Shift-Reduce-Parsing
Laufzeitanalysen

Vorlesung "Computerlinguistische Techniken"

Alexander Koller

23. Oktober 2015

Kontextfreie Grammatiken

T = {Hans, isst, Käsebrot, ein}

N = {S, NP, VP, V, N, Det}; Startsymbol: S

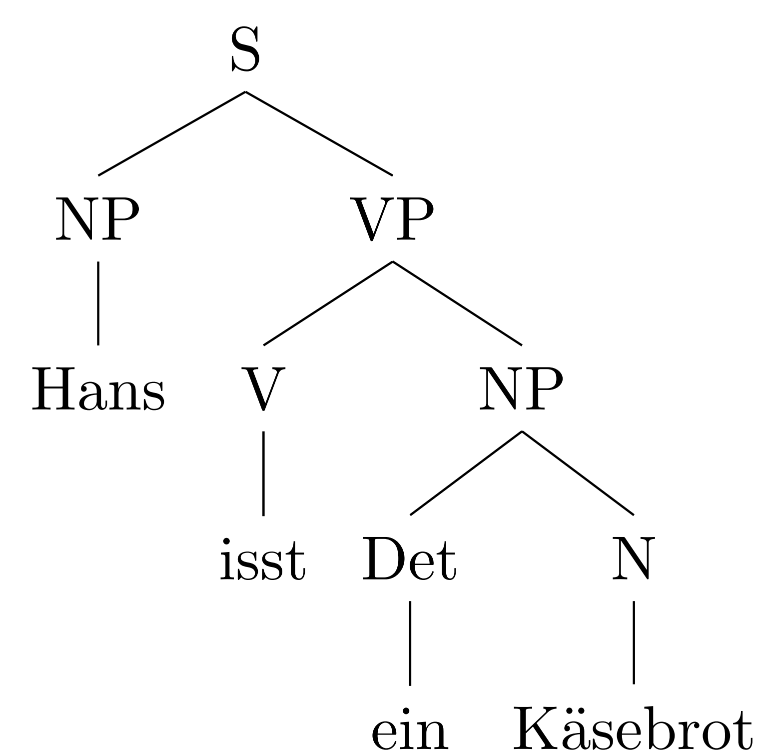
Produktionsregeln:

$S \rightarrow NP\ VP$	$V \rightarrow$ isst	$Det \rightarrow$ ein
$NP \rightarrow Det\ N$	$NP \rightarrow$ Hans	$N \rightarrow$ Käsebrot
$VP \rightarrow V\ NP$		

Ableitung

$$\begin{aligned} S &\Rightarrow NP\ \ VP \ \Rightarrow \text{Hans}\ \ VP \\ &\Rightarrow \text{Hans}\ \ V\ NP \ \Rightarrow \text{Hans isst } NP \\ &\Rightarrow \text{Hans isst } Det\ N \\ &\Rightarrow \text{Hans isst ein } N \\ &\Rightarrow \text{Hans isst ein Käsebrot} \end{aligned}$$

Parsebaum

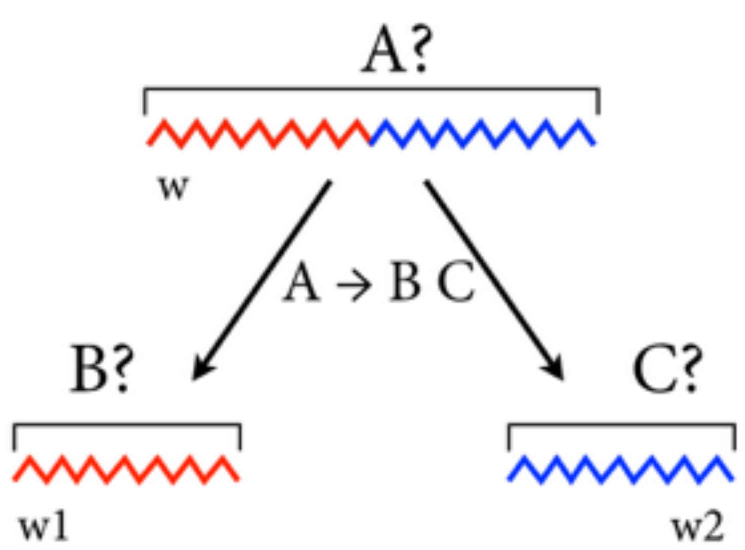


Probleme und Algorithmen

Problem

Wortproblem(G,w) = 1
gdw $w \in L(G)$

Algorithmus

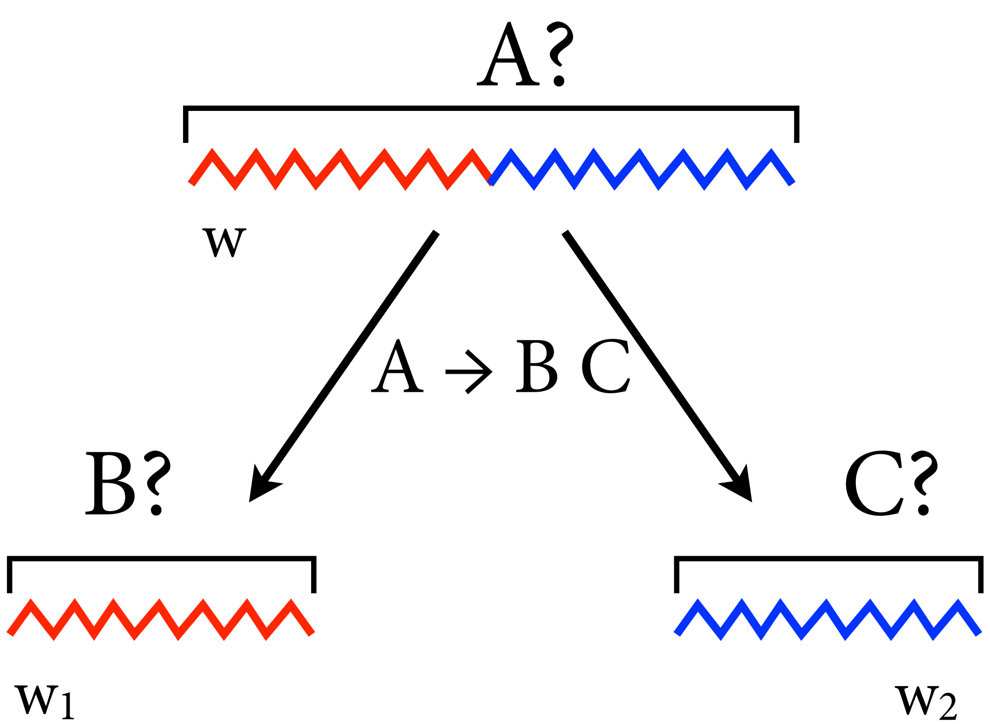



```
function S(w, i, k):
  for j = i+1, ..., k-1 do
    if NP(w, i, j) and VP(w, j, k) then
      return true
    else
      return false
    end if
  end for
```

