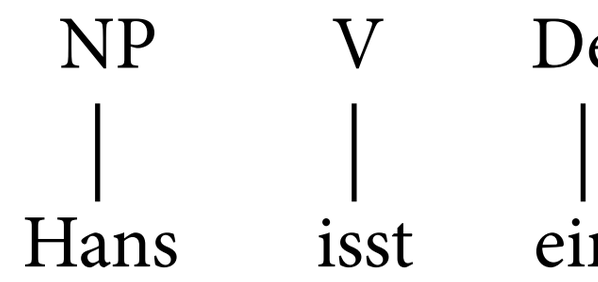
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.)

Shift-Reduce: Beispiel

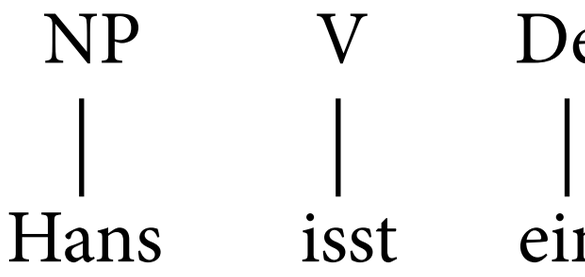
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)

Shift-Reduce: Beispiel

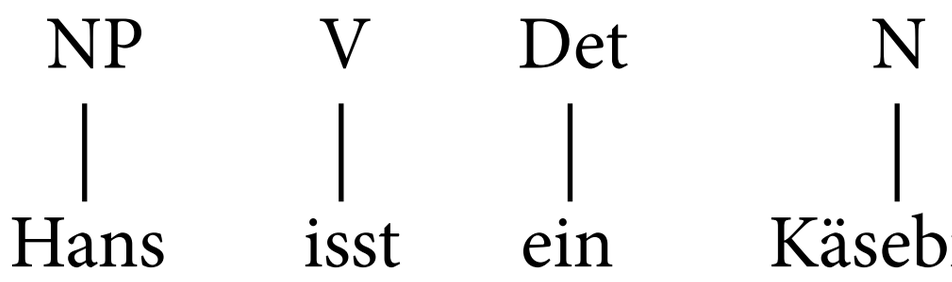
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.)

Shift-Reduce: Beispiel

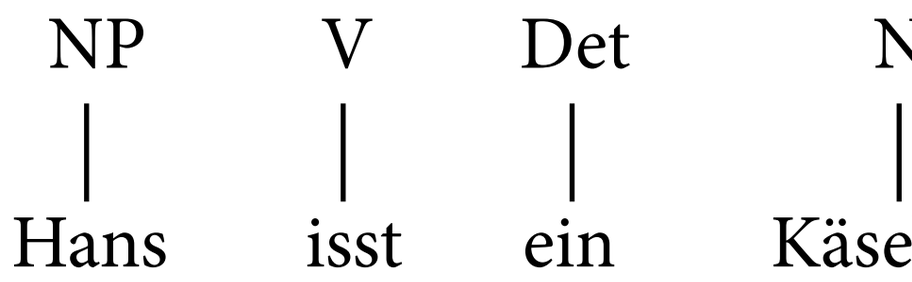
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε)

