

Burchard v. Braunmühl

GTI-Skript

Grundlagen der Theoretischen Informatik

Informatik III

2 Kontextfreie Sprachen

2.1 Kontextfreie Grammatiken

2.1.1 Definitionen

Definition 2.1.1 (kontextfreie Grammatik, CFG)

Seien N, T disjunkt endliche Alphabete, P eine endliche Teilmenge von $N \times (N \cup T)^*$ und $S \in N$. Dann heißt die Struktur $G = (N, T, P, S)$ *kontextfreie Grammatik* (i.Z. CFG, engl.: context free grammar).

Für zwei Worte $W, W' \in (N \cup T)^*$ erklären wir, wann W' aus W in einem Schritt oder in mehreren Schritten mithilfe der Regeln der Grammatik abgeleitet werden kann:

Die *Einschrittrelation* \vdash_G ist definiert durch:

$$W \vdash_G W' \quad \text{gdw.} \quad \begin{array}{l} W = UAV \quad (U, V \in (N \cup T)^*, A \in N), \\ W' = UXV \quad (X \in (N \cup T)^*) \quad \text{und} \\ (A, X) \in P. \end{array}$$

Ist $(A, X) \in P$, so werden auch wir die übliche Schreibweise $A \longrightarrow X$ verwenden (welche suggeriert: ersetze A durch X).

Die *i-Schrittrelation* \vdash_G^i ist definiert vermöge

$$\begin{array}{l} W \vdash_G^0 W \\ W \vdash_G^i W' \wedge W' \vdash_G W'' \implies W \vdash_G^{i+1} W''. \end{array}$$

Die *Mehrschrittrelation* \vdash_G^* ist die reflexive und transitive Hülle von \vdash_G , d.h.

$$\begin{array}{ll} W \vdash_G^* W & \text{(reflexiv)} \\ W \vdash_G^* W' \wedge W' \vdash_G W'' \implies W \vdash_G^* W'' & \text{(transitiv) oder} \\ W \vdash_G^* W' \quad \text{gdw.} \quad \exists i \in \mathbb{N} : W \vdash_G^i W'. & \end{array}$$

Eine Folge W_0, \dots, W_n nennen wir eine *Ableitung* der Länge n von W_n aus W_0 in G , wenn $W_{i-1} \vdash_G W_i$ ($i = 1, \dots, n$). Die W_i nennen wir *Ableitungsworte*. Die *Sprache* von G ist $L(G) := \{w \in T^* \mid S \vdash_G^* w\}$ und entsprechend heißt S das *Startsymbol*.

Wir nennen die Zeichen aus N *Nonterminals* oder *Variablen*, die aus T *Terminals* und die Paare aus P (*Ableitungs-*) *Regeln*. Die Terminals heißen so, weil sie nicht weiter ableitbar sind, bei ihnen terminiert die Ableitung. Sie spielen dieselbe Rolle wie die Eingabezeichen bei den finiten Automaten: es sind die Worte über diesen Zeichen, die den Gegenstand unserer Betrachtungen ausmachen, über die wir reden und nachdenken, die wir durch bestimmte Kalküle beschreiben (erkennen oder erzeugen) wollen. Die Nonterminals sind eben die nichtterminierenden Zeichen, für die es (im Prinzip) Regeln gibt, die sie weiter ableiten. Wir nennen sie auch Variablen, weil sie nicht die Objekte unserer Betrachtung sind, sondern Zeichen des Kalküls, mit dessen Hilfe wir Terminalworte beschreiben: die Variablen

stehen für die Worte, die aus ihnen ableitbar sind. (Es bedarf allerdings einiger Phantasie, in ihnen die Variablen zu entdecken, wie sie in der Mathematik gebraucht werden, nämlich Zeichen, die für Objekte stehen und ihre Bedeutung erst bekommen, wenn sie durch einen Operator gebunden und ihnen ein Objektbereich zugewiesen wurde.) Ein Zeichen, das für ein Zeichen steht, also eine Zeichenvariable oder Zeichenkonstante, wird oft Metazeichen genannt. Entsprechend findet man für die Nonterminals auch die Bezeichnung Metazeichen (mit denselben Bedenken, mit denen man sie Variable nennt).

Wir nützen die Gelegenheit zu erklären was ein Hüllenoperator ist.

Definition 2.1.2 (Hüllenoperator)

Sei \mathfrak{M} ein Mengensystem, also eine Klasse von Mengen, und sei \mathfrak{H} ein Operator auf \mathfrak{M} , d.h. eine Abbildung von \mathfrak{M} in \mathfrak{M} . Dann nennt man \mathfrak{H} einen Hüllenoperator, wenn für alle Mengen $R_1, R_2 \in \mathfrak{M}$ gilt:

1. $R_1 \subseteq \mathfrak{H}(R_1)$ (monoton)
2. $R_1 \subseteq R_2 \implies \mathfrak{H}(R_1) \subseteq \mathfrak{H}(R_2)$ (inklusionserhaltend)
3. $\mathfrak{H}(\mathfrak{H}(R_1)) = \mathfrak{H}(R_1)$ (idempotent).

Ein Hüllenoperator schafft aus einer Menge R die kleinste Menge, die R umfaßt und unter gewissen Operationen oder Relationen abgeschlossen ist.

Definition 2.1.3 (Reflexive und transitive Hülle einer Relation)

Ist R eine Relation aus $A \times A$, so können wir die kleinste Relation $R^* \subseteq A \times A$ betrachten, die R enthält und zudem transitiv und reflexiv ist:

1. $\forall a \in A : (a, a) \in R^*$
2. $\forall a, b, c \in A : (a, b), (b, c) \in R^* \implies (a, c) \in R^*$

Der Operator $*$ ist ein Hüllenoperator, denn $R \subseteq R^*$, $(R^*)^* = R^*$ und $R_1 \subseteq R_2 \implies R_1^* \subseteq R_2^*$.

2.1.2 Beispiele

Als nächstes behandeln wir ein paar Beispiele und zeigen, wie man beweist, daß eine gegebene Sprache von einer gegebenen Grammatik erzeugt wird.

Notation 2.1.4

Abkürzend schreiben wir statt $\{(A, W_1), \dots, (A, W_k)\}$ auch $\{A \longrightarrow W_1 \mid \dots \mid W_k\}$

Beispiel 2.1.5

Eine CFG für die Sprache $L = \{a^n b^n \mid n \in \mathbb{N}\}$ ist $G = (\{S\}, \{a, b\}, \{S \longrightarrow aSb \mid \varepsilon\}, S)$

Beispiel 2.1.6

Die Sprache $PAL = \{w \in \{a, b\}^* \mid w = \overleftarrow{w}\} =$ Menge der Palindrome über $\{a, b\}$. (Palindrome sind Worte, die von vorn und von hinten gelesen gleich lauten, z.B. OTTO). Die Palindrome kann man auf Grund folgender Bildungsregeln erzeugen:

1. $\varepsilon, a, b \in PAL$
2. $w \in PAL \implies awa, bwb \in PAL$
3. keine anderen Worte sind aus PAL, d.h. PAL ist die kleinste Menge, die die Regeln 1. und 2. erfüllt.

Daraus können wir uns sofort eine CFG konstruieren:

$$G = (\{S\}, \{a, b\}, \{S \longrightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb\}, S)$$

Notation 2.1.7

$\#_a(w)$ = Anzahl der Vorkommen von a im Wort w .

Beispiel 2.1.8

Wir suchen eine CFG für die Sprache $L = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$. Eine mögliche Idee aus der Welt der finiten Automaten ist: Erzeugen wir ein a , so merken wir uns, daß zum Ausgleich auch ein b erzeugt werden muß. Sei also A die Variable, die die Information trägt: es ist ein (überschüssiges) b erzeugt worden, es muß noch ein a erzeugt werden. Sei B die Variable, die besagt, daß wir jetzt ein a zuviel haben und S die Startvariable, die ausgeglichene Worte erzeugt. D.h.

$$\begin{aligned} S \text{ erzeuge alle Worte } w \text{ mit } & \#_a(w) = \#_b(w) , \\ A \text{ erzeuge alle Worte } w \text{ mit } & \#_a(w) = \#_b(w) + 1 , \\ B \text{ erzeuge alle Worte } w \text{ mit } & \#_a(w) = \#_b(w) - 1 . \end{aligned}$$

Die resultierende Grammatik ist dann: $G = (\{S, A, B\}, \{a, b\}, P, S)$ mit

$$\begin{aligned} P = \{ & S \longrightarrow aB \mid bA \mid \varepsilon \\ & A \longrightarrow aS \mid bAA \\ & B \longrightarrow bS \mid aBB \end{aligned} \}.$$

Um zu zeigen, daß unsere Bildungsidee stimmt, daß also $L = L(G)$, nennen wir

$$\begin{aligned} L_0 & := \{w \mid \#_a(w) = \#_b(w)\} & , & \quad L_S := \{w \mid S \vdash^* w\} & , \\ L_a & := \{w \mid \#_a(w) = \#_b(w) + 1\} & , & \quad L_A := \{w \mid A \vdash^* w\} & , \\ L_b & := \{w \mid \#_b(w) = \#_a(w) + 1\} & , & \quad L_B := \{w \mid B \vdash^* w\} & . \end{aligned}$$

und zeigen, daß $L_0 = L_S$, $L_a = L_A$ und $L_b = L_B$ (Beachte: L_A, L_B, L_S sind paarweise disjunkt):

Beweis: Induktion über die Länge der Worte.

1. \subseteq : Induktionsanfang $|w| \leq 1$:
- | | | | | | |
|---------------------|-----------------------|-----|---------------------------------|------|-----------------------|
| $w = \varepsilon$: | $\varepsilon \in L_0$ | und | $S \longrightarrow \varepsilon$ | also | $\varepsilon \in L_S$ |
| $w = a$: | $a \in L_a$ | und | $A \vdash a$ | also | $a \in L_A$ |
| $w = b$: | $b \in L_b$ | und | $B \vdash b$ | also | $b \in L_B$ |

Annahme: die Behauptung gilt für die Worte einer Länge kleiner k .

$$\begin{aligned} L_0 \subseteq L_S : & \quad w \in L_0 \wedge w = av \implies v \in L_b \xrightarrow{\text{Annahme}} B \vdash^* v \implies S \vdash aB \vdash^* av = w \implies w \in L_S. \\ & \quad w \in L_0 \wedge w = bv \implies v \in L_a \implies A \vdash^* v \implies S \vdash bA \vdash^* bv = w \implies w \in L_S. \\ L_a \subseteq L_A : & \quad w \in L_a \wedge w = av \implies v \in L_0 \implies S \vdash^* v \implies A \vdash aS \vdash^* av = w \implies w \in L_A. \\ & \quad w \in L_a \wedge w = bv \implies \exists v_1 \in L_a, v_2 \in L_a : v = v_1v_2 \implies \\ & \quad \quad A \vdash^* v_1, A \vdash^* v_2 \implies A \vdash bAA \vdash^* bv_1v_2 = w. \\ L_b \subseteq L_B : & \quad w \in L_b \wedge w = av \implies \exists v_1 \in L_b, v_2 \in L_b : v = v_1v_2 \implies \\ & \quad \quad B \vdash^* v_1, B \vdash^* v_2 \implies B \vdash aBB \vdash^* av_1v_2 = w. \\ & \quad w \in L_b \wedge w = bv \implies v \in L_0 \implies S \vdash^* v \implies B \vdash bS \vdash^* bv = w. \end{aligned}$$

2. \supseteq : Induktionsanfang $|w| \leq 1$ wie oben.

Annahme: Behauptung gilt für Worte einer Länge kleiner k .

$$\begin{aligned} L_S \subseteq L_0 : & \quad S \vdash^* w, w = av \implies S \vdash aB \vdash^* av \implies v \in L_b \implies w \in L_0. \\ & \quad S \vdash^* w, w = bv \implies S \vdash bA \vdash^* bv \implies v \in L_a \implies w \in L_0. \\ L_A \subseteq L_a : & \quad A \vdash^* w, w = av \implies A \vdash aS \vdash^* av \implies v \in L_0 \implies w \in L_a \\ & \quad A \vdash^* w, w = bv \implies A \vdash bAA \implies \exists u_1, u_2 : A \vdash^* u_1 \wedge A \vdash^* u_2 \wedge v = u_1u_2 \implies \\ & \quad \quad u_1, u_2 \in L_a \implies w \in L_a \\ L_B \subseteq L_b : & \quad B \vdash^* w, w = av \implies B \vdash aBB \implies \exists u_1, u_2 : B \vdash^* u_1 \wedge B \vdash^* u_2 \wedge v = u_1u_2 \implies \\ & \quad \quad u_1, u_2 \in L_b \implies w \in L_b \\ & \quad B \vdash^* w, w = bv \implies B \vdash bS \vdash^* bv \implies v \in L_0 \implies w \in L_b. \end{aligned}$$

■

Beispiel 2.1.9

Betrachten wir noch einmal die Sprache $L = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$. Eine etwas elegantere Grammatik erhalten wir, wenn wir uns überlegen, daß wir wieder ein Wort aus L erhalten, wenn wir an ein Wort aus L links a und rechts b anfügen oder umgekehrt links b und rechts a , oder wenn wir zwei Worte aus L konkatenieren, und daß wir so alle Worte aus L aus dem leeren Wort erzeugen können. Dann bekommen wir folgende CFG: $G = (\{X\}, \{a, b\}, P, X)$ mit $P = \{X \rightarrow aXb \mid bXa \mid XX \mid \varepsilon\}$ Per Induktion über die Länge der Worte zeigen wir, daß G die Sprache L erzeugt, d.h. $L = L(G)$.

Beweis: per Induktion (alle Worte aus L haben gerade Länge):

1. \subseteq : Sei $w \in L$: Ist $w = \varepsilon$, so stimmt die Behauptung: $\varepsilon \in L$ und $\varepsilon \in L(G)$. Wir nehmen an, die Behauptung gelte für Worte bis zur Länge $2k$. Betrachten wir also Worte der Länge $2k + 2$:
 - Hat w die Form avb , so ist $v \in L$. Nach Annahme $X \vdash^* v$. Damit läßt sich w aus X ableiten: $X \vdash aXb \vdash^* avb = w$.
 - Hat w die Form bva , so gilt das Analoge.
 - Hat w die Form ava , so läßt sich w zerlegen in zwei Teile: $w = w_1w_2$ mit w_1 und $w_2 \in L(G)$ (Man gehe in w soweit, bis zum ersten Mal $\#_a = \#_b$. Eine solche Stelle muß es in v geben, da $w \in L$ und v folglich zwei b zuviel haben muß.) Nach Annahme gilt also: $X \vdash^* w_1, X \vdash^* w_2$. Damit läßt sich w aus X ableiten: $X \vdash XX \vdash^* w_1w_2 = w$.
 - Hat w die Form bvb , so gilt das Analoge.
2. \supseteq : Sei $X \vdash^* w$. Ist $w = \varepsilon$, so stimmt die Behauptung: $\varepsilon \in L$ und $\varepsilon \in L(G)$. Wir nehmen an, die Behauptung gelte für Worte bis zur Länge $2k$. Betrachten wir also Worte der Länge $2k + 2$:
 - Hat w die Form avb , so muß in der Ableitung von w die erste Regel $X \rightarrow aXb$ sein. Dann muß aber $X \vdash^* v$ gelten und nach Annahme $v \in L$ sein. Dann ist aber auch $w \in L$.
 - Hat w die Form bva , so gilt das Analoge.
 - Hat w die Form ava , so muß in der Ableitung von w die erste Regel $X \rightarrow XX$ sein. Dann muß es aber Worte $w_1, w_2 \in T^*$ geben mit $X \vdash^* w_1, X \vdash^* w_2$ und $w = w_1w_2$. Nach Annahme sind dann aber $w_1, w_2 \in L$ und folglich $w \in L$.
 - Hat w die Form bvb , so gilt das Analoge.

■

2.1.3 Ableitungsbaum

Im Folgenden benötigen wir häufig den Begriff des Baumes. Hier also eine kurze Einführung:

Definition 2.1.10 (Digraph)

Sei Q eine Menge und $K \subseteq Q \times Q$. Dann heißt die Struktur $G = (Q, K)$ *Digraph* (gerichteter Graph, engl. directed graph). Q ist die Menge der *Knoten* und K die Menge der *gerichteten Kanten* (Pfeile). Eine Abbildung $f : Q \rightarrow A$ nennen wir auch eine *Bewertung* der Knoten durch Elemente aus A . Die Struktur $G = (Q, K, f)$ nennt man demgemäß auch einen *knotenbewerteten Digraph*.

Eine Reihe von n in G aufeinanderfolgender Kanten nennt man einen *Pfad der Länge n* . Man sagt:

- p ist *Vorgänger* von q bzw. q ist *Nachfolger* von p , wenn $(p, q) \in K$.
- p ist *Vorfahre* von q bzw. q ist *Nachkomme* von p , wenn es einen Pfad von p nach q gibt.
- p ist *echter Vorfahre* von q bzw. q ist *echter Nachkomme* von p , wenn p *Vorfahre* von q bzw. q *Nachkomme* von p und $p \neq q$.