Programm

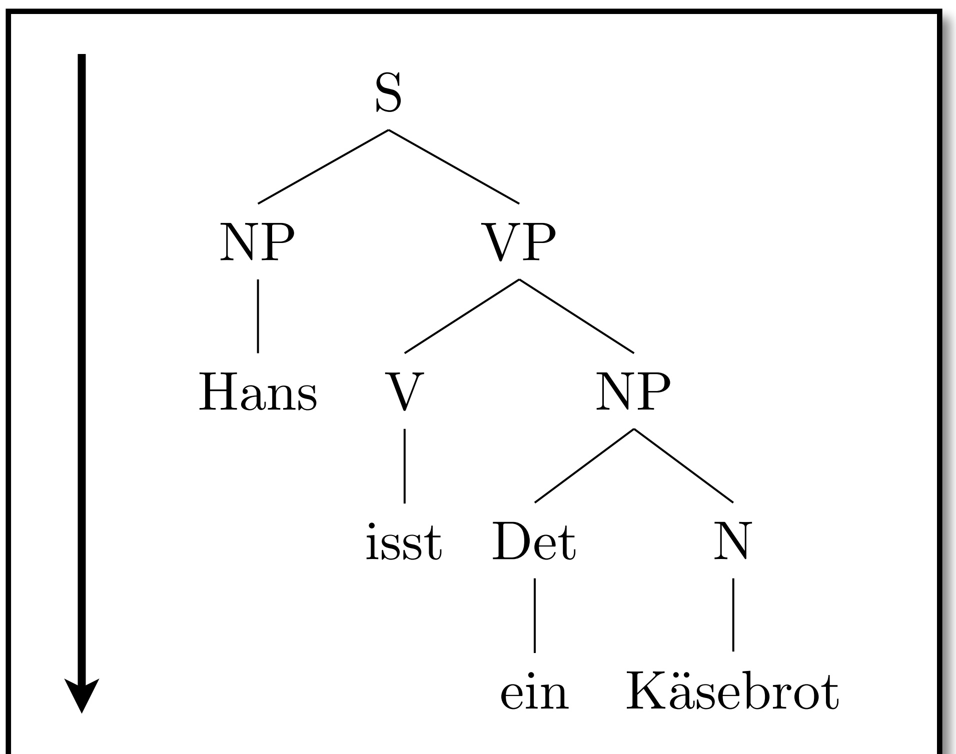
Recursive-Descent-Parsing

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



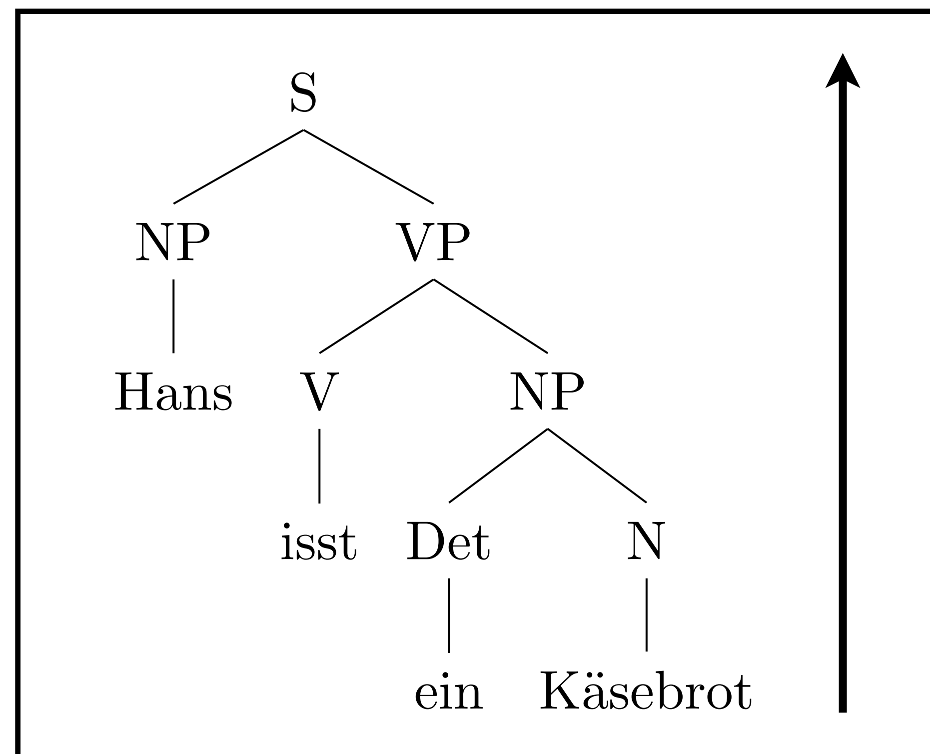
Top-Down vs. Bottom-Up

- 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		

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		

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.

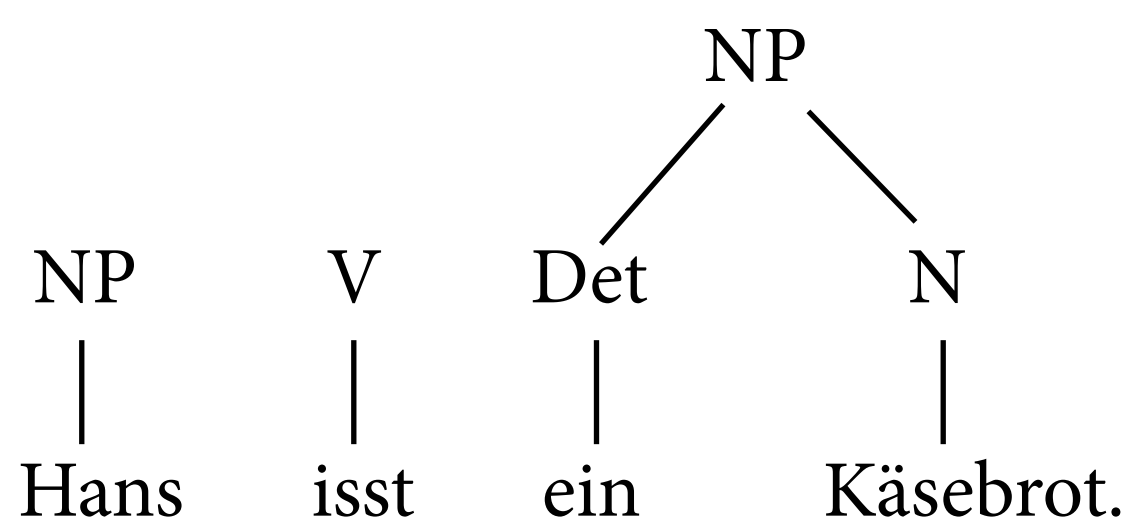
Bottom-Up-Parsing

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.

