

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2018 - Thomas Schwentick

Teil B: Kontextfreie Sprachen

11: Wortproblem und Syntaxanalyse

Version von: 24. Mai 2018 (14:05)

# Wortproblem und Syntaxanalyse für kontextfreie Sprachen


- Phasen eines Compilers (schematisch):
  - Lexikalische Analyse
  - Syntaktische Analyse
  - Semantische Analyse
  - Zwischencode-Erzeugung
  - Zwischencode-Optimierung
  - Code-Erzeugung
- Bei der **Syntaxanalyse** wird die Struktur eines Programmes überprüft und in Form eines Syntax-Baumes repräsentiert
- Sie wird vom **Parser** durchgeführt
- Hierbei spielen kontextfreie Sprachen eine wichtige Rolle

- Die Syntaxanalyse liefert die Information, ob das gegebene Programm  $w$  syntaktisch korrekt ist
- Dies entspricht folgendem algorithmischen Problem:

Definition (Wortproblem für kontextfreie Grammatiken)

**Gegeben:** Wort  $w \in \Sigma^*$ , Grammatik  $G$

**Frage:** Ist  $w \in L(G)$ ?

- Wenn das Programm syntaktisch korrekt ist, soll die Syntaxanalyse auch einen Ableitungsbaum liefern, da dieser das Rückgrat für die Codeerzeugung darstellt
- Außerdem sollte bei syntaktisch inkorrekten Programmen  $w$  eine Begründung geliefert werden, warum  $w \notin L(G)$   Das betrachten wir nicht

Definition (Syntaxanalyse-Problem für kfr. Grammatiken)

**Gegeben:** Wort  $w \in \Sigma^*$ , Grammatik  $G$

**Gesucht:** Falls  $w \in L(G)$ : Ableitungsbaum

# Übersicht

- Das Wortproblem für reguläre Sprachen kann für jede feste reguläre Sprache in linearer Zeit gelöst werden
  - durch Auswertung eines DFA
- Auch deterministische Kellerautomaten können in linearer Zeit ausgewertet werden
  - aber leider gibt es nicht für jede kontextfreie Sprache einen deterministischen Kellerautomaten
- Wir werden in diesem Kapitel sehen:
- Die naive Auswertung von PDAs mit Backtracking kann zu exponentieller Laufzeit führen kann
- Es gibt einen Algorithmus, der das Syntaxanalyse-Problem für kontextfreie Grammatiken in polynomieller Zeit löst:
  - Der **CYK-Algorithmus** basiert auf dynamischer Programmierung und hat Laufzeit  $\mathcal{O}(|G||w|^3)$
- Da kubische Laufzeit für viele Zwecke nicht akzeptabel ist, betrachten wir danach zwei eingeschränkte Grammatiktypen, die eine Syntaxanalyse in *linearer Zeit* erlauben:
  - LL(1)-Grammatiken: recht einfach zu definieren
  - LR(1)-Grammatiken: komplizierter zu definieren, aber gleichmächtig zu DPDAs

# Syntaxanalyse: Herangehensweisen

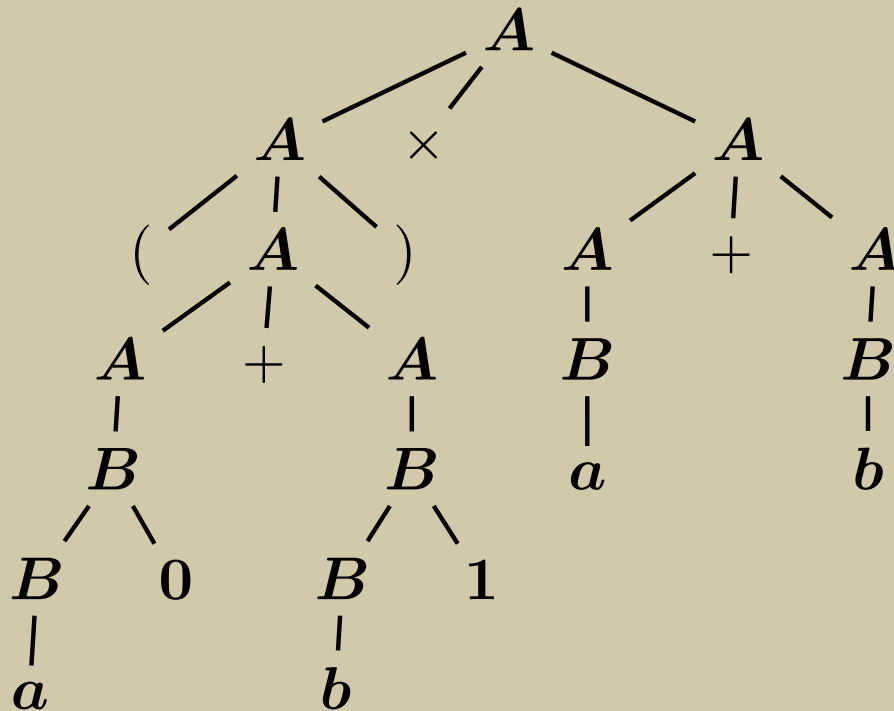
- Wir betrachten zwei Arten von Algorithmen für das Wortproblem für kontextfreie Grammatiken
- Algorithmen, die versuchen beim Lesen des Eingabestrings von links nach rechts eine Ableitung zu erzeugen
  - Backtracking (Linksableitung, top-down)
  - $LL(k)$  (Linksableitung, top-down)
  - $LR(k)$  (Rechtsableitung, bottom-up)
- Algorithmen, die den Eingabestring „als Ganzes“ analysieren
  - CYK-Algorithmus
- Bevor wir uns dem Backtracking-Algorithmus zuwenden, werfen wir zunächst einen Blick auf den Ansatz der Top-down Syntaxanalyse
  - Bottom-up Syntaxanalyse werden wir gegen Ende des Kapitels betrachten

# Top-down Syntaxanalyse (1/3)

## Beispiel-Grammatik

$A \rightarrow B \mid A + A \mid A \times A \mid (A)$   
 $B \rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1$

## Beispiel-Ableitungsbaum



- Bei der Top-down Syntaxanalyse wird der Ableitungsbaum von oben nach unten und (üblicherweise) von links nach rechts konstruiert
- Dieses Vorgehen ergibt eine Linksableitung

## Beispiel-Ableitung: Top-down

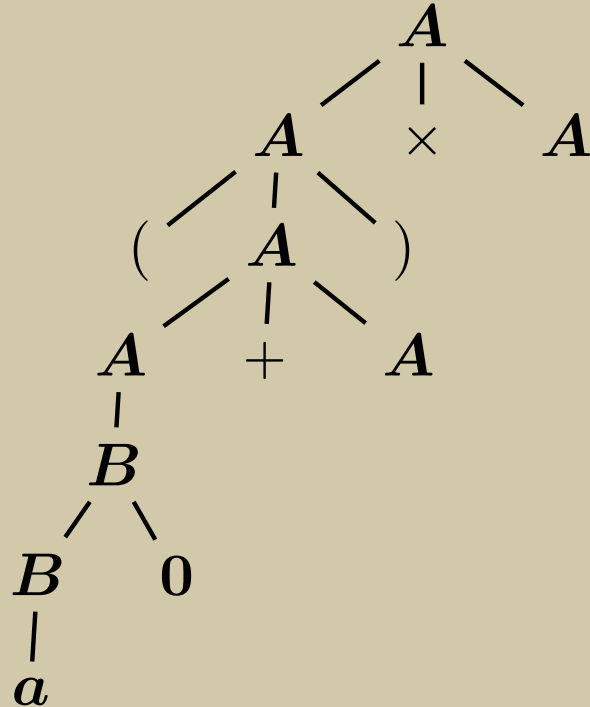
$A \Rightarrow A \times A$   
 $\Rightarrow (A) \times A$   
 $\Rightarrow (A + A) \times A$   
 $\Rightarrow (B + A) \times A$   
 $\Rightarrow (B0 + A) \times A$   
 $\Rightarrow (a0 + A) \times A$   
 $\Rightarrow (a0 + B) \times A$   
 $\Rightarrow (a0 + B1) \times A$   
 $\Rightarrow (a0 + b1) \times A$   
 $\Rightarrow (a0 + b1) \times A + A$   
 $\Rightarrow (a0 + b1) \times B + A$   
 $\Rightarrow (a0 + b1) \times a + A$   
 $\Rightarrow (a0 + b1) \times a + B$   
 $\Rightarrow (a0 + b1) \times a + b$

## Top-down Syntaxanalyse (2/3)

### Beispiel

$A \rightarrow B \mid A + A \mid A \times A \mid (A)$

$B \rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1$



- Eingabe:  $(a0 + b1) \times a + b$

### Beispiel

- Der angegebene unvollständige Ableitungsbaum stellt eine Zwischensituation bei der Erzeugung einer Linksableitung für  $(a0 + b1) \times a + b$  dar

- Der linke Teil der Blätter des Baumes stimmt mit dem Anfang der Eingabe überein:  $(a0 +$

- Der unvollständige Baum entspricht der Satzform:  $(a0 + A) \times A$

- Als nächstes muss also die Variable  $A$  ersetzt werden

- Der Rest der Satzform ist:  $) \times A$

## Top-down Syntaxanalyse (3/3)

- Die allgemeine Situation bei der Bestimmung des nächsten Schrittes einer Linksableitung für einen Eingabestring  $w$  ist wie folgt:

