

# **Informatik IV**

## **Theoretische Informatik**

(früher: Grundlagen der Theoretischen Informatik:  
Formale Sprachen und Automaten,  
Berechenbarkeit und NP-Vollständigkeit)

**Vorlesungsskript**

**Sommersemester 2024**

**Skript: Prof. Dr. J. Rothe**  
**Dozentin: M. Mutz**



## Vorbemerkungen

Dieses Skript wurde von Prof. Dr. J. Rothe erstellt und wird seit dem Jahr 2000 regelmäßig überarbeitet,

- in der Regel von ihm selbst: 2000–2007, 2010, 2011, 2013, 2016, 2019, 2022;
- von PD Dr. F. Gurski: 2008;
- von Dr. G. Erdélyi: 2009;
- von Prof. Dr. M. Leuschel: 2012, 2014, 2017, 2020 und 2023;
- von Jun.-Prof. Dr. D. Baumeister: 2015, 2018 und 2021;
- und von Mareike Mutz: 2024.

Das Skript soll und kann kein Lehrbuch ersetzen! Es soll und kann Ihnen nicht die Teilnahme an den Vorlesungen und Übungen ersparen. Für eine erfolgreiche Teilnahme an der Veranstaltung „Informatik IV“ müssen Sie zusätzlich an den Vorlesungen und Übungen aktiv teilnehmen und in Lehrbüchern lesen.

Dieses Skript wird ständig aktualisiert und kann von der Vorlesungs-Ilias-Seite heruntergeladen werden. Am Ende des Semesters enthält es alle für die Prüfung nötigen Begriffe, Definitionen und Sätze, zum Teil auch zusätzliche Erläuterungen, Beispiele und Beweise. Manche Abschnitte dieses Skripts sind grau statt schwarz dargestellt. Das bedeutet, dass diese Teile nicht prüfungsrelevant sind und in der Vorlesung und den Übungen auch nicht besprochen werden. Wer möchte, kann diese Teile natürlich trotzdem lesen.

*Stand der Änderungen: 9. April 2024.*

## Literatur

- Alexander Asteroth und Christel Baier: „Theoretische Informatik. Eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen mit 101 Beispielen“. Pearson Studium, 2002.
- Norbert Blum: „Theoretische Informatik. Eine anwendungsorientierte Einführung“. Oldenbourg, 2. Auflage, 2001.
- Katrin Erk und Lutz Priese: „Theoretische Informatik. Eine umfassende Einführung“. Springer-Verlag, 3. Auflage, 2008.
- Gerhard Goos: „Vorlesungen über Informatik 3. Berechenbarkeit, formale Sprachen, Spezifikationen“. Springer-Verlag, 1997.
- John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman: „Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie“. Pearson Studium, 2. Auflage, 2002.
- Jörg Rothe: „Komplexitätstheorie und Kryptologie. Eine Einführung in Kryptokomplexität“. eXamen.press, Springer-Verlag, Berlin, Heidelberg, 2008. (Nur die Abschnitte 2.2 und 3.5.)
- Arto K. Salomaa: „Formale Sprachen“. Springer-Verlag, 1978.
- Uwe Schöning: „Theoretische Informatik – kurz gefasst“. Spektrum Akademischer Verlag, 5. Auflage, 2008.
- Gottfried Vossen und Kurt-Ulrich Witt: „Grundkurs Theoretische Informatik“. Vieweg Verlag, 4. Auflage, 2006.
- Klaus W. Wagner: „Theoretische Informatik. Eine kompakte Einführung“. Springer-Verlag, 2. Auflage, 2003.
- Ingo Wegener: „Theoretische Informatik. Eine algorithmenorientierte Einführung“. Teubner, 3. Auflage, 2005.



# Inhaltsverzeichnis

<b>I</b>	<b>Formale Sprachen und Automaten</b>	<b>9</b>
<b>1</b>	<b>Grundbegriffe</b>	<b>11</b>
1.1	Wörter, Sprachen und Grammatiken . . . . .	11
1.2	Die Chomsky-Hierarchie . . . . .	17
<b>2</b>	<b>Reguläre Sprachen</b>	<b>23</b>
2.1	Endliche Automaten . . . . .	23
2.2	Reguläre Ausdrücke . . . . .	32
2.3	Gleichungssysteme . . . . .	37
2.4	Das Pumping-Lemma . . . . .	40
2.5	Satz von Myhill und Nerode und Minimalautomaten . . . . .	43
2.5.1	Der Satz von Myhill und Nerode . . . . .	43
2.5.2	Minimalautomaten . . . . .	46
2.6	Abschlusseigenschaften regulärer Sprachen . . . . .	49
2.7	Charakterisierungen regulärer Sprachen . . . . .	50
<b>3</b>	<b>Kontextfreie Sprachen</b>	<b>51</b>
3.1	Normalformen . . . . .	51
3.2	Das Pumping-Lemma . . . . .	58
3.3	Der Satz von Parikh . . . . .	62
3.4	Abschlusseigenschaften kontextfreier Sprachen . . . . .	65
3.5	Der Algorithmus von Cocke, Younger und Kasami . . . . .	67
3.6	Kellerautomaten . . . . .	71
<b>4</b>	<b>Deterministisch kontextfreie Sprachen</b>	<b>83</b>
4.1	Deterministische Kellerautomaten . . . . .	83
4.2	LR( $k$ )- und LL( $k$ )-Grammatiken . . . . .	86
4.3	Anwendung: Syntaxanalyse durch LL( $k$ )-Parser . . . . .	93