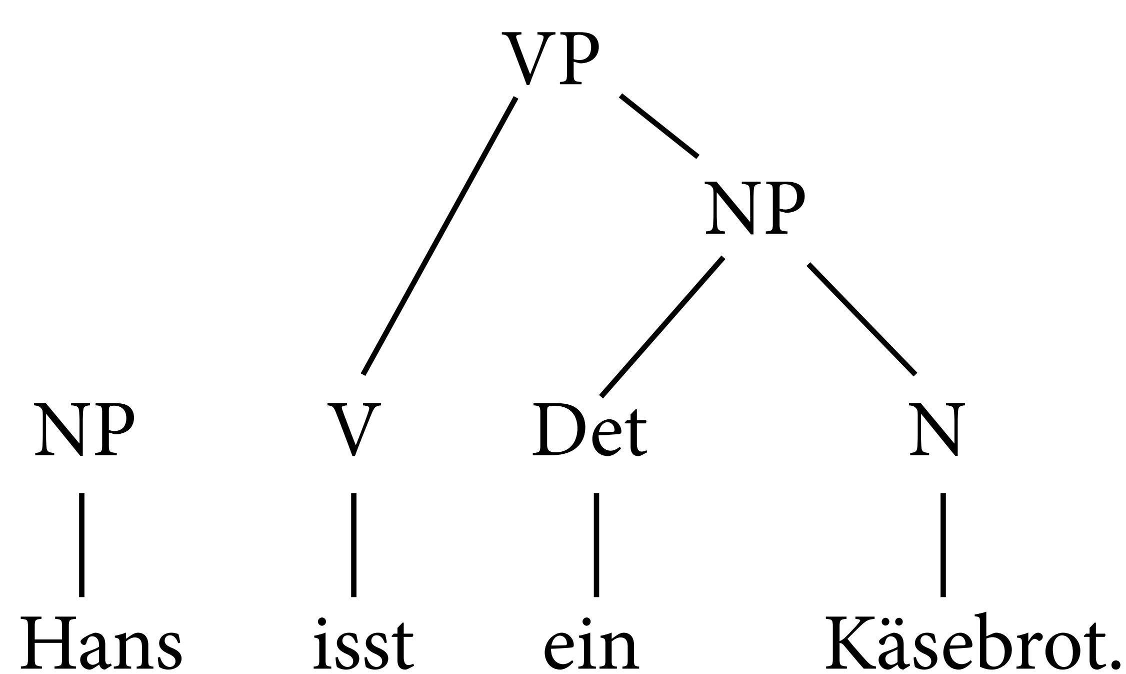
Bottom-Up-Parsing

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



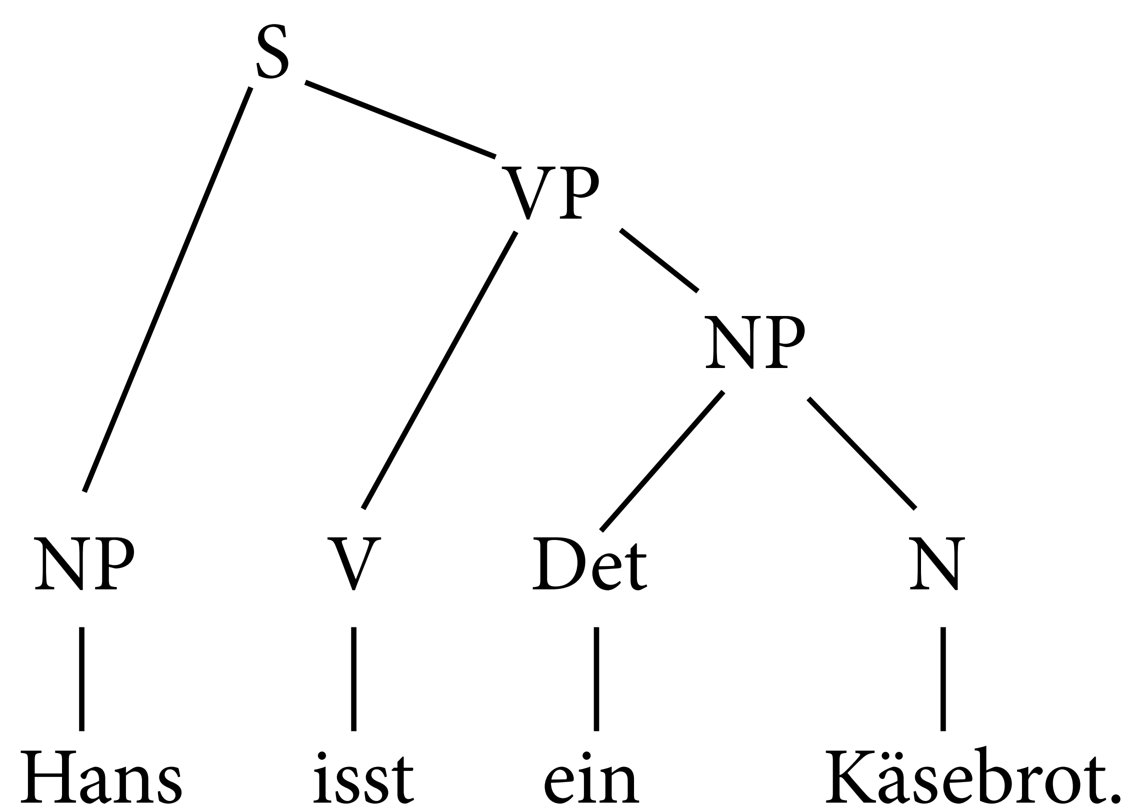
Bottom-Up-Parsing

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		



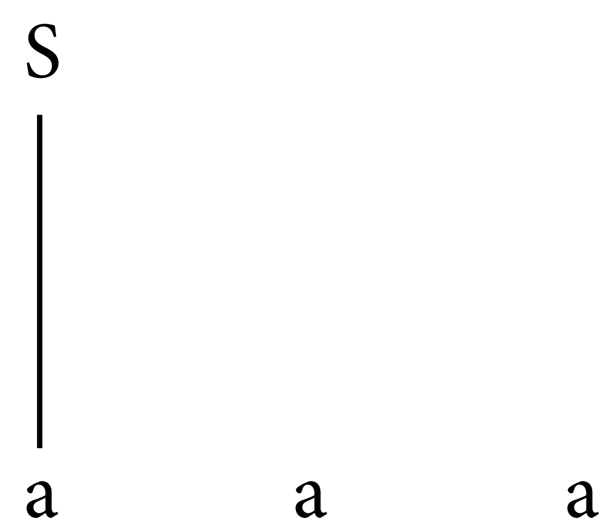
Bottom-Up-Parsing

S → S S	S → a

a a a

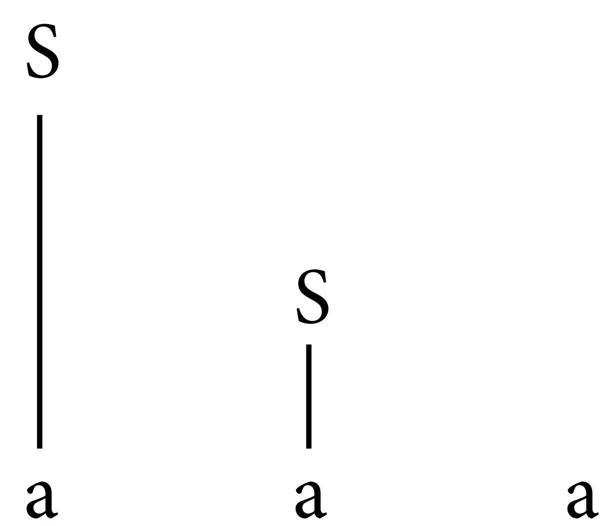
Bottom-Up-Parsing

S → S S   S → a



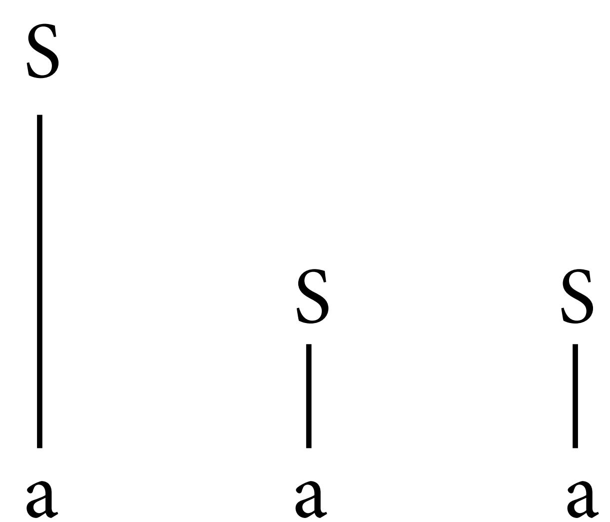