Man erkennt, daß die Vorfahrenrelation die reflexive und transitive Hülle der Vorgängerrelation ist. (Pfade können auch die Länge 0 haben).

Definition 2.1.11 (Baum)

Ein Digraph $G = (Q, K)$ ist ein (gerichteter) *Baum*, wenn gilt:

1. es gibt genau einen Knoten ohne Vorgänger, die Wurzel,
2. alle Knoten werden von der Wurzel aus durch genau einen Pfad erreicht.

Den Knoten ohne Vorgänger nennen wir die *Wurzel* und die Knoten ohne Nachfolger die *Blätter* des Baumes. Den Vorgänger eines Knoten nennen wir hier auch *Vater*, die Nachfolger auch *Söhne*. Söhne desselben Vaters heißen *Brüder*. Die *Tiefe* eines Knotens q ist die Länge des Pfades von der Wurzel zu q . Die *Höhe* des Baums ist die Länge des längsten Pfades (= maximale Tiefe eines Knotens).

Definition 2.1.12 (Teilbaum)

Ist G ein Baum, so ist G' ein Teilbaum von G , wenn G' ein Baum ist, und alle Knoten und Kanten von G' auch Knoten bzw. Kanten von G sind.

Wenn wir einen Baum an die Tafel oder auf Papier zeichnen, so müssen wir zwei Brüder notgezwungen in eine Reihenfolge bringen, den einen links vom anderen zeichnen, wodurch der Eindruck entsteht, als sei der eine Bruder in irgendeiner Hinsicht vor dem anderen. Diese durch die Zeichnung erzwungene Ordnung ist aber nicht durch die Definition selbst gegeben. Wollen wir eine solche Ordnung festlegen, so müssen wir sie gesondert definieren.

Definition 2.1.13 (orientierter Baum)

Ein Baum G ist ein *orientierter Baum*, wenn die Söhne jedes Knotens linear geordnet sind ($<_B$). Diese Brüderordnung kann man in verschiedener Weise zu einer Ordnung aller Knoten fortsetzen:

Depth First: $a <_{DF} b$ gdw.

1. a ist echter Vorfahre von b oder
2. es gibt Vorfahren a' von a und b' von b mit $a' <_B b'$.

Breadth First: $a <_{BF} b$ gdw.

1. Tiefe (a) $<$ Tiefe(b) oder
2. Tiefe (a) = Tiefe (b) und es gibt Vorfahren a' von a und b' von b mit $a' <_B b'$.

Definition 2.1.14 (Front)

In einem orientierten knotenbewerteten Baum, nennen wir das Wort der Bewertungen der Blätter in ihrer depth-first Reihenfolge die *Front* des Baums.

Definition 2.1.15 (Teilbaum eines Knotens)

Ist G ein (orientierter) Baum, so begründet jeder Knoten a von G einen orientierten Teilbaum von G , nämlich den Baum $G(a)$ aller Nachfahren von a in G (der maximale Teilbaum mit der Wurzel a).

Definition 2.1.16 (binärer, (fast) vollständiger Baum)

Ein Baum ist *binär*, wenn jeder Knoten höchstens 2 Söhne hat. Ein binärer Baum der Höhe n heißt *vollständig*, wenn alle Knoten einer Tiefe $< n$ genau 2 Söhne haben. Er heißt *fast vollständig*, wenn gilt

1. alle Knoten einer Tiefe $< n - 1$ haben genau 2 Söhne,
2. hat ein Knoten p der Tiefe $n - 1$ weniger als 2 Söhne, so hat kein Knoten q mit $q \geq_{BF} p$ einen Sohn.

Lemma 2.1.17

Ein binärer Baum der Höhe n hat höchstens $2^{n+1} - 1$ Knoten.

Beweis: Unter den binären Bäumen der Höhe n hat der vollständige die meisten Knoten. Er hat 2^i Knoten der Tiefe i , also insgesamt $2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^n = 2^{n+1} - 1$ Knoten. ■

$$\underbrace{2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^n}_{2^2-1} = 2^{n+1} - 1$$

$$\underbrace{\hspace{10em}}_{2^3-1}$$

$$\underbrace{\hspace{15em}}_{2^4-1 \dots etc}$$

Anmerkung 2.1.18

Die Zahl der Blätter ist beim vollständig binären Baum um 1 größer als die Zahl der übrigen Knoten.

Eine CFG ist ein "nichtdeterministischer" Kalkül: es ist i. Allg. möglich eine von mehreren Regeln anzuwenden und dies an mehreren Orten des abzuleitenden Wortes. Allerdings ist es unwesentlich, in welcher Reihenfolge wir die Variablen behandeln, erst ein weiter links stehendes oder erst ein weiter rechts stehendes, solange die Variablen jeweils mit derselben Regel abgeleitet werden. Um einer Ableitung diese unnötige Mehrdeutigkeit zu nehmen, wollen wir alle Ableitungen als äquivalent ansehen, die sich nur in der Reihenfolge der Anwendung der Regeln unterscheiden. Repräsentant für ein Klasse äquivalenter Ableitungen ist die Linksableitung, in der in jedem Ableitungsschritt, immer das jeweils linke Nonterminal abgeleitet wurde.

Definition 2.1.19 (Ableitungsbaum)

Ein orientierter knotenbewerteter Baum Γ ist ein Ableitungsbaum des Wortes w bzgl. $G = (N, T, P, S)$, wenn gilt:

1. Die Wurzel von Γ ist mit S bewertet, die Blätter mit ε oder mit Terminals und die übrigen Knoten mit Nonterminals.
2. Ist ein Knoten in Γ mit A bewertet und seine k Söhne in ihrer Reihenfolge mit B_1, \dots, B_k , so ist $A \rightarrow B_1 \dots B_k$ eine Regel aus P .
3. Ein mit ε bewertetes Blatt hat keine Brüder.
4. w ist die Front von Γ .

Ein Ableitungsbaum entspricht gerade einer Klasse äquivalenter Ableitungen bzw. einer Linksableitung, wie folgendes Lemma sagt:

Lemma 2.1.20

Zu jeder Linksableitung gehört genau ein Ableitungsbaum und umgekehrt.

Definition 2.1.21

- Eine CFG G heißt *eindeutig*, gdw. für alle $w \in L(G)$ genau ein Ableitungsbaum existiert.
- Eine CFG G heißt *mehrdeutig*, wenn sie nicht eindeutig ist.
- CFL L heißt *eindeutig*, wenn es eine eindeutige Grammatik für L gibt.
- CFL L heißt *inherent mehrdeutig*, gdw. L nicht eindeutig.

Aufgabe 2.1.22

Zeigen Sie: Die CFG $G = (\{X\}, \{a, b\}, \{X \rightarrow aXb \mid bXa \mid XX \mid \varepsilon\}, X)$ ist nicht eindeutig.

2.2 Rechtlineare Grammatiken

Definition 2.2.1 (rechtslineare Grammatik)

Ein CFG $G = (N, T, P, S)$ heißt *rechtslinear* gdw. $P \subseteq N \times (T \cup N \cup TN \cup \{\varepsilon\})$, d.h. die Regeln haben die Form $A \rightarrow \varepsilon$, $A \rightarrow a$, $A \rightarrow B$ oder $A \rightarrow aB$ ($A, B \in N$, $a \in T$).

Beispiel 2.2.2

$$\begin{aligned}
 L &= \{ab, aba\}^* \\
 G &= (\{S, B, C\}, \{a, b\}, P, S) \\
 P &= S \longrightarrow aB \mid \varepsilon \\
 &\quad B \longrightarrow bS \mid bC \\
 &\quad C \longrightarrow aS
 \end{aligned}$$

eine Folge von Ableitungsworten wäre z.B.: $S, aB, abC, abaS, abaaB, abaabS, abaab$

Satz 2.2.3

Zu jedem FA existiert eine äquivalente rechtslineare Grammatik und umgekehrt.

Beweis:

I. Gegeben sei ein FA $M = (Q, \Sigma, \delta, q_0, F)$ ohne ε -Befehle. Wir definieren die CFG $G = (Q, \Sigma, P, q_0)$ mit $P = \{p \longrightarrow aq \mid (p, a, q) \in \delta\} \cup \{q \longrightarrow \varepsilon \mid q \in F\}$.

II. Gegeben sei eine rechtslineare CFG $G = (N, T, P, S)$. Wir definieren dazu den FA $M = (N \cup \{E\}, T, \delta, S, \{E\})$ ($E \notin N$) mit:

$$\begin{aligned}
 (A, a, B) \in \delta &\iff A \longrightarrow aB \quad \text{aus } P \\
 (A, \varepsilon, B) \in \delta &\iff A \longrightarrow B \quad \text{aus } P \\
 (A, a, E) \in \delta &\iff A \longrightarrow a \quad \text{aus } P \\
 (A, \varepsilon, E) \in \delta &\iff A \longrightarrow \varepsilon \quad \text{aus } P
 \end{aligned}$$

Beispiel 2.2.4

FA:	<table style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="padding: 5px;">δ</td> <td style="padding: 5px;">a</td> <td style="padding: 5px;">b</td> </tr> <tr> <td style="padding: 5px;">SF</td> <td style="padding: 5px;">1</td> <td style="padding: 5px;">2 -</td> </tr> <tr> <td></td> <td style="padding: 5px;">2</td> <td style="padding: 5px;">- 1,3</td> </tr> <tr> <td></td> <td style="padding: 5px;">3</td> <td style="padding: 5px;">1 -</td> </tr> </table>	δ	a	b	SF	1	2 -		2	- 1,3		3	1 -	CFG:	<table style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="padding: 5px;">S</td> <td style="padding: 5px;">1</td> <td style="padding: 5px;">\longrightarrow</td> <td style="padding: 5px;">ε</td> </tr> <tr> <td></td> <td style="padding: 5px;">1</td> <td style="padding: 5px;">\longrightarrow</td> <td style="padding: 5px;">$a2$</td> </tr> <tr> <td></td> <td style="padding: 5px;">2</td> <td style="padding: 5px;">\longrightarrow</td> <td style="padding: 5px;">$b1 \mid b3$</td> </tr> <tr> <td></td> <td style="padding: 5px;">3</td> <td style="padding: 5px;">\longrightarrow</td> <td style="padding: 5px;">$a1$</td> </tr> </table>	S	1	\longrightarrow	ε		1	\longrightarrow	$a2$		2	\longrightarrow	$b1 \mid b3$		3	\longrightarrow	$a1$
δ	a	b																													
SF	1	2 -																													
	2	- 1,3																													
	3	1 -																													
S	1	\longrightarrow	ε																												
	1	\longrightarrow	$a2$																												
	2	\longrightarrow	$b1 \mid b3$																												
	3	\longrightarrow	$a1$																												

Beispiel 2.2.5

CFG:	$S \longrightarrow 0S \mid 1S \mid 1$	FA:	<table style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="padding: 5px;">δ</td> <td style="padding: 5px;">0</td> <td style="padding: 5px;">1</td> </tr> <tr> <td style="padding: 5px;">S</td> <td style="padding: 5px;">S</td> <td style="padding: 5px;">S, E</td> </tr> <tr> <td style="padding: 5px;">F</td> <td style="padding: 5px;">E</td> <td style="padding: 5px;">- -</td> </tr> </table>	δ	0	1	S	S	S, E	F	E	- -
δ	0	1										
S	S	S, E										
F	E	- -										

Definition 2.2.6 (linkslineare Grammatik)

Seien N, T disjunkte endliche Mengen und $P \subseteq N \times (T \cup N \cup NT \cup \{\varepsilon\})$, $S \in N$. Dann heißt die Struktur $G = (N, T, P, S)$ *linkslineare* Grammatik. (Die Regeln von G haben also die Form $A \longrightarrow \varepsilon$, $A \longrightarrow a$, $A \longrightarrow B$ oder $A \longrightarrow Ba$ ($A, B \in N$, $a \in T$)).

Satz 2.2.7

Zu jeder rechtslinearen Grammatik existiert eine äquivalente linkslineare und umgekehrt.

Beweis: Sei $G = (N, T, P, S)$ eine rechtslineare Grammatik.

1. Vertausche die Symbole auf den rechten Regelseiten: ist $A \longrightarrow aB$ aus P , so sei $A \longrightarrow Ba$ aus P' . Außerdem enthält P' alle Regeln aus P mit einer rechten Seite einer Länge < 2 . Die linkslineare Grammatik $G' = (N, T, P', S)$ erzeugt die Sprache $L(G') = \overleftarrow{L(G)}$ ($L(G)$ gespiegelt).
2. Vertausche die Variablen der linken und rechten Seite einer Regel ($S'' \notin N$):

$$\begin{aligned}
 A \longrightarrow Ba \quad \text{aus } P' &\implies B \longrightarrow Aa \quad \text{aus } P'' \\
 A \longrightarrow B \quad \text{aus } P' &\implies B \longrightarrow A \quad \text{aus } P'' \\
 A \longrightarrow a \quad \text{aus } P' &\implies S'' \longrightarrow Aa \quad \text{aus } P'' \\
 A \longrightarrow \varepsilon \quad \text{aus } P' &\implies S'' \longrightarrow A \quad \text{aus } P'' \\
 \text{Außerdem sei} & \implies S \longrightarrow \varepsilon \quad \text{aus } P''
 \end{aligned}$$

Die linkslineare Grammatik $G'' = (N \cup \{S''\}, T, P'', S'')$ erzeugt $L(G'') = \overleftarrow{L(G')} = L(G)$

Die Umkehrung wird analog bewiesen. ■

Beispiel 2.2.8

$M :$	δ	a	b	c
S	S	A	$-$	$-$
F	A	$-$	B	$-$
	B	$-$	$-$	A

$G :$	$S \longrightarrow aA$	z.B. $S \vdash aA, A \vdash \varepsilon$	$R(G) = a(bc)^*$
	$A \longrightarrow bB \mid \varepsilon$	$A \vdash bB \vdash bcA$	
	$B \longrightarrow cA$		

$G' :$	$S \longrightarrow Aa$	z.B. $S \vdash Aa, A \vdash \varepsilon$	$R(G') = (cb)^*a$
	$A \longrightarrow Bb \mid \varepsilon$	$A \vdash Bb \vdash Acb$	
	$B \longrightarrow Ac$		

$G'' :$	$S'' \longrightarrow A$	z.B. $S'' \vdash A$	$R(G'') = a(bc)^*$
	$S \longrightarrow \varepsilon$	$A \vdash Sa \vdash a$	
	$A \longrightarrow Sa \mid Bc$	$A \vdash Bc \vdash Abc$	
	$B \longrightarrow Ab$		

2.3 Chomsky-Normalform

Lemma 2.3.1

Zu jeder CFG G mit $\varepsilon \notin L(G)$ existiert eine äquivalente CFG G' , die keine ε -Regeln enthält (d.h. Regeln der Form $A \longrightarrow \varepsilon$).

Beweis: I. Sei $G = (N, T, P, S)$. Wir definieren eine Operation auf P : Man nehme eine ε -Regel $A \longrightarrow \varepsilon$ aus P und ersetze jede Regel R mit k Vorkommen von A auf der rechten Seite durch die 2^k neuen Regeln, die man erhält, wenn man auf der rechten Seite von R die Vorkommen von A beläßt oder löscht. Man bekommt die Hülle von P unter dieser Operation, wenn man diesen Schritt solange iteriert, bis keine neue Regel mehr entsteht. Dann löscht man alle ε -Regeln.

Die Hüllenbildung kann man z.B. so beschreiben: Die Variablen von N seien nummeriert: A_1, \dots, A_k .

```

begin
  P' = ∅;
  while P' ≠ P do begin
    P' = P;
    for i = 1 to k do
      if Ai → ε aus P do
        for all Regeln B → B1 ··· Bl ∈ P do
          P = P ∪ {B → B'1 ··· B'l | B'j = Ai ∨ ε, falls Bj = Ai, und B'j = Bj sonst}
    end;
  end;
end.
    
```

Bevor man die ε -Regeln löscht, wird sicher jedes Wort, daß von P erzeugt wird, auch jetzt erzeugt und umgekehrt, wird jedes Wort, was mithilfe einer neuen Regel erzeugt wird auch nur mit den alten erzeugt. Nach dem Entfernen der ε -Regeln kann man aber immer noch alle früher erzeugten Worte

ableiten, denn wird in einer Ableitung das Vorkommen einer Variablen B mit einer ε -Regel gelöscht, und wurde dieses Vorkommen durch die Regel $A \rightarrow UBV$ gesetzt, so hätte man statt dieser Regel auch die neue Regel $A \rightarrow UV$ anwenden können, und wäre so auf dasselbe Ergebnis gekommen.