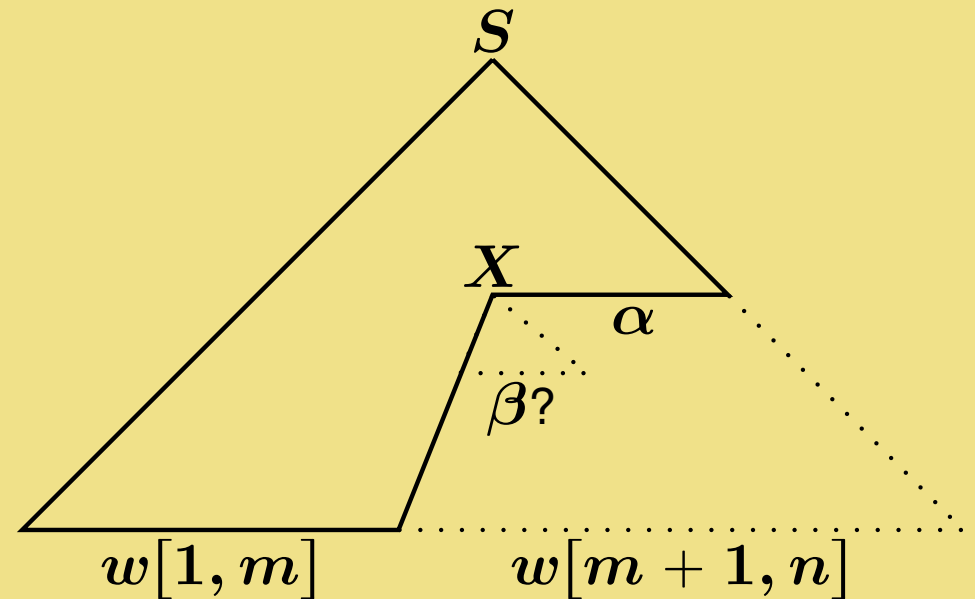
- Es ist schon eine Satzform abgeleitet
- Ihre erste Variable bezeichnen wir mit  $X$
- Die davor stehenden Zeichen aus dem Alphabet  $\Sigma$  müssen mit dem Anfang der Eingabe übereinstimmen

☞  $m \stackrel{\text{def}}{=} \text{Anzahl dieser Zeichen}$

- Den Rest der Satzform bezeichnen wir mit  $\alpha$

- Wir haben also:  $S \Rightarrow_l^* w[1, m]X\alpha$
- Damit insgesamt  $w$  erzeugt wird, muss also aus  $X\alpha$  der restliche String  $w[m+1, n]$  erzeugt werden

- Der nächste Ableitungsschritt ist gesucht:  
 $w[1, m]X\alpha \Rightarrow_l w[1, m]\beta\alpha$



## 11.1 Algorithmen für beliebige kontextfreie Sprachen

### ▷ 11.1.1 Backtracking

#### 11.1.2 Der CYK-Algorithmus

## 11.2 Effiziente Syntaxanalyse



## Backtracking-Algorithmus: Idee

- Der Backtrackingalgorithmus versucht systematisch eine Linksableitung zu erzeugen
- Er probiert dazu nach Ableitung von  $S \Rightarrow_i^* w[1, m] X \alpha$  alle Regeln der Form  $X \rightarrow \beta$  nacheinander aus
- Wenn die entstehende Satzform nicht zur Eingabe passt oder zu lang wird, wählt er beim nächsten Mal die nächste Regel
- Dabei kann es nötig sein, Schritte wieder rückgängig zu machen
- Die Laufzeit des Backtracking-Algorithmus kann exponentiell werden, wie das folgende Beispiel zeigt

# Backtracking-Algorithmus: Beispiel (1/2)

## • Backtracking-Algorithmus:

- Versuche, Linksableitung zu erzeugen
- Wähle immer jeweils die erste passende Regel
- Falls nicht erfolgreich:  
\* zurücksetzen und nächste Regel wählen

## Beispiel

### • Grammatik:

$$S \rightarrow aA0 \quad (1)$$

$$S \rightarrow aB1 \quad (2)$$

$$A \rightarrow aA0 \quad (3)$$

$$A \rightarrow aB1 \quad (4)$$

$$A \rightarrow c \quad (5)$$

$$B \rightarrow aA0 \quad (6)$$

$$B \rightarrow aB1 \quad (7)$$

$$B \rightarrow c \quad (8)$$

### • Eingabe: aaac111

## Lauf des Backtracking-Algorithmus

Eingabe	Satzform	Regeln	Letzte Aktion
aaac111	S		
a aac111	aA0	1	Regel 1
a a ac111	a A0	1	Vergleich: ok
a a a ac111	a aA00	1 3	Regel 3
aa a ac111	aa A00	1 3	Vergleich: ok
aa a a ac111	aa aA000	1 3 3	Regel 3
aaa a ac111	aaa A000	1 3 3	Vergleich: ok
aaa a a ac111	aaa aA0000	1 3 3 3	Regel 3
aaac a ac111	aaaa A0000	1 3 3 3	Vergleich: nicht ok
aaa a ac111	aaa A000	1 3 3 (3)	zurück
aaa a a ac111	aaa aB1000	1 3 3 4	Regel 4
aaac a a ac111	aaaa B1000	1 3 3 4	Vergleich: nicht ok
aaa a a ac111	aaa A000	1 3 3 (4)	zurück
aaa a c ac111	aaa c000	1 3 3 5	Regel 5
aaac a c ac111	aaac 000	1 3 3 5	Vergleich: ok
aaac1 a c ac111	aaac0 00	1 3 3 5	Vergleich: nicht ok
aaa a c ac111	aaa A000	1 3 3 (5)	zurück
aa a a ac111	aa A00	1 3 (3)	zurück
aa a a a ac111	aa aB100	1 3 4	Regel 4
aaa a a a ac111	aaa B100	1 3 4	Vergleich: ok
aaa a c a ac111	aaa aA0100	1 3 4 6	Regel 6
aaac a c a ac111	aaaa A0100	1 3 4 6	Vergleich: nicht ok

# Backtracking-Algorithmus: Beispiel (2/2)

## • Backtracking-Algorithmus:

- Versuche, Linksableitung zu erzeugen
- Wähle immer jeweils die erste passende Regel
- Falls nicht erfolgreich:  
\* zurücksetzen und nächste Regel wählen

## Beispiel

### • Grammatik:

- $S \rightarrow aA0$  (1)
- $S \rightarrow aB1$  (2)
- $A \rightarrow aA0$  (3)
- $A \rightarrow aB1$  (4)
- $A \rightarrow c$  (5)
- $B \rightarrow aA0$  (6)
- $B \rightarrow aB1$  (7)
- $B \rightarrow c$  (8)

### • Eingabe: aaac111

## Lauf des Backtracking-Algorithmus (Forts.)

Eingabe	Satzform	Regeln	Letzte Aktion
aaac 111	aaaa A0100	1 3 4 6	Vergleich: nicht ok
aaa c111	aaa B100	1 3 4 (6)	zurück
aaa c111	aaa aB1100	1 3 4 7	Regel 7
aaac 111	aaaa B1100	1 3 4 (7)	Vergleich: nicht ok
aaa c111	aaa B100	1 3 4 (7)	zurück
aaa c111	aaa c100	1 3 4 8	Regel 8
aaac 111	aaac 100	1 3 4 8	Vergleich: ok
aaac1 11	aaac1 00	1 3 4 8	Vergleich: ok
aaac11 1	aaac10 0	1 3 4 8	Vergleich: nicht ok
aaa c111	aaa B100	1 3 4 (8)	zurück
aa ac111	aa A00	1 3 (4)	zurück
aa ac111	aa c00	1 3 5	Regel 5
aaa c111	aac 00	1 3 5	Vergleich: nicht ok
aa ac111	aa A00	1 3 (5)	zurück
a aac111	a A0	1 (3)	zurück
a aac111	a aB10	1 4	Regel 4
...	...	...	...
aaac111	aaac111	2 7 7 8	Vergleich: ok

- Beobachtung: Bei Eingabe  $a^n c 1^n$  kommen alle  $n$ -stelligen Binärzahlen  $x$  in einer Satzform  $a^n c x$  vor  
➔ exponentiell viele Schritte

## 11.1 Algorithmen für beliebige kontextfreie Sprachen

11.1.1 Backtracking

▷ **11.1.2 Der CYK-Algorithmus**

11.2 Effiziente Syntaxanalyse

# Der CYK-Algorithmus

- Exponentieller Aufwand ist bei der Syntaxanalyse natürlich inakzeptabel
- Wir betrachten jetzt einen Algorithmus, der das Wortproblem für *beliebige* kontextfreie Grammatiken in polynomieller Zeit löst  
☞ für CNF in Zeit  $\mathcal{O}(|G||w|^3)$
- Der **CYK-Algorithmus** wurde von Cocke, Younger und Kasami (unabhängig voneinander) entwickelt  
☞ um 1965
  - „Richtig“ veröffentlicht wurde er nur von Younger  
[Younger 67]
- Der CYK-Algorithmus verwendet dynamische Programmierung
- Die hier betrachtete Variante nutzt aus, dass die Grammatik  $G$  für  $L$  in CNF ist, der Algorithmus lässt sich aber für kontextfreie Grammatiken, die nicht in CNF sind, anpassen