Shift-Reduce: Beispiel

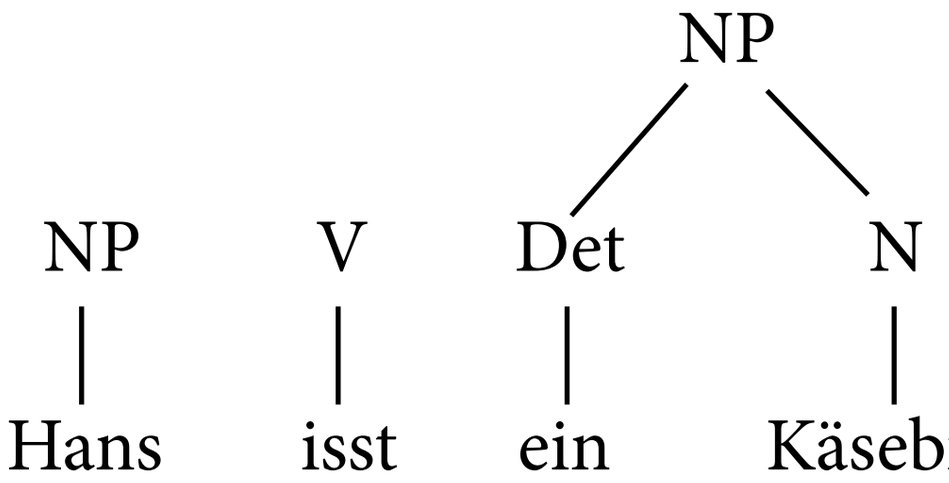
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε) → (NP V Det N, ε)

Shift-Reduce: Beispiel

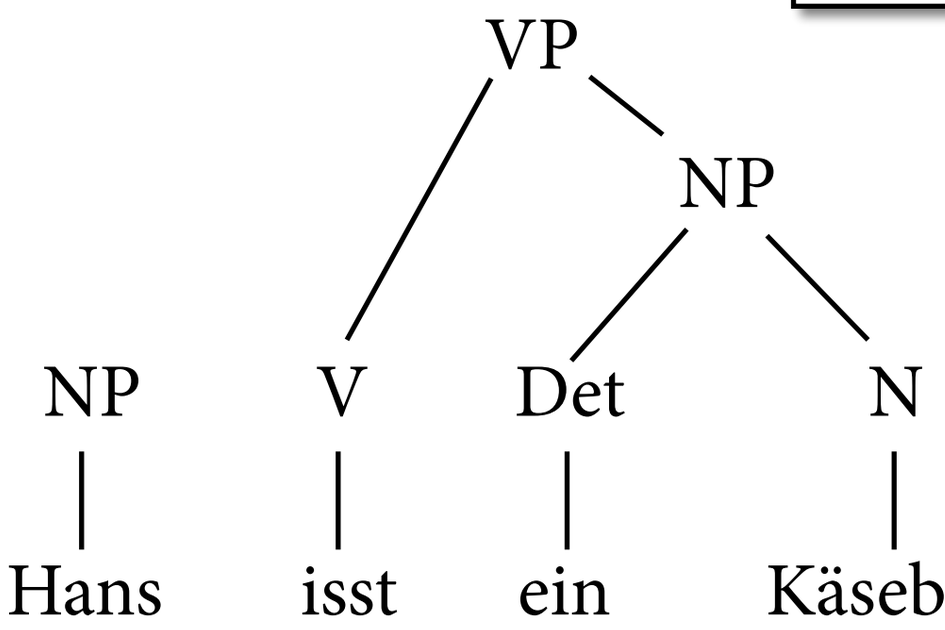
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε) → (NP V Det N, ε)
→ (NP V NP, ε)