Bottom-Up-Parsing

S → S S　　　　S → a



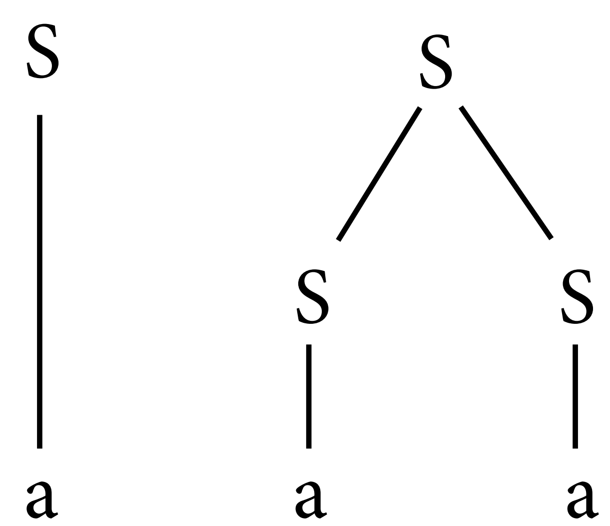
Bottom-Up-Parsing

S → S S   S → a



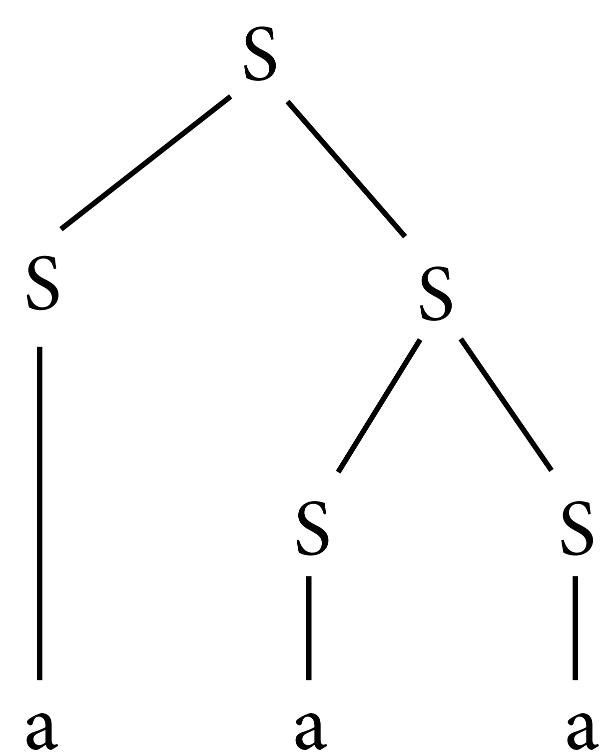
Bottom-Up-Parsing

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



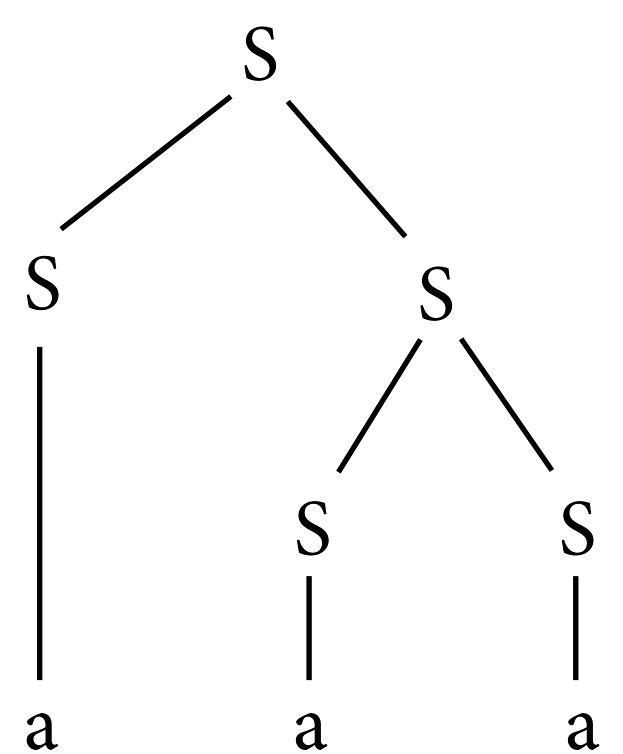
Bottom-Up-Parsing

S → S S　　　　S → a



Bottom-Up-Parsing

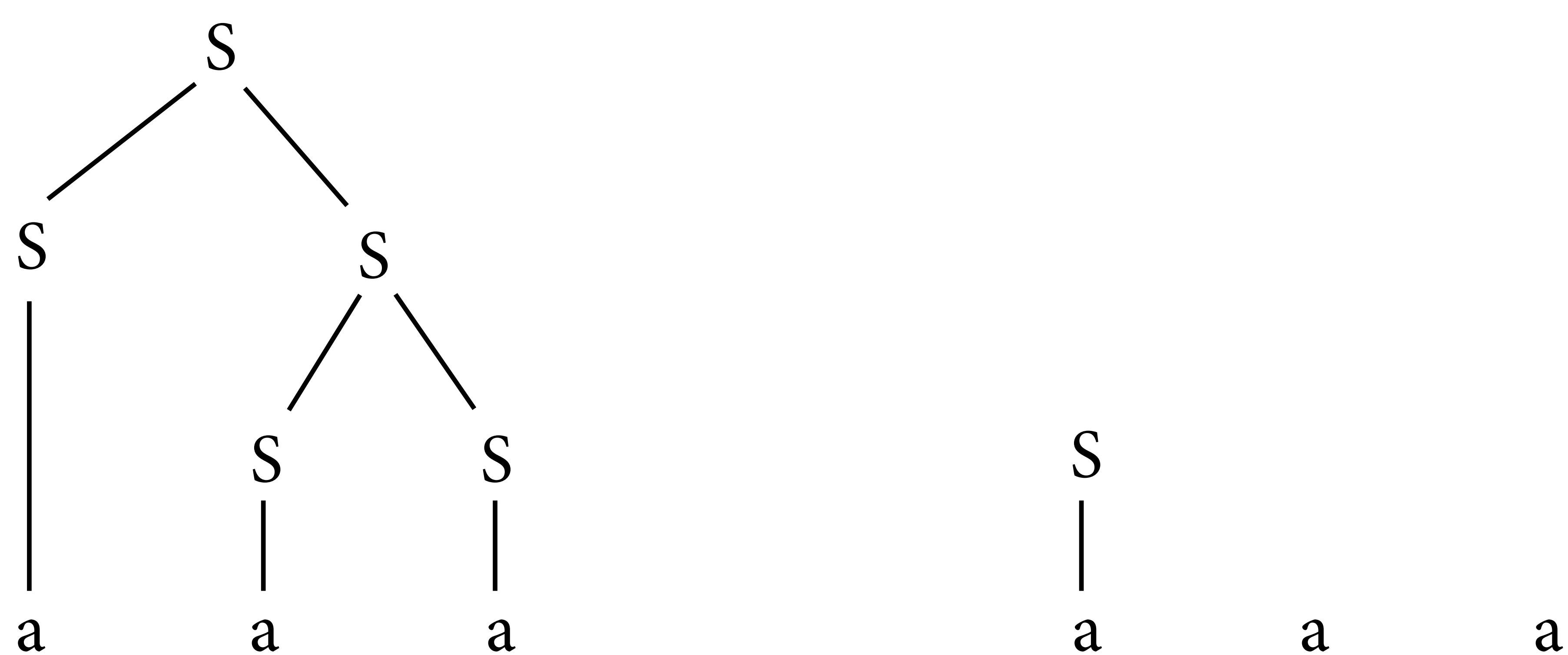
$S \rightarrow S\ S$ $\qquad\qquad$ $S \rightarrow a$