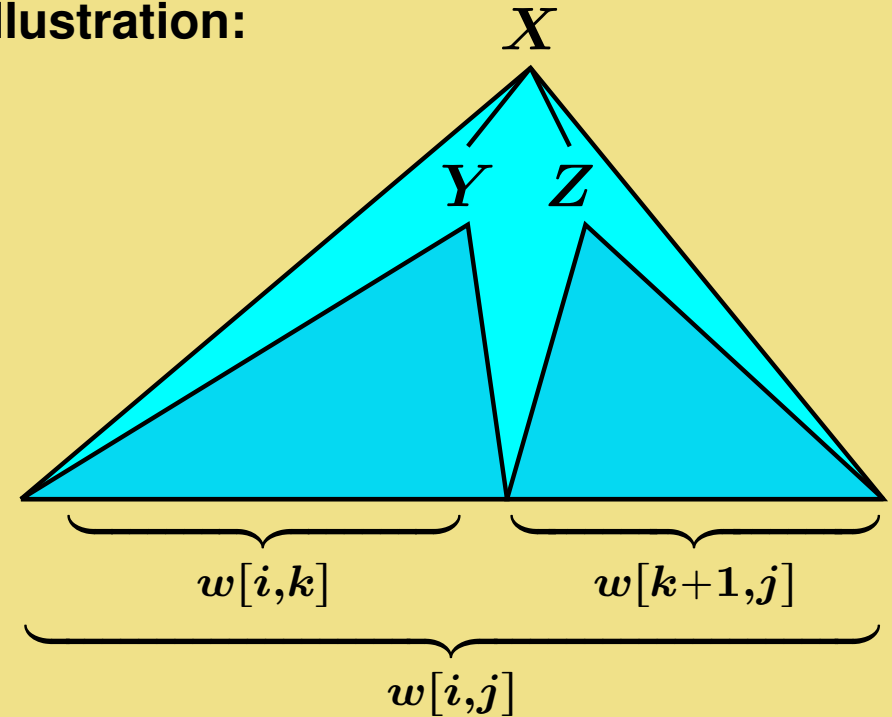
# Der CYK-Algorithmus: Grundidee

- Sei  $G$  in CNF gegeben und  $w$  ein String der Länge  $n$
- Zunächst wird das Problem in Teilprobleme zerlegt, die durch Parameter repräsentiert werden
  - ✎ Das Problem wird also „parametrisiert“

- Für jede Wahl von  $i, j \in \{1, \dots, n\}$  mit  $i \leq j$  sei
$$\underline{V_{i,j}} \stackrel{\text{def}}{=} \{X \in V \mid X \Rightarrow^* w[i, j]\}$$
  - ✎  $V_{ij}$  ist also die Menge aller Variablen, aus denen  $w[i, j]$  abgeleitet werden kann

- Klar:  $w \in L(G) \iff S \in V_{1,n}$
- Der CYK-Algorithmus berechnet die Mengen  $V_{i,j}$  **bottom-up**

## Illustration:



- Er nutzt aus, dass bei einer CNF-Grammatik für  $X \in V$ ,  $1 \leq i < j \leq n$  äquivalent sind:
  - $X \in V_{ij}$
  - es gibt  $Y, Z \in V$  und  $k \in \{i, \dots, j-1\}$  mit:
    - \*  $X \rightarrow YZ$  ist Regel von  $G$ ,
    - \*  $Y \in V_{i,k}$  und
    - \*  $Z \in V_{k+1,j}$

# Der CYK-Algorithmus

## Algorithmus 11.1 (CYK-Algorithmus)

**Eingabe:**  $w \in \Sigma^*$ ,  $G = (V, \Sigma, S, P)$  in CNF

**Ausgabe:** „ja“, falls  $w \in L(G)$

{Einzelne Zeichen}

```
1: for  $i := 1$  TO  $n$  do
2:    $V_{i,i} := \{X \in V \mid X \rightarrow w[i] \text{ in } P\}$ 
   { Teilstrings der Länge  $\ell + 1$  }
3: for  $\ell := 1$  TO  $n - 1$  do
4:   for  $i := 1$  TO  $n - \ell$  do
5:      $V_{i,i+\ell} := \emptyset$ 
6:     for  $k := i$  TO  $i + \ell - 1$  do
7:        $V_{i,i+\ell} := V_{i,i+\ell} \cup$ 
          $\{X \mid X \rightarrow YZ \text{ in } P, Y \in V_{i,k}, Z \in V_{k+1,i+\ell}\}$ 
8: if  $S \in V_{1,n}$  then
9:   Akzeptieren
10: else
11:   Ablehnen
```

- Anweisung 7 kann durch eine Schleife über alle Regeln von  $G$  implementiert werden

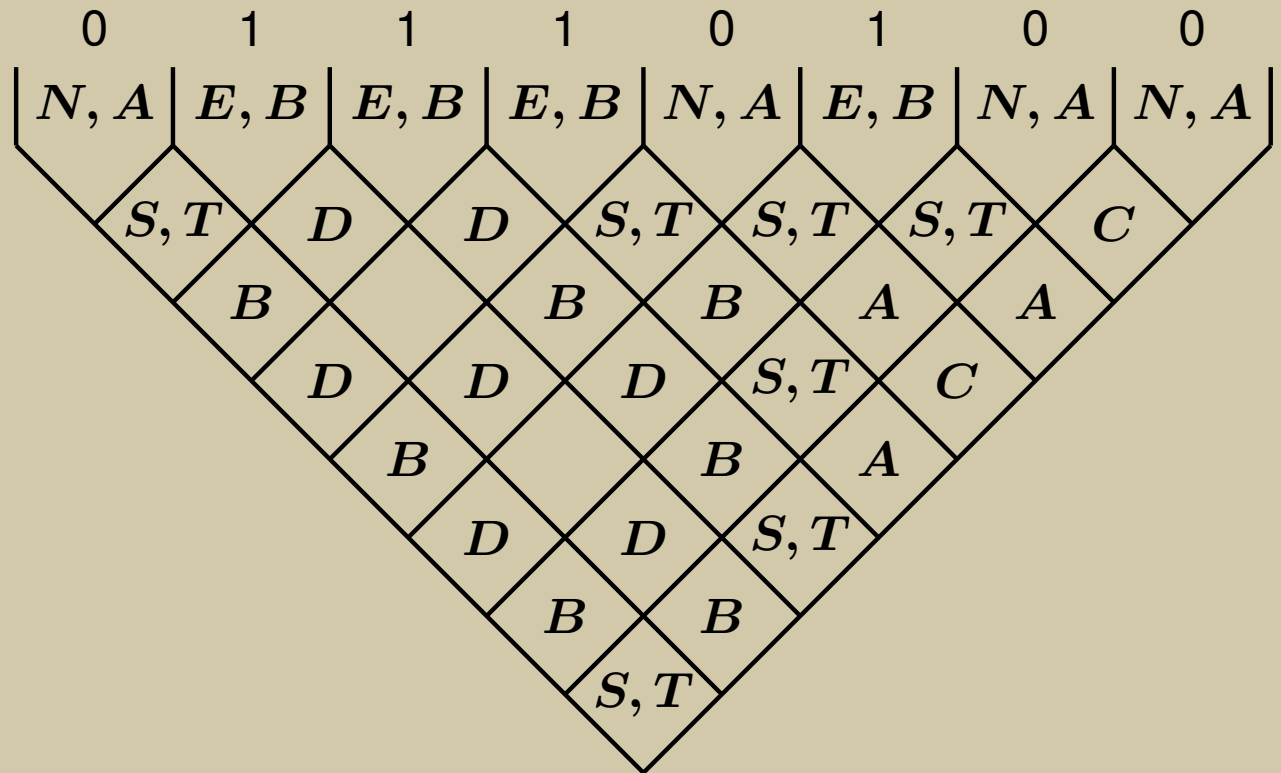
- Dass der Aufwand  $\mathcal{O}(n^3|G|)$  ist (für  $n = |w|$ ), lässt sich an den verschachtelten Schleifen des Algorithmus direkt ablesen

# CYK-Algorithmus: Beispiel

## Beispiel-Grammatik

$S \rightarrow NB \mid EA \mid \epsilon$   
 $T \rightarrow NB \mid EA$   
 $N \rightarrow 0$   
 $E \rightarrow 1$   
 $A \rightarrow 0 \mid NT \mid EC$   
 $B \rightarrow 1 \mid ET \mid ND$   
 $C \rightarrow AA$   
 $D \rightarrow BB$

## Verlauf der Bearbeitung



- Ergebnis:  $S \in V_{1,8}$  deshalb:  $01110100 \in L(G)$
- Wie lässt sich nun ein Ableitungsbaum für **01110100** gewinnen?
- Durch eine kleine Erweiterung des CYK-Algorithmus: er merkt sich jeweils nicht nur  $X$  sondern auch das zugehörige  $k$




# Der erweiterte CYK-Algorithmus

## Algorithmus 11.2 (Erweiterter CYK-Algorithmus 11.3)

**Eingabe:**  $w \in \Sigma^*$ ,  $G = (V, \Sigma, S, P)$  in CNF

**Ausgabe:** Ableitungsbaum, falls  $w \in L(G)$

- 1: **for**  $i := 1$  **TO**  $n$  **do**
- 2:    $V_{i,i} := \{(X, i) \mid X \rightarrow w[i, i] \text{ in } P\}$
- 3: **for**  $\ell := 1$  **TO**  $n - 1$  **do**
- 4:   **for**  $i := 1$  **TO**  $n - \ell$  **do**
- 5:      $V_{i,i+\ell} := \emptyset$
- 6:     **for**  $k := i$  **TO**  $i + \ell - 1$  **do**
- 7:        $V_{i,i+\ell} := V_{i,i+\ell} \cup \{(X, k) \mid$   
           $X \rightarrow YZ \text{ in } P, Y \in V_{i,k}, Z \in V_{k+1,i+\ell}\}$
- 8:   **if** es gibt kein  $k$  mit  $(S, k) \in V_{1,n}$  **then**
- 9:     Ablehnen
- 10: Konstruiere Ableitungsbaum rekursiv durch Aufruf von  $\text{Tree}(S, 1, n)$

 Dabei ist „ $Y \in V_{i,k}$ “ eine Abkürzung für:  
„es gibt ein  $m$  mit  $(Y, m) \in V_{i,k}$ “