II. Eine andere Variante für den Beweis ist die folgende: Wir nennen eine Variable A *nullierbar* (in G), wenn $A \vdash_G^* \varepsilon$. Zunächst finden wir alle nullierbaren Variablen:

```
begin
  for all  $A \in N$  do if  $A \rightarrow \varepsilon$  aus  $P$  then markiere  $A$ ;
  while neue Variable wurde markiert do
    for all  $A \rightarrow B_1 \dots B_k$  aus  $P$  do
      if  $B_1, \dots, B_k$  markiert, then markiere  $A$ ;
end.
```

Dann ersetzt man für jede nullierbare Variable A jede Regel R mit m Vorkommen von A auf der rechten Seite durch die 2^m neuen Regeln, die man erhält, wenn man auf der rechten Seite von R die Vorkommen von A beläßt oder löscht. D.h. Alle Variablen auf den rechten Seiten von Regeln, aus denen das leere Wort ε abgeleitet werden kann, können auch gleich weggelassen werden. Da man aus solchen Variablen aber auch anderes ableiten könnte, müssen sie in einer anderen Kopie dieser Regeln auch noch vorkommen.

Sei n die Zahl der Variablen, k die Zahl der Regeln und l die Länge der längsten Regel. Dann dauert das Markieren der nullierbaren Variablen höchstens nkl Schritte und Teil 2 des Verfahrens $k2^l$ Schritte. Insgesamt ist der Zeitaufwand dieses Verfahrens kleiner als $k^2l + k2^l$ Schritte ($n < k$). ■

Korollar 2.3.2

Zu jeder CFG G existiert eine äquivalente CFG G' , die keine ε -Regeln enthält außer eventuell der Regel $S \rightarrow \varepsilon$. Enthält sie die Regel $S \rightarrow \varepsilon$, so taucht das Startsymbol S nie auf der rechten Seite einer Regel auf.

Beweis: Falls $\varepsilon \in L(G)$, würde das Verfahren des vorigen Beweises zu einer CFG G' führen, die $L(G)$ nicht erzeugt, da sie ε nicht erzeugt. Ist S das Startsymbol von G' , führen wir daher ein neues Startsymbol S' ein und die beiden Regeln $S' \rightarrow S \mid \varepsilon$. ■

Lemma 2.3.3

Zu jeder CFG G mit $\varepsilon \notin L(G)$ existiert eine äquivalente CFG $G' = (N, T, P, S)$, die keine Regel aus $N \times (\{\varepsilon\} \cup N)$ enthält, also keine Regeln der Form $A \rightarrow \varepsilon$ und $A \rightarrow B$ ($A, B \in N$).

Beweis: Nach dem vorigen Lemma gibt es eine zu G äquivalente Grammatik G' ohne ε -Regeln. Wir müssen noch die unären Regeln entfernen, d.h. Regeln der Form $A \rightarrow B$. Wir definieren wieder eine Operation auf der Regelmenge P' von G' : Man nehme eine Regel der Form $A \rightarrow B$ aus P und füge zu jeder Regel $B \rightarrow W$ aus P' noch die Regel $A \rightarrow W$ hinzu. Dies wiederholen wir solange, bis wir keine neue Regel mehr bekommen, d.h. wir bilden die Hülle von P' unter dieser Operation. Danach löschen wir alle Regeln der Form $A \rightarrow B$. Sei P'' die so erhaltene Regelmenge. Dann ist die Grammatik G'' , die man aus G' erhält, wenn man ihre Regelmenge P' durch P'' ersetzt, äquivalent zu G' und damit zu G .

Um zu sehen, daß die Grammatik G'' äquivalent ist zu G' muß man sich nur überlegen, daß die Anwendung einer Operation, die von den Regeln erzeugte Sprache gleich läßt. Haben wir eine Regel der Form $A \rightarrow B$ aus P und füge wir zu jeder Regel $B \rightarrow W$ noch die Regel $A \rightarrow W$ hinzu, so werden sicher nicht weniger Worte erzeugt. Ist w aber ein Wort, das mithilfe einer neuen Regel $A \rightarrow W$ erzeugt würde, so könnte man diese aber auch immer durch die Regelfolge $A \rightarrow B$,

$B \rightarrow W$ ersetzen, d.h. das Wort würde auch mithilfe der alten Regeln erzeugt. Somit wird auch kein weiteres Wort erzeugt.

Dann überlegt man sich, daß auch das Entfernen der Regeln der Form $A \rightarrow B$ am Schluß nichts ausmacht. Nehmen wir an, das Wort w wird mithilfe von Regeln der Form $A \rightarrow B$ erzeugt und sei $A_1 \vdash A_2 \vdash \dots \vdash A_l$ eine maximale Ableitungsfolge nur durch unäre Regeln, d.h. A_l wird durch eine Regel weiter abgeleitet, die keine unäre Regel ist, z.B. $A_l \rightarrow W$, so gibt es in P'' auch eine Regel $A_1 \rightarrow W$, die die Anwendung aller unären Regeln in obiger Ableitung erübrigt.

Schließlich überlegt man sich noch, daß die Hüllenbildung ein endender Prozeß ist: hat G' etwa k Variablen, so können höchstens k^2 viele unäre Regeln in P' sein oder durch unsere Operationen erzeugt werden. Haben wir alle unären Regeln erzeugt, d.h. bekommen wir durch Anwendung einer Operation keine weitere unäre Regel, so genügt es unsere Operation höchstens noch einmal für jede dieser unären Regeln anzuwenden und wir sind fertig. ■

Definition 2.3.4

Die CFG $G = (N, T, P, S)$ hat *Chomsky-Normalform (CNF)*, wenn

$$P \subseteq N \times (T \cup (N - \{S\})^2) \cup \{S \rightarrow \varepsilon\},$$

d.h. wenn alle Regeln die Form haben

$$A \rightarrow BC \text{ oder } A \rightarrow a \text{ oder } S \rightarrow \varepsilon \quad (A \in N, B, C \in N - \{S\}, a \in T).$$

Satz 2.3.5

Zu jeder CFG G mit $\varepsilon \notin L(G)$ existiert eine äquivalente CFG in CNF.

Beweis: Nach Lemma 2.3.3, existiert zu einer CFG $G = (N, T, P, S)$ eine äquivalente CFG $G' = (N, T, P', S)$ ohne ε -Regeln oder Regeln der Form $A \rightarrow B$ (A, B Variablen). Sei $N' = N \cup \{C_a \mid a \in T\}$ eine neue Variablenmenge.

Ersetze in jeder Regel aus P' , deren rechte Seite eine Länge größer 1 hat, alle Terminals a durch die zugeordneten neuen Variablen C_a und füge die Regeln $C_a \rightarrow a$ ($a \in T$) zu P' hinzu. Sei l die maximale Regellänge von Regeln aus P' und sei $N'' = N' \cup \{[A_1 \dots A_k] \mid A_i \in N' \text{ für } i = 1, \dots, k \text{ und } 1 < k < l\}$ wieder eine neue Variablenmenge. Jede Regel aus P' , die noch nicht die richtige Form hat, also von der Form $A \rightarrow B_1 \dots B_k$ ist ($k > 2, A, B_1, \dots, B_k \in N'$), wird jetzt ersetzt durch die Regeln:

$$\begin{aligned} A &\rightarrow B_1[B_2 \dots B_k] \\ [B_2 \dots B_k] &\rightarrow B_2[B_3 \dots B_k] \\ &\vdots \\ [B_i \dots B_k] &\rightarrow B_i[B_{i+1} \dots B_k] \\ &\vdots \\ [B_{k-1} B_k] &\rightarrow B_{k-1} B_k \end{aligned}$$

Korollar 2.3.6

Zu jeder CFG G mit $\varepsilon \notin L(G)$ existiert eine äquivalente CFG in CNF.

Beweis:

1. $\varepsilon \notin L(G)$ wie oben.

2. $\varepsilon \in L(G) =: L$. Sei $G' = (N, T, P, S)$ eine CFG in CNF für $L - \{\varepsilon\}$.
 Wir bilden $G'' = (N \cup \{S'\}, T, P', S')$ für L mit $S' \notin N$ (mit S' neuer Startvariable) und
 $P' = P \cup \{S' \rightarrow \varepsilon\} \cup \{S' \rightarrow AB \mid S \rightarrow AB \in P\}$.

■

Beispiel 2.3.7

$\{S \rightarrow aSb \mid ab\}$ erzeugt die Sprache $L = \{a^n b^n \mid n > 0\}$.

Terminalbehandlung: $S \rightarrow C_a S C_b \mid C_a C_b, C_a \rightarrow a, C_b \rightarrow b$.

Behandlung langer Regeln : $S \rightarrow C_a S_1 \mid C_a C_b, S_1 \rightarrow S C_b, C_a \rightarrow a, C_b \rightarrow b$.

Die Variablen $[S C_b]$ haben wir hier in S_1 umbenannt.

(passend zur Definition: $S \rightarrow C_a S_1 \mid C_a C_b, S_1 \rightarrow S' C_b, C_a \rightarrow a, C_b \rightarrow b, S' \rightarrow C_a S_1 \mid C_a C_b$.)

2.4 Pumpinglemma, Ogdenlemma

Definition 2.4.1 (Pumpingeigenschaft)

$L \subseteq \Sigma^*$ hat die Pumping-Eigenschaft (i.Z. PE) gdw.

$$\begin{array}{l} \exists k \in \mathbb{N} \quad \forall z \in L \quad \exists u, v, w, x, y \in \Sigma^* \quad \forall i \in \mathbb{N} : uv^iwx^iy \in L \\ |z| \geq k \quad z = uvwxy \\ |vwx| \leq k \\ vx \neq \varepsilon \end{array}$$

In Worten: Es gibt eine natürliche Zahl k , so daß es für alle genügend langen (wenigstens k langen) Worte z aus der Sprache L eine Zerlegung $z = uvwxy$ gibt mit folgenden Eigenschaften:

1. die mittleren Teile v, w, x zusammen haben höchstens die Länge k (liegen k -nah zusammen),
2. nicht beide "Pumpstellen" v und x sind leer,
3. die "Pumpstellen" v und x können synchron gepumpt werden, d.h. gelöscht oder gleich oft eingefügt werden, ohne daß das Ergebnis aus L herausfällt.

Satz 2.4.2 (Pumpinglemma)

Jede CFL hat die Pumpingeigenschaft.

Beweis: Sei L eine CFL und $G = (N, T, P, S)$ eine CFG in CNF für L mit $|N| = m$. Wir wählen $k = 2^m$ und zeigen, daß es für jedes wenigstens k -lange $z \in L$ eine Zerlegung $z = uvwxy$ gibt, die die Eigenschaften 1) bis 3) hat. Sei also $z = a_1 \cdots a_n$ ($a_i \in \Sigma$ für alle $i = 1, \dots, n$) irgendein Wort aus L von wenigstens der Länge k . Zu z gibt es einen Ableitungsbaum Γ in G . Nun wählen wir eine Knotenfolge $\pi = q_0, \dots, q_r$ in Γ mit

1. q_0 ist Wurzel von Γ ,
2. q_r ist Vater eines Blattes von Γ ,
3. q_{i+1} ist der Sohn von q_i , der den Teilbaum mit den meisten Blättern begründet. Haben die Teilbäume beider Söhne gleichviel Blätter, so sei q_{i+1} der linke Sohn von q_i . ($i = 0, \dots, r - 1$)

Die Zahl der Blätter vom Vater zum Sohn halbiert sich höchstens, d.h. hat ein Knoten q_i in π noch b Blätter unter sich, so hat der Sohn q_{i+1} mindestens noch $b/2$ Blätter unter sich. q_r hat aber nur noch ein Blatt unter sich. Da man k aber mindestens m -mal halbieren muß, um auf 1 zu kommen, muß $r \geq m$ sein.

Wir betrachten nun die $m + 1$ letzten Knoten in π (also von hinten gezählt) p_1, \dots, p_{m+1} . Da wir nur m Variablen haben und alle Knoten mit Variablen bewertet sind, muß es unter den $m + 1$ Knoten p_1, \dots, p_{m+1} wenigstens zwei geben, die mit derselben Variablen bewertet sind. Seien es p_i und p_j

($i < j$) und die von ihnen begründeten Teilbäume in Γ nennen wir $\Gamma(p_i)$ und $\Gamma(p_j)$. Die Front von $\Gamma(p_j)$ sei w , die von $\Gamma(p_i)$ sei vwx (sie umfaßt ja die Front von $\Gamma(p_j)$) und schließlich seien u, y die restlichen Teile von z , sodaß $z = uvwxy$. Dann gilt:

1. $\Gamma(p_1)$ enthält höchstens 2^m Blätter, da nach p_1 in π höchstens noch m Knoten kommen, so daß sich die Zahl der unter p_1 liegenden Blätter höchstens noch m -mal halbieren läßt.
2. p_{i+1} ist ein Sohn von p_i . Sei p'_{i+1} der andere Sohn. Auch dieser Sohn hat ein Blatt unter sich (es gibt keine ε -Regeln). Eine der beiden Pumpstellen (v oder x) umfaßt die Front von $\Gamma(p'_{i+1})$ (ist p_{i+1} der linke Sohn von p_i so umfaßt x die Front von $\Gamma(p'_{i+1})$, andernfalls umfaßt v diese Front).
3. Ersetzen wir den Teilbaum $\preceq(p_i)$ durch den Teilbaum $\Gamma(p_j)$, so erhalten wir einen Ableitungsbaum Γ_0 für $uv^0wx^0y = uwy$. Ersetzen wir umgekehrt in Γ den Teilbaum $\Gamma(p_j)$ durch den Teilbaum $\preceq(p_i)$, so erhalten wir einen Ableitungsbaum Γ_2 für das Wort uv^2wx^2y . Dies letztere können wir wiederholt tun, sagen wir i -mal und erhalten einen Ableitungsbaum Γ_i für das Wort uv^iwx^iy .

■

Beispiel 2.4.3

1. $\{a^n b^n \mid n \in \mathbb{N}\}$ hat die Pumpingeigenschaft.
2. $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ hat nicht die Pumpingeigenschaft.

Leider ist das Pumpinglemma nicht umkehrbar. Es gibt Sprachen, die die Pumping-Eigenschaft haben und dennoch nicht kontextfrei sind.

Beispiel 2.4.4

1. $\{a^n b^n c^n d^m \mid n, m \in \mathbb{N}\} \cup L(a^* b^* c^*)$ hat die Pumpingeigenschaft. Man pumpt im d -Teil, solange vorhanden, sonst wo man will. Der kritische Fall ist $m = 1$: pumpt man das letzte d heraus, so fällt man in die Wortmenge $L(a^* b^* c^*)$, wo nun beliebig gepumpt werden darf.
2. $\{a^n b^n c^m \mid n, m \in \mathbb{N}, n \neq m\}$ hat die Pumpingeigenschaft. Wir setzen $k = 7$ und unterscheiden 3 Fälle: Bei $n > m + 1$ pumpen wir in $a^n b^n$ das letzte a und das erste b . Bei $n < m - 1$ pumpen wir in c^m das letzte c . Wenn $n = m + 1$ oder $n = m - 1$, so pumpen wir in c^m die letzten beiden c . Man überlege sich, daß alle Wörter einer Länge größergleich 7 pumpbar sind.

Daß diese beiden Sprachen nicht kontextfrei sind, werden wir später sehen. Ein Grund, warum man 1. pumpen kann, liegt offenbar darin, daß wir eine an sich nicht pumpbare Sprache mit einer künstlichen Pumpstelle versehen haben. 2. ist auch nur pumpbar, weil wir frei sind, die Stelle, wo gepumpt werden soll, zu wählen.

Wir können das Lemma aber verschärfen, sodaß man die Pumpstellen bis zu einem gewissen Grade einschränken kann:

Definition 2.4.5

(k -Punktierung) Sei Σ ein endliches Alphabet und $\dot{\Sigma} = \{\dot{a} \mid a \in \Sigma\}$ und h ein Homomorphismus: $(\Sigma \cup \dot{\Sigma})^* \rightarrow \Sigma^*$ vermöge $h(\dot{a}) = h(a) = a$ für alle $a \in \Sigma$. Dann heißt \dot{w} eine k -Punktierung von $w \in \Sigma^*$, wenn $h(\dot{w}) = w$ und \dot{w} mindestens k punktierte Symbole aus $\dot{\Sigma}$ enthält.