Shift-Reduce: Beispiel

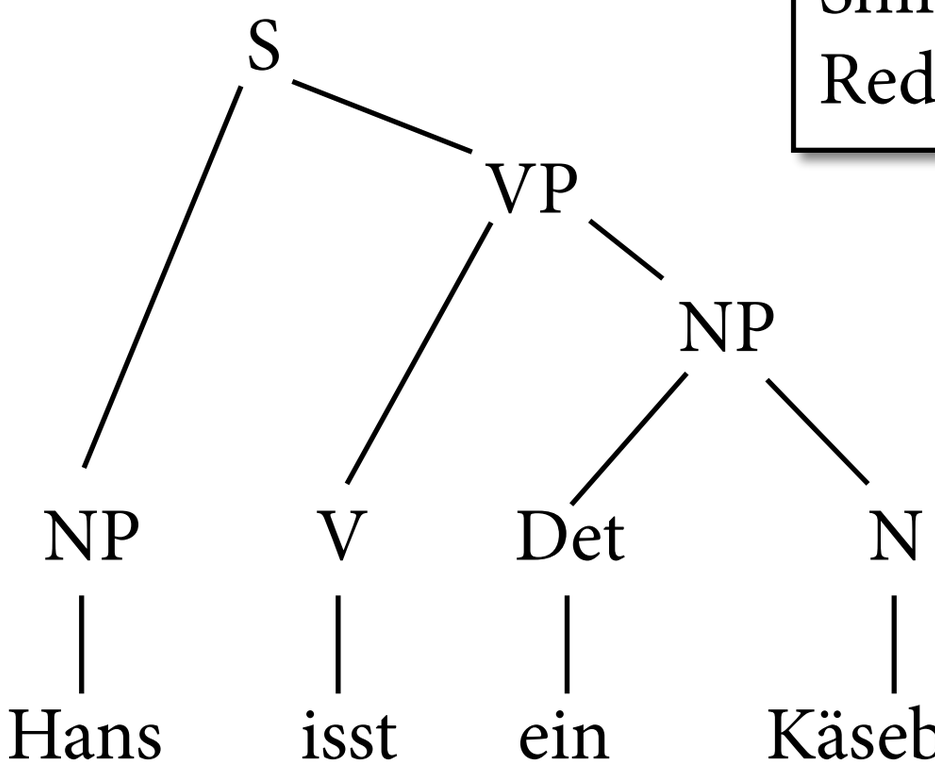
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)



(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε) → (NP V Det N, ε)
→ (NP V NP, ε) → (NP VP, ε)

Shift-Reduce: Beispiel

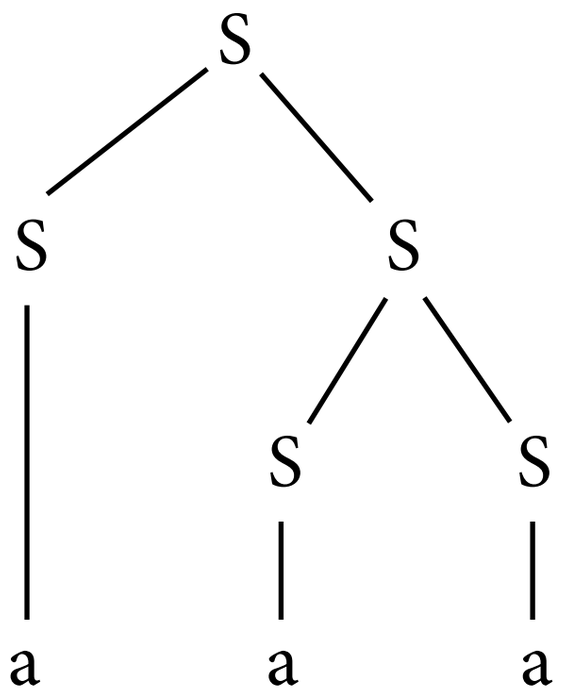
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)

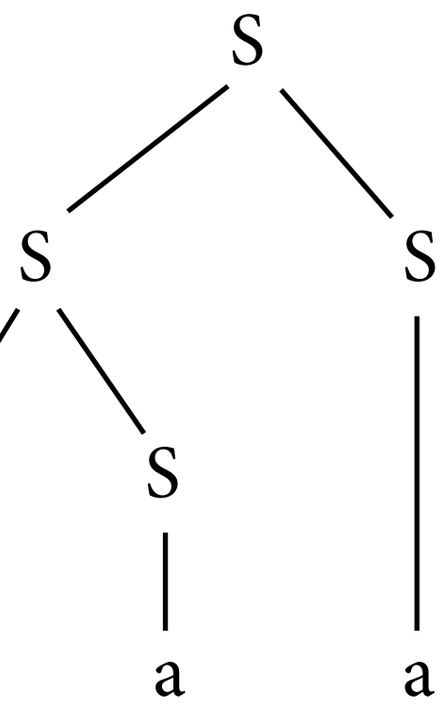


(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε) → (NP V Det N, ε)
→ (NP V NP, ε) → (NP VP, ε) → (S, ε)