## Algorithmus (Tree)

**Eingabe:**  $X, i, j$

**Ausgabe:** Ableitungsbaum für  
 $w[i, j]$  aus  $X$

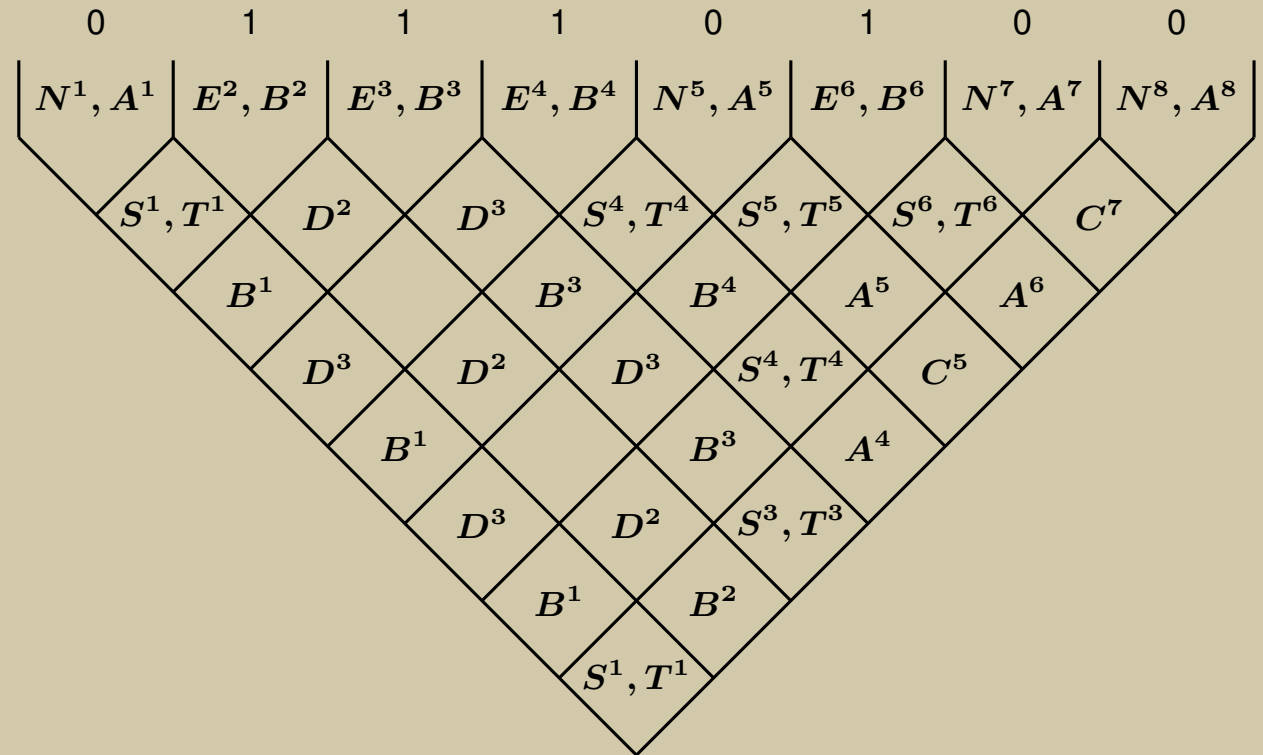
- 1: **if**  $i = j$  **then**
- 2:   RETURN Blatt  $\sigma_i$
- 3: Wähle ein  $k$  mit  $(X, k) \in V_{i,j}$
- 4: Wähle  $Y, Z$ , so dass
  - $X \rightarrow YZ$ ,
  - $Y \in V_{i,k}$  und
  - $Z \in V_{k+1,j}$
- 5: RETURN Baum mit Wurzel  $X$ , linkem Teilbaum  $\text{Tree}(Y, i, k)$  und rechtem Teilbaum  $\text{Tree}(Z, k + 1, j)$

# Erweiterter CYK-Algorithmus: Beispiel

## Beispiel-Grammatik

$S \rightarrow NB \mid EA \mid \epsilon$   
 $T \rightarrow NB \mid EA$   
 $N \rightarrow 0$   
 $E \rightarrow 1$   
 $A \rightarrow 0 \mid NT \mid EC$   
 $B \rightarrow 1 \mid ET \mid ND$   
 $C \rightarrow AA$   
 $D \rightarrow BB$

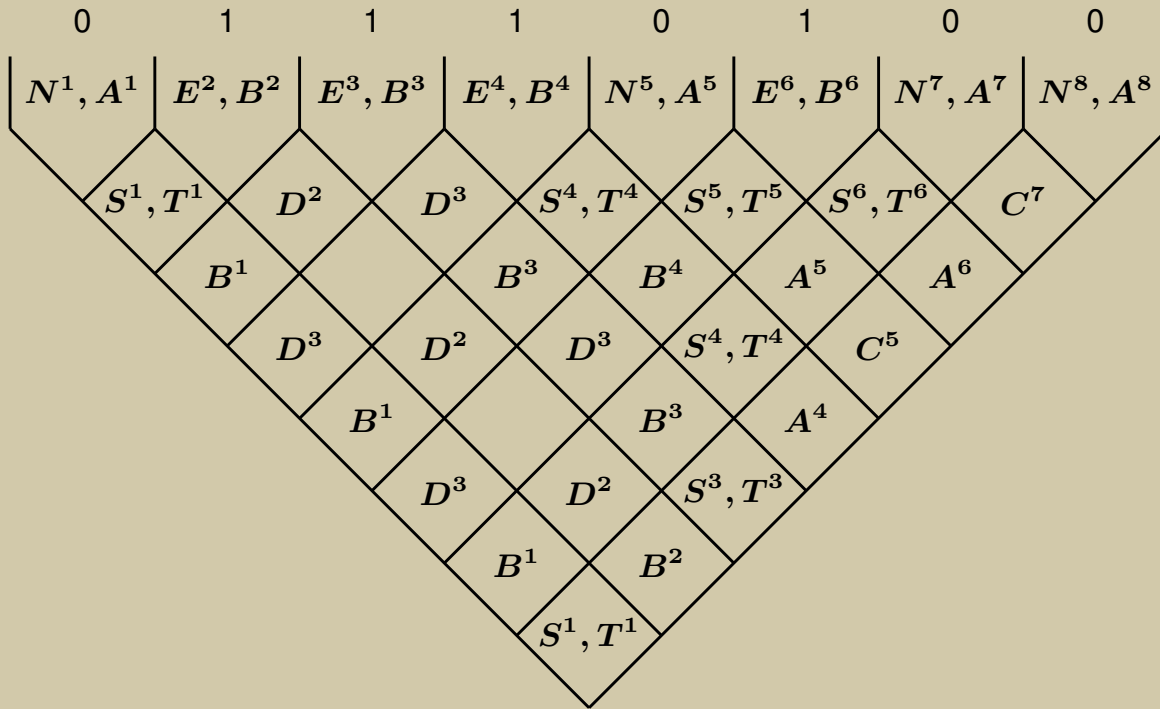
## Beispiel



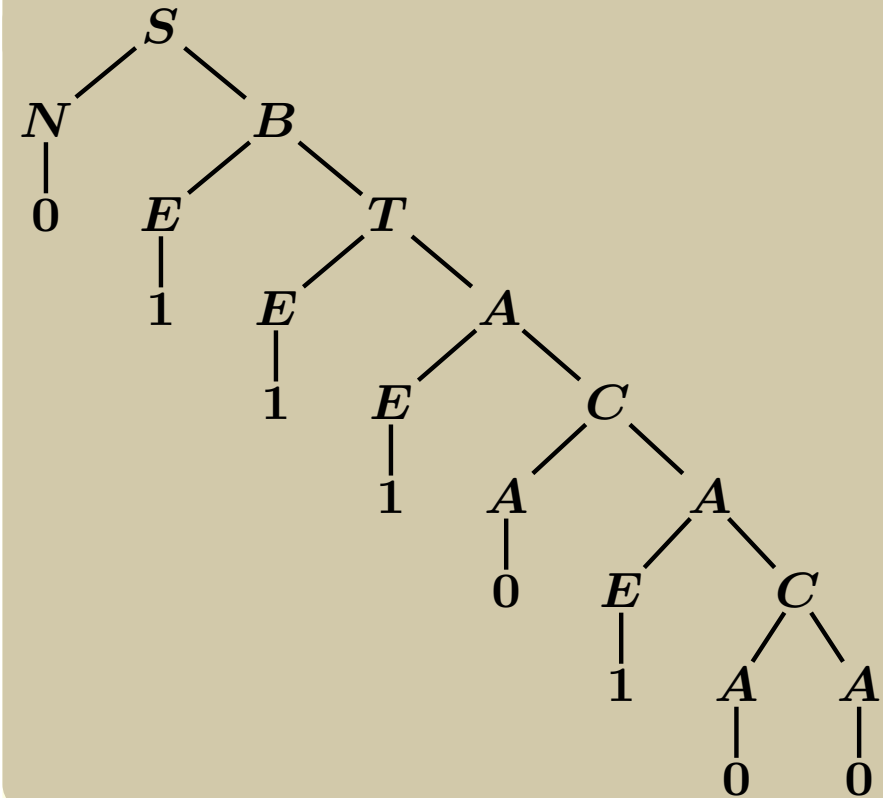
- Aus Platzgründen steht hier statt  $(X, k)$  jeweils  $X^k$
- Außerdem ist nur jeweils höchstens *ein*  $k$  mit  $(X, k) \in V_{i,j}$  angegeben