Definition 2.4.6

$L \subseteq \Sigma^*$ hat die Ogden-Eigenschaft gdw.

$$\begin{array}{ccccccc} \exists & \forall & \forall & \exists & & \forall & : \quad wv^iwx^i y \in L \\ k \in \mathbb{N} & z \in L & k\text{-Pktng} & \dot{u}, \dot{v}, \dot{w}, \dot{x}, \dot{y} \in (\Sigma \cup \dot{\Sigma})^* \text{ mit} & & i \in \mathbb{N} & \\ & |z| \geq k & \dot{z} \text{ von } z & \dot{z} = \dot{u}\dot{v}\dot{w}\dot{x}\dot{y}, & & & \\ & & & \dot{v}\dot{w}\dot{x} \text{ hat h\u00f6chstens } k \text{ Punkte und} & & & \\ & & & \dot{u}, \dot{v}, \dot{w} \text{ oder } \dot{w}, \dot{x}, \dot{y} \text{ haben je wenigstens einen Punkt} & & & \\ & & & (u = h(\dot{u}), v = h(\dot{v}), w = h(\dot{w}), x = h(\dot{x}), y = h(\dot{y})). & & & \end{array}$$

In Worten: Es gibt eine nat\u00fcrliche Zahl k , so da\u00df es f\u00fcr alle gen\u00fcgend langen (wenigstens k langen) Worte z aus der Sprache L und zu jeder k -Punktierung (Verteilung von wenigstens k Punkten) \dot{z} von z eine Zerlegung $\dot{z} = \dot{u}\dot{v}\dot{w}\dot{x}\dot{y}$ von \dot{z} gibt mit folgenden Eigenschaften:

1. die mittleren Teile $\dot{v}, \dot{w}, \dot{x}$ zusammen haben h\u00f6chstens k Punkte (liegen k -punkte-nah zusammen),
2. die ersten drei Teile \dot{u}, \dot{v} , und \dot{w} oder die letzten drei Teile \dot{w}, \dot{x} und \dot{y} haben je mindestens einen Punkt (sind nicht punkt-leer), (das "oder" ist nicht ausschlie\u00dfend.),

so da\u00df die "Pumpstellen" v und x synchron gepumpt werden k\u00f6nnen (d.h. gel\u00f6scht oder gleich oft eingef\u00fcgt) ohne da\u00df das Ergebnis aus L herausf\u00e4llt.

Jetzt stellen wir fest, da\u00df die beiden Sprachen aus Beispiel 2.4.4, obwohl sie pumpbar sind, die Ogden-eigenschaft nicht besitzen:

Beispiel 2.4.7

1. $\{a^n b^n c^n d^m \mid n, m \in \mathbb{N}\} \cup \{a^* b^* c^*\}$ hat nicht die Ogden-eigenschaft.
2. $\{a^n b^n c^m \mid n, m \in \mathbb{N}, n \neq m\}$ hat nicht die Ogden-eigenschaft.

Beweis:

ad 1) Wir legen alle Punkte auf den a -Teil.

ad 2) Sei $k \in \mathbb{N}$ und $n > k$. Betrachte $a^{n+n!} b^{n+n!} c^n$ und eine Punktierung mit Punkten nur auf dem c -Teil. Eine der Pumpstellen - v oder x - hat einen Punkt. Da auch w einen Punkt haben mu\u00df, liegt x ganz im c -Teil. Liegt v nicht ganz im c -Teil, so kann man nicht pumpen (aus demselben Grund, aus dem $\{a^n b^n \mid n \in \mathbb{N}\}$ nicht die regul\u00e4re Pumpingeigenschaft hat). Liegt v aber ganz im c -Teil und hat vx die L\u00e4nge l , so erreichte man durch pumpen auch die Worte $a^{n+n!} b^{n+n!} c^{n+il}$ f\u00fcr alle $i \in \mathbb{N}$, also auch $a^{n+n!} b^{n+n!} c^{n+n!}$ ($i = n!/l$), was nicht in der Sprache liegt. W! ■

Satz 2.4.8 (Ogden-Lemma)

Jede CFL L hat die Ogden-eigenschaft.

Beweis: Sei L eine CFL, $G = (N, T, P, S)$ eine CFG f\u00fcr L in CNF und $|N| = m$. Wir w\u00e4hlen $k = 2^{2m+3}$, und zeigen, da\u00df es f\u00fcr jedes wenigstens k -lange $z \in L$ und jede k -Punktierung \dot{z} von z eine Zerlegung $\dot{z} = \dot{u}\dot{v}\dot{w}\dot{x}\dot{y}$ von \dot{z} gibt, die die Eigenschaften 1) und 2) aus der Definition der Ogden-Eigenschaft hat.

Sei also z irgendein Wort aus L von wenigstens der L\u00e4nge k und \dot{z} irgend eine Punktierung von z mit wenigstens k Punkten. Zu z gibt es einen Ableitungsbaum Γ in G . $\dot{\Gamma}$ sei der Baum, der aus Γ entsteht, wenn man die Blattlabels gem\u00e4\u00df \dot{z} punktiert. Nun w\u00e4hlen wir eine Knotenfolge $\pi = q_1, \dots, q_r$ in $\dot{\Gamma}$ mit

1. $q_1 =$ Wurzel von $\dot{\Gamma}$,
2. $q_r =$ Blatt von $\dot{\Gamma}$,
3. q_{i+1} ist der Sohn von q_i , dessen Teilbaum $\dot{\Gamma}(q_{i+1})$ die meisten punktierten Bl\u00e4tter enth\u00e4lt ($i = 1, \dots, r - 1$).

Wir erinnern uns, da\u00df jeder Knoten q eines Baumes den Teilbaum seiner Nachfahren begr\u00fcndet.

- q_i heißt *Verzweigungsknoten* in π , wenn beide Söhne Teilbäume mit punktierten Blättern begründen (wenn unter beiden Söhnen Punkte liegen).
- q_i heißt *Linksverzweigungsknoten* in π , wenn q_i Verzweigungsknoten ist und q_{i+1} linker Sohn von q_i ist (d.h. wenn wir in π von q_i aus nach links gegangen sind, weil der linke Sohn mehr Punkte unter sich hatte.)
- q_i heißt *Rechtsverzweigungsknoten*, wenn q_i Verzweigungsknoten ist und q_{i+1} rechter Sohn von q_i ist.

Feststellung: π enthält mindestens $2m + 3$ Verzweigungsknoten.

Begründung: In $\dot{\Gamma}$ liegen unter der Wurzel q_1 wenigstens $k = 2^{2m+3}$ Punkte. Laufen wir den Pfad π entlang, so liegen unter den erreichten Knoten immer weniger Punkte, bis zuletzt der Knoten q_{r-1} erreicht ist, der genau einen Punkt unter sich hat (nämlich das punktierte Blatt q_r). Allerdings wird sich die Punktzahl nicht verringern, wenn wir von einem Knoten, der nicht ein Verzweigungsknoten ist, zu seinem Sohn weitergehen. Nur in Verzweigungsknoten wird sich die Punktzahl verringern, aber auch da um höchstens die Hälfte, da wir ja immer zum punktereicheren Sohn gehen. Da sich die Punktzahl also in jedem Verzweigungsknoten höchstens halbiert, benötigen wir wenigstens $2m + 3$ Verzweigungsknoten um bei q_{r-1} zu landen, der nur noch einen Punkt unter sich hat (man muß k schon $2m + 3$ -mal halbieren, um auf 1 zu kommen).

Wir betrachten nun die $2m + 3$ letzten Verzweigungsknoten in π (also von unten gezählt). Unter diesen kann es mehr Linksverzweigungsknoten oder mehr Rechtsverzweigungsknoten geben. Nehmen wir den letzteren Fall an. (Im ersten Fall kann man analog argumentieren.) Dann gibt es also darunter mindestens $m + 2$ Rechtsverzweigungsknoten. Seien p_0, p_1, \dots, p_{m+1} die letzten $m + 2$ darunter (von oben nach unten gezählt). Da wir nur m Variablen haben und alle Knoten bis auf die Blätter mit Variablen bewertet sind, muß es unter den $m + 1$ Rechtsverzweigungsknoten p_1, \dots, p_{m+1} wenigstens zwei geben, die dieselbe Variable tragen. Nennen wir sie p_i und p_j (mit $0 < i < j \leq m + 1$) und die von ihnen begründeten Teilbäume in $\dot{\Gamma}$ entsprechend $\dot{\Gamma}(p_i)$ und $\dot{\Gamma}(p_j)$. Die Front von $\dot{\Gamma}(p_j)$ sei \dot{w} , die von $\dot{\Gamma}(p_i)$ sei $\dot{v}\dot{w}\dot{x}$ (sie umfaßt ja die Front von $\dot{\Gamma}(p_j)$) und schließlich seien \dot{u}, \dot{y} die restlichen Teile von \dot{z} , sodaß $\dot{z} = \dot{u}\dot{v}\dot{w}\dot{x}\dot{y}$. Dann gilt:

1. Da p_i zu den letzten $2m + 3$ Verzweigungsknoten gehört, enthält $\dot{\Gamma}(p_i)$ höchstens $k = 2^{2m+3}$ punktierte Blätter (enthielte dieser Teilbaum mehr Punkte, so müßte er auch mehr als $2m + 3$ Verzweigungsknoten enthalten), und damit enthält $\dot{v}\dot{w}\dot{x}$ höchstens k Punkte.
2. Da p_0 Verzweigungsknoten ist, enthält auch die Front des vom linken Sohn von p_0 begründeten Teilbaums einen Punkt. Diese Front liegt aber in \dot{u} , somit enthält \dot{u} einen Punkt. (Wir haben in π zwar den rechten Sohn gewählt, weil der mehr Punkte unter sich hatte, aber auch der linke Sohn hat wenigstens einen Punkt unter sich. Der Teil \dot{u} umfaßt die Front dieses linken Sohns, da \dot{v} in der Front von p_i liegt, also unter dem rechten Sohn von p_0 . Darum muß \dot{u} also diesen Punkt der unter dem linken Sohn von p_0 liegt, enthalten.)
Da p_i Verzweigungsknoten ist und die Front des vom linken Sohn von p_i begründeten Teilbaums in \dot{v} liegt, enthält \dot{v} einen Punkt (dieselbe Argumentation wie eben, wobei p_i jetzt die Rolle von p_0 und p_j die Rolle von p_i spielt.)
3. Da p_j Verzweigungsknoten ist, liegen in der Front \dot{w} des von p_j begründeten Teilbaums Punkte.
4. Ersetzen wir den Teilbaum $\dot{\Gamma}(p_i)$ durch den Teilbaum $\dot{\Gamma}(p_j)$, so erhalten wir einen Ableitungsbaum Γ_0 für $uw\dot{x} = uv^0wx^0y$. Ersetzen wir umgekehrt in Γ den Teilbaum $\dot{\Gamma}(p_j)$ durch den Teilbaum $\dot{\Gamma}(p_i)$, so erhalten wir einen Ableitungsbaum Γ_2 für das Wort uv^2wx^2y . Dies letztere können wir wiederholt tun, sagen wir l -mal und erhalten einen Ableitungsbaum Γ_l für das Wort uv^lwx^ly .

■

Anmerkung 2.4.9

Würden wir nicht zwischen Linksverzweigungsknoten und Rechtsverzweigungsknoten unterscheiden

und nur Verzweigungsknoten betrachten, so erhalte man die schwächere Aussage, daß \dot{w} einen Punkt, \dot{v} oder \dot{x} einen Punkt und \dot{u} oder \dot{y} einen Punkt hat, was der PE bezüglich der gepunkteten Symbole entspricht.

Leider ist auch das Ogden-Lemma nicht umkehrbar. Auch hier gibt es Sprachen, die zwar die Ogden-Eigenschaft haben, aber dennoch nicht kontextfrei sind. Beim folgenden Beispiel nehmen wir vorweg, daß die Klasse CFL der kontextfreien Sprachen abgeschlossen ist unter Homomorphismus und GSM-Abbildung und Schnitt mit regulären Mengen.

Satz 2.4.10

Es gibt Sprachen mit der Ogdeneigenschaft, die nicht kontextfrei sind.

Beweis: Sei $L \subseteq \Sigma^*$, irgendeine nicht kontextfreie Sprache über Σ , die nicht die Ogdeneigenschaft hat. Sei $\Sigma' = \{a' \mid a \in \Sigma\}$ eine Kopie von Σ mit gestrichenen Symbolen. Definiere L' vermöge: $w = a_1 \dots a_n \in L \implies w' = b_1 \dots b_n \in L'$, wo $b_i = a_i$ falls i ungerade und $b_i = a'_i$ falls i gerade, d.h. jedes zweite Symbol in w' ist gestrichelt. Jetzt stehen keine zwei gleichen Symbole mehr nebeneinander. Definiere \widehat{L} vermöge:

1. $w' = a_1 \dots a_n \in L' \implies \{a_1\}^+ \dots \{a_n\}^+ \subseteq \widehat{L}$
2. Sei $a, b, c \in (\Sigma \cup \Sigma')$ mit $a \neq b, b \neq c$ und abc Teilwort von $w \in (\Sigma \cup \Sigma')^*$, so ist $w \in \widehat{L}$. (steht ein Symbol in w isoliert, so gehört w zu \widehat{L}).

Jedes Symbol in w' wird beliebig wiederholt. Steht ein Symbol allein, so gehört das Wort ohnehin zur Sprache. Man erhält also die Worte von \widehat{L} aus denen von L , indem man erst jedes zweite Symbol strichelt, und dann jedes Symbol vervielfacht (aufbläst). Umgekehrt bekommt man jedes Wort von L aus den Worten von \widehat{L} , in denen kein Symbol isoliert steht, indem man die Wortteile, die aus demselben Symbol bestehen, wieder zu einem Symbol schrumpft und die Striche entfernt.

L und \widehat{L} sind also leicht auseinander konstruierbar, sie sind sich sehr ähnlich. Da L nicht kontextfrei war, ist auch \widehat{L} das nicht:

CFL ist abgeschlossen unter Homomorphismus, GSM-Abbildung und Schnitt mit regulären Mengen (Beweis dafür später).

1. Definiere FA $M = (Q, \Sigma \cup \Sigma', \delta, q_0, \{q_0\})$ mit $Q = \{q_a \mid a \in \Sigma \cup \Sigma'\} \cup \{q_0\}$ und

$$\delta = \{(q_0, a, q_a), (q_a, a, q_a), (q_a, a, q_0) \mid a \in \Sigma \cup \Sigma'\}.$$

Sei $R = L(M)$. Dann enthält $\widehat{L} \cap R$ alle Worte aus \widehat{L} , die keine isolierten Symbole haben.

2. Definiere GSM $M' = (Q', \Sigma \cup \Sigma', \delta', q'_0)$ mit $Q' = \{q'_0\} \cup \{q'_a \mid a \in \Sigma \cup \Sigma'\}$, und

$$\delta' = \{(q'_0, a, a, q'_a), (q'_b, a, a, q'_a), (q'_a, a, \varepsilon, q'_a) \mid a \neq b, a, b \in \Sigma \cup \Sigma'\}.$$

Ist g die GSM-Abbildung zu M' , so ist $g(\widehat{L} \cap R) = L'$.

3. Definiere Homomorphismus h auf $(\Sigma \cup \Sigma')^*$ vermöge $h(a') = h(a) = a$ ($a \in \Sigma$).

Dann ist $L = h(L') = h(g(\widehat{L} \cap R))$. Wäre nun \widehat{L} kontextfrei, so auch L . W!

Aber \widehat{L} hat jetzt die Ogdeneigenschaft, die L nicht hat (wir haben eben jedes Symbol pumpbar gemacht): Wenn es keine isolierten Symbole gibt, können wir jedes Symbol hinein- oder heraus-pumpen. Gibt es ein isoliertes Symbol, so können wir das auch tun, wenn wir nicht ausgerechnet das einzige isolierte Symbol pumpen (heraus oder hinein). ■

Anmerkung 2.4.11

Das Pumping- und das Ogdenlemma ist also nicht umkehrbar. Daher benutzen wir diese Lemmata gewöhnlich in ihrer negativen Form: Um zu beweisen, daß eine Sprache nicht kontextfrei ist, zeigen wir, daß sie nicht die Pumpingeigenschaft bzw. die Ogdeneigenschaft besitzt.

Um zu zeigen, daß die Sprache L nicht die Pumpingeigenschaft besitzt, geben wir also für alle $k \in \mathbb{N}$ ein Wort $z \in L$ einer Länge von wenigstens k an, sodaß bei jeder Zerlegung $z = uvwxy$, die erlaubt ist ($vx \neq \varepsilon$ und $|vwx| \leq k$) das Pumpwort uv^iwx^iy für ein $i \in \mathbb{N}$ nicht mehr aus L ist:

$$\forall k \in \mathbb{N} \quad \exists z \in L \quad \forall u, v, w, x, y \in \Sigma^* \quad \exists i \in \mathbb{N} : uv^iwx^iy \notin L.$$

$$|z| \geq k \quad z = uvwxy \quad |vwx| \leq k \quad vx \neq \varepsilon$$

Zur besseren Anschauung können wir das Pumpingspiel spielen, wir gegen einen Zweifler:

1. der Zweifler wählt ein $k \in \mathbb{N}$.
2. Wir wählen ein Wort z aus L von wenigstens der Länge k .
3. der Zweifler zerlegt unser Wort z in 5 Teile $uvwxy$, sodaß der Mittelteil nicht zu groß ist ($|vwx| \leq k$) und nicht beide Pumpstellen leer sind ($vx \neq \varepsilon$).
4. Wir geben jetzt ein $i \in \mathbb{N}$ an. Liegt das i -fach gepumpte Wort uv^iwx^iy nicht in der Sprache L , so haben wir gewonnen, der Zweifler muß aufgeben und einräumen, daß wir Recht hatten.