Shift-Reduce: Beispiel

$S \rightarrow S\,S$ $\qquad$ $S \rightarrow a$





$(\varepsilon, aaa) \rightarrow (a, aa) \rightarrow (S, aa)$
$\rightarrow (Sa, a) \rightarrow (SS, a) \rightarrow (SSa, \varepsilon)$
$\rightarrow (SSS, \varepsilon) \rightarrow (SS, \varepsilon) \rightarrow (S, \varepsilon)$

$(\varepsilon, aaa) \rightarrow (a, aa) \rightarrow (S, aa)$
$\rightarrow (Sa, a) \rightarrow (SS, a) \rightarrow (S, a)$
$\rightarrow (Sa, \varepsilon) \rightarrow (SS, \varepsilon) \rightarrow (S, \varepsilon)$

Shift-Reduce in Prolog

- Implementiere Ausführung eines Schrittes mit einem Schritt-Prädikat step:
  - 1. Argument: noch zu lesender Eingabestring
  - 2. Argument: Stack, mit oberstem Element links (weil Prolog leicht auf das erste Element einer Liste zugreifen kann, aber nicht auf das letzte)
  - probiert shift und reduce durch

- Das bedeutet: Wir repräsentieren den Stack in umgekehrter Reihenfolge.

Shift-Reduce in Prolog

```
parse(Satz) :- step(Satz, []).

step([], [s]).
step(Satz, Stack) :-
   reduce(Satz, Stack, Satz1, Stack1), step(Satz1,Stack1).
step(Satz, Stack) :-
   shift(Satz, Stack, Satz1, Stack1), step(Satz1, Stack1).

shift([W|Rest], Stack, Rest, [W|Stack]).

reduce(Satz, [B,C|Rest], Satz, [A|Rest]) :- rule(A, C, B).
reduce(Satz, [B|Rest], Satz, [A|Rest]) :- lex(A, B).

rule(s, np, vp).    rule(vp, tv, np).

lex(np, hans).  lex(n, kaesebrot).  lex(det, ein).  lex(v, isst).
```

Vom Erkenner zum Parser

```
parse(Satz, T) :- step(Satz, [], T).

step([], [(s,T)], T).
step(Satz, Stack, T) :- reduce(Satz, Stack, Satz1, Stack1),
       step(Satz1,Stack1, T).
step(Satz, Stack, T) :- shift(Satz, Stack, Satz1, Stack1),
       step(Satz1, Stack1, T).

shift([W|Rest], Stack, Rest, [(W,W)|Stack]).

reduce(Satz, [(B,T1), (C,T2) | Rest], Satz, [(A,T)|Rest]) :-
       rule_with_trees(A, C, B, T2, T1, T).

reduce(Satz, [(B,T1)|Rest], Satz, [(A,T)|Rest]) :-
       lex_with_trees(A, B, T1, T).

rule_with_trees(A, B, C, T1, T2, T) :-
   rule(A, B, C), T =.. [A, T1, T2].

lex_with_trees(A, B, T1, T) :- lex(A,B), T =.. [A, T1].
```

Shift-Reduce: Korrektheit

- Zeige:
  - wenn Shift-Reduce-Berechnung (s,w) $\rightarrow^*$ (s',w') existiert,
  - dann gibt es kfG-Ableitung s'w' $\Rightarrow^*$ sw.

- Daraus folgt dann Korrektheit:
  - SR-Erkenner behauptet $w \in L(G)$
  - gdw $(\varepsilon, w) \rightarrow^* (S, \varepsilon)$
  - daher $S \Rightarrow^* w$ (s.o.)
  - d.h. $w \in L(G)$ ist wahr.