# Erweiterter CYK-Algorithmus: Beispiel-Ableitung

## Beispiel



## Beispiel



11.1 Algorithmen für beliebige kontextfreie Sprachen

▷ **11.2 Effiziente Syntaxanalyse**

11.2.1 Top-down-Syntaxanalyse

11.2.2 Bottom-up-Syntaxanalyse

# Effizientere Syntaxanalyse

- Syntaxanalyse von Programmtexten sollte möglichst in linearer Zeit erfolgen
  - Die beiden bisher betrachteten Algorithmen für das Syntaxanalyse-Problem sind also nicht effizient genug

- Für allgemeine kontextfreie Grammatiken sind aber leider keine Linearzeit-Algorithmen für das Syntaxanalyse-Problem bekannt


- Ein möglicher Ausweg ist, Grammatiken so einzuschränken, dass die Syntaxanalyse in linearer Zeit möglich wird
  - Das Ziel ist dabei, möglichst viele Sprachen mit den eingeschränkten Grammatiken beschreiben zu können

- Wir werden zwei Einschränkungen von Grammatiken kennen lernen
- Bei beiden wird die Eingabe von links nach rechts gelesen
- Bei beiden hängt die nächste Regelanwendung nur vom nächsten Zeichen der Eingabe ab
  - Damit kann uferloses Backtracking vermieden werden

- Bei  $LL(1)$ -Grammatiken wird, ausgehend vom Startsymbol, eine *Linksableitung* für  $w$  erzeugt
  - Top-down-Syntaxanalyse

- Bei  $LR(1)$ -Grammatiken wird, ausgehend von  $w$ , durch „Rückwärtsanwendung“ von Regeln eine *Rechtsableitung* erzeugt
  - Bottom-up-Syntaxanalyse

- Beide Grammatik-Typen gibt es auch mit Abhängigkeit von den nächsten  $k$  Zeichen

  $LL(k), LR(k)$

# Inhalt

11.1 Algorithmen für beliebige kontextfreie Sprachen

## **11.2 Effiziente Syntaxanalyse**

### ▷ **11.2.1 Top-down-Syntaxanalyse**

11.2.2 Bottom-up-Syntaxanalyse

# Effiziente Top-down-Syntaxanalyse: Vorüberlegungen (1/5)

- Im Folgenden betrachten wir nur Grammatiken  $G$ , die nicht *linksrekursiv* sind
  - ☞ d.h.:  $X \Rightarrow_G^* X\alpha$  mit  $\alpha \neq \epsilon$  ist verboten
    - Sonst bestünde die Gefahr von Endlosschleifen
- **Problem:** welche Regel soll angewendet werden, wenn mehrere Anwendungen möglich sind?
  - Wir wissen: Backtracking ist zu ineffizient
- **Idee** zur Vermeidung exponentiellen Aufwandes:
  - Wir schränken  $G$  so ein, dass immer direkt erkennbar ist, welche Regel angewendet werden muss
  - „Direkt erkennbar“ heißt dabei, dass für die Entscheidung nur das nächste Zeichen der Eingabe angeschaut werden muss

# Effiziente Top-down-Syntaxanalyse: Vorüberlegungen (2/5)

## Beispiel

- Wir betrachten die Grammatik für Palindrome:

$$P \rightarrow aPa \mid bPb \mid a \mid b \mid \epsilon$$

und versuchen eine Linksableitung für den String *abba* zu finden:

- Der Versuch scheitert schon im ersten Schritt, da aus der Kenntnis des ersten Zeichens *a* nicht hervorgeht, ob wir die Regel  $P \rightarrow a$  oder  $P \rightarrow aPa$  anwenden sollen
- Wenn wir korrekt mit  $P \Rightarrow aPa$  beginnen, stellt sich im zweiten Schritt wieder das Problem:

$$P \rightarrow b \text{ oder } P \rightarrow bPb$$

- Nach dem korrekten zweiten Schritt  $aPa \Rightarrow abPba$  gibt es wieder zwei Möglichkeiten:  $P \Rightarrow bPb$  oder  $P \Rightarrow \epsilon$

- Diese Grammatik ist also sicher nicht für die Top-down-Analyse mit nur einem „Vorschau-Zeichen“ geeignet
- Ein offensichtlicher Grund hierfür ist, dass es mehrere Regeln derselben Variablen gibt, deren rechte Seite mit demselben Terminalsymbol beginnt  
→ das verbieten wir in effizienten Top-down-Grammatiken



# Effiziente Top-down-Syntaxanalyse: Vorüberlegungen (3/5)

## Beispiel

- Wir betrachten die Grammatik

$$S \rightarrow aA \mid BC$$

$$A \rightarrow bA \mid c$$

$$B \rightarrow bC \mid Ca$$

$$C \rightarrow ba \mid ab$$

- Wir suchen eine Linksableitung für *ababa*:

$$S \Rightarrow aA$$

$$\Rightarrow abA$$

$$\Rightarrow ?$$

- Wir haben im ersten Schritt schon einen „Fehler“ gemacht

- Eine Ableitung ergäbe sich durch:

$$S \Rightarrow BC$$

$$\Rightarrow CaC$$

$$\Rightarrow abaC$$

$$\Rightarrow ababa$$

- Was ist hier schiefgelaufen?

- Zwar haben die rechten Regelseiten hier jeweils verschiedene erste Terminalsymbole

- Aber aus *B* lässt sich der String *aba* ableiten

- Damit stehen die beiden rechten Regelseiten *aA* und *BC* miteinander in Konkurrenz, obwohl sie nicht mit dem selben Terminalsymbol beginnen

- Wir müssen also auch berücksichtigen, welche ersten Zeichen sich durch weitere Ableitung aus der ersten Variablen einer rechten Regelseite ergeben können

# Effiziente Top-down-Syntaxanalyse: Vorüberlegungen (4/5)

## Beispiel

- Wir betrachten die Grammatik

$$S \rightarrow aA \mid BC$$

$$A \rightarrow bA \mid c$$

$$B \rightarrow c \mid \epsilon$$

$$C \rightarrow ab$$

- Wir suchen eine Linksableitung für  $ab$ :

$$S \Rightarrow aA$$

$$\Rightarrow abA$$

$$\Rightarrow ?$$

Wir haben schon wieder im ersten Schritt einen „Fehler“ gemacht

- Eine Ableitung ergäbe sich durch:

$$S \Rightarrow BC$$

$$\Rightarrow C$$

$$\Rightarrow ab$$

- Was ist hier schiefgelaufen?

- Die rechten Regelseiten haben hier jeweils verschiedene erste Terminalsymbole, auch bei Berücksichtigung der möglichen ersten Terminalsymbole, die sich aus ersten Variablen rechter Regelseiten ableiten lassen

- Aber  $B$  lässt sich zu  $\epsilon$  ableiten und  $C$  zu  $ab$

- Deshalb stehen die beiden rechten Regelseiten  $aA$  und  $BC$  miteinander in Konkurrenz, obwohl aus  $B$  nicht als erstes Terminalsymbol  $a$  ableitbar ist

- Wir müssen also auch berücksichtigen, welche ersten Zeichen sich durch Ableitung aus den gesamten rechten Regelseiten ergeben können und dabei insbesondere  $\epsilon$ -Ableitungen berücksichtigen

# Effiziente Top-down-Syntaxanalyse: Vorüberlegungen (5/5)

## Beispiel

- Wir betrachten die Grammatik

$$S \rightarrow AB$$

$$A \rightarrow \epsilon \mid aB$$

$$B \rightarrow aaB \mid b$$

- Wir suchen eine Linksableitung für  $aab$ :

$$S \Rightarrow AB$$

$$\Rightarrow aBB$$

$$\Rightarrow ?$$

- Wir haben hier im zweiten Schritt einen „Fehler“ gemacht
- Eine Ableitung ergäbe sich durch:

$$S \Rightarrow AB$$

$$\Rightarrow B$$

$$\Rightarrow aaB$$

$$\Rightarrow aab$$

- Was ist hier schiefgelaufen?

- Zwar haben die rechten Regelseiten jeweils verschiedene erste Terminalsymbole, auch bei Berücksichtigung der ableitbaren Strings
- Aber in  $AB$  kann das nächste Zeichen  $a$  sowohl aus  $A$  entstehen als auch (nach Anwendung von  $A \rightarrow \epsilon$ ) aus  $B$
- Dies führt dazu, dass bei der Satzform  $AB$  und nächstem Symbol  $a$  nicht klar ist, ob als nächstes  $A \rightarrow \epsilon$  oder  $A \rightarrow aB$  angewendet werden muss
- Wir müssen deshalb in einem solchen Fall ( $A \Rightarrow^* \epsilon$ ) auch darauf achten, welche Zeichen **hinter**  $A$  erzeugt werden können
- Diese Beobachtungen führen uns zur Definition von  $LL(1)$ -Grammatiken

# LL(1)-Grammatiken: Definition

- Zur Definition von LL(1)-Grammatiken verwenden wir die folgenden beiden Operatoren:

- Für eine Satzform  $\alpha$  sei  
$$\underline{\text{FIRST}(\alpha)} \stackrel{\text{def}}{=} \{\sigma \in \Sigma \mid \alpha \Rightarrow^* \sigma v, v \in \Sigma^*\} \cup \{\epsilon \mid \alpha \Rightarrow^* \epsilon\}$$

- Für eine Variable  $X$  sei  
$$\underline{\text{FOLLOW}(X)} \stackrel{\text{def}}{=} \{\sigma \in \Sigma \mid S \Rightarrow^* uX\sigma v, u, v \in \Sigma^*\}$$

- $\text{FIRST}(\alpha)$  enthält also alle ersten Terminalzeichen von Strings, die aus  $\alpha$  abgeleitet werden können (und evtl.  $\epsilon$ , wenn dieses aus  $\alpha$  abgeleitet werden kann)