Genauso können wir ein Ogdenspiel spielen, wenn wir zeigen wollen, daß eine Sprache nicht die Ogdeneigenschaft hat:

$$\forall k \in \mathbb{N} \quad \exists z \in L \quad \exists k\text{-Pktng } \dot{z} \text{ von } z \quad \forall \dot{u}, \dot{v}, \dot{w}, \dot{x}, \dot{y} \in (\Sigma \cup \dot{\Sigma})^* \text{ mit } \dot{z} = \dot{u}\dot{v}\dot{w}\dot{x}\dot{y},$$

$\dot{v}\dot{w}\dot{x}$ hat höchstens k Punkte und
 $\dot{u}, \dot{v}, \dot{w}$ oder $\dot{w}, \dot{x}, \dot{y}$ haben je einen Punkt

$$(u = h(\dot{u}), v = h(\dot{v}), w = h(\dot{w}), x = h(\dot{x}), y = h(\dot{y}))$$

2.5 Pushdown-Automat, Kellerautomat

2.5.1 Definitionen

Wir erweitern jetzt das Modell des finiten Automaten durch hinzufügen einer besonderen Speicherstruktur, die wir Keller oder Pushdown-Stack nennen wollen.

Die Menge Q der Zustände eines finiten Automaten kann man sich als Menge der speicherbaren Informationen vorstellen: jede dieser Informationen wird durch ein eigenes Symbol - das Zustandszeichen - repräsentiert. Diese Informationsmenge ist aber fest vorgegeben und endlich, sie ist mit dem Automaten gegeben und nicht durch den jeweiligen Input veränderbar.

Der Kellerautomat oder Pushdown-Automat ist ein finiter Automat angereichert mit einer neuen, in der Größe nicht beschränkten Informationsmenge. Die neue Informationsmenge ist Γ^* , wobei Γ ein Alphabet ist. Hier besteht eine Information also nicht aus einem einzelnen Zeichen aus Γ , sondern aus einem im Prinzip beliebig langen Wort über Γ .

Im Unterschied zu den Zuständen ist allerdings der Zugriff auf diese Information eingeschränkt. Der Automat kann nur das erste Zeichen des Informationswortes $W \in \Gamma^*$ sehen und es ist nicht möglich, eine Speicherinformation aus Γ^* durch eine beliebige andere Information zu ersetzen - dies würde man einen random access Zugang nennen - sondern die Information $W \in \Gamma^*$ kann in einem Schritt nur an der linken Peripherie verändert werden: Das erste Symbol kann gelöscht werden oder es kann vor das erste Symbol ein weiteres Symbol gesetzt werden. Man kann sich vorstellen, daß das Wort $W \in \Gamma^*$ einen Stapel aus Symbolen darstellt, das erste Symbol ist das oberste des Stapels, das letzte Symbol

das unterste des Stapels. Gesehen werden kann immer nur das oberste Symbol des Stapels. Erlaubt ist, das oberste Symbol des Stapels zu entfernen oder auf den Stapel ein weiteres Symbol zu legen. Diese Vorstellung hat der Maschine den Namen gegeben: Stapel- oder Stack-Automat. Beim Keller haben wir das Bild: Was zuletzt durch die Kellerluke eingekellert wird, wird als erstes entnommen (last in - first out). Das ist aber gerade das Stapelprinzip.

Formal wollen wir das wie bei den Grammatiken durch Termersetzung ausdrücken: wir ersetzen jeweils das oberste Symbol des Kellers - wir sagen auch: das Symbol an der Kellerspitze, das Topsymbol oder das Spitzensymbol - durch ein Wort. Dabei stellen wir uns - der besseren Schreibweise wegen - den Keller nach links gekippt vor, so daß wir den Kellerinhalt durch ein Wort darstellen können, dessen erstes Symbol oben auf dem Keller sitzt, also Topsymbol des Kellers ist.

Hierdurch können wir das oben Beschriebene erreichen: Wollen wir über das Topsymbol A ein Symbol B setzen, so ersetzen wir A durch das Wort BA . Wollen wir das Topsymbol A löschen, so ersetzen wir es durch das leere Wort ε . Umgekehrt kann man diesen Ersetzungsprozeß - allerdings in mehreren Schritten - auch durch die Operationen Entfernen und Auflegen simulieren: Ersetzen wir A durch $B_1 \cdots B_k$ (B_1 wird neues Topsymbol), so ahmen wir dies nach durch die Operationsfolge: entferne A , setze B_k auf die Kellerspitze, ..., setze B_1 auf die Kellerspitze.

Um die beiden Speicher - Zustandsmenge und Keller- gleich zu behandeln, stellen wir uns den Zustand als ein einelementiges Wort über dem Zustandsalphabet Q vor. Der Kellerautomat bekommt also drei Informationen: das Inputsymbol, das Zustandsymbol und das Topsymbol des Kellers. Daraufhin reagiert er, indem er das Zustandsymbol durch ein neues Zustandsymbol und das Topsymbol durch ein Wort ersetzt. D.h. die Befehle des Kellerautomaten haben die Form (p, a, A, q, W) (Zustand, Inputsymbol, Topsymbol, neuer Zustand, Ersetzungswort).

Der Verlauf einer Rechnung eines Kellerautomaten hängt von seiner gespeicherten Information ab, also vom Zustand und vom Kellerinhalt. Geben wir auch noch an, welcher Input bisher gelesen wurde, so nennen wir das eine Situation des Automaten. Der Kellerautomat befindet sich also in der Situation (q, w, W) , wenn er im Zustand q ist, bisher das Inputwort w gelesen hat und er im Keller das Wort W stehen hat. In jedem Schritt ändert er seine Situation, wie, das gibt die Befehlsmenge an.

Definition 2.5.1 (Kellerautomat, Pushdown-Automat, NPDA)

Seien Q, Γ endliche Alphabeten, $\Sigma \subseteq \Gamma$, $q_0 \in Q$, $\perp \in \Gamma$ und δ eine endliche Teilmenge aus $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$. Dann heißt die Struktur

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \perp)$$

(nichtdeterministischer, one-way) Pushdown-Automat (NPDA) oder Kellerautomat. Wir nennen Q die Menge der Zustände, Γ das Kelleralphabet, Σ das Inputalphabet, q_0 den Startzustand, \perp das Bodensymbol (Bottom-Symbol) des Kellers und δ die Befehlsmenge.

Wir nennen M einen deterministischen Pushdown-Automaten (DPDA), wenn es keine zwei Befehle gibt, die dieselben beiden ersten Komponenten haben, und keinen ε -Befehl, der mit dem Anfangszustand eines anderen Befehls beginnt.

Menge der Situationen ist $S = Q \times \Sigma^* \times \Gamma^*$ (Zustand, gelesener Input, Kellerinhalt). Die Änderungen der Situationen durch die Anwendung von Befehlen wird durch die Schrittrelation \vdash aus $S \times S$ beschrieben:

$$(q, w, AW) \vdash (q', wa, VW) \text{ gdw. } (q, a, A, q', V) \in \delta,$$

wobei $q, q' \in Q$; $a \in \Sigma \cup \{\varepsilon\}$; $w \in \Sigma^*$; $A \in \Gamma$ und $V, W \in \Gamma^*$.

Die Mehrschrittrelation \vdash^* ist die reflexive und transitive Hülle von \vdash : $s \vdash^0 s$, $s \vdash^{n+1} s''$ gdw. $s \vdash^n s'$ und $s' \vdash s''$, d.h. $s \vdash^* s''$, wenn s'' in 0 oder mehr Schritten aus s hervorgeht. Eine Folge s_1, \dots, s_n nennen wir einen Lauf der Länge n von M , wenn $s_i \vdash s_{i+1}$ für alle $i \in \{1, \dots, n-1\}$. Ein

w -Lauf von M ist ein Lauf, der mit der Startsituation $(q_0, \varepsilon, \perp)$ beginnt und mit einer Situation endet, in der kein Befehl mehr anwendbar ist (Stopsituation) und in der w in der 2. Komponente steht (das gelesene Wort ist). Ist in dieser Stopsituation der Keller leer, so ist der w -Lauf akzeptierend (das Wort w wird akzeptiert). Man beachte, daß das Kriterium für Akzeptanz hier nicht ein erreichter Endzustand ist, wie bei den finiten Automaten, sondern ein leerer Keller.

Die Sprache von M ist $L(M) = \{w \in \Sigma^* \mid (q_0, \varepsilon, Z) \vdash^* (q, w, \varepsilon) \text{ für ein } q \in Q\}$. Ein Wort w ist also genau dann aus L , wenn es einen akzeptierenden w -Lauf gibt.

Beispiel 2.5.2

NPDA für $\{a^n b^n \mid n \in \mathbb{N}\}$.

Befehle	Ein Lauf
$(1, \varepsilon, \perp, 1, \varepsilon)$	$(1, \varepsilon, \perp)$
$(1, a, \perp, 1, A)$	$(1, a, A)$
$(1, a, A, 1, AA)$	$(1, aa, AA)$
$(1, b, A, 2, \varepsilon)$	$(2, aab, A)$
$(2, b, A, 2, \varepsilon)$	$(2, aabb, \varepsilon)$

2.5.2 NPDA-Sprachen sind kontextfrei

Lemma 2.5.3

Zu jedem NPDA M existiert ein äquivalenter NPDA M' mit nur *einem* Zustand.

Beweis: Der NPDA M' muß sich die Zustandsinformation (das Zustandssymbol) wohl oder übel auf seinem anderen Speicher - dem Keller - merken. Dies kann er tun, indem er das Zustandssymbol noch zu dem Topsymbol an der Kellerspitze schreibt (d.h. das neue Topsymbol ist ein Paar aus Zustandssymbol und altem Topsymbol). Solange M' nur druckt, macht das kein Problem. Schwierig wird es erst, wenn M' löscht: dann löscht er nämlich auch das aktuelle Zustandssymbol. Das Symbol darunter, das jetzt sichtbar wird, und an dem wir das neue aktuelle Zustandssymbol erwarten, wird es nicht enthalten, es sei denn, es ist damals mit dem jetzt erst aktuellen Zustandssymbol versehen worden. Dies kann nur geschehen, wenn M' damals das jetzt erst aktuelle Zustandssymbol geraten hat. Eben dies soll M' tun (M' ist also nichtdeterministisch).

Allerdings taucht ein zweites Problem auf: Ist das Topsymbol (p, A) (aktueller Zustand, altes Topsymbol) und der Input a und hat M den Befehl $(p, a, A, q, \varepsilon)$. Dann muß M' das Topsymbol (p, A) löschen. Es ist aber für M' unmöglich, beim Lesen von (p, A) schon zu wissen, ob unter der Kellerspitze ein Symbol der Form (q, B) mit dem richtigen aktuellen Zustand q steht. Nach dem Löschen des Topsymbols (p, A) aber liest M' das neue Topsymbol (q, B) , weiß aber nicht mehr, ob das jetzt sichtbare q der richtige aktuelle Zustand ist (M' hat ja keine Möglichkeit, sich Information außerhalb des Kellers auch nur einen Schritt lang zu merken). Hier hilft die Tatsache, daß M' beim Ersetzen des Topsymbols gleich 2 Felder bedrucken kann und damit garantieren kann, daß eine bestimmte Information gleichermaßen auf beiden Feldern auftaucht. Die Symbole im Keller von M' sind Tripel der Form (p, A, r) (aktueller Zustand, altes Topsymbol, Zustand, den M erreicht, wenn er unter dieses aktuelle Feld kommt). Ist (p, a, A, q, BC) ein Befehl von M , (p, A, r) das aktuelle Topsymbol von M' , so ersetzt M' dieses durch $(q, B, s)(s, C, r)$. Jetzt kann M' beim Lesen des Topsymbols (q, B, s) sehen, ob der Befehl $(q, a, B, s, \varepsilon)$ anwendbar ist, weil M' schon jetzt sieht, ob unter der Kellerspitze der korrekte neue Zustand s verzeichnet ist.

Formal: Sei $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp)$. Wir definieren $M' = (\{1\}, \Sigma, \Gamma', \delta', 1, \perp)$ mit $\Gamma' = Q \cdot \Gamma \cdot Q \cup \{\perp\}$ und δ' wie folgt:

1. $(\varepsilon, \perp, (q_0 \perp r)) \in \delta'$ für alle $r \in Q$ (das Bottomsymbol wird in die Tripelform gebracht).

2. $(p, a, A, q, B_k \cdots B_1) \in \delta$ ($k \geq 1$), so ist $(a, (pAr), (qB_kq_k)(q_kB_{k-1}q_{k-1}) \cdots (q_2B_1r)) \in \delta'$ für alle $r, q_2, \dots, q_k \in Q$. (für jedes Kellerfeld - außer dem untersten - wird der Zustand geraten, der von M' eingenommen wird, wenn M' zum erstenmal wieder unter dieses Kellerfeld kommt)
3. $(p, a, A, q, \varepsilon) \in \delta$, so ist $(a, (pAq), \varepsilon) \in \delta'$ (das Topsymbol wird gelöscht, wenn der Folgezustand, den M' mit diesem Schritt einnimmt, auch ein Feld tiefer steht.)

■

Beispiel 2.5.4

NPDA für $L = \{a^n b^{n+1} \mid n \in \mathbb{N}\}$.

NPDA	(1, a, \perp , 1, $\perp\perp$)	1-Zust.-NPDA	(ε , \perp , (1 \perp 1))	(ε , \perp , (1 \perp 2))
	(1, b, \perp , 2, ε)		(a, (1 \perp 1), (1 \perp 1)(1 \perp 1))	(a, (1 \perp 1), (1 \perp 2)(2 \perp 1))
	(2, b, \perp , 2, ε)		(a, (1 \perp 2), (1 \perp 1)(1 \perp 2))	(a, (1 \perp 2), (1 \perp 2)(2 \perp 2))
			(b, (1 \perp 2), ε)	
			(b, (2 \perp 2), ε)	

(Bei den Befehlen des 1-Zustands-NPDA lassen wir die Zustandskomponenten weg).

Satz 2.5.5

Jede NPDA-Sprache ist kontextfrei.

Beweis: Sei $P = (Q, \Sigma, \Gamma, \delta, q_0, \perp)$ NPDA (o.B.d.A.: P hat nur einen Zustand, kein Zeichen aus Σ wird in den Keller geschrieben). Definiere CFG $G = (\Gamma - \Sigma, \Sigma, P, \perp)$ für $L(P)$ mit $A \rightarrow aV$ aus P gdw. $(a, A, V) \in \delta$ (Die Linksableitung für w in G entspricht einem w -Lauf von P). ■

Beispiel 2.5.6

die CFG zu dem 1-Zustands-NPDA aus dem vorigen Beispiel:

\perp	\rightarrow	(1 \perp 1)		(1 \perp 2)
(1 \perp 1)	\rightarrow	$a(1\perp 1)(1\perp 1)$		$a(1\perp 2)(2\perp 1)$
(1 \perp 2)	\rightarrow	$a(1\perp 1)(1\perp 2)$		$a(1\perp 2)(2\perp 2)$ b
(2 \perp 2)	\rightarrow	b		

2.5.3 Analyse kontextfreier Sprachen

Satz 2.5.7

Zu jeder CFG $G = (N, T, P, S)$ existiert ein NPDA M , der dieselbe Sprache beschreibt.

Beweis: 1. **Top-down-Analyse:** Wir geben einen NPDA $M = (Q, T, T \cup N, \delta, q_0, S)$ an, der nach den Regeln der Grammatik G aus dem Startsymbol S sein Inputwort ableitet. Dabei simuliert er eine Linksableitung, er ersetzt immer das linkeste Nonterminal des Ableitungswortes. Ist das Topsymbol eine Variable, so ersetzt er sie gemäß einer Regel, ist es ein Terminal, so überprüft er, ob es mit dem momentanen Inputsymbol übereinstimmt, ob es also gerade eingelesen wird. Nur dann kann er es löschen und weitermachen. Der Keller wird also genau dann leer, wenn M nach und nach das Inputwort erzeugt hat.