a a a

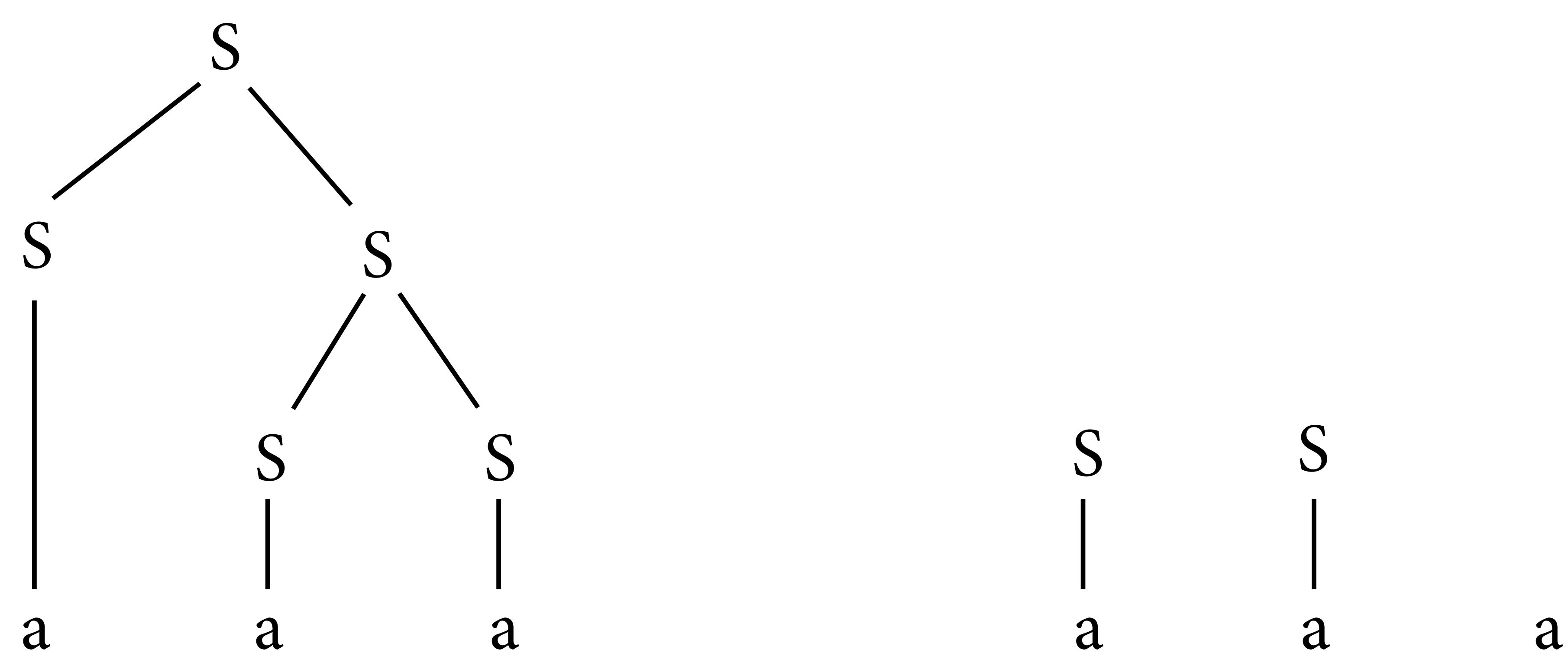
Bottom-Up-Parsing

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



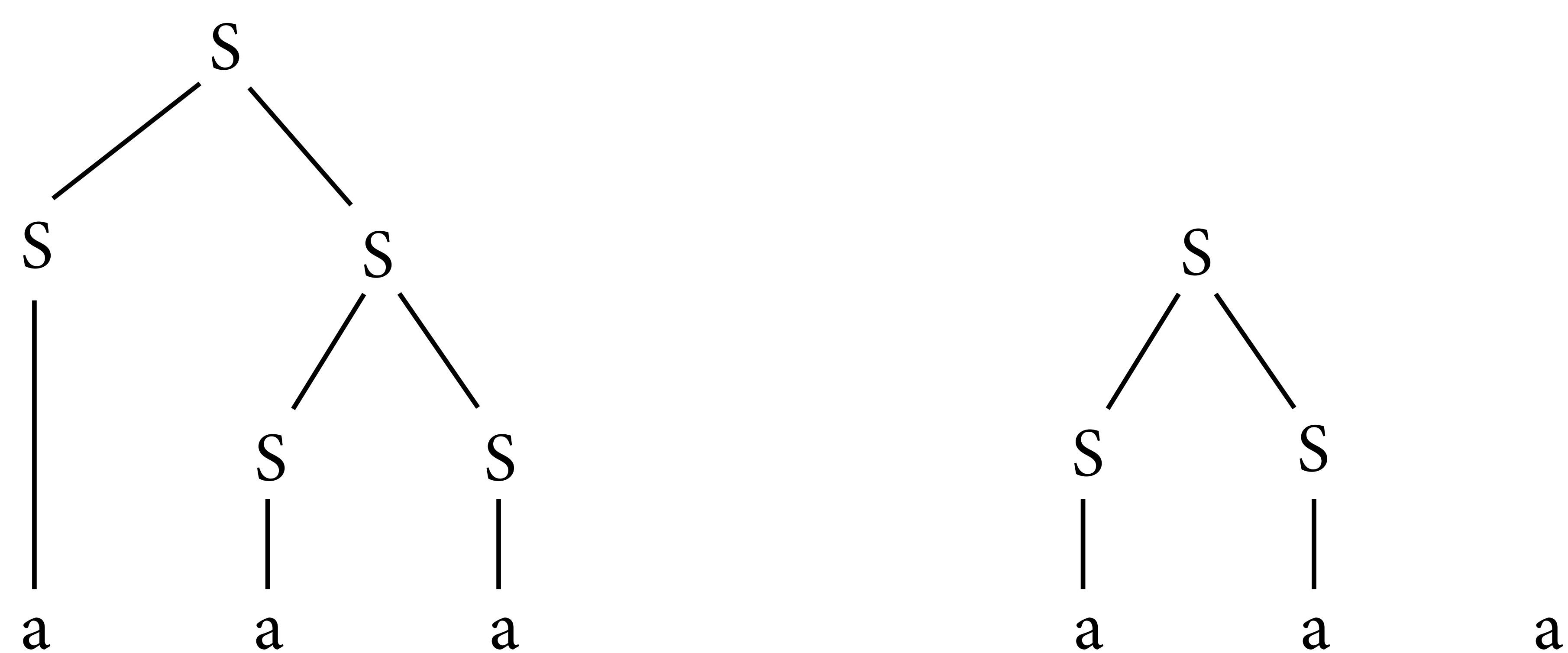
Bottom-Up-Parsing

S → S S S → a



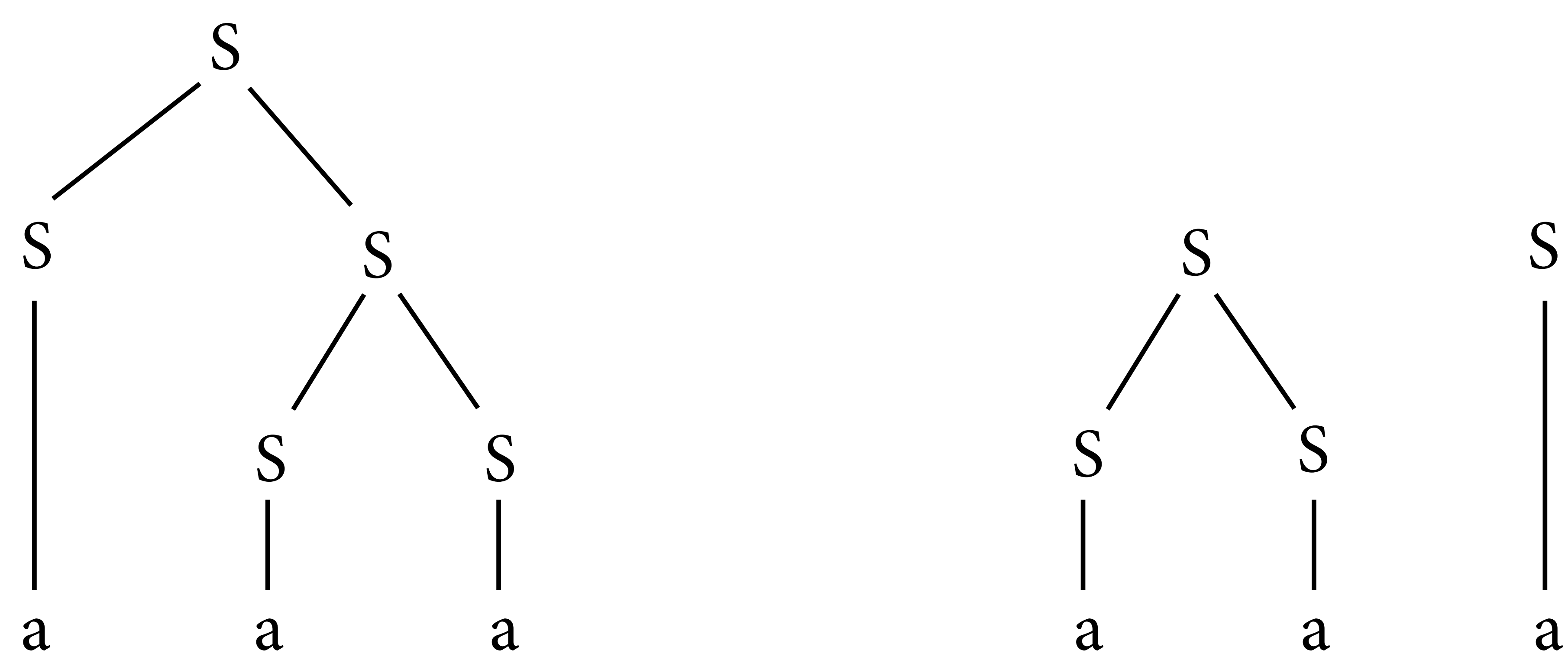
Bottom-Up-Parsing

S → S S S → a



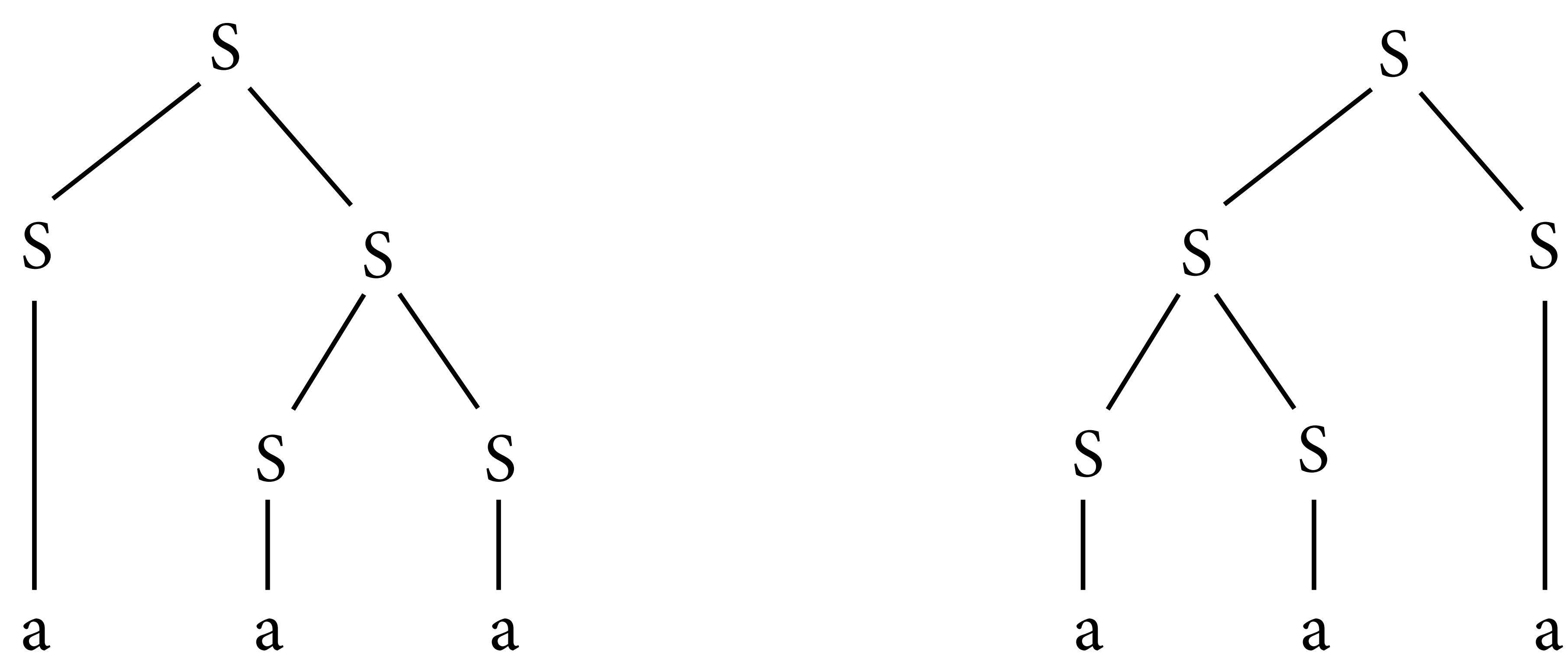
Bottom-Up-Parsing

S → S S   S → a



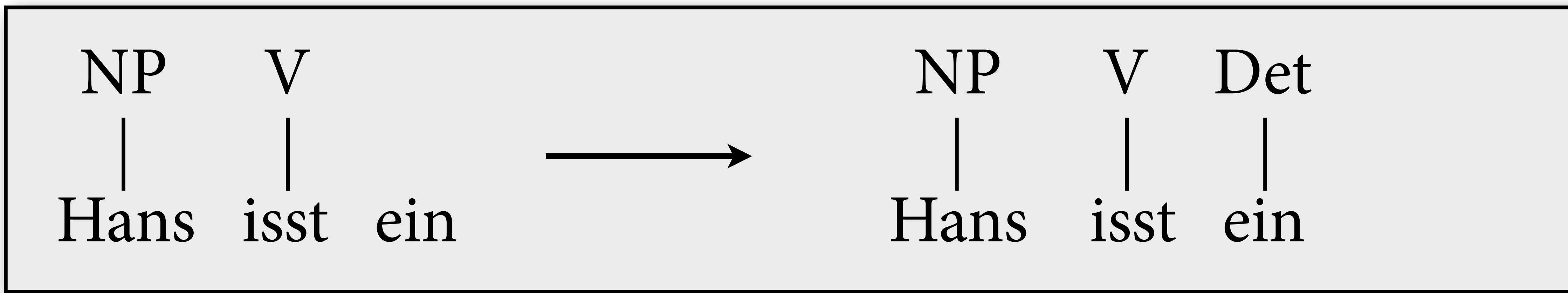
Bottom-Up-Parsing

S → S S S → a

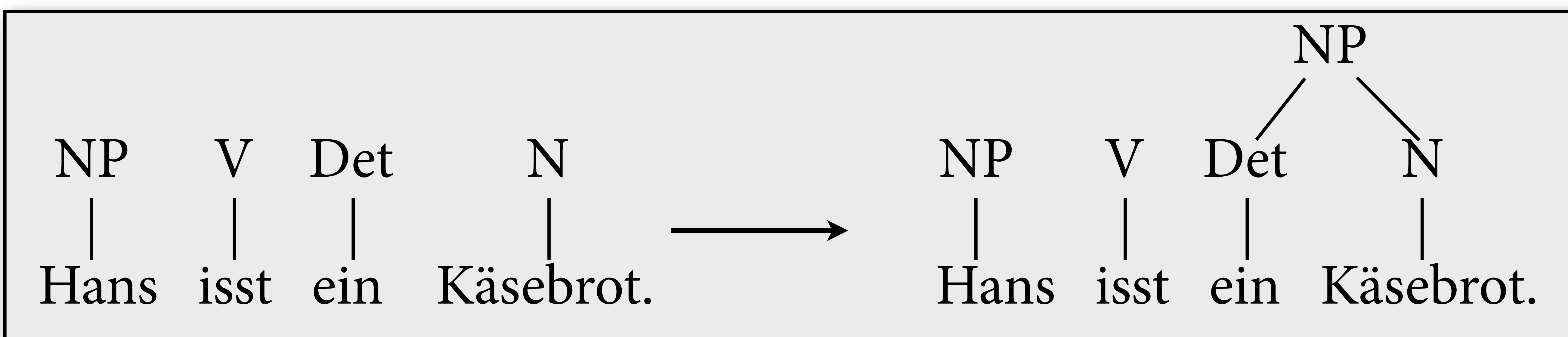


Shift-Reduce-Parsing

Shift: nächstes Terminalsymbol lesen



Reduce: Regel auf letzte offene Symbole anwenden

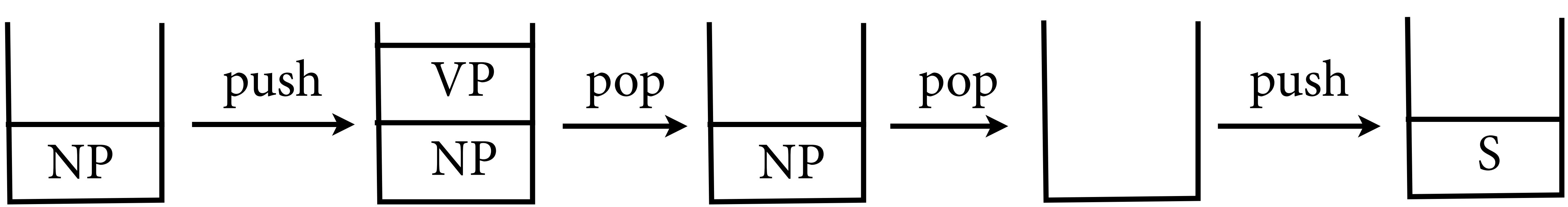


Shift-Reduce-Parsing

- 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.

$(\varepsilon$, Hans isst ein K.)

Shift-Reduce: Beispiel

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

NP
|
Hans isst ein Käsebrot.

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

Shift-Reduce: Beispiel

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

NP
|
Hans isst ein Käsebrot.

$(\varepsilon$, Hans isst ein K.$)$ $\rightarrow$ (Hans, isst ein K.) $\rightarrow$ (NP, isst ein K.)

Shift-Reduce: Beispiel