<b>5</b>	<b>Kontextsensitive und <math>\mathcal{L}_0</math>-Sprachen</b>	<b>97</b>
5.1	Turingmaschinen . . . . .	97
5.2	Linear beschränkte Automaten . . . . .	102
5.3	Zusammenfassung . . . . .	106
<b>II</b>	<b>Berechenbarkeit</b>	<b>109</b>
<b>6</b>	<b>Intuitiver Berechenbarkeitsbegriff und die These von Church</b>	<b>111</b>
<b>7</b>	<b>Turing-Berechenbarkeit</b>	<b>117</b>
<b>8</b>	<b>LOOP-, WHILE- und GOTO-Berechenbarkeit</b>	<b>121</b>
8.1	LOOP-Berechenbarkeit . . . . .	121
8.2	WHILE-Berechenbarkeit . . . . .	124
8.3	GOTO-Berechenbarkeit . . . . .	125
<b>9</b>	<b>Primitiv rekursive und partiell rekursive Funktionen</b>	<b>129</b>
9.1	Primitiv rekursive Funktionen . . . . .	129
9.2	Die Ackermann-Funktion . . . . .	133
9.3	Allgemein und partiell rekursive Funktionen . . . . .	136
9.4	Der Hauptsatz der Berechenbarkeitstheorie . . . . .	139
<b>10</b>	<b>Entscheidbarkeit und Aufzählbarkeit</b>	<b>141</b>
10.1	Einige grundlegende Sätze . . . . .	141
10.2	Entscheidbarkeit . . . . .	143
10.3	Rekursiv aufzählbare Mengen . . . . .	148
<b>11</b>	<b>Unentscheidbarkeit</b>	<b>161</b>
11.1	Der Satz von Rice . . . . .	161
11.2	Reduzierbarkeit . . . . .	163
11.3	Das Postsche Korrespondenzproblem . . . . .	167
11.4	Unentscheidbarkeit in der Chomsky-Hierarchie . . . . .	173
11.5	Zusammenfassung . . . . .	177
<b>III</b>	<b>NP-Vollständigkeit</b>	<b>179</b>
<b>12</b>	<b>Probleme in P und NP</b>	<b>181</b>
12.1	Deterministische Polynomialzeit . . . . .	181
12.2	Das Erfüllbarkeitsproblem der Aussagenlogik . . . . .	182

12.3 Nichtdeterministische Polynomialzeit . . . . .	186
<b>13 NP-Vollständigkeit und der Satz von Cook</b>	<b>189</b>
13.1 NP-Vollständigkeit . . . . .	190
13.2 Der Satz von Cook . . . . .	192





# **Teil I**

## **Formale Sprachen und Automaten**



# Kapitel 1

## Grundbegriffe

### 1.1 Wörter, Sprachen und Grammatiken

#### Definition 1.1 (Alphabet, Wort und Sprache)

- Ein Alphabet ist eine endliche, nichtleere Menge  $\Sigma$  von Buchstaben (oder Symbolen).  
*Beispiel:*  $\Sigma_1 = \{a, b, c\}$  oder  $\Sigma_2 = \{0, 1\}$ .
- Ein Wort über  $\Sigma$  ist eine endliche Folge von Elementen aus  $\Sigma$ .  
*Beispiel:*  $w_1 = aabc$  und  $w_2 = caba$  sind Wörter über  $\Sigma_1$ ;  $w_3 = 1110$  und  $w_4 = 0101$  sind Wörter über  $\Sigma_2$ .
- Die Länge eines Wortes  $w$  (bezeichnet mit  $|w|$ ) ist die Anzahl der Symbole in  $w$ .  
*Beispiel:*  $|010| = 3$ .
- Das leere Wort ist das eindeutig bestimmte Wort der Länge 0 und wird mit dem griechischen Buchstaben  $\lambda$  („Lambda“) bezeichnet<sup>1</sup>.  
(Beachte:  $\lambda$  ist nicht als Symbol eines Alphabets erlaubt.)  
Es gilt  $|\lambda| = 0$ .
- Die Menge aller Wörter über  $\Sigma$  bezeichnen wir mit  $\Sigma^*$ .  
*Beispiel:*  $\{0, 1\}^* = \{\lambda, 0, 1, 00, 01, 10, 11, 000, \dots\}$ .
- Eine (formale) Sprache (über  $\Sigma$ ) ist eine jede Teilmenge von  $\Sigma^*$ .  
*Beispiel:*  $L = \{00, 10\} \subseteq \Sigma^* = \{0, 1\}^*$ .

---

<sup>1</sup>In einigen Büchern (z.B. Schöning: Theoretische Informatik - kurz gefasst) wird das leere Wort auch mit dem griechischen Buchstaben  $\epsilon$  („Epsilon“) bezeichnet.

- Die leere Sprache ist die Sprache, die keine Wörter enthält, und wird mit  $\emptyset$  bezeichnet. (Beachte:  $\emptyset \neq \{\lambda\}$ .)
- Die Kardinalität einer Sprache  $L$  ist die Anzahl der Wörter von  $L$  und wird mit  $\|L\|$  bezeichnet.  
Beispiel:  $\|\{00, 10\}\| = 2$ .

Zusätzlich zu den üblichen mengentheoretischen Operationen definieren wir weitere grundlegende Operationen, die auf Wörter oder Sprachen angewandt werden können.

**Definition 1.2 (Operationen auf Wörtern und Sprachen)** Seien  $A$  und  $B$  beliebige Sprachen über dem Alphabet  $\Sigma$ , d.h.,  $A, B \subseteq \Sigma^*$ . Definiere

- die Vereinigung von  $A$  und  $B$  durch

$$A \cup B = \{x \in \Sigma^* \mid x \in A \text{ oder } x \in B\};$$

Beispiel:  $\{10, 1\} \cup \{1, 0\} = \{10, 1, 0\}$ .

- den Durchschnitt von  $A$  und  $B$  durch

$$A \cap B = \{x \in \Sigma^* \mid x \in A \text{ und } x \in B\};$$

Beispiel:  $\{10, 1\} \cap \{1, 0\} = \{1\}$ .

- die Differenz von  $A$  und  $B$  durch

$$A - B = \{x \in \Sigma^* \mid x \in A \text{ und } x \notin B\};$$

Beispiel:  $\{10, 1\} - \{1, 0\} = \{10\}$ .

- das Komplement von  $A$  durch

$$\overline{A} = \Sigma^* - A = \{x \in \Sigma^* \mid x \notin A\}.$$

Beispiel:  $\overline{\{10, 1\}} = \{0, 1\}^* - \{10, 1\} = \{\lambda, 0, 00, 11, 01, 000, \dots\}$ .

- Die Konkatenation (Verketzung) von Wörtern  $u, v \in \Sigma^*$  ist ein Wort  $uv \in \Sigma^*$ , das wie folgt definiert ist.

- Ist  $u = v = \lambda$ , so ist  $uv = vu = \lambda$ .
- Ist  $v = \lambda$ , so ist  $uv = u$ .