M braucht keine Zustände, d.h. formal hat M nur einen Zustand ($Q = \{1\}$, $q_0 = 1$). Ist $A \rightarrow W$ eine Regel aus P , so ist (ε, A, W) aus δ (M ersetzt in einem ε -Schritt das Topsymbol, wenn es eine Variable ist, gemäß der Regel). Außerdem ist für alle $a \in T$ noch $(a, a, \varepsilon) \in \delta$ (M löscht das Topsymbol, wenn es mit dem gerade gelesenen Inputsymbol übereinstimmt). ■

Beispiel 2.5.8

Regeln: $S \longrightarrow aSb \mid \varepsilon$	Befehle:	$(\varepsilon, S, \varepsilon)$	Lauf:	(ε, S)
		(ε, S, aSb)		(ε, aSb)
		(a, a, ε)		(a, Sb)
		(b, b, ε)		$(a, aSbb)$
				(aa, Sbb)
				(aa, bb)
				(aab, b)
				$(aabb, \varepsilon)$

Beweis: 2. Bottom-up-Analyse. Wir geben einen NPDA $M = (Q, T, N \cup T \cup \{\perp\}, \delta, q_0, \perp)$ an, der in seinem Keller aus dem Inputwort nach den Regeln der Grammatik G eine Rechtsableitung rückwärts durchführt, bis nur noch das Startsymbol S über dem Bodensymbol \perp steht. Immer wenn der Anfang des Kellerwortes (von oben bzw. links gelesen) der rechten Seite einer Regel entspricht, so kann M den Anfang löschen und durch die Variable der linken Seite dieser Regel ersetzen.

Am Anfang steht nur das Bodensymbol \perp im Keller. Zunächst kopiert M den Beginn des Inputworts in den Keller, bis M nichtdeterministisch entscheidet, daß jetzt eine Regel rückwärts ausgeführt werden kann. Hat M richtig entschieden, dann findet M die rechte Seite einer Regel am Anfang seines Kellers und ersetzt sie durch die Variable auf der linken Seite dieser Regel. Ist die rechte Seite der Regel länger als ein Symbol, braucht M natürlich mehrere Schritte um diese rechte Seite im Keller zu löschen.

Dann liest M wieder weiteren Input in den Keller, bis M wieder annimmt, daß eine Regel rückwärts ausführbar sei. Ist das Inputwort aus der Sprache $L(G)$, d.h. gibt es eine Rechtsableitung für dieses Wort und hat M immer richtig geraten, so wird der Input irgendwann abgearbeitet sein und im Keller nur noch $S\perp$ stehen. Dies kann nun M noch löschen, um den Keller leer zu machen. Damit hat M das Inputwort akzeptiert.

Wir wählen also:

$\Gamma := N \cup T \cup \{\perp\}$ (im Keller können Variable sowie Terminals stehen). Sei m die maximale Regellänge von P . $Q := \{[W, A] \mid W \in (N \cup T)^*, |W| \leq m, A \in N\} \cup \{[\varepsilon], [S]\}$ (M merkt sich im Zustand, welche Regel er rückwärts ausführen will, was er noch im Keller zu löschen hat und was er am Ende dafür setzen will: W wird noch gelöscht, und A wird am Ende auf den Keller gesetzt). Als Startzustand q_0 wählen wir $[\varepsilon]$.

δ definieren wir wie folgt:

$([\varepsilon], a, X, [\varepsilon], aX)$ für alle $a \in T$ und $X \in \Gamma$. (Damit kopiert M Input in den Keller.)

Ist $A \longrightarrow B_1 \cdots B_k$ aus P , so ist $([\varepsilon], \varepsilon, X, [B_k \cdots B_1, A], X) \in \delta$ für alle $X \in \Gamma$. (M entschließt sich für eine Regel. Beachte, daß die Regel $A \longrightarrow B_1 \cdots B_k$ für $k = 0$ die Form $A \longrightarrow \varepsilon$ hat.)

Hat M sich für eine Regel entschieden, so wird sie mithilfe folgender Befehle rückwärts ausgeführt:

$([C_l \cdots C_1, A], \varepsilon, C_l, [C_{l-1} \cdots C_1, A], \varepsilon)$ mit $k \geq 1$ für alle A, C_1, \dots, C_l aus Γ , $1 \leq l \leq m$
 $([\varepsilon, A], \varepsilon, X, [\varepsilon], AX)$ für alle $A, X \in \Gamma$.

Steht am Schluß nur noch $S\perp$ im Keller, so kann M den Keller mit folgenden Regeln leeren

$([\varepsilon], \varepsilon, S, [S], \varepsilon),$
 $([S], \varepsilon, \perp, [S], \varepsilon).$

■

Beispiel 2.5.9

Regeln: $S \longrightarrow aSb \mid \varepsilon$

Befehle:

$([\varepsilon], a, X, [\varepsilon], aX)$	}	für alle X (liest Input in den Keller)
$([\varepsilon], b, X, [\varepsilon], bX)$		
$([\varepsilon], \varepsilon, X, [\varepsilon, S], X)$	}	für alle X (ersetzt ε durch S)
$([\varepsilon, S], \varepsilon, X, [\varepsilon], SX)$		
$([\varepsilon], \varepsilon, X, [bSa, S], X)$	}	für alle X (ersetzt bSa durch S)
$([bSa, S], \varepsilon, b, [Sa, S], \varepsilon)$		
$([Sa, S], \varepsilon, S, [a, S], \varepsilon)$	}	(Schlußroutine)
$([a, S], \varepsilon, a, [\varepsilon, S], \varepsilon)$		
$([\varepsilon, S], \varepsilon, X, [\varepsilon], SX)$		
$([\varepsilon], \varepsilon, S, [S], \varepsilon)$		
$([S], \varepsilon, \perp, [S], \varepsilon)$		

aabb-Lauf:

Start	$([\varepsilon], \perp)$	
lese a	$([\varepsilon], a\perp)$	
lese a	$([\varepsilon], aa\perp)$	
simuliere $S \rightarrow \varepsilon$	$([\varepsilon], Saa\perp)$	(2 Schritte)
lese b	$([\varepsilon], bSaa\perp)$	
simuliere $S \rightarrow aSb$	$([\varepsilon], Sa\perp)$	(mehrere Schritte)
lese b	$([\varepsilon], bSa\perp)$	
simuliere $S \rightarrow aSb$	$([\varepsilon], S\perp)$	(mehrere Schritte)
lösche S	$([S], \perp)$	
lösche \perp	$([S], \varepsilon)$	

Beispiel 2.5.10

Regeln $S \rightarrow SS \mid aSb \mid bSa \mid \varepsilon$; Input $w = aabbab$

Die Rechtsableitung von w ist $\underline{S}, \underline{SS}, \underline{SaSb}, \underline{Sab}, \underline{aSbab}, \underline{aaSbbab}, aabbab$

Der dazugehörige w -Lauf ist, der diese Rechtsableitung von hinten simuliert, ist:

Start	$([\varepsilon], \perp)$	
lese a	$([\varepsilon], a\perp)$	
lese a	$([\varepsilon], aa\perp)$	
sim $S \rightarrow \varepsilon$	$([\varepsilon], Saa\perp)$	
lese b	$([\varepsilon], bSaa\perp)$	
sim $S \rightarrow aSb$	$([\varepsilon], Sa\perp)$	mehrere Schritte
lese b	$([\varepsilon], bSa\perp)$	
sim $S \rightarrow aSb$	$([\varepsilon], S\perp)$	
lese a	$([\varepsilon], aS\perp)$	
sim $S \rightarrow \varepsilon$	$([\varepsilon], SaS\perp)$	
lese b	$([\varepsilon], bSaS\perp)$	
sim $S \rightarrow aSb$	$([\varepsilon], SS\perp)$	mehrere Schritte
sim $S \rightarrow SS$	$([\varepsilon], S\perp)$	
lösche S	$([S], \perp)$	
lösche \perp	$([\varepsilon], \varepsilon)$	

Anmerkung 2.5.11

Obwohl die Top-down-Analyse gedanklich näher liegt, sind die Syntaxanalyseverfahren der modernen Compiler alle Bottom-up. Da der Compilierungsprozeß schnell gehen soll, bedient man sich besonders einfach zu analysierender Grammatiken, der $LL(k)$ - und der $LR(k)$ -Grammatiken.

Die Sprachen von $LL(k)$ -Grammatiken kann man mit einem look-ahead von k Inputsymbolen deterministisch top-down analysieren. Die von $LR(k)$ -Grammatiken kann man mit einem look-ahead von k Inputsymbolen deterministisch bottom-up analysieren.

Kontextfreie Sprachen heißen deterministisch, wenn sie von deterministischen Kellerautomaten (DPDA's) mit Endzustand erkannt werden. Für jede deterministische kontextfreie Sprache (DCFL) kann man eine $LR(1)$ -Grammatik angeben. Aber es gibt DCFL's, für die es für kein k eine $LL(k)$ -Grammatik gibt (z.B. alle inhärent mehrdeutigen (ambiguous) Sprachen oder die Sprache $\{a^n 0 b^n \mid n \in \mathbb{N}_+\} \cup \{a^n 1 b^{2n} \mid n \in \mathbb{N}_+\}$). Ja man kann nicht einmal entscheiden, ob es zu einer gegebenen CFG für ein festes k überhaupt eine äquivalente $LL(k)$ -Grammatik gibt.

Heutzutage werden die meisten imperativen Programmiersprachen durch $LR(1)$ -Grammatiken beschrieben. Der Compiler-Generator YACC (yet another compiler compiler) beruht auf der $LR(1)$ -Syntaxanalyse.

2.5.4 Der CYK-Algorithmus

Die oben beschriebenen Analyseverfahren sind sehr schnell (sie können in Linearzeit laufen) und benötigen wenig Speicher (nur linearen Speicherplatz), sind aber nichtdeterministisch. Wir suchen aber ein deterministisches Verfahren, das alle (nichtdeterministischen) kontextfreien Sprachen analysiert. Hier stellen wir ein deterministisches Verfahren von Cocke, Younger, Kasami vor, das bei Vorgabe einer CFG G ein Wort w der Länge n in n^3 Zeit und n^2 Speicherplatz analysiert. (Für deterministische kontextfreie Sprachen gibt es bessere Verfahren, die auf Turingmaschinen in $n^2 / \log^2 n$ Zeit und $\log^2 n$ Platz laufen oder allgemeiner in einem Zeit-Platz-Produkt von n^2 , auf Maschinen mit random access input in Zeit $n^{1+\epsilon}$ und Platz $\log^2 n$ bzw. in linearer Zeit und Platz n^ϵ .)

Problem: Gegeben sei eine CFG $G = (N, T, P, S)$ (o.B.d.A. in CNF) und ein Wort $w \in \Sigma^*$. Wir fragen: ist $w \in L(G)$?

Lösung 1: Probiere alle Ableitungen durch.

Zeitaufwand: Ist $|w| = n$, so haben die Ableitungen für w die Länge n , wenn man von den Terminalregeln absieht. Gibt es bis zu r Regeln mit gleicher linker Seite, so gibt es bis zu r^n viele Ableitungen dieser Länge, d.h. der Zeitaufwand dieses Verfahrens ist exponentiell.

Lösung 2: Sei $w = a_1 \cdots a_n$. Jede Ableitung ist von der Form $S \vdash AB \vdash^* a_1 \cdots a_k a_{k+1} \cdots a_n$, wo $A \vdash^* a_1 \cdots a_k$, und $B \vdash^* a_{k+1} \cdots a_n$. Wir untersuchen alle diese Möglichkeiten für alle A, B mit $S \rightarrow AB$ aus P und alle $k \in \{1, \dots, n\}$. Die dadurch entstehenden Teilprobleme $A \vdash^* a_1 \cdots a_k$, und $B \vdash^* a_{k+1} \cdots a_n$, werden genauso behandelt, d.h. selbst wieder auf alle möglichen Weisen geteilt und die so entstandenen Subprobleme untersucht. Dies wird rekursiv solange weiterverfolgt, als eine Teilung noch möglich ist. In der ersten Rekursionsstufe sind $n - 1$ Teilungen möglich, in der zweiten Rekursionsstufe führt die Teilung an der Stelle i zu $i - 1$ verschiedenen Teilungen des ersten Teils und $n - i - 1$ verschiedenen Teilungen des zweiten Teils, usw.

Zeitaufwand: Das Grundproblem $S \vdash^* a_1 \cdots a_k a_{k+1} \cdots a_n$ wird in zwei Teile von ähnlicher Form zerlegt $A \vdash^* a_1 \cdots a_k$ und $B \vdash^* a_{k+1} \cdots a_n$. Um das Grundproblem zu lösen, lösen wir die beiden Teilprobleme und zwar für jede mögliche Zerlegung. Es gibt $n - 1$ mögliche Zerlegungen des Wortes $a_1 \cdots a_n$ und für jede Zerlegung bis zu r viele Wahlen von A und B , wenn r die maximale Zahl von Regeln mit gleicher linker Seite ist. Wenn $T(n)$ der Zeitaufwand ist, der benötigt wird, um eine Frage der Form: gilt $C \vdash^* w$? für ein beliebiges $C \in N$ und ein Wort w der Länge n , so bekommen wir folgende Ungleichung:

$$T(n) \geq \sum_{i=1}^{n-1} (T(i) + T(n-i))$$

Da $T(i) + T(n - i) \geq T(\frac{n}{2})$, folgt daraus $T(n) \geq n \cdot T(\frac{n}{2})$. Damit ist aber

$$T(n) \geq n \cdot \frac{n}{2} \cdots \frac{n}{2^{\log n}} = \frac{n^{\log n}}{2^{0+\dots+\log n}} = \frac{n^{\log n}}{2^{\frac{1}{2}(\log^2 n - \log n)}} \geq \frac{n^{\log n}}{2^{\frac{1}{2} \log^2 n}} = \frac{n^{\log n}}{n^{\frac{1}{2} \log n}} = n^{\frac{\log n}{2}}.$$

Der Zeitaufwand zwar kleiner geworden, aber immer noch größer als jedes Polynom.

Lösung 3: (Cocke, Younger, Kasami)

Im Lösungsansatz 2 haben wir dasselbe Problem mehrmals gelöst: Sei $w = a_1 \cdots a_i \cdots a_k \cdots a_n$, und $A \rightarrow BC$ eine Regel für ein B , so wird das Problem $A \vdash^* a_1 \cdots a_i$ gelöst, wenn die Teilung an der Stelle i geschieht, aber noch einmal, wenn die Teilung an der Stelle k geschieht, und zur Lösung des Problems $A \vdash^* a_1 \cdots a_k$ - eine Rekursionsstufe tiefer - das Wort $a_1 \cdots a_k$ wieder an der Stelle i geteilt wird. Da dies für alle $k \in \{i + 1, \dots, n\}$ passiert, kommt dasselbe Problem also schon in der zweiten Rekursionsstufe $(n - i - 1)$ -mal vor. In der dritten Rekursionsstufe wiederholt sich das Problem noch öfter, usw.

Um das wiederholte Lösen immer desselben Problems zu vermeiden, lösen wir alle möglicherweise vorkommenden Probleme im voraus - aber nur einmal - und merken uns die Ergebnisse. Diese Vorgehensweise nennt man "Dynamisches Programmieren". Man zahlt den Preis des hohen Speicheraufwands und der Lösung möglicherweise gar nicht benötigter Probleme, gewinnt aber die Sicherheit, kein Problem zweimal lösen zu müssen. Dieser Preis zahlt sich aus, wenn - wie hier - absehbar ist, daß die Zahl der Wiederholungen sehr groß ist.

Wir speichern die Ergebnisse in einer $n \times n$ - Matrix T ab, wobei das Feld $T_{i,j}$ alle Variablen aus N enthält, aus denen das Teilwort $a_i \cdots a_j$ ableitbar ist, also

$$T_{i,j} = \{A \mid A \vdash^* a_i \cdots a_j\} \quad (i, j \in \{1, \dots, n\}, i \leq j).$$

Ist $|j - i| > 1$, und $A \in T_{i,j}$, so muß es eine Regel $A \rightarrow BC$ geben und ein $k \in \{i, \dots, j - 1\}$, sodaß $B \vdash^* a_i \cdots a_k$ und $C \vdash^* a_{k+1} \cdots a_j$, d.h. $B \in T_{i,k}$ und $C \in T_{k+1,j}$. Für $j > i + 1$ ist also

$$T_{i,j} := \bigcup_{k=i}^{j-1} \{A \in N \mid \exists B, C \in N : A \rightarrow BC \text{ aus } P, B \in T_{i,k} \text{ und } C \in T_{k+1,j}\}.$$

Außerdem gilt $S \vdash^* a_1 \cdots a_n$ gdw. $S \in T_{1,n}$.

Auch ohne die konkrete Implementierung dieses Verfahrens zu kennen, können wir den Zeitaufwand schon abschätzen: Die Matrix T hat $\frac{1}{2}n^2$ Einträge $T_{i,j}$ (Nur die obere Hälfte über der Diagonalen wird gefüllt). Zur Berechnung eines Eintrags $T_{i,j}$ greifen wir auf $2(j - i)$ andere Einträge $T_{i,k}$ und $T_{k+1,j}$ zu. Da $j - i \leq n - 1 < n$, sind dies höchstens $\frac{n^2}{2} \cdot 2n = n^3$ Zugriffe. Also Zeit ist proportional zu n^3 .