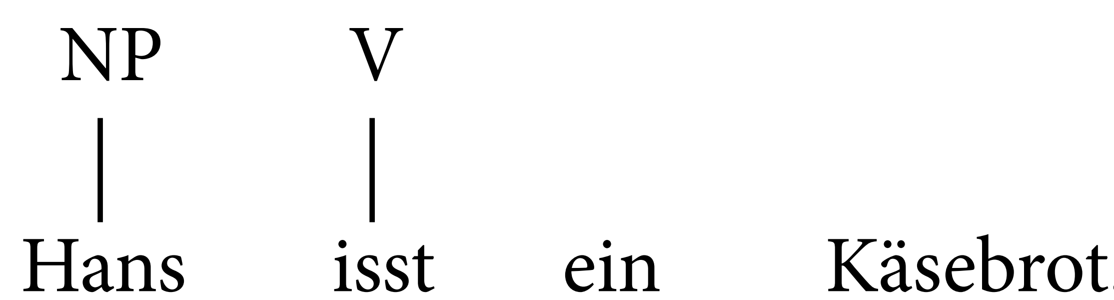
Shift: $(s, a \cdot w) \rightarrow (s \cdot a, w)$
Reduce: $(s \cdot w', w) \rightarrow (s \cdot A, w)$

NP V
Hans isst ein Käsebrot.

$(\varepsilon$, Hans isst ein K.$)$ → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.)

Shift-Reduce: Beispiel

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



$(\varepsilon$, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.)

Shift-Reduce: Beispiel

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

NP V Det
Hans isst ein Käsebrot.

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

NP V Det

Hans isst ein Käsebrot.

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

NP V Det N

Hans isst ein Käsebrot.

(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

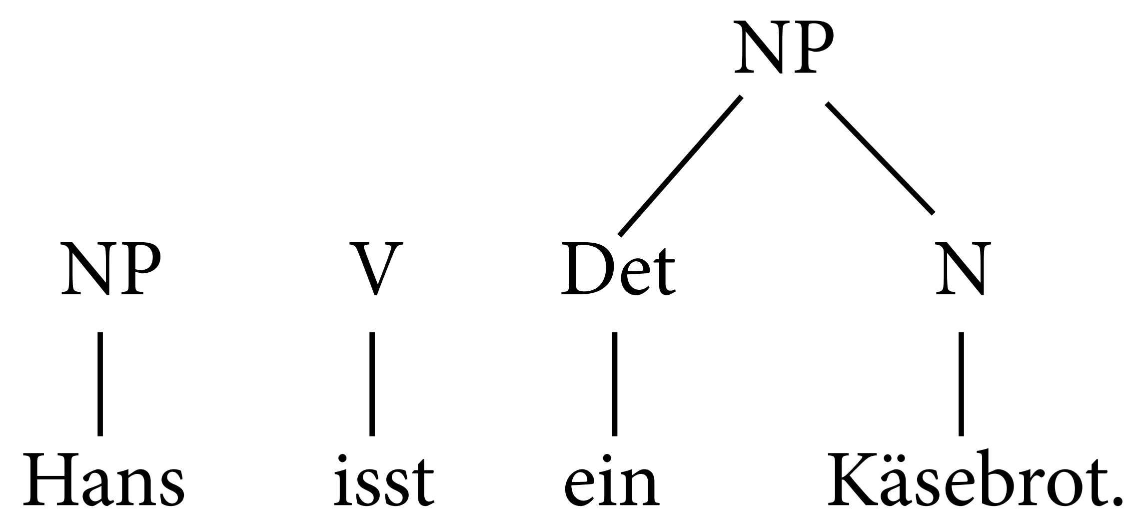
Shift: $(s, a \cdot w) \rightarrow (s \cdot a, w)$
Reduce: $(s \cdot w', w) \rightarrow (s \cdot A, w)$

NP	V	Det	N
Hans	isst	ein	Käsebrot.

(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

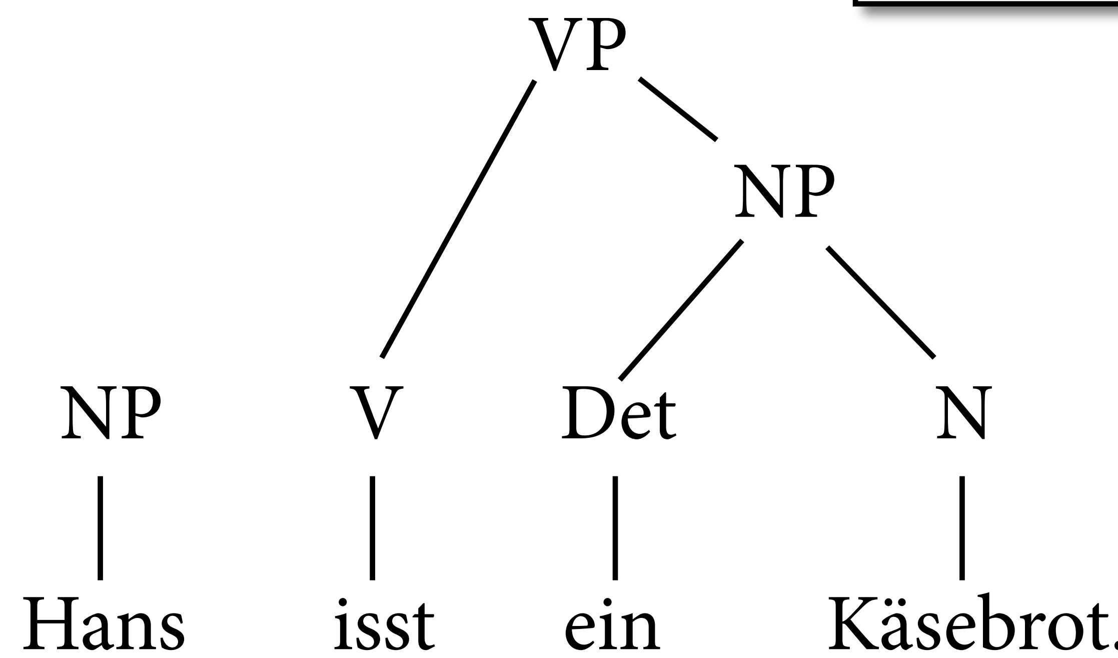
Shift: $(s, a \cdot w) \rightarrow (s \cdot a, w)$
Reduce: $(s \cdot w', w) \rightarrow (s \cdot A, w)$



(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

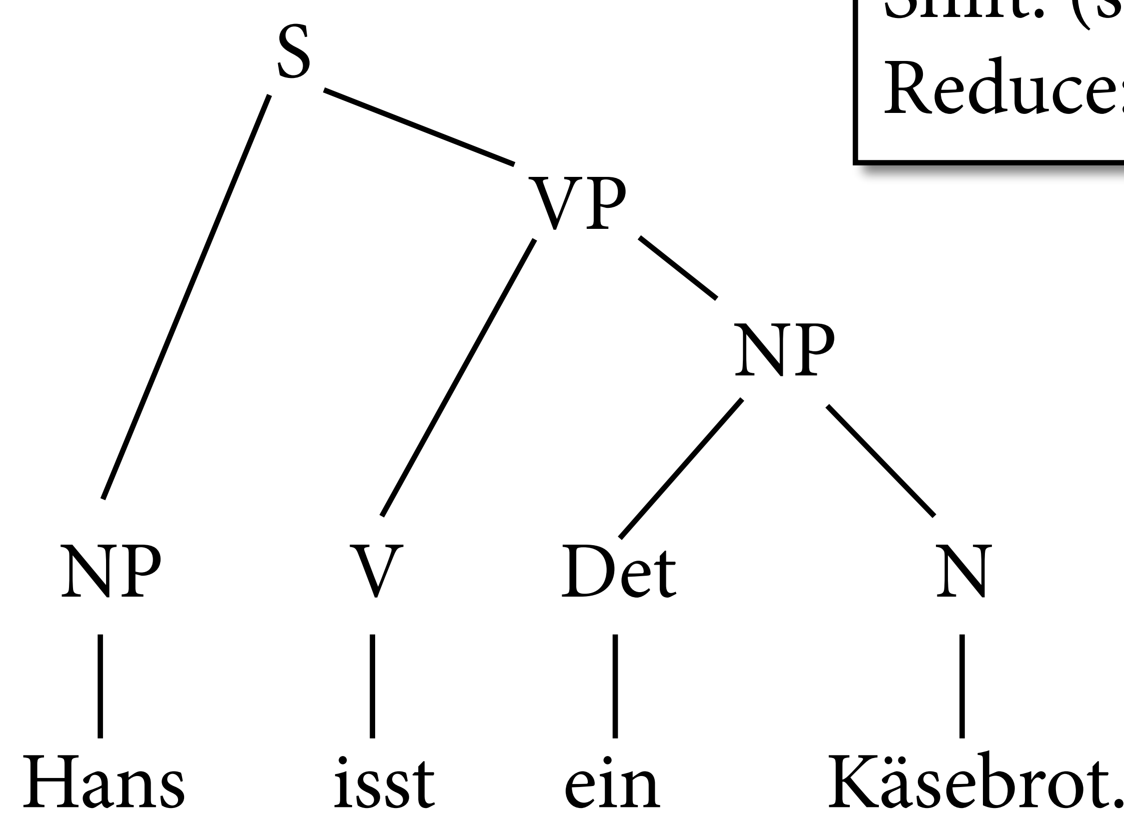
Shift: $(s, a \cdot w) \rightarrow (s \cdot a, w)$
Reduce: $(s \cdot w', w) \rightarrow (s \cdot A, w)$



(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

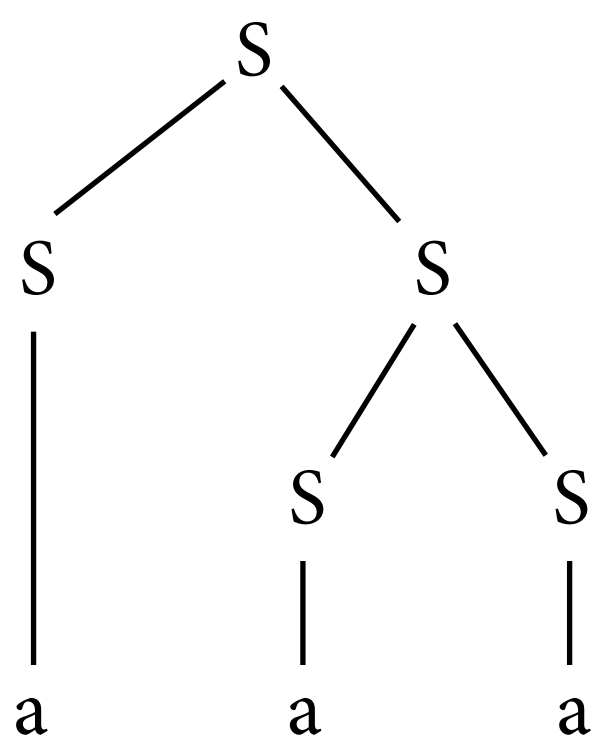
Shift: (s, a · w) → (s · a, w)
Reduce: (s · w', w) → (s · A, w)

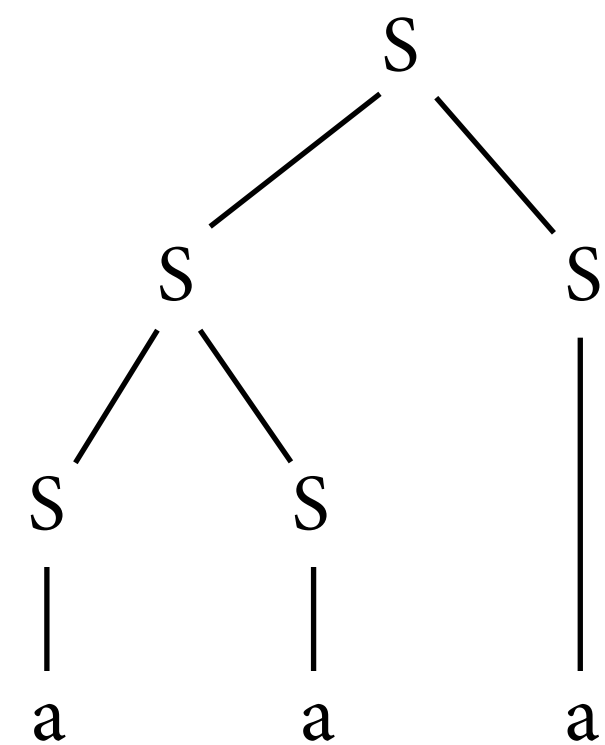


(ε, Hans isst ein K.) → (Hans, isst ein K.) → (NP, isst ein K.)
→ (NP isst, ein K.) → (NP V, ein K.) → (NP V ein, K.)
→ (NP V Det, K.) → (NP V Det K., ε) → (NP V Det N, ε)
→ (NP V NP, ε) → (NP VP, ε) → (S, ε)

Shift-Reduce: Beispiel

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





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

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

Shift-Reduce: Korrektheit

- Zeige mit Induktion über Anzahl der S/R-Schritte:
  - 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.

Shift-Reduce: Vollständigkeit

- Zeige mit Induktion über Länge der Ableitung:
  - wenn kfG-Ableitung $A \Rightarrow^* w$ existiert mit $w \in T^*$,
  - dann gibt es SR-Berechnung $(\varepsilon, w) \to^* (A, \varepsilon)$
- 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 $(\varepsilon, w) \to^* (S, \varepsilon)$ (s.o.)
  - also behauptet SR-Erkenner, dass $w \in L(G)$ gilt.

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

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

$$b \quad b \quad b \quad c$$

$$\to^* \quad C \quad C \quad C \quad T \quad \to^* \quad T \ ✘$$

Shift-Reduce: Ein Problem

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

$$
\begin{array}{lllllll}
 & b & b & b & c & & \\
\to^* & C & C & C & T & \to^* & T \ ✗ \\
\to^* & C & C & B & T \ ✗ & &
\end{array}
$$

Shift-Reduce: Ein Problem

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

	b	b	b	c		
$\to^*$	C	C	C	T	$\to^*$	T ✗
$\to^*$	C	C	B	T ✗		
$\to^*$	C	B	C	T	$\to^*$	C B T ✗

Shift-Reduce: Ein Problem

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

	b	b	b	c		
$\to^*$	C	C	C	T	$\to^*$	T ✗
$\to^*$	C	C	B	T ✗		
$\to^*$	C	B	C	T	$\to^*$	C B T ✗
$\to^*$	C	B	B	T ✗		etc.

Shift-Reduce: Ein Problem

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

	b	b	b	c		
→*	C	C	C	T	→*	T
→*	C	C	B	T		
→*	C	B	C	T	→*	C B T
→*	C	B	B	T		
						etc.

Shift-Reduce: Ein Problem

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

	b	b	b	c		
$\to^*$	C	C	C	T	$\to^*$	T
$\to^*$	C	C	B	T		
$\to^*$	C	B	C	T	$\to^*$	C B T
$\to^*$	C	B	B	T		
						etc.

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

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)

Beispiel

- Problem: Liste von Zahlen auf Sortiertheit testen.
  - gegeben Liste L von ints der Länge n
  - gibt es Indizes $1 \leq i < j \leq n$ mit $L_i > L_j$?
- Wir schauen uns zwei Algorithmen für dieses Problem an.

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100		
1000		
10000		
100.000		
1.000.000		

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	
1000		
10000		
100.000		
1.000.000		

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000		
100.000		
1.000.000		

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000	4.5 sec	
100.000		
1.000.000		

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	
1.000.000		

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		179 ms

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		179 ms

$\approx n \cdot 120$ ns

Laufzeitvergleich

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Laufzeit

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		179 ms

$\approx n^2 \cdot 45$ ns

$\approx n \cdot 120$ ns

Analyse

- Wichtige Eckdaten der Laufzeit:
  - Eingabegröße len(L): Länge der Liste.
  - Im worst-case (alles sortiert) wird jede Schleife len(L)-mal durchlaufen.
  - Details der Zeit pro Schleifendurchlauf sind uns egal.

- Uns reicht es zu sagen, dass Laufzeit *linear* bzw. *quadratisch* 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 $f = O(g)$.

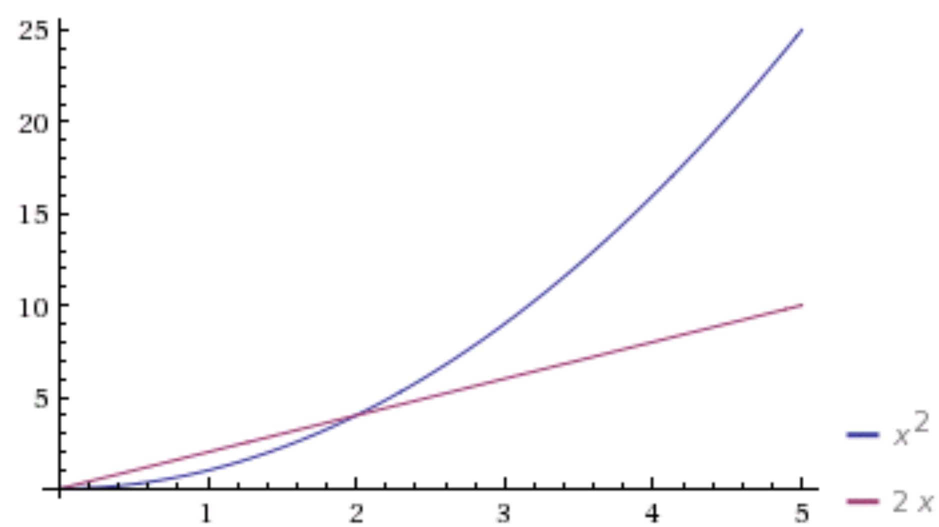
Zurück zu den Listen

$f = O(g)$ 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
  $c \cdot n \leq c' \cdot n^2$

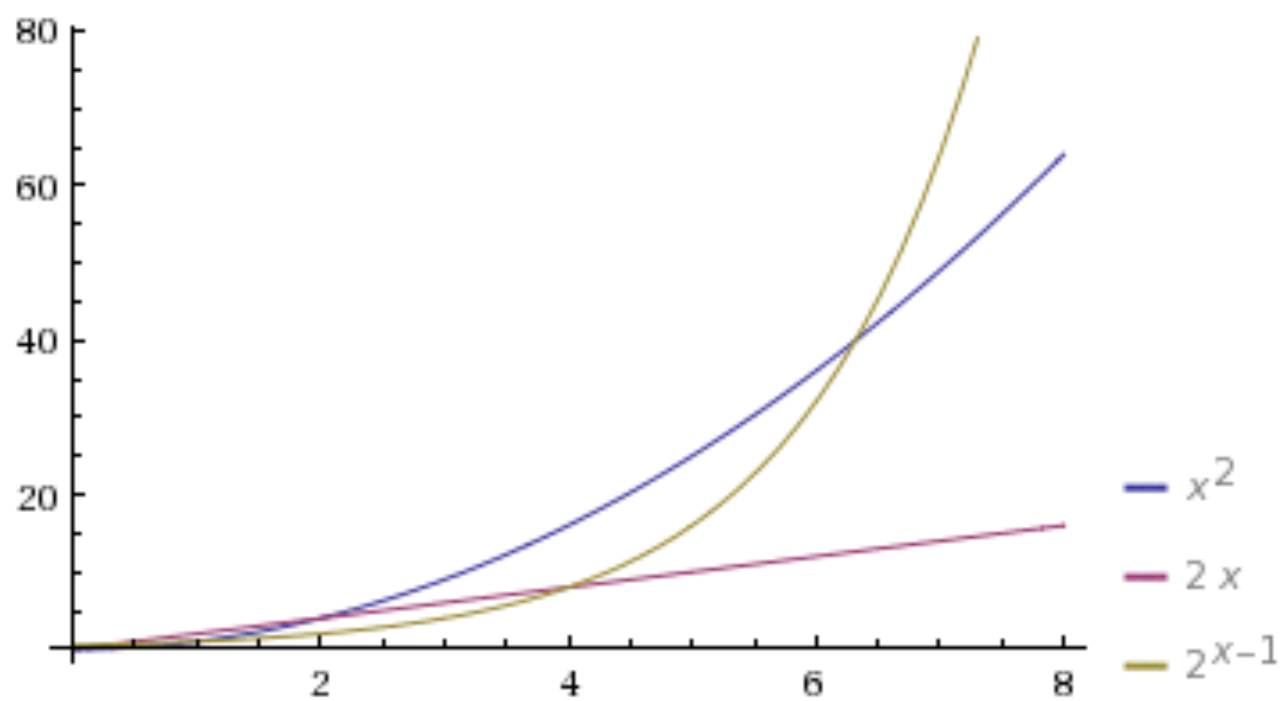


- Für große n also kleine Polynome schneller:
  - $O(n)$ linear < $O(n^2)$ quadratisch
    (auch für $n + 5$, $100 \cdot n - 27$ usw.)
  - $O(n^2)$ quadratisch < $O(n^3)$ kubisch
  - etc.

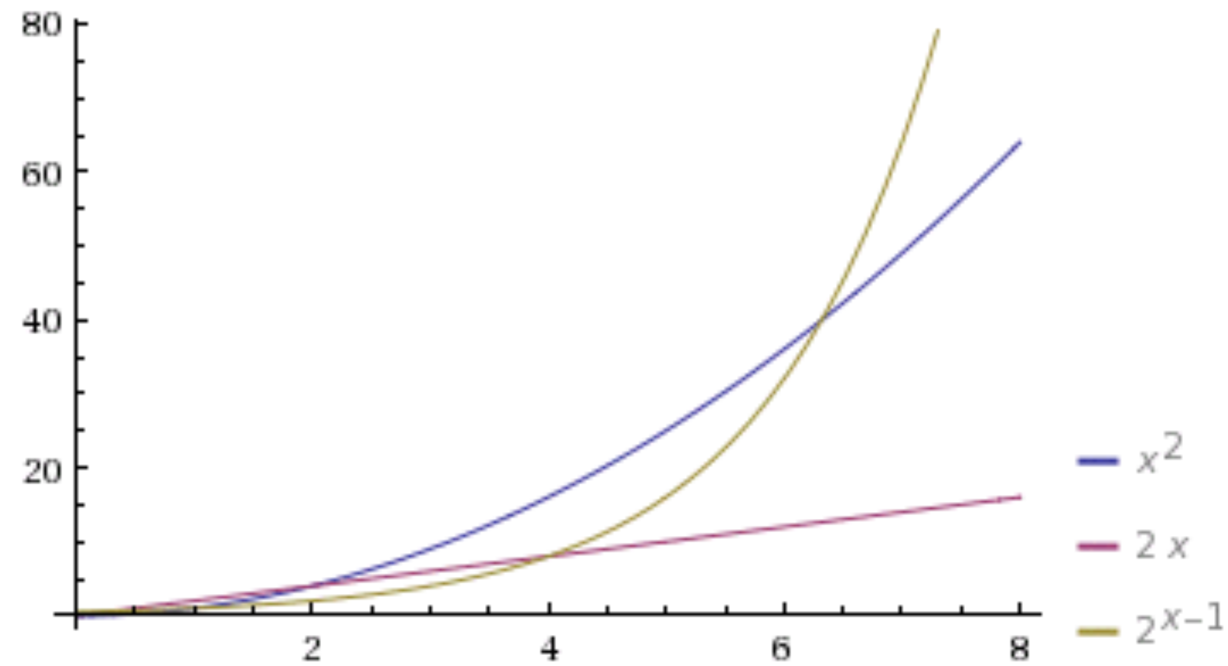
Exponentielle Laufzeit

- 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?

Probleme: Übersicht

- Parsertypen haben komplementäre Probleme:

	top-down	bottom-up
Regeln raten	$A \rightarrow w_1$ vs. $A \rightarrow w_2$	$A \rightarrow w$ vs. $B \rightarrow w$
Zerlegung raten	String aufteilen	Shift vs. Reduce entscheiden

- Allgemein nicht zu vermeiden (Ambiguität).

Zusammenfassung

- Grundlegende Parsingstrategien:
  - top-down — z.B. Recursive Descent
  - bottom-up — z.B. Shift-Reduce
- Backtracking-basierte Parser (RD, SR) können exponentielle Laufzeit brauchen.
  - weil Grammatiken für natürliche Sprachen Ambiguitäten erlauben müssen
  - in der Praxis viel zu langsam
- Nächstes Mal machen wir es schneller.