- Ist  $u = \lambda$ , so ist  $uv = v$ .
- Sind  $u = u_1u_2 \cdots u_n$  und  $v = v_1v_2 \cdots v_m$  mit  $u_i, v_i \in \Sigma$ , so ist

$$uv = u_1u_2 \cdots u_nv_1v_2 \cdots v_m.$$

*Beispiel:*  $u = 01$ ,  $v = 10$ ,  $uv = 0110$  und  $vu = 1001$ .

Die wiederholte Verkettung eines Wortes  $u$  wird induktiv definiert durch  $u^0 = \lambda$  und  $u^n = u(u^{n-1})$  für  $n > 0$ . Zum Beispiel ist für  $u = 01$ :  $u^1 = 01$ ,  $u^2 = 0101$  und  $u^3 = 010101$ .

- Die Konkatenation (Verkettung) von Sprachen  $A$  und  $B$  (also von Mengen von Wörtern) ist die Sprache

$$AB = \{ab \mid a \in A \text{ und } b \in B\}.$$

*Beispiel:*  $\{10, 1\}\{1, 0\} = \{101, 100, 11, 10\}$ .

- Die Iteration einer Sprache  $A \subseteq \Sigma^*$  (die Kleene-Hülle von  $A$ ) ist die Sprache  $A^*$ , die induktiv definiert ist durch

$$A^0 = \{\lambda\}, \quad A^n = AA^{n-1}, \quad A^* = \bigcup_{n \geq 0} A^n.$$

Definiere die  $\lambda$ -freie Iteration von  $A$  durch  $A^+ = \bigcup_{n \geq 1} A^n$ . Es gilt:  $A^+ \cup \{\lambda\} = A^*$ . Folgende Eigenschaft gilt aber nicht:  $A^+ = A^* - \{\lambda\}$ , nämlich dann nicht, wenn  $\lambda \in A$ .

*Beispiel:*

$$\begin{aligned} \{10, 1\}^0 &= \{\lambda\}, \\ \{10, 1\}^1 &= \{10, 1\}, \\ \{10, 1\}^2 &= \{10, 1\}\{10, 1\} = \{1010, 101, 110, 11\}, \\ \{10, 1\}^3 &= \{10, 1\}\{10, 1\}^2 = \{101010, \dots\}. \end{aligned}$$

- Die Spiegelbildoperation für ein Wort  $u = u_1u_2 \cdots u_n \in \Sigma^*$  ist definiert durch

$$sp(u) = u_n \cdots u_2u_1.$$

*Beispiel:*  $sp(0100101) = 1010010$ .

- Die Spiegelung einer Sprache  $A \subseteq \Sigma^*$  ist definiert durch

$$sp(A) = \{sp(w) \mid w \in A\}.$$

*Beispiel:*  $sp(\{100, 011\}) = \{001, 110\}$ .

- Die Teilwortrelation auf  $\Sigma^*$  ist definiert durch

$$u \sqsubseteq v \iff (\exists v_1, v_2 \in \Sigma^*) [v_1 u v_2 = v].$$

Gilt  $u \sqsubseteq v$ , so bezeichnen wir  $u$  als ein Teilwort (einen Infix) von  $v$ .

Beispiel:  $100 \sqsubseteq 1010010$ .

- Die Anfangswortrelation auf  $\Sigma^*$  ist definiert durch

$$u \sqsubseteq_a v \iff (\exists w \in \Sigma^*) [uw = v].$$

Gilt  $u \sqsubseteq_a v$ , so bezeichnen wir  $u$  als ein Anfangswort (einen Präfix) von  $v$ .

Beispiel:  $101 \sqsubseteq_a 1010010$ .

Will man eine neue Sprache wie etwa Holländisch erlernen, so muss man zunächst das Vokabular der Sprache und ihre Grammatik lernen. Das Vokabular ist einfach die Menge der Wörter, die in der Sprache vorkommen. Die Grammatik ist eine Liste von Regeln, die festlegen, wie man Wörter und Wortgruppen so zusammensetzen und mit der geeigneten Interpunktion versehen kann, dass syntaktisch korrekt gebildete Sätze entstehen. Später lernt man dann, semantisch korrekte (womöglich sogar sinnvolle) Sätze zu bilden.

Dies trifft auch auf formale Sprachen zu. Betrachte zum Beispiel die Sprache

$$L = \{a^n b^n \mid n \geq 0\},$$

die genau aus den Wörtern besteht, die mit einem Block von  $n$  Symbolen  $a$  beginnen, gefolgt von einem Block von  $n$  Symbolen  $b$ , für beliebiges  $n \geq 0$ . Beachte, dass wegen  $n = 0$  auch das leere Wort  $\lambda$  zu  $L$  gehört. Eine formale Sprache wie  $L$  hat eines gemeinsam mit einer natürlichen Sprache wie Holländisch: Sie beide brauchen eine Grammatik, die ihre Syntax festlegt. Grammatiken formaler Sprachen sind so genannte *generative* Grammatiken, denn sie legen fest, wie man ein beliebiges Wort der Sprache aus dem Startsymbol erzeugen kann. Grammatiken sind also endliche Objekte, die unendliche Objekte beschreiben können, nämlich die von ihnen erzeugten Sprachen.

Operatorpräzedenz: In unserer Schreibweise  $a^n$  gilt, dass die Konkatenation durch  $^n$  stärker bindet als die Konkatenation durch hintereinanderschreiben. Es gilt also z.B.  $\{ab^n\} = \{a(b)^n\}$  oder konkret  $ab^2 = abb = a(b)^2$ , im Gegensatz zu  $(ab)^2 = abab$ .

**Definition 1.3 (Grammatik)** Eine Grammatik ist ein Quadrupel  $G = (\Sigma, N, S, P)$ , wobei

- $\Sigma$  ein Alphabet (von so genannten Terminalsymbolen) ist,
- $N$  eine endliche Menge (von so genannten Nichtterminalen) mit  $\Sigma \cap N = \emptyset$ ,
- $S \in N$  das Startsymbol und

- $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$  die endliche Menge der Produktionen (Regeln).

Dabei ist  $(N \cup \Sigma)^+ = (N \cup \Sigma)^* - \{\lambda\}$ . Regeln  $(p, q)$  in  $P$  schreiben wir auch so:  $p \rightarrow q$ .

Für Grammatikregeln mit gleicher linker Seite  $A \in N$  schreiben wir auch kurz

$$A \rightarrow q_1 \mid q_2 \mid \cdots \mid q_n \quad \text{statt} \quad \begin{array}{l} A \rightarrow q_1 \\ A \rightarrow q_2 \\ \vdots \\ A \rightarrow q_n \end{array}$$

(aus der so genannten *BNF (Backus-Naur-Form)*).

**Definition 1.4 (Ableitungsrelation, Sprache einer Grammatik)** Sei  $G = (\Sigma, N, S, P)$  eine Grammatik, und seien  $u$  und  $v$  Wörter in  $(N \cup \Sigma)^*$ .

- Definiere die unmittelbare Ableitungsrelation bzgl.  $G$  so:

$$u \vdash_G v \iff u = xpz, v = xqz,$$

wobei  $x, z \in (N \cup \Sigma)^*$  und  $p \rightarrow q$  eine Regel in  $P$  ist.

- Durch  $n$ -malige Anwendung von  $\vdash_G$  erhalten wir  $\vdash_G^n$ . Das heißt:

$$u \vdash_G^n v \iff u = x_0 \vdash_G x_1 \vdash_G \cdots \vdash_G x_n = v$$

für  $n \geq 0$  und für eine Folge von Wörtern  $x_0, x_1, \dots, x_n \in (\Sigma \cup N)^*$ . Insbesondere ist  $u \vdash_G^0 u$ .

- Die Folge  $(x_0, x_1, \dots, x_n)$ ,  $x_i \in (\Sigma \cup N)^*$ ,  $x_0 = S$  und  $x_n \in \Sigma^*$  heißt Ableitung von  $x_n$  in  $G$ , falls  $x_0 \vdash_G x_1 \vdash_G \cdots \vdash_G x_n$ .
- Definiere  $\vdash_G^* = \bigcup_{n \geq 0} \vdash_G^n$ . Man kann zeigen, dass  $\vdash_G^*$  die reflexive und transitive Hülle von  $\vdash_G$  ist, d.h. die kleinste binäre Relation, die reflexiv und transitiv ist und  $\vdash_G$  umfasst.
- Die von der Grammatik  $G$  erzeugte Sprache ist definiert als

$$L(G) = \{w \in \Sigma^* \mid S \vdash_G^* w\}.$$

- Zwei Grammatiken  $G_1$  und  $G_2$  heißen äquivalent, falls  $L(G_1) = L(G_2)$ .

**Beispiel 1.5 (Grammatik)** Betrachte die folgenden Grammatiken.

1. Sei  $G_1 = (\Sigma_1, \Gamma_1, S_1, R_1)$  die folgende Grammatik:

- das terminale Alphabet ist  $\Sigma_1 = \{a, b\}$ ,
- das nichtterminale Alphabet ist  $\Gamma_1 = \{S_1\}$  und
- die Menge der Regeln ist gegeben durch

$$R_1 = \{S_1 \rightarrow aS_1b \mid \lambda\}.$$

Eine Ableitung für  $G_1$ :

$$S_1 \vdash_{G_1} aS_1b \vdash_{G_1} aaS_1bb \vdash_{G_1} aabb \Rightarrow aabb \in L(G_1).$$

Man sieht leicht, dass  $G_1$  die Sprache  $L = \{a^n b^n \mid n \geq 0\}$  erzeugt, d.h.,  $L(G_1) = L$ .

2. Sei  $G_2 = (\Sigma_2, \Gamma_2, S_2, R_2)$  die folgende Grammatik:

- das terminale Alphabet ist  $\Sigma_2 = \{a, b, c\}$ ,
- das nichtterminale Alphabet ist  $\Gamma_2 = \{S_2, B, C\}$  und
- die Menge der Regeln ist gegeben durch

$$R_2 = \left\{ \begin{array}{l} S_2 \rightarrow aS_2BC \mid aBC, \\ CB \rightarrow BC, \\ aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc \end{array} \right\}.$$

Eine Ableitung für  $\frac{1}{2}r$   $G_2$ :

$$\begin{array}{lll} S_2 \vdash_{G_2} aS_2BC & \vdash_{G_2} aaS_2BCBC & \vdash_{G_2} aaaBCBCBC \\ \vdash_{G_2} aaaBBCCBC & \vdash_{G_2} aaaBBCBCC & \vdash_{G_2} aaaBBBCCC \\ \vdash_{G_2} aaabBBCCC & \vdash_{G_2} aaabbBCCC & \vdash_{G_2} aaabbbCCC \\ \vdash_{G_2} aaabbbcCC & \vdash_{G_2} aaabbbccC & \vdash_{G_2} aaabbbccc \end{array}$$

$$\Rightarrow aaabbbccc \in L(G_2).$$

Man sieht leicht, dass  $G_2$  die Sprache  $L = \{a^n b^n c^n \mid n \geq 1\}$  erzeugt, d.h.,  $L(G_2) = L$ . (Übungsaufgabe)

3. Sei  $G_3 = (\Sigma_3, \Gamma_3, S_3, R_3)$  die folgende Grammatik:

- das terminale Alphabet ist  $\Sigma_3 = \{*, +, (, ), a\}$ ,
- das nichtterminale Alphabet ist  $\Gamma_3 = \{S_3\}$  und
- die Menge der Regeln ist gegeben durch

$$R_3 = \{S_3 \rightarrow S_3 + S_3 \mid S_3 * S_3 \mid (S_3) \mid a\}.$$



Eine Ableitung für  $\frac{1}{2}r$   $G_3$ :

$$\begin{array}{ll} S_3 \vdash_{G_3} S_3 + S_3 & \vdash_{G_3} S_3 * S_3 + S_3 \\ \vdash_{G_3} (S_3) * S_3 + S_3 & \vdash_{G_3} (S_3 + S_3) * S_3 + S_3 \\ \vdash_{G_3} \dots & \vdash_{G_3} (a + a) * a + a \end{array}$$

$$\Rightarrow (a + a) * a + a \in L(G_3).$$

$G_3$  erzeugt die Sprache aller verschachtelten Klammerausdrücke mit den Operationen  $+$  und  $*$  und einem Zeichen  $a$ .

Der Prozess des Ableitens von Wörtern ist inhärent nichtdeterministisch, denn im Allgemeinen kann mehr als eine Regel in einem Ableitungsschritt angewandt werden.

Verschiedene Grammatiken können dieselbe Sprache erzeugen. Tatsächlich hat jede durch eine Grammatik definierbare Sprache unendlich viele sie erzeugende Grammatiken. Das heißt, eine Grammatik ist ein syntaktisches Objekt, das mittels der in Definition 1.4 eingeführten Begriffe ein semantisches Objekt erzeugt, nämlich ihre Sprache.

## 1.2 Die Chomsky-Hierarchie

Grammatiken werden gemäß bestimmter Restriktionen klassifiziert, die ihrer Regelmenge auferlegt sind. In dieser Weise wird die Chomsky-Hierarchie definiert, eine Hierarchie von Sprachklassen.

**Definition 1.6 (Chomsky-Hierarchie)**  $G = (\Sigma, N, S, P)$  sei eine Grammatik.

- $G$  ist Typ-0-Grammatik, falls  $P$  keinerlei Einschränkungen unterliegt.
- $G$  ist Typ-1-Grammatik (bzw. kontextsensitiv bzw. nichtverkürzend oder monoton), falls für alle Regeln  $p \rightarrow q$  in  $P$  gilt:  $|p| \leq |q|$ .
- Eine Typ-1-Grammatik  $G$  ist vom Typ 2 (bzw. kontextfrei), falls für alle Regeln  $p \rightarrow q$  in  $P$  gilt:  $p \in N$ .
- Eine Typ-2-Grammatik  $G$  ist vom Typ 3 (bzw. regulär bzw. rechtslinear), falls für alle Regeln  $p \rightarrow q$  in  $P$  gilt:  $p \in N$  und  $q \in \Sigma \cup \Sigma N$ .
- Eine Sprache  $A \subseteq \Sigma^*$  ist genau dann vom Typ  $i \in \{0, 1, 2, 3\}$ , wenn es eine Typ- $i$ -Grammatik  $G$  gibt mit  $L(G) = A$ .
- Die Chomsky-Hierarchie besteht aus den vier Sprachklassen:

$$\mathcal{L}_i = \{L(G) \mid G \text{ ist Typ-}i\text{-Grammatik}\},$$

wobei  $i \in \{0, 1, 2, 3\}$ . Übliche Bezeichnungen:

- $\mathcal{L}_0$  ist die Klasse aller Sprachen, die durch eine Grammatik erzeugt werden können;
- $\mathcal{L}_1 = \text{CS}$  ist die Klasse der kontextsensitiven Sprachen;
- $\mathcal{L}_2 = \text{CF}$  ist die Klasse der kontextfreien Sprachen;
- $\mathcal{L}_3 = \text{REG}$  ist die Klasse der regulären Sprachen.

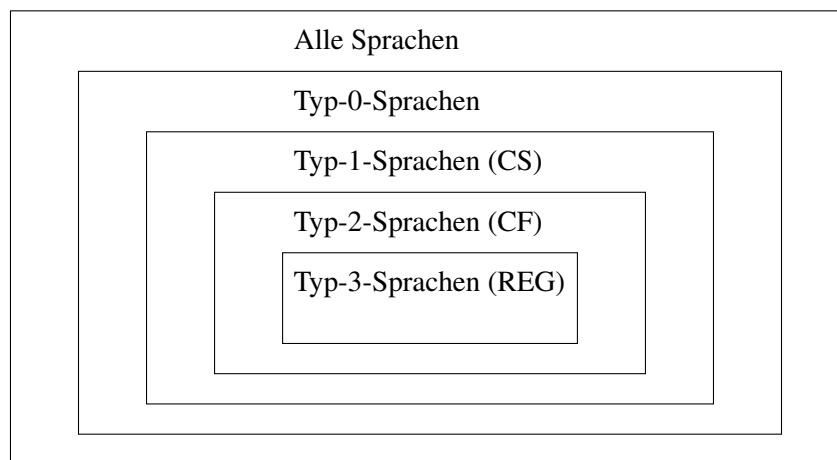


Abbildung 1.1: Die Chomsky-Hierarchie

Offensichtlich sind die Grammatiken  $G_2$  und  $G_3$  aus Beispiel 1.5 kontextsensitiv. Die Grammatik  $G_1$  enthält die Regel  $S_1 \rightarrow \lambda$  und ist wegen  $|S_1| = 1 > 0 = |\lambda|$  nach Definition 1.6 nicht kontextsensitiv (vgl. „Sonderregelung für das leere Wort“). Da  $G_1$  eine Grammatik ist, ist  $G_1$  eine Typ-0-Grammatik.

Der Ausdruck „kontextsensitiv“ für Typ-1-Grammatiken kann damit erklärt werden, dass eine Grammatik mit *nichtverkürzenden* Regeln, d.h., Regeln von der Form  $p \rightarrow q$  mit  $|p| \leq |q|$ , in eine äquivalente Grammatik  $G = (\Sigma, \Gamma, S, R)$  umgewandelt werden kann, deren Regeln *kontextsensitiv* sind, also die Form  $uAv \rightarrow uvw$  haben, wobei  $A \in \Gamma$ ,  $u, v, w \in (\Sigma \cup \Gamma)^*$  und  $w \neq \lambda$ . Die Anwendung einer solchen Regel von  $G$  heißt, dass ein Nichtterminal  $A$  nur im Kontext von  $u$  und  $v$  ersetzt werden kann.

Offensichtlich ist die Grammatik  $G_3$  aus Beispiel 1.5 sogar kontextfrei. Der Ausdruck „kontextfrei“ für Typ-2-Grammatiken kann damit erklärt werden, dass nur Regeln der Form  $A \rightarrow q$  für ein Nichtterminal  $A$  angewandt werden dürfen, d.h.,  $A$  wird durch  $q$  ersetzt, unabhängig vom Kontext von  $A$ .

Der Begriff „rechtlinear“ für Typ-3-Grammatiken ist durch die spezielle Form der Regeln (wenn ein Nichtterminalsymbol auf einer rechten Seite auftritt, so steht es dort

ganz rechts) leicht nachvollziehbar. Entsprechend verwendet man den Begriff „linkslinear“, wenn alle für alle Regeln  $p \rightarrow q$  in  $P$  gilt:  $p \in N$  und  $q \in \Sigma \cup N\Sigma$  (wenn ein Nichtterminalsymbol auf einer rechten Seite auftritt, so steht es dort ganz links). Man kann zeigen, dass beide Begriffe äquivalent bzgl. der definierbaren Sprachen sind (Übungsaufgabe).

Fakt 1.7 sowie Abbildung 1.1 zeigen die Inklusionsstruktur der Chomsky-Hierarchie. Wir werden später sehen, dass alle diese Inklusionen echt sind.

### Fakt 1.7

$$\text{REG} \subseteq \text{CF} \subseteq \text{CS} \subseteq \mathcal{L}_0.$$

Unter den Klassen der Chomsky-Hierarchie sind die kontextfreien Sprachen besonders wichtig, z.B. im Zusammenhang mit der Syntaxanalyse beim Compilerbau. Ebenso sind die regulären Sprachen bei der lexikalischen Analyse von Programmen wichtig. Kontextsensitive und Typ-0-Sprachen spielen in der Theoretischen Informatik, speziell in der Berechenbarkeitstheorie, eine wichtige Rolle.

### Sonderregelung für das leere Wort

Für Typ- $i$ -Grammatiken  $G = (\Sigma, N, S, P)$  mit  $i \in \{1, 2, 3\}$  gilt nach obiger Definition für alle Produktionen  $p \rightarrow q$  die Bedingung  $|p| \leq |q|$ , daraus folgt dass  $\lambda \notin L(G)$ . Dies ist jedoch nicht immer wünschenswert (s. Beispiel 1.5). Um das leere Wort in Sprachen von Typ- $i$ -Grammatiken für  $i \in \{1, 2, 3\}$  aufzunehmen, treffen wir die folgende Vereinbarung:

- (a) Die Regel  $S \rightarrow \lambda$  ist als einzige verkürzende Regel für Grammatiken vom Typ 1, 2, 3 zugelassen.
- (b) Tritt die Regel  $S \rightarrow \lambda$  auf, so darf  $S$  auf keiner rechten Seite einer Regel vorkommen.

Dies ist keine Beschränkung der Allgemeinheit, denn: Gibt es in  $P$  Regeln mit  $S$  auf der rechten Seite, so verändern wir die Regelmenge  $P$  zur neuen Regelmenge  $P'$  wie folgt:

1. In allen Regeln der Form  $S \rightarrow u$  aus  $P$  mit  $u \in (N \cup \Sigma)^*$  wird jedes Vorkommen von  $S$  in  $u$  durch ein neues Nichtterminal  $S'$  ersetzt.
2. Zusätzlich enthält  $P'$  alle Regeln aus  $P$ , mit  $S$  ersetzt durch  $S'$ .
3. Die Regel  $S \rightarrow \lambda$  wird hinzugefügt.

Die so modifizierte Grammatik  $G' = (\Sigma, N \cup \{S'\}, S, P')$  ist (bis auf  $S \rightarrow \lambda$ ) vom selben Typ wie  $G$  und erfüllt  $L(G') = L(G) \cup \{\lambda\}$ .

**Beispiel 1.8** Wir betrachten die Grammatik  $G = (\Sigma, N, S, P)$  mit

- das terminale Alphabet ist  $\Sigma = \{a, b\}$ ,
- das nichtterminale Alphabet ist  $N = \{S\}$  und
- die Menge der Regeln ist gegeben durch

$$P = \{S \rightarrow aSb \mid ab\}.$$

Man sieht leicht, dass  $G$  kontextfrei ist und die Sprache  $L = \{a^n b^n \mid n \geq 1\}$  erzeugt, d.h.,  $L(G) = L$ .

Wir modifizieren die Grammatik  $G$  nun gemäß der Sonderregelung für das leere Wort, um eine kontextfreie Grammatik für die Sprache  $\{a^n b^n \mid n \geq 0\} = L \cup \{\lambda\}$  zu gewinnen. Nach obiger Konstruktion erhalten wir eine kontextfreie Grammatik  $G' = (\Sigma, \{S, S'\}, S, P')$  mit

$$P' = \left\{ \begin{array}{ll} S \rightarrow aS'b \mid ab & \text{gemäß } \frac{1}{2} \ddot{v}_G \frac{1}{2} \text{ 1.)} \\ S' \rightarrow aS'b \mid ab & \text{gemäß } \frac{1}{2} \ddot{v}_G \frac{1}{2} \text{ 2.)} \\ S \rightarrow \lambda & \text{gemäß } \frac{1}{2} \ddot{v}_G \frac{1}{2} \text{ 3.)} \end{array} \right\}$$

und  $L(G') = L(G) \cup \{\lambda\}$ .

## Syntaxbaum

Um Ableitungen von Worten in Grammatiken vom Typ 2 oder 3 anschaulich darzustellen, verwenden wir einen wie folgt definierten Syntaxbaum.

- Sei  $G = (\Sigma, N, S, P)$  eine Grammatik vom Typ 2 oder 3, und sei  $S = w_0 \vdash_G w_1 \vdash_G \dots \vdash_G w_n = w$  eine Ableitung von  $w \in L(G)$ .
- Die Wurzel (d.h. die Etage 0) des Syntaxbaums ist das Startsymbol  $S$ .
- Wird wegen der Regel  $A \rightarrow z$  im  $i$ -ten Ableitungsschritt ( $w_{i-1} \vdash_G w_i$ ) das Nichtterminal  $A$  in  $w_{i-1}$  durch das Teilwort  $z$  von  $w_i$  ersetzt, so hat der entsprechende Knoten  $A$  im Syntaxbaum  $|z|$  Söhne, die v.l.n.r. mit den Symbolen aus  $z$  beschriftet sind. Falls  $z = \lambda$  fügen wir ein Kind beschriftet mit  $\lambda$  hinzu.
- Die Blätter des Baumes ergeben von links nach rechts gelesen  $w$ .

Ein Beispiel für einen Syntaxbaum ist in Abbildung 1.2 zu sehen.

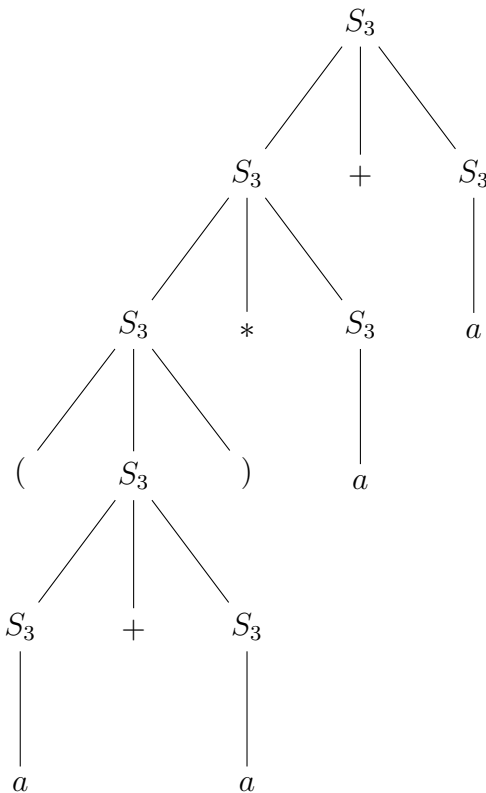


Abbildung 1.2: Syntaxbaum für die Ableitung des Wortes  $(a + a) * a + a$  in der Typ 2-Grammatik  $G_3$  aus Beispiel 1.5

## Mehrdeutigkeit

**Definition 1.9 (Mehrdeutige Grammatik)** Eine Grammatik  $G$  heißt mehrdeutig, falls es ein Wort  $w \in L(G)$  gibt, das zwei verschiedene Syntaxbäume hat. Eine Sprache  $A$  heißt inhärent mehrdeutig, falls jede Grammatik  $G$  mit  $A = L(G)$  mehrdeutig ist.

**Beispiel 1.10 (Mehrdeutige Grammatik)** Offensichtlich ist die Grammatik  $G_3$  aus Beispiel 1.5 mehrdeutig, wie die zwei Syntaxbäume aus Abbildung 1.2 und Abbildung 1.3 für Ableitungen des Wortes  $(a + a) * a + a$  zeigen.

Beispiele inhärent mehrdeutiger Sprachen werden wir später kennenlernen.

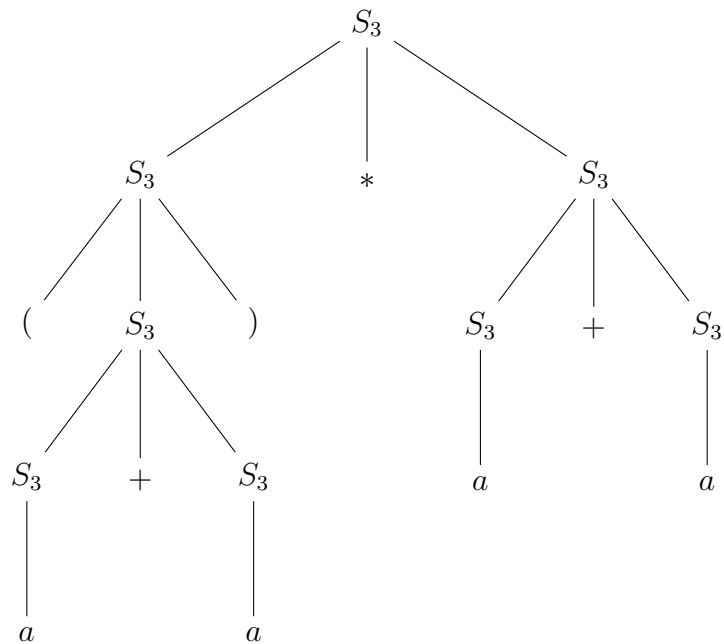


Abbildung 1.3: Ein weiterer Syntaxbaum für die Ableitung des Wortes  $(a + a) * a + a$  in der Typ 2-Grammatik  $G_3$  aus Beispiel 1.5

# Kapitel 2

## Reguläre Sprachen

### 2.1 Endliche Automaten

Jede Klasse der Chomsky-Hierarchie kann durch einen geeigneten Automatentypen charakterisiert werden. Beispielsweise kann jede reguläre Sprache durch einen endlichen Automaten erkannt werden, und jede von einem endlichen Automaten erkannte Sprache ist regulär.

**Definition 2.1 (DFA)** Ein deterministischer endlicher Automat (kurz DFA für “deterministic finite automaton”) ist ein Quintupel  $M = (\Sigma, Z, \delta, z_0, F)$ , wobei

- $\Sigma$  ein Alphabet ist,
- $Z$  eine endliche Menge von Zuständen mit  $\Sigma \cap Z = \emptyset$ ,
- $\delta : Z \times \Sigma \rightarrow Z$  die Überföhrungsfunktion,
- $z_0 \in Z$  der Startzustand und
- $F \subseteq Z$  die Menge der Endzustände (Finalzustände).

Wir erlauben auch partiell definierte Überföhrungsfunktionen. Durch Hinzunahme eines weiteren Zustandes kann man solche Funktionen leicht total definieren.

**Beispiel 2.2 (DFA)** Ein Beispiel für einen DFA ist  $M = (\Sigma, Z, \delta, z_0, F)$  mit

$\Sigma$	$=$	$\{0, 1\}$	$\delta$ :	<table><tr><td><math>\delta</math></td><td><math>z_0</math></td><td><math>z_1</math></td><td><math>z_2</math></td><td><math>z_3</math></td></tr></table>	$\delta$	$z_0$	$z_1$	$z_2$	$z_3$
$\delta$	$z_0$	$z_1$		$z_2$	$z_3$				
$Z$	$=$	$\{z_0, z_1, z_2, z_3\}$		<table><tr><td>0</td><td><math>z_1</math></td><td><math>z_3</math></td><td><math>z_2</math></td><td><math>z_3</math></td></tr></table>	0	$z_1$	$z_3$	$z_2$	$z_3$
0	$z_1$	$z_3$	$z_2$	$z_3$					
$F$	$=$	$\{z_2\}$	<table><tr><td>1</td><td><math>z_3</math></td><td><math>z_2</math></td><td><math>z_2</math></td><td><math>z_3</math></td></tr></table>	1	$z_3$	$z_2$	$z_2$	$z_3$	
1	$z_3$	$z_2$	$z_2$	$z_3$					

## Arbeitsweise eines DFA

Ein DFA  $M = (\Sigma, Z, \delta, z_0, F)$  akzeptiert bzw. verwirft eine Eingabe  $x$  wie folgt:

- $M$  beginnt beim Anfangszustand  $z_0$  und führt insgesamt  $|x|$  Schritte aus.
- Der Lesekopf wandert dabei v.l.n.r. über das Eingabewort  $x$ , Symbol für Symbol, und ändert dabei seinen Zustand jeweils gemäß der Überföhrungsfunktion  $\delta$ : Ist  $M$  im Zustand  $z \in Z$  und liest das Symbol  $a \in \Sigma$  und gilt  $\delta(z, a) = z'$ , so ändert  $M$  seinen Zustand in  $z'$ .
- Ist der letzte erreichte Zustand (nachdem  $x$  abgearbeitet ist) ein Endzustand, so akzeptiert  $M$  die Eingabe  $x$ ; andernfalls lehnt  $M$  sie ab.

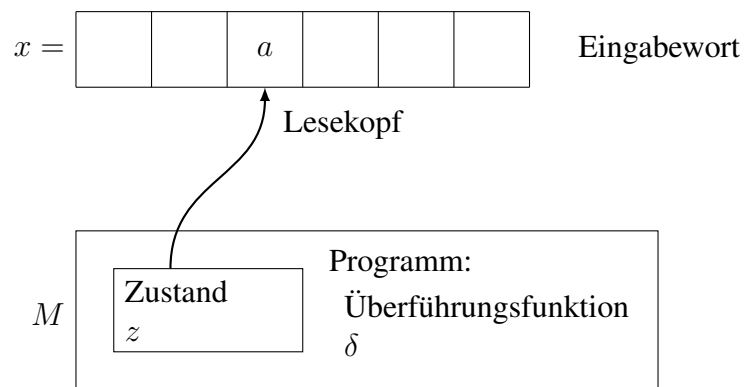


Abbildung 2.1: Arbeitsweise eines endlichen Automaten

## Zustandsgraph eines DFA

Ein DFA  $M = (\Sigma, Z, \delta, z_0, F)$  lässt sich anschaulich durch seinen *Zustandsgraphen* darstellen, dessen Knoten die Zustände von  $M$  und dessen Kanten Zustandsübergänge gemäß der Überföhrungsfunktion  $\delta$  repräsentieren. Gilt  $\delta(z, a) = z'$  für ein Symbol  $a \in \Sigma$  und für zwei Zustände  $z, z' \in Z$ , so hat dieser Graph eine gerichtete Kante von  $z$  nach  $z'$ , die mit  $a$  beschriftet ist. Der Startzustand wird durch einen Pfeil auf  $z_0$  dargestellt. Endzustände sind durch einen Doppelkreis markiert, siehe Beispiel 2.3.

**Beispiel 2.3 (Zustandsgraph eines DFA)** *Abbildung 2.2 zeigt den Zustandsgraph für den DFA aus Beispiel 2.2.*

Um die Sprache eines DFA zu definieren, erweitern wir unsere Überföhrungsfunktion  $\delta$  auf Eingaben aus  $Z \times \Sigma^*$ .



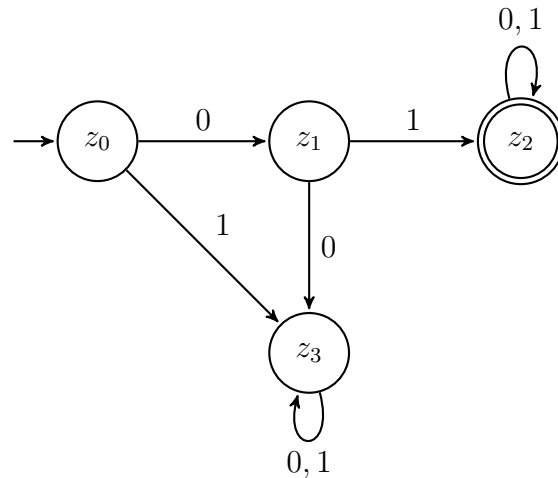


Abbildung 2.2: Zustandsgraph eines deterministischen endlichen Automaten

**Definition 2.4 (Sprache eines DFA)** Sei  $M = (\Sigma, Z, \delta, z_0, F)$  ein DFA. Die erweiterte Überföhrungsfunktion  $\hat{\delta} : Z \times \Sigma^* \rightarrow Z$  von  $M$  ist induktiv definiert:

$$\begin{aligned}\hat{\delta}(z, \lambda) &= z \\ \hat{\delta}(z, ax) &= \hat{\delta}(\delta(z, a), x)\end{aligned}$$

für alle  $z \in Z$ ,  $a \in \Sigma$  und  $x \in \Sigma^*$ .

Die vom DFA  $M$  akzeptierte Sprache ist definiert durch

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(z_0, w) \in F\}.$$

Für  $a \in \Sigma$  gilt  $\hat{\delta}(z, a) = \delta(z, a)$ , und für  $x = a_1 a_2 \cdots a_n$  in  $\Sigma^*$  gilt:

$$\hat{\delta}(z, x) = \delta(\cdots \delta(\delta(z, a_1), a_2) \cdots, a_n).$$

**Beispiel 2.5 (Sprache eines DFA)** Wie arbeitet der DFA in Abbildung 2.2 bei Eingabe von 0111?

$$\begin{aligned}\hat{\delta}(z_0, 0111) &= \delta(\delta(\delta(\delta(z_0, 0), 1), 1), 1) \\ &= \delta(\delta(\delta(z_1, 1), 1), 1) \\ &= \delta(\delta(z_2, 1), 1) \\ &= \delta(z_2, 1) \\ &= z_2 \in F.\end{aligned}$$

Wie arbeitet der DFA in Abbildung 2.2 bei Eingabe von 0011?

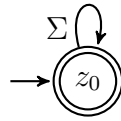
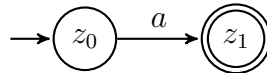
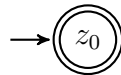
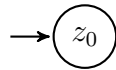
$$\begin{aligned}
\widehat{\delta}(z_0, 0011) &= \delta(\delta(\delta(\delta(z_0, 0), 0), 1), 1), 1) \\
&= \delta(\delta(\delta(z_1, 0), 1), 1) \\
&= \delta(\delta(z_3, 1), 1) \\
&= \delta(z_3, 1) \\
&= z_3 \quad \notin F.
\end{aligned}$$

Der DFA in Abbildung 2.2 akzeptiert offenbar die Sprache

$$L = \{x \in \{0, 1\}^* \mid x \text{ beginnt mit } 01\}.$$

Im nächsten Beispiel geben wir DFAs für einfache formale Sprachen an.

**Beispiel 2.6 (DFA)** Es sei  $\Sigma$  ein Alphabet.

$L$	DFA $M$ mit $L(M) = L$	