Zur Implementierung: Uns interessiert nur das Feld $T_{1,n}$ der Matrix. Dennoch müssen wir dazu alle Felder berechnen. Wir berechnen die Felder Diagonale für Diagonale. Wir beginnen mit der Hauptdiagonalen (der 0-ten Nebendiagonalen) ($T_{i,i} \mid i = 1, \dots, n$): das ist einfach und kostet n Schritte. Dann beginnen wir mit der ersten Nebendiagonale darüber und zwar nacheinander von Zeile 1 bis Zeile $n - 1$. Angenommen wir haben die Felder der ersten $l - 1$ Diagonalen berechnet und von der Diagonalen l auch schon die ersten $i - 1$ Felder. Als nächstes wollen wir das Feld i in Diagonale l berechnen, d.h. das Feld $T_{i,j}$, wobei $j = l + i$. Um das Feld $T_{i,j}$ zu berechnen, benützen wir zwei Pointer. Mit dem einen laufen wir in der Zeile i von Feld i bis Feld $j - 1$ und mit dem anderen gleichzeitig synchron in der Spalte j von Feld $i + 1$ bis Feld j . Man beachte, daß wir bei unserem Vorgehen diese Felder alle schon berechnet haben, sie liegen alle links bzw. unter dem Feld $T_{i,j}$, also in einer der früher berechneten Diagonalen. Zur Berechnung des Feldes $T_{i,j}$ laufen wir also mit unseren Pointern je über $j - i$ Felder. Dies machen wir bis zur $(n - 1)$ -ten Nebendiagonalen, die nur noch aus dem Feld $T_{1,n}$ besteht.

Das führt zu folgendem Algorithmus:

```

for i = 1 to n do T(i, i) = {A | A → ai};
for l = 1 to n - 1 do begin
  for i = 1 to n - l do begin
    j := l + i; T(i, j) = ∅;
    for k = i to j - 1 do begin
      T(i, j) = T(i, j) ∪ {A | A → BC, B ∈ T(i, k), C ∈ T(k + 1, j)}
    endfor
  endfor
endfor

```

Beispiel 2.5.12

$w = baaba$

Regeln:

$S \rightarrow AB \mid BC$
 $A \rightarrow BA \mid a$
 $B \rightarrow CC \mid b$
 $C \rightarrow AB \mid a$

Matrix:

	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>
	1	2	3	4	5
1	<i>B</i>	<i>S, A</i>	–	–	<i>A, S, C</i>
2		<i>A, C</i>	<i>B</i>	<i>B</i>	<i>S, A, C</i>
3			<i>A, C</i>	<i>S, C</i>	<i>B</i>
4				<i>B</i>	<i>S, A</i>
5					<i>A, C</i>

Wir finden: $S \in T(1, 5)$, also ist $w \in L(G)$

Anmerkung 2.5.13 (Im Vorgriff auf Kapitel 3)

Wir können diesen Algorithmus auch auf einer Turingmaschine implementieren: Auf Band 1 schreiben wir die sich verkürzenden Zeilen der Matrix T :

$$1 \dots\dots\dots n\#2 \dots\dots\dots n\#3 \dots\dots\dots n\# \dots\dots\dots \#n\# .$$

Auf Band 2 schreiben wir die immer länger werdenden Spalten der Matrix T :

$$1\#12\#123\# \dots\dots\dots \#1 \dots\dots\dots n\# .$$

Um $T_{i,j}$ zu berechnen läuft der Kopf auf Band 1 im i -ten Block $\#i \dots\dots\dots \#n\#$ von i bis $j - 1$ synchron mit dem Kopf auf Band 2 im j -ten Block $\#1 \dots\dots\dots \#j\#$ von $i + 1$ bis j , und legen das Ergebnis auf jedem Band ab, und zwar auf Band 1 in Block i auf Feld j (das ist das Feld hinter dem zuletzt erreichten Feld) und auf Band 2 in Block j auf Feld i (das ist das Feld vor seinem Startfeld, hier muß der Kopf nochmal ein Stück zurücklaufen). Das nächste Feld in der Diagonale ist $T_{i+1,j+1}$. Also gehen die Köpfe einfach weiter in Block $i + 1$ bzw. $j + 1$. D.h. um eine Diagonale zu berechnen laufen die Köpfe jeweils einmal über ihr Band (Kopf 2 läuft höchstens das 3-fache der Bandlänge), legen also einen Weg von weniger als $3n^2$ Feldern zurück. Für alle n Diagonalen benötigt die Turingmaschine also höchstens $3n^3$ Schritte.

2.6 Abschlußeigenschaften der kontextfreien Sprachen

2.6.1 Reguläre und Boolesche Operationen und Homomorphismen.

Notation 2.6.1

Die Klasse der kontextfreien Sprachen bezeichnen wir mit CFL.

Satz 2.6.2

CFL ist abgeschlossen unter den Operationen Vereinigung, Konkatenation, Sternbildung und Homomorphismus.

Beweis: Sind $G_i = (N_i, T, P_i, S_i)$ ($i = 1, 2$, o.B.d.A. N_1, N_2, T paarweise disjunkt) die CFG's für die CFL's L_1 und L_2 , so ist $G = (N, T, P, S)$ die Grammatik für $L_1 \cup L_2$, $L_1 \cdot L_2$ bzw L_1^* , wenn $N = N_1 \cup N_2$, $S \notin N_1 \cup N_2 \cup T$, $P = P_1 \cup P_2 \cup P'$, wobei

$$P' := \begin{array}{ll} \{S \longrightarrow S_1 \mid S_2\} & \text{bei } \cup, \\ \{S \longrightarrow S_1 S_2\} & \text{bei } \cdot, \\ \{S \longrightarrow SS \mid S_1 \mid \varepsilon\} & \text{bei } * . \end{array}$$

Ist h ein Homomorphismus, so wird die Sprache $h(L_1)$ durch die Grammatik $G_h = (N_h, T, P_h, S_1)$ erzeugt mit $N_h := N \cup \{C_a \mid a \in T\}$, $P_h := P'' \cup \{C_a \longrightarrow h(a) \mid a \in T\}$, wo P'' aus P entsteht, indem man in allen Regeln von P die Terminals a durch die zugehörigen neuen Variablen C_a ersetzt. ■

Satz 2.6.3

CFL ist nicht abgeschlossen unter den Operationen Schnitt und Komplement.

Beweis: $\{a^i b^j c^j \mid i, j \in \mathbb{N}_+\} \cap \{a^i b^i c^j \mid i, j \in \mathbb{N}_+\} = \{a^n b^n c^n \mid n \in \mathbb{N}_+\}$ ist der Schnitt zweier kontextfreier Sprachen, selbst aber nicht kontextfrei. Wegen $A \cap B = \overline{\overline{A} \cup \overline{B}}$ kann CFL dann auch unter Komplement nicht abgeschlossen sein. ■

2.6.2 Produktautomat

Für das weitere ist es hilfreich den uns von den finiten Automaten her geläufigen Akzeptierungsbegriff durch Endzustand auch für NPDA einzuführen. Die NPDA's, wie wir sie bisher betrachtet haben akzeptieren durch leeren Keller, d.h. ein Inputwort gilt als erkannt, wenn der Keller des NPDA's leer wird (und der NPDA damit stoppt, da kein Befehl anwendbar ist, wenn der Keller leer ist). Genauso gut hätten wir dem NPDA aber auch eine Menge von Endzuständen geben und erklären können, daß ein Inputwort erkannt ist, wenn der NPDA eine Situation mit Endzustand erreicht. Dieses Modell ist geeigneter, wenn wir einen NPDA mit einem FA koppeln wollen, z.B. als Produktautomaten. Wir führen als nachträglich ein:

Definition 2.6.4 (Kellerautomat mit Endzuständen)

Seien Q, Γ endliche Alphabeten, $\Sigma \subseteq \Gamma$, $q_0 \in Q$, $F \subseteq Q$, $\perp \in \Gamma$, δ eine endliche Teilmenge von $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$. Dann heißt die Struktur

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$$

Kellerautomat mit Endzuständen. Wir nennen Q die Menge der Zustände, Γ das Kelleralphabet, Σ das Inputalphabet, q_0 den Startzustand, \perp das Bodensymbol, F die Menge der Endzustände (oder akzeptierenden Zustände) und δ die Befehlsmenge.

Die Sprache von M ist $L(M) = \{w \in \Sigma^* \mid (q_0, \varepsilon, \perp) \vdash^* (q, w, V) \text{ für ein } q \in F \text{ und ein } V \in \Gamma^*\}$.

Lemma 2.6.5

Zu jedem NPDA (der mit leerem Keller akzeptiert) existiert ein äquivalenter Kellerautomat mit Endzuständen und umgekehrt.

Beweis: \implies : Sei $P = (Q, \Sigma, \Gamma, \delta, q_0, \perp)$ ein NPDA (der mit leerem Keller akzeptiert). O.B.d.A. druckt und überschreibt M das Bodensymbol \perp nie. Wir definieren einen zu M äquivalenten Kellerautomaten mit Endzuständen $K = (Q \cup \{f\}, \Sigma, \Gamma, \delta', q_0, \perp, \{f\})$ wie folgt:

Ersetze jeden Befehl $(p, a, \perp, q, \varepsilon)$ von P $(p, q \in Q, a \in \Sigma \cup \{\varepsilon\})$ durch den Befehl $(p, a, \perp, f, \varepsilon)$ (f ist

ein neuer Zustand, der einzige Endzustand). Beim Leeren des Kellers geht K in seinen Endzustand f über.

\Leftarrow : Sei $K = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ ein Kellerautomat mit Endzuständen. Wir konstruieren einen äquivalenten NPDA $P = (Q \cup \{f\}, \Sigma, \Gamma, \delta', q_0, \perp)$ (der $L(K)$ mit leerem Keller akzeptiert). Für jeden Endzustand $q \in F$ füge zu δ noch folgende Befehle hinzu: $(q, \varepsilon, X, f, \varepsilon)$ für alle $X \in \Gamma$, d.h. in einem Endzustand kann P den Zustand spontan in den neuen Zustand f wechseln. Außerdem seien in δ' noch folgende neue Löschbefehle $(f, \varepsilon, X, f, \varepsilon)$ für alle $X \in \Gamma$, d.h. in diesem neuen Zustand f löscht P den Keller bis er leer ist, ohne weiteren Input zu lesen. ■

Satz 2.6.6

CFL ist abgeschlossen unter Schnitt mit regulären Sprachen.

Beweis: Sei R eine reguläre Sprache und $L \in \text{CFL}$. Sei $M = (Q, \Sigma, \delta_M, q_0, F)$ ein finiter Automat (ohne ε -Befehle) für R , und $K = (Q', \Sigma, \Gamma, \delta_K, q'_0, \perp, F')$ ein Kellerautomat mit Endzuständen für L . Wir konstruieren einen Kellerautomaten P mit Endzuständen, der die Worte akzeptiert, die von beiden Automaten M und K akzeptiert werden. Diese Konstruktion kennen wir schon aus dem ersten Kapitel: der Produktautomat simuliert sozusagen parallel die Rechnungen beider Maschinen. Wir definieren den Produktautomaten von M und K wie folgt:

$$P = M \times K = (Q \times Q', \Sigma, \Gamma, \delta_P, (q_0, q'_0), \perp, F \times F') \text{ mit} \\ ((p, p'), a, A, (q, q'), V) \in \delta_P \text{ gdw. } (p, a, q) \in \delta_M \text{ und } (p', a, A, q', V) \in \delta_K$$

für alle $p, q \in Q$; $p', q' \in Q'$; $a \in \Sigma$; $A \in \Gamma$, $V \in \Gamma^*$. Damit der finite Automat auf den Kellerautomaten wartet, falls der noch spontane Befehle (ohne Input) ausführt, seien außerdem noch folgende Befehle aus δ_P : $((p, p'), \varepsilon, A, (p, q'), V) \in \delta_K$ für alle $p \in Q$, falls $(p', \varepsilon, A, q', V) \in \delta_K$. ■

Satz 2.6.7

CFL ist abgeschlossen unter inversem Homomorphismus.

Beweis: Sei L eine CFL, $M = (Q, \Sigma, \Gamma, \delta, s, Z, F)$ ein NPDA mit Endzuständen für L und h ein Homomorphismus. Es ist $h^{-1}(L) = \{w \in \Sigma^* \mid h(w) \in L\}$. Die Idee für einen NPDA mit Endzuständen $M' = (Q', \Sigma, \Gamma, \delta', (s, \varepsilon), Z, F')$ für $h^{-1}(L)$ ist folgende: Liest M' den Input a , so notiert M' sich $h(a)$ in der Kontrolleinheit und arbeitet $h(a)$ ab, als ob dies der Input wäre.

Zunächst seien also für alle $a \in \Sigma$ die Befehle $((q, \varepsilon), a, X, (q, h(a)), X)$ aus δ' (für alle $q \in Q$ und $X \in \Gamma$). Damit speichert sich M' das Bild $h(a)$ des Inputsymbols a im Zustand. Zur Abarbeitung dieses Bildes habe M' für jeden Befehl (p, a, A, q, V) aus δ die Befehle $((p, aw), \varepsilon, A, (q, w), V)$ aus δ' (für alle $w \in \Sigma^*$ mit $|w| < \max\{|h(a)| \mid a \in \Sigma\}$). Die Zustandsmenge von M' ist also $Q' = \{(q, w) \mid q \in Q, w \in \Sigma^* \text{ mit } |w| \leq \max\{|h(a)| \mid a \in \Sigma\}\}$. Desweiteren darf M' nur akzeptieren, wenn das zum letzten gelesenen a in der Kontrolleinheit notierte $h(a)$ vollständig abgearbeitet ist. Deshalb setzen wir $F' := \{(p, \varepsilon) \mid p \in F\}$. ■

Satz 2.6.8

CFL ist abgeschlossen unter inverser GSM-Abbildung.

Beweis: Sei $M = (Q_M, \Sigma, \Sigma, \delta_M, q_M)$ eine GSM für die GSM-Abbildung $g : \Sigma^* \rightarrow \Sigma^*$ und $K = (Q_K, \Sigma, \Gamma, \delta_K, q_K, \perp, F_K)$ ein Kellerautomat mit Endzuständen für L . Gesucht ist ein Kellerautomat K' , der die Sprache $g^{-1}(L) = \{w \in \Sigma^* \mid g(w) \in L\}$ erkennt. Wie beim inversen Homomorphismus ist die Idee: Liest K' ein Inputsymbol a , so wandelt er es in das Bild unter g und simuliert nun K auf diesem neuen Input. Anders als beim Homomorphismus ist das Bild von a abhängig vom

Zustand der GSM M , die also parallel zu K auf dem eingelesenen Inputwort mitlaufen mu. Damit bentigen wir eine Produktautomatenkonstruktion:

$K' = (Q_M \times Q_K \times G, \Sigma, \Gamma, \delta', (q_M, q_K, \varepsilon), \perp, Q_M \times F_K \times \{\varepsilon\})$ mit
 $G = \{w \in \Sigma^* \mid |w| \leq \max\{|v| \mid (p, a, v, q) \in \delta_M (a \in \Sigma; p, q \in Q_M)\}\}$ und
 $((p, q, \varepsilon), a, X, (p', q, \nu), X) \in \delta'$ fr alle $q \in Q_K, X \in \Gamma$, falls $(p, a, v, p') \in \delta_M$
 (K' liest a und schreibt das gegenwrtige Bild von a unter M in seinen Zustand),
 $((p, q, bu), \varepsilon, A, (p, q', u), V) \in \delta'$ fr alle $p \in Q_M$ und $bu \in G$ mit $b \in \Sigma \cup \{\varepsilon\}$, falls $(q, b, A, q', V) \in \delta_K$
 (K' simuliert K auf dem im Zustand gespeicherten Bild seines Inputsymbols a). ■

Satz 2.6.9

CFL ist abgeschlossen unter GSM-Abbildung.

Beweis: Sei $M = (Q_M, \Sigma, \Sigma, \delta_M, q_M)$ eine GSM fr die GSM-Abbildung $g : \Sigma^* \rightarrow \Sigma^*$ und $K = (Q_K, \Sigma, \Gamma, \delta_K, q_K, F_K)$ ein Kellerautomat mit Endzustnden fr L . Gesucht ist ein Kellerautomat K' fr die Sprache $g(L) = \{g(w) \mid w \in L\}$. Die Idee ist folgende: K' liest ein Stck seines Inputs - sagen wir $b_1 \cdots b_n$ - speichert ihn in seinem Zustand, rt ein Symbol a als Urbild von $b_1 \cdots b_n$, berprft, ob das Bild von a unter der GSM M , die parallel mit simuliert wird, gerade $b_1 \cdots b_n$ ist, und simuliert, wenn alles stimmt, den Kellerautomat K auf diesem Input a .