$(s,w) \to^* (s',w') \quad \Rightarrow \quad s'w' \Rightarrow^* sw$

Induktion über Länge n der Berechnung
(d.h. Anzahl der Shift- und Reduce-Schritte)

- n = 0: $sw \Rightarrow^* sw$ trivial
- n → n+1: Betrachte ersten Schritt von $(s,w) \to^* (s',w')$
  - erster Schritt shift: $(s, aw) \to (sa, w) \to^* (s',w')$
    es gilt: $s'w' \underset{IA}{\Rightarrow}^* sa \cdot w \Rightarrow^* s \cdot aw$
  - erster Schritt reduce: $(sw', w) \to (sA, w) \to^* (s',w')$
    es gilt: $s'w' \underset{IA}{\Rightarrow}^* sAw \Rightarrow sw'w$

Shift-Reduce: Vollständigkeit

- Zeige:
  - wenn kfG-Ableitung $A \Rightarrow^* w$ existiert mit $w \in T^*$,
  - dann gibt es SR-Berechnung $(\varepsilon, w) \rightarrow^* (A, \varepsilon)$
- Daraus folgt dann Vollständigkeit:
  - $w \in L(G)$ gilt
  - gdw kfG-Ableitung $S \Rightarrow^* w$ existiert mit $w \in T^*$
  - dann gibt es SR-Berechnung $(\varepsilon, w) \rightarrow^* (S, \varepsilon)$ (s.o.)
  - also behauptet SR-Erkenner, dass $w \in L(G)$ gilt.

$A \Rightarrow^* w \quad \Rightarrow \quad (\varepsilon, w) \rightarrow^* (A, \varepsilon)$

Induktion über Länge n der kfG-Ableitung
(für CNF-Grammatiken; allgemeiner Fall nur mehr Schreibarbeit)

- n = 1: d.h. $A \rightarrow w \in P$, also $(\varepsilon, w) \underset{s}{\rightarrow} (w, \varepsilon) \underset{r}{\rightarrow} (A, \varepsilon)$

- n → n+1: d.h. $A \Rightarrow B\ C \Rightarrow^* B\ w_2 \Rightarrow^* w_1\ w_2 = w$
  - IA1: $(\varepsilon, w_1) \rightarrow^* (B, \varepsilon)$; daher auch $(\varepsilon, w_1\ v) \rightarrow^* (B, v)$
  - IA2: $(\varepsilon, w_2) \rightarrow^* (C, \varepsilon)$; daher auch $(u, w_2) \rightarrow^* (u\ C, \varepsilon)$
  - also: $(\varepsilon, w_1\ w_2) \underset{IV1}{\rightarrow^*} (B, w_2) \underset{IV2}{\rightarrow^*} (B\ C, \varepsilon) \underset{r}{\rightarrow} (A, \varepsilon)$

Effizienzprobleme?

- Auch SR-Parser muss Kombinationen von Shift und Reduce blind probieren, wird ggf. langsam:

$S \rightarrow A\ S'$	$S' \rightarrow S\ B$	$S \rightarrow A\ B$
$B \rightarrow a$	$A \rightarrow a$	$B \rightarrow b$

a a b b ?

- Parser muss jedes a als A analysieren, kann aber auch B auswählen. Fehler merkt man erst, wenn ganzer String auf Stack steht.

Probleme: Übersicht

- Parsertypen haben komplementäre Probleme:

	top-down	bottom-up
Regeln raten	$A \rightarrow w$ vs. $A \rightarrow w$	$A \rightarrow w$ vs. $B \rightarrow w$
Zerlegung raten	String aufteilen	Shift vs. Reduce entscheiden

- Allgemein nicht zu vermeiden (Ambiguität).

Zusammenfassung

- Bottom-up vs. Top-down:
  - top-down: Recursive Descent
  - bottom-up: Shift-Reduce

- Korrektheit und Vollständigkeit
  - Beide Parsertypen haben komplementäre Probleme beim Parsing verschiedener Grammatiken.