- $\text{FOLLOW}(X)$  enthält alle Terminalzeichen, die in irgendeiner aus  $S$  ableitbaren Satzform *unmittelbar hinter*  $X$  vorkommen können

## Definition (LL(1)-Grammatik)

- Sei  $G$  eine kontextfreie Grammatik  $G$  ohne nutzlose Variablen und ohne Linksrekursion
- $G$  ist eine **LL(1)-Grammatik**, wenn für alle Variablen  $X$  und alle Regeln  $X \rightarrow \alpha$  und  $X \rightarrow \beta$  mit  $\alpha \neq \beta$  die beiden folgenden Bedingungen gelten:
  - (i)  $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
  - (ii) Falls  $\alpha \Rightarrow^* \epsilon$  so ist  
$$\text{FOLLOW}(X) \cap \text{FIRST}(\beta) = \emptyset$$

- $\text{FIRST}(\alpha)$  und  $\text{FOLLOW}(X)$  können effizient berechnet werden

- LL(1)-Grammatiken lassen sich sehr einfach rekursiv implementieren
- Die Erzeugung von Code lässt sich dabei oft sehr leicht integrieren: *rekursiver Abstieg*

# LL(1)-Grammatiken: Beispiele

## Beispiel

$$\begin{aligned} S &\rightarrow aB \mid bA \mid c \\ A &\rightarrow aS \mid bAA \\ B &\rightarrow bS \mid aBB \end{aligned}$$

- Offensichtlich erfüllt die Grammatik alle Bedingungen der Art (i)
- Da keine  $\epsilon$ -Regeln vorkommen, erfüllt sie auch (ii)
- Also ist es eine LL(1)-Grammatik

## Beispiel

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow \epsilon \mid aB \\ B &\rightarrow aA \mid b \end{aligned}$$


- FOLLOW( $A$ ) = { $a, b$ }
  - FIRST( $aB$ ) = { $a$ }
  - Da es eine Regel  $A \rightarrow \epsilon$  gibt, müssten diese beiden Mengen disjunkt sein
- keine LL(1)-Grammatik

## Beispiel

$$\begin{aligned} A &\rightarrow BC \mid ab \\ B &\rightarrow cAA \mid bc \mid \epsilon \\ C &\rightarrow cC \mid \epsilon \end{aligned}$$

- Es gilt:
    - FIRST( $A$ ) = { $a, b, c, \epsilon$ }
    - FIRST( $B$ ) = { $b, c, \epsilon$ }
    - FIRST( $C$ ) = { $c, \epsilon$ }
    - FOLLOW( $A$ ) = { $a, b, c$ }
    - FOLLOW( $B$ ) = { $a, b, c$ }
    - FOLLOW( $C$ ) = { $a, b, c$ }
    - $BC \Rightarrow^* \epsilon$
  - Also haben wir:
    - $A \rightarrow BC$  und  $A \rightarrow ab$ ,
    - $BC \Rightarrow^* \epsilon$
    - FOLLOW( $A$ )  $\cap$  FIRST( $ab$ ) = { $a$ }  $\neq \emptyset$
- ➡ Dies ist keine LL(1)-Grammatik

# LL(1)-Grammatiken: Parsing

- Parsing-Algorithmen für LL(1)-Grammatiken verwenden eine Tabelle, die dem Compiler in jeder Situation (Variable und nächstes Zeichen) sagt, welche Regel anzuwenden ist
- Die Berechnung dieser Tabelle wird hier nicht betrachtet
- Stattdessen schauen wir ein Beispiel an  
 Die Beispieldtabelle vernachlässigt das Wortende-Symbol
- LL( $k$ )-Grammatiken (für  $k \geq 2$ ) erlauben Parsing-Algorithmen, die die nächsten  $k$  Zeichen der Eingabe berücksichtigen, und verallgemeinern LL(1)-Grammatiken

## Beispiel

- Sei  $G$  die LL(1)-Grammatik

$$S \rightarrow AB \mid (S)S$$

$$A \rightarrow CA \mid \epsilon$$

$$B \rightarrow ba$$

$$C \rightarrow ca$$

- $\text{FIRST}(B) = \{b\}$ ,  $\text{FIRST}(C) = \{c\}$
- $\text{FIRST}(A) = \{c, \epsilon\}$
- $\text{FIRST}(S) = \{b, c, (\}$
- $\text{FOLLOW}(A) = \{b\}$

- Die zugehörige Tabelle ist:

	$a$	$b$	$c$	$($	$)$
$S$		$AB$	$AB$	$(S)S$	
$A$		$\epsilon$	$CA$		
$C$			$ca$		
$B$		$ba$			

# Inhalt

11.1 Algorithmen für beliebige kontextfreie Sprachen

## **11.2 Effiziente Syntaxanalyse**

11.2.1 Top-down-Syntaxanalyse

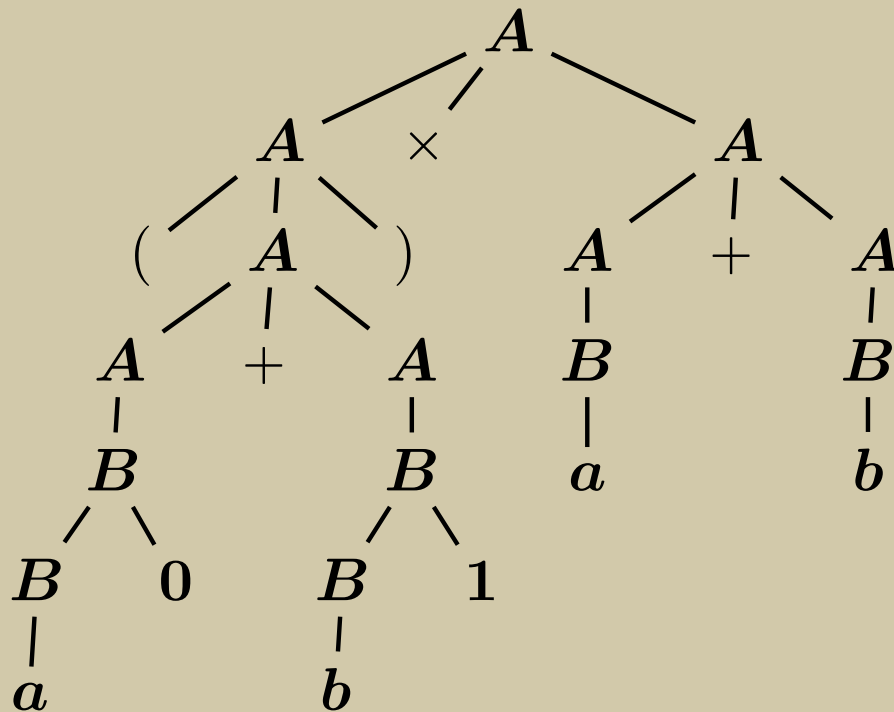
▷ **11.2.2 Bottom-up-Syntaxanalyse**

# Bottom-up Syntaxanalyse (1/3)

## Beispiel-Grammatik

$A \rightarrow B \mid A + A \mid A \times A \mid (A)$   
 $B \rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1$

## Beispiel-Ableitungsbaum



- Bei der Bottom-up Syntaxanalyse wird der Ableitungsbaum von unten nach oben und (üblicherweise) von links nach rechts konstruiert
- Dieses Vorgehen ergibt eine Rechtsableitung

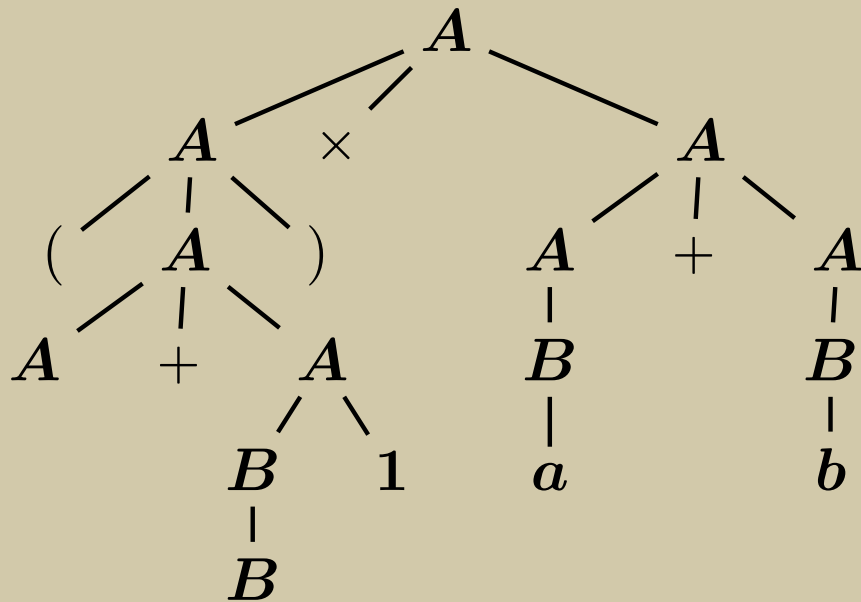
## Beispiel-Ableitung: Bottom-up (rückwärts)

$(a0 + b1) \times a + b \Leftarrow$   
 $(B0 + b1) \times a + b \Leftarrow$   
 $(B + b1) \times a + b \Leftarrow$   
 $(A + b1) \times a + b \Leftarrow$   
 $(A + B1) \times a + b \Leftarrow$   
 $(A + B) \times a + b \Leftarrow$   
 $(A + A) \times a + b \Leftarrow$   
 $(A) \times a + b \Leftarrow$   
 $A \times a + b \Leftarrow$   
 $A \times B + b \Leftarrow$   
 $A \times A + b \Leftarrow$   
 $A \times A + B \Leftarrow$   
 $A \times A + A \Leftarrow$   
 $A \times A \Leftarrow A$



## Bottom-up Syntaxanalyse (2/3)

### Beispiel



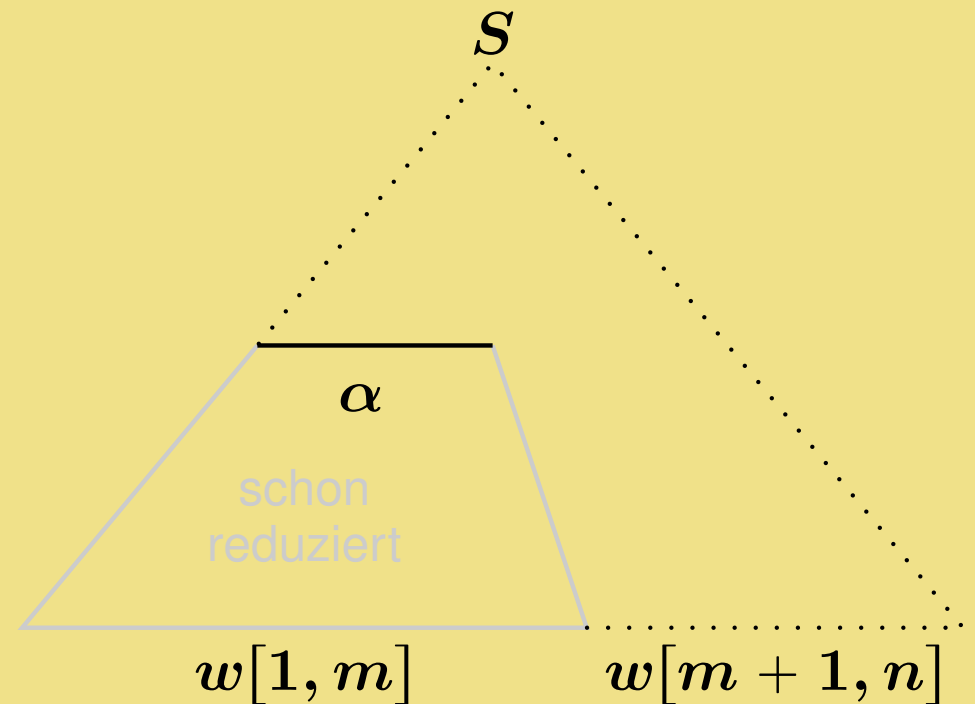
- Eingabe:  $(a0 + b1) \times a + b$

### Beispiel

- Der abgebildete unvollständige Ableitungsbaum stellt eine Zwischensituation bei der Erzeugung einer Rechtsableitung für  $(a0 + b1) \times a + b$  dar
- Das Anfangsstück  $(a0 + b1$  der Eingabe wurde schon gelesen und auf  $(A + B1$  reduziert
- Der unvollständige Baum entspricht der Satzform::  $(A + B1) \times a + b$
- Die restliche Eingabe  $) \times a + b$  muss noch gelesen werden
- Im nächsten Schritt muss wieder eine Regel rückwärts angewendet werden

## Bottom-up Syntaxanalyse (3/3)

- Die allgemeine Situation bei der Bestimmung des nächsten Schrittes einer Rechtsableitung für einen Eingabestring  $w$  ist wie folgt:
  - Für ein Anfangsstück  $w[1, m]$  des Strings wurden schon Regeln rückwärts angewendet
  - Die daraus entstandene Satzform  $\alpha$  liegt auf dem Keller
- Wir haben also:  $\alpha \Rightarrow_r^* w[1, m]$
- Damit insgesamt  $S$  erreicht wird, muss also  $\alpha w[m + 1, n]$  auf  $S$  „zurück abgeleitet werden“
- Im nächsten Schritt muss eine passende rechte Regelseite  $\beta$  und eine Regel  $X \rightarrow \beta$  identifiziert und dann (rückwärts) angewendet werden



# Bottom-up Syntaxanalyse: Vorüberlegungen

## Beispiel: Grammatik

$$S \rightarrow aABe$$

$$A \rightarrow Abc \mid b$$

$$B \rightarrow d$$

- Wir betrachten die Bottom-up Syntaxanalyse für den String *abbcede*
- Eine Rechtsableitung für diesen String:

## Beispiel: Rechtsableitung

$$S \Rightarrow aABe$$

$$\Rightarrow aAde$$

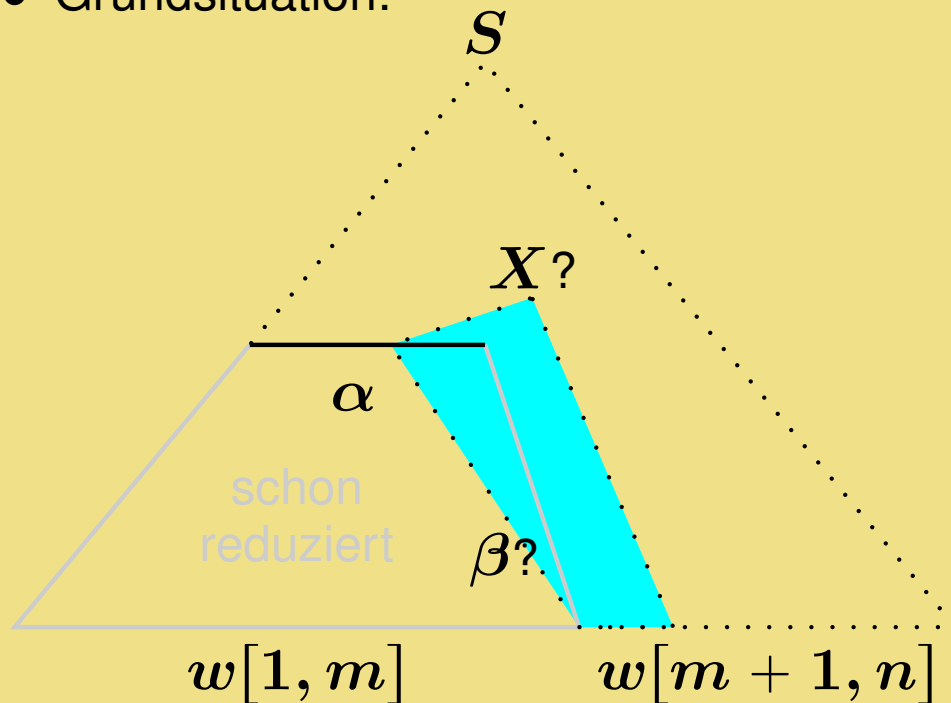
$$\Rightarrow aAbcde$$

$$\Rightarrow abbcde$$

- Bottom-up-Syntaxanalyse für *abbcede*
- Ziel: durch „umgekehrte“ Regelanwendung mit Rechtsableitung zu *S* kommen
- 1. Ableitungsschritt:
  - Wir suchen zuerst rechte Seiten von Produktionen in *abbcede*
  - Kandidaten: *b* und *d*
  - Wir entscheiden uns für  $A \rightarrow b$  und erhalten die Satzform *aAbcde*
- 2. Ableitungsschritt:
  - Kandidaten (in *aAbcde*): *Abc*, *b*, *d*
  - Welche Produktion sollen wir anwenden?
- Die Grammatik soll garantieren, dass das Finden der „richtigen rechten Seite“ immer „einfach“ ist
- Wir nennen diese „richtige rechte Seite“ den *Schlüssel* (engl.: *handle*)
  - Der Schlüssel von *aAbcde* ist *Abc*



# Bottom-up Syntaxanalyse: Prinzip (1/2)

- Grundsituation:



- $w[1, m]$  ist schon reduziert auf  $\alpha$
- $w[m + 1, n]$  ist noch zu lesen
- Nächster Schritt: Passenden Schlüssel  $\beta$  und Regel  $X \rightarrow \beta$  identifizieren und anwenden

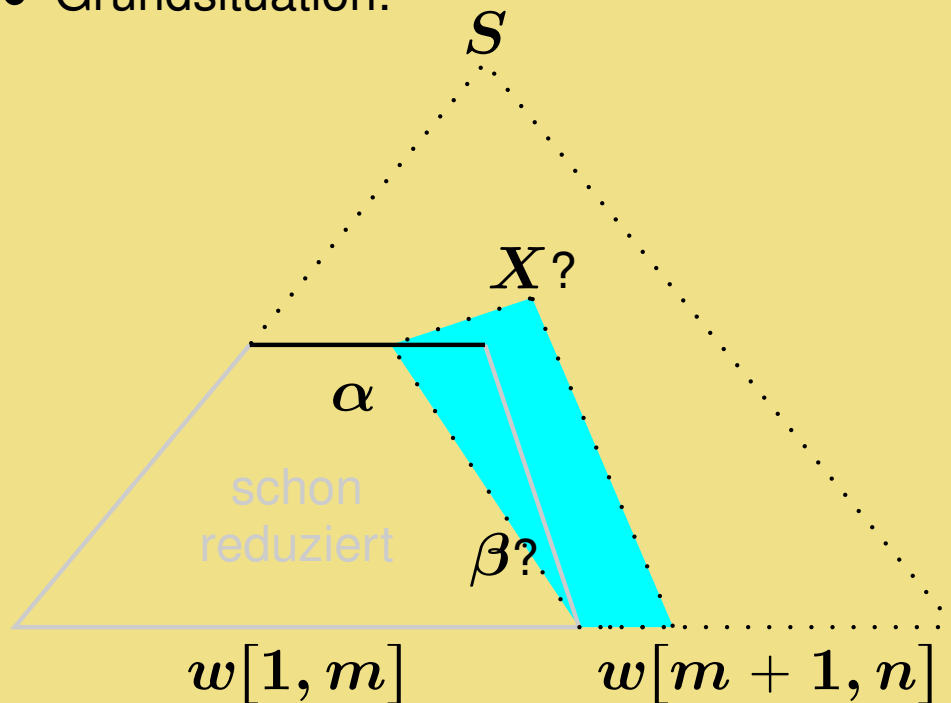
- $\beta$  kann
  - (a) ein Suffix von  $\alpha$  sein,
  - (b) ein Teilstring von  $w[m + 1, n]$  sein, oder
  - (c) aus einem Suffix von  $\alpha$  und einem Präfix von  $w[m + 1, n]$  bestehen

- Die jeweils auszuführende Aktion:
  - (a) Ersetze auf dem Keller  $\beta$  durch eine passende linke Regelseite  $X$   
 *Reduce*
  - (b,c) oder lege (zunächst)  $w[m + 1]$  auf den Keller  
 *Shift*

→ *Shift-Reduce-Parsing*

## Bottom-up Sytaxanalyse: Prinzip (2/2)

- Grundsituation:




- Damit Shift-Reduce-Parsing ohne Backtracking möglich ist, muss die Grammatik folgende Bedingungen erfüllen:
  - (A) Der Parser muss den Schlüssel  $\beta$  identifizieren können, um zu entscheiden, ob er einen Reduce-Schritt ausführen kann
  - (B) Er muss erkennen können, bezüglich welcher Variablen  $X$  ein Reduktions-schritt mit Regel  $X \rightarrow \beta$  angewandt wird
- **Ziel bei LR(1)-Grammatiken:** um diese Entscheidungen zu treffen, muss nur das nächste Zeichen hinter dem (kürzesten möglichen) Schlüssel gelesen werden

# LR(1)-Grammatiken: Definition

## Definition (LR(1)-Grammatik)

- Eine Grammatik  $G$  heißt **LR(1)-Grammatik**, falls für alle  $X \in V$ ,  $x, y \in \Sigma^*$ , und alle Satzformen  $\alpha, \beta, \gamma$  gelten:
  - (1) Falls für ein  $\sigma \in \Sigma$ 
$$S \Rightarrow_r^* \alpha X \sigma x \Rightarrow_r \alpha \beta \sigma x$$
und
$$\gamma \Rightarrow_r \alpha \beta \sigma y$$
gelten mit  $\gamma \neq \alpha X \sigma y$ , dann ist  $\gamma$  *nicht* von  $S$  aus ableitbar
  - (2) Falls
$$S \Rightarrow_r^* \alpha X \Rightarrow_r \alpha \beta$$
und
$$\gamma \Rightarrow_r \alpha \beta$$
gelten mit  $\gamma \neq \alpha X$ , dann ist  $\gamma$  nicht von  $S$  aus ableitbar

 Bedingung (1) sagt also, dass es mit aktueller Satzform  $\alpha\beta$  und nächstem Zeichen  $\sigma$  keine Alternative zur Rückwärtsanwendung von  $X \rightarrow \beta$  gibt

- Die Definition garantiert also gerade die Gültigkeit der Bedingungen (A) & (B) der vorherigen Folie
- Nach Lesen von  $\sigma$  (oder am Ende der Eingabe) und mit  $\alpha\beta$  auf dem Keller „weiß“ der Algorithmus, dass er  $X \rightarrow \beta$  anwenden muss
- Um zu einer LR(1)-Grammatik einen Shift-Reduce-Algorithmus zu gewinnen, ist eine genauere Analyse der Grammatik nötig
  - Dann können die anzuwendenden Regeln jeweils aus einer Tabelle abgelesen werden
  - Diese Analyse geht aber über den Rahmen dieser Vorlesung hinaus
- LR(1)-Grammatiken lassen sich verallgemeinern für eine weitere Vorausschau (*look-ahead*) von  $k$  Zeichen statt einem Zeichen

# LR( $k$ )-Grammatiken: Beispiele (1/2)

## Beispiel

$$S \rightarrow CD$$

$$C \rightarrow a$$

$$D \rightarrow EF \mid aG$$

$$E \rightarrow ab$$

$$F \rightarrow bb$$

$$G \rightarrow bba$$

- Die Grammatik hat nur zwei Rechtsableitungen:
  - $S \Rightarrow_r CD \Rightarrow_r CEF \Rightarrow_r CEbb \Rightarrow_r Cabbb$
  - $S \Rightarrow_r CD \Rightarrow_r CaG \Rightarrow_r Cabba$

➡ Sie erfüllt *nicht* die LR(1)-Bedingung:

- $\sigma = b$
- $\alpha = C, X = E, x = b, \beta = ab$
- $\gamma = CaG, y = a$
- Aber:  $\gamma = CaG \neq CEbb = \alpha X \sigma x$

- Sie ist aber eine LR(2)-Grammatik

# LR( $k$ )-Grammatiken

## LR( $k$ )-Grammatik

- Für jedes  $k \geq 0$  heißt eine Grammatik  $G$  **LR( $k$ )-Grammatik**, falls für alle  $X \in V$ ,  $x, y \in \Sigma^*$ , und alle Satzformen  $\alpha, \beta, \gamma$  gelten:

(1) Falls für ein  $z \in \Sigma^k$

$$S \Rightarrow_r^* \alpha X z x \Rightarrow_r \alpha \beta z x$$

und

$$\gamma \Rightarrow_r \alpha \beta z y$$

gelten mit  $\gamma \neq \alpha X z y$ , dann ist  $\gamma$  nicht von  $S$  aus ableitbar

(2) Falls für ein  $z \in \Sigma^{<k}$

$$S \Rightarrow_r^* \alpha X z \Rightarrow_r \alpha \beta z$$

und

$$\gamma \Rightarrow_r \alpha \beta z$$

gelten mit  $\gamma \neq \alpha X z$ , dann ist  $\gamma$  nicht von  $S$  aus ableitbar



## LR( $k$ )-Grammatiken: Beispiele (2/2)

### Beispiel

$$\begin{aligned} S &\rightarrow Cc \mid Dd \\ C &\rightarrow Ca \mid \epsilon \\ D &\rightarrow Da \mid \epsilon \end{aligned}$$

- Die Grammatik erzeugt Strings der Form  $a^n c$  und  $a^n d$
- Ableitungen:
  - $S \Rightarrow_r Cc \Rightarrow_r^* Ca^n c \Rightarrow_r a^n c$
  - $S \Rightarrow_r Dd \Rightarrow_r^* Da^n d \Rightarrow_r a^n d$
- Der Parser müsste zuerst  $a^n c$  oder  $a^n d$  lesen, um zu wissen, ob als letzter Ableitungsschritt der Rechtsableitung die Regel  $C \rightarrow \epsilon$  oder  $D \rightarrow \epsilon$  angewendet werden muss
- ➡ Die Grammatik erfüllt für kein  $k$  die LR( $k$ )-Bedingung

# LL( $k$ )- und LR( $k$ )-Grammatiken: Eigenschaften

## Satz 11.4

- Für jedes  $k \geq 1$  lassen sich durch LL( $k+1$ )-Grammatiken mehr Sprachen beschreiben als durch LL( $k$ )-Grammatiken

## Satz 11.5

- Für jedes  $k \geq 1$  sind äquivalent:
  - $L$  ist deterministisch kontextfrei
  - $L = L(G)$  für eine LR( $k$ )-Grammatik  $G$

## Folgerung 11.6

- (a) Für jedes  $k \geq 1$  sind LR( $k$ )-Grammatiken und LR(1)-Grammatiken gleich ausdrucksstark
- (b) Das Syntaxanalyseproblem für LR( $k$ )-Grammatiken lässt sich in linearer Zeit lösen

(b) ...mittels DPDAs

- LR(0)-Grammatiken entsprechen gerade den deterministischen Kellerautomaten, die mit leerem Keller akzeptieren

- Jede LL( $k$ )-Grammatik ist auch eine LR( $k$ )-Grammatik
- Aber: nicht zu jeder LR( $k$ )-Grammatik gibt es eine äquivalente LL( $k$ )-Grammatik

- LR( $k$ )-Grammatiken sind eindeutig

## Folgerung 11.7

- Jede deterministisch kontextfreie Sprache hat eine eindeutige Grammatik

- Mehr zur Syntaxanalyse mit LR( $k$ )-Grammatiken findet sich im Buch von Ingo Wegener
- Es gibt eine Vielzahl weiterer eingeschränkter kontextfreier Grammatiktypen, die für die Konstruktion von Compilern von Bedeutung sind
- Näheres (und natürlich sehr viel mehr) können Sie in der Vorlesung *Übersetzerbau* erfahren

# Zusammenfassung

- Das Wortproblem für kontextfreie Grammatiken lässt sich mit dem CYK-Algorithmus in Zeit  $O(|G||w|^3)$  lösen
- Um eine sehr effiziente Syntaxanalyse von Programm-Code zu gewährleisten, ist es nötig, eingeschränkte Grammatiken zu verwenden
- Dabei gibt es zwei wichtige Ansätze:
  - Top-down: LL(1)-Grammatiken
  - Bottom-up: LR(1)-Grammatiken
- LL(1)-Grammatiken sind konzeptionell einfacher, aber LR(1)-Grammatiken sind ausdrucksstärker
- LR(1)-Grammatiken können genau die deterministisch kontextfreien Sprachen beschreiben
- Parser lassen sich zum Beispiel mit **yacc** automatisch aus kontextfreien Grammatiken erzeugen
- Dabei lässt sich im Falle eines (einfachen) Compilers sogar die Codeerzeugung integrieren
  - Zusammenspiel mit **lex**

# Literatur

- **CYK-Algorithmus:**

- Daniel H. Younger. Recognition and parsing of context-free languages in time  $n^3$ . *Information and Control*, 10(2):189–208, 1967