Der Produktautomat, der dies leistet ist:

$K' = (Q_M \times Q_K \times G, \Sigma, \Gamma, \delta', (q_M, q_K, \varepsilon), Q_M \times F_K \times \{\varepsilon\})$ mit
 $G = \{w \in \Sigma^* \mid |w| \leq \max\{|v| \mid (p, a, v, q) \in \delta_M (a \in \Sigma; p, q \in Q_M)\}\}$ und
 $((p, q, u), b, X, (p, q, ub), X) \in \delta'$ fr alle $p \in Q_M, q \in Q_K, ub \in G, b \in \Sigma, X \in \Gamma$
 (K' liest ein Stck seines Inputs und speichert ihn in seinem Zustand),
 $((p, q, u), \varepsilon, A, (p', q', \varepsilon), V)$, falls $(p, a, u, p') \in \delta_M$ und $(q, a, A, q', V) \in \delta_K$
 (K' simuliert K auf a , wenn das im Zustand gespeicherte Stck Input Bild von a unter M ist),
 $((p, q, \varepsilon), \varepsilon, A, (p, q', \varepsilon), V)$, falls $(q, \varepsilon, A, q', V) \in \delta_K$
 (K macht eine ε -Bewegung, M bleibt stehen). ■

Aufgabe 2.6.10

Zeigen Sie mithilfe dieser Stze, da die erste Sprache aus Beispiel 2.4.4 nicht kontextfrei ist.

2.6.3 AFL [Ergnzung]

Definition 2.6.11

Eine Klasse von Sprachen, die nicht leer ist und auch nicht nur die leere Sprache enthlt heit

- *Zylinder*, wenn sie unter inversem Homomorphismus und Schnitt mit regulren Mengen abgeschlossen ist $(h^{-1}, \cap R)$,
- *full Trio (Kegel)*, wenn sie unter Homomorphismus, inversem Homomorphismus und Schnitt mit regulren Mengen abgeschlossen ist $(h, h^{-1}, \cap R)$,
- *full AFL*, wenn sie zudem unter Vereinigung, Konkatenation und ε -freier Iteration abgeschlossen ist $(\cup, \cdot, ^+, h, h^{-1}, \cap R)$,

Definition 2.6.12

Eine Sprache L_0 aus einer Sprachklasse \mathcal{L} heit full-Trio-Generator von \mathcal{L} , wenn sich jede Sprache $L \in \mathcal{L}$ aus L_0 mit Hilfe der Trio-Operationen Homomorphismus, inverser Homomorphismus und Schnitt mit regulren Mengen $(h, h^{-1}, \cap R)$ erzeugen lt.

Anmerkung 2.6.13

Damit haben wir gezeigt, daß CFL ein full-AFL ist. Wir werden unten zeigen, daß D_2 (Dyck-Sprache vom Grad 2) ein full-Trio-Generator von CFL ist.

2.7 Homomorphiesatz der kontextfreien Sprachen

Definition 2.7.1 (Dyck-Sprache D_r vom Grade r)

Sei $\Sigma_r = \{a_1, \dots, a_r, b_1, \dots, b_r\}$ mit paarweise einander zugeordneten Symbolen (a_i, b_i) . (Wir können uns r Klammertypen vorstellen mit jeweils öffnenden und schließenden Klammern)

Definition der Worte von D_r über den Aufbau:

1. das leere Wort ε ist in D_r
2. sind die Worte u, v in D_r , so sind auch die Worte uv und $a_i u b_i$ für alle $i \leq r$ in D_r
3. keine anderen Worte sind in D_r .

Aus dieser induktiven Definition können wir einfach eine CFG $G = (\{S\}, \Sigma_r, P, S)$ entwickeln, die D_r erzeugt, wobei $P = \{S \rightarrow a_i S b_i \mid S S \mid \varepsilon, \text{ wo } i = 1, \dots, r\}$.

Anmerkung 2.7.2

Eine mehr algebraische Möglichkeit, die Worte aus D_r zu beschreiben, ist, sich die schließende Klammer eines Typs als die (bzgl. der Konkatenation) Inverse der öffnenden Klammer desselben Typs vorzustellen. Dann besteht D_r gerade aus den Worten, die zu ε werden, wenn für jeden der r Klammertypen folgende Gleichung gilt: $() = \varepsilon$, genauer: $a_i b_i = \varepsilon$ ($i = 1, \dots, r$).

Wir sagen auch: D_r ist die Menge der korrekten Klammersausdrücke über r Klammertypen. So ist etwa das Klammer-Wort $()$ aus D_2 , aber nicht die Worte: $(()$ und $) [] (.$

Anmerkung 2.7.3

Man kann sich unter einem korrekten Klammersausdruck aus D_r auch einen orientierten binären Baum mit Knoten vorstellen, die durch Labels aus $\{1, \dots, r\}$ bewertet sind: Jeder Knoten entspricht einem Klammerpaar, das die Söhne des Knotens umklammert.

Das Durchlaufen des Klammersausdrucks von links nach rechts entspricht einem depth-first-Durchlauf durch den Baum: Erreichen einer offenen Klammer entspricht dem ersten Besuch des entsprechenden Knotens, den man von oben erreicht und von dem aus dann der linke Sohn (der einzige Sohn, falls es nur einen Sohn gibt) angelaufen wird. Das Erreichen der zugehörigen geschlossenen Klammer entspricht dem letzten Besuch des entsprechenden Knotens, der jetzt von unten angelaufen wird, nachdem der unter ihm liegende Teilbaum abgearbeitet worden ist und von dem aus der Vater angelaufen wird.

Beispiel 2.7.4

Der Klammersausdruck mit den 7 Klammertypen $\overset{i}{(} \overset{i}{)}$ ($i = 1, \dots, 7$)

$$\overset{1}{(} \overset{4}{(} \overset{5}{(} \overset{5}{)} \overset{4}{)} \overset{2}{(} \overset{6}{(} \overset{7}{(} \overset{7}{)} \overset{6}{)} \overset{3}{(} \overset{4}{(} \overset{5}{(} \overset{5}{)} \overset{4}{)} \overset{4}{(} \overset{5}{(} \overset{5}{)} \overset{4}{)} \overset{3}{)} \overset{2}{)} \overset{6}{(} \overset{7}{(} \overset{7}{)} \overset{6}{)} \overset{1}{)}$$

entspricht einem orientierten, binären, knotenbewerteten Baum der Höhe 4.

Satz 2.7.5 (Homomorphiesatz für kontextfreie Sprachen)

Für jede kontextfreie Sprache L gibt es Homomorphismen g und h und einen regulären Ausdruck R , so daß $L = g(h^{-1}(D_2) \cap R)$.

Zum Beweis dieses Satzes führen wir noch einmal eine veränderte, aber äquivalente Form des Pushdown-Automaten ein, den Push-Pop-Automaten:

Definition 2.7.6 (Push-Pop-Automat)

Seien Q (Zustandsmenge), Σ (Inputalphabet) und Γ (Kelleralphabet) Alphabete, $q_0 \in Q$ (Startzustand), $F \subseteq Q$ (Endzustandsmenge), $OP = \{\text{Push } A, \text{Pop } A \mid A \in \Gamma\} \cup \{\text{nop}\}$ (Operatormenge) und $\delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times OP \times Q$ (Befehlsmenge), dann heißt die Struktur

$$M = (Q, \Sigma, \Gamma, OP, \delta, q_0, F)$$

Push-Pop-Automat (PPA). Die Situation eines PPA wird wie bei den anderen Varianten beschrieben durch ein Tripel $(q, w, W) \subseteq Q \times \Sigma^* \times \Gamma^*$ (Zustand, gelesener Input, Kellerinhalt). Die Einschrittrelation \vdash wird definiert durch:

$$\begin{aligned} (p, w, W) &\vdash (q, wa, BW) && \text{falls } (p, a, \text{Push } B, q) \in \delta, \\ (p, w, BW) &\vdash (q, wa, W) && \text{falls } (p, a, \text{Pop } B, q) \in \delta, \\ (p, w, W) &\vdash (q, wa, W) && \text{falls } (p, a, \text{nop}, q) \in \delta. \end{aligned}$$

M akzeptiert genau dann, wenn der Keller in einem Endzustand leer wird. M heißt *binärer PPA*, wenn $|\Gamma| = 2$.

Aufgabe 2.7.7

Zu jedem NPDA existiert ein äquivalenter PPA und umgekehrt.

Aufgabe 2.7.8

Zu jedem PPA existiert ein äquivalenter binärer PPA. [Hinweis: kodieren Sie die Kellerinhalte binär]

Beweis: (des Homomorphiesatzes): Sei also L eine CFL und $M = (Q, \Sigma, \{A, B\}, OP, \delta, q_0, F)$ ein binärer PPA, der L erkennt. Zunächst definieren wir den Homomorphismus h , der auf Befehle angewandt wird und dort die Operation herausfiltert und in Form einer Klammer abbildet (offene Klammer für Push, geschlossene für Pop)

$$\begin{aligned} h \text{ von } (Q \times \Sigma \cup \{\varepsilon\} \times OP \times Q)^* &\longrightarrow \{ ((,) , ((,)))^* \text{ vermöge:} \\ h(p, a, \text{push } X, q) &= ((,)^X, \\ h(p, a, \text{pop } X, q) &=)^X, \\ h(p, a, \text{nop}, q) &= \varepsilon \quad (X \in \{A, B\}). \end{aligned}$$

Ist W aus D_2 so ist $h^{-1}(W)$ die Menge aller Befehlsfolgen, deren Kelleroperationen durch die Klammern von W beschrieben werden. Da W ein korrekt geklammerter Ausdruck ist, ist die Folge der Kelleroperationen ebenfalls korrekt, d.h. im Keller wird nur gelöscht, was zuvor an dieser Stelle gedruckt worden ist. $h^{-1}(D_2)$ enthält also alle kellerkorrekten Befehlsfolgen.

Von einer korrekten Befehlsfolge erwarten wir allerdings, daß sie nicht nur kellerkorrekt ist: es sollen auch die Zustände zusammenpassen, d.h. hat ein Befehl der Folge den PPA M in einen Zustand q gebracht, so soll der nachfolgende Befehl, mit diesem Zustand beginnen (sonst wäre er ja nicht anwendbar). Wir schneiden uns aus der Menge $h^{-1}(D_2)$ der kellerkorrekten Befehlsfolgen also die heraus, die auch noch zustandskorrekt sind. Die zustandskorrekten Befehlsfolgen beschreiben wir durch eine reguläre Sprache R .

Dazu konstruieren wir uns einen finiten Automaten F , der alle Worte (alle Befehlsfolgen) aus $(Q \times \Sigma \cup \{\varepsilon\} \times OP \times Q)^*$ erkennt mit

1. alle Befehle sind aus δ ,
2. der erste Befehl beginnt mit dem Startzustand von M ,
3. jeder andere Befehl beginnt mit dem Zustand, mit dem der vorhergehende Befehl endet,

4. der letzte Befehl endet mit dem Endzustand von M .

R sei nun die Sprache von F . Dann ist $h^{-1}(D_2) \cap R$ die Menge aller keller- und zustandskorrekten Befehlsfolgen mit Befehlen von M , d.h. also aller Befehlsfolgen von M , die eine korrekte akzeptierende Folge von Situationen (einen akzeptierenden w -Lauf für ein $w \in \Sigma^*$) von M produzieren.

Jetzt benötigen wir nur noch einen Homomorphismus g , der uns aus einem akzeptierenden w -Lauf den Input w herausfiltert, d.h. jeden Befehl einer korrekten Befehlsfolge von M auf sein Inputsymbol abbildet:

$g : (Q \times \Sigma \cup \{\varepsilon\} \times OP \times Q)^* \rightarrow \Sigma^*$ vermöge $(p, a, \text{op}, q) \mapsto a$ für alle $p, q \in Q$, $\text{op} \in OP$ und $a \in \Sigma$. Jetzt ist $g(h^{-1}(D_2) \cap R)$ gerade die Menge der Inputworte $w \in \Sigma^*$, für die es eine korrekte Befehlsfolge von M , d.h. einen akzeptierenden w -Lauf von M gibt. Also ist $g(h^{-1}(D_2) \cap R) = L(M)$. ■

Damit hat CFL also einen full Trio-Generator, nämlich die Dycksprache D_2 vom Grad 2. CFL ist also ein principal full Trio. Wir können darüber hinaus zeigen, daß CFL sogar einen Zylindergenerator hat, ja einen Generator, aus dem jede kontextfreie Sprache durch inversen Homomorphismus erzeugt werden kann. (In der Sprache der Reduktionen bedeutet das, daß CFL ein bzgl. homomorpher Reduktion vollständiges Problem H (the hardest context-free language) besitzt: jede kontextfreie Sprache läßt sich durch einen Homomorphismus auf H reduzieren).

Aufgabe 2.7.9

Zu jedem PPA gibt es einen PPA ohne ε -Befehle.

Satz 2.7.10

Die Klasse CFL der kontextfreien Sprachen hat einen Zylindergenerator H (the hardest context-free language): jede ε -freie CFL ist homomorphes Urbild von H , d.h. ist L eine ε -freie CFL ($\varepsilon \notin L$), so gibt es einen Homomorphismus h_L mit $L = h_L^{-1}(H)$.

Beweis: Der Einfachheit halber nehmen wir an, daß alle Alphabete Teilmengen von \mathbb{N} sind. Zu jeder CFL L existiert ein PPA $M = (Q, \Sigma, \Gamma, OP, \delta, 0, \{1\})$ mit Alphabeten aus \mathbb{N} . (Damit ist macht der Term 1^p Sinn, nämlich ein Wort aus p Einsen. Ebenso ist $(^A$ ein Wort von A offenen runden Klammern).

I. Zunächst definieren wir die hardest context-free language H : Ein Wort der Form

$$\begin{array}{lll} 1^p \# [(^A \# 1^q & \text{oder} & (\text{entspricht } (p, -, \text{Push } A, q)) \\ 1^p \#)^A \# 1^q & \text{oder} & (\text{entspricht } (p, -, \text{Pop } A, q)) \\ 1^p \# [] \# 1^q & (p, q, A \in \mathbb{N}) & (\text{entspricht } (p, -, \text{nop}, q)) \end{array}$$

nennen wir eine *Anweisung*. Ein *Block* B ist eine Folge von Anweisungen: $B = I_1 \& \dots \& I_k$ ($k \in \mathbb{N}$). Ein *Blockwort* W ist eine Folge von Blöcken: $W = B_1 \$ \dots \$ B_l \$$ ($l \in \mathbb{N}$).

Ein Blockwort $W = B_1 \$ \dots \$ B_l \$$ ($l > 0$) ist aus H , gdw. gilt: es gibt eine Folge I_1, \dots, I_l von Anweisungen mit

1. $I_i = 1^{p_i} \# u_i \# 1^{q_i}$ ist eine Anweisung von B_i ($i = 1, \dots, l$),
2. $p_1 = 0, q_l = 1, q_i = p_{i+1}$ ($i = 1, \dots, l - 1$),
3. $u_1 \dots u_l \in D_2$ über $\{(,), [,]\}$.

Demnach ist $\varepsilon \notin H$.

II. Wir zeigen: $H \in \text{CFL}$:

Der PPA M für H rät aus jedem Block des Inputworts eine Anweisung und überprüft, die Bedingungen 2) und 3). Für beides benötigt M seinen Keller.

III. Wir definieren h_L :

Sei $P = (Q, \Sigma, \Gamma, \text{OP}, \delta, q_0, F)$ ein PPA für L ohne ε -Befehle. OBdA sei $Q = \{0, \dots, k\}$ für ein $k \in \mathbb{N}$, $q_0 = 0$ und $F = \{1\}$. Wir geben den Befehlen aus δ eine Ordnung, indem wir sie durchnummerieren.

Eine a -Anweisung von P ($a \in \Sigma$) ist:

$$\begin{aligned} 1^p \# [(A \# 1^q) & , \text{ wenn } (p, a, \text{push } A, q) \text{ ein Befehl von } P \text{ ist, bzw.} \\ 1^p \#)^A] \# 1^q & , \text{ wenn } (p, a, \text{pop } A, q) \text{ ein Befehl von } P \text{ ist, bzw} \\ 1^p \# [] \# 1^q & , \text{ wenn } (p, a, \text{nop}, q) \text{ ein Befehl von } P \text{ ist.} \end{aligned}$$

Sei B_a die Folge der a -Anweisungen von P getrennt durch $\$$ in ihrer Ordnung und $h(a) = B_a \$$.

IV. Zeige $L = h_L^{-1}(H)$:

$$\begin{aligned} a_1 \cdots a_n \in L & \iff \text{es existiert ein akzeptierender } a_1 \cdots a_n\text{-Lauf } \mathcal{L} \text{ von } P \\ & \iff \text{es existiert eine Anweisungsfolge } I_1, \dots, I_n \text{ zu } \mathcal{L} \\ & \iff B_{a_1} \$ \cdots \$ B_{a_n} \$ \in H. \end{aligned}$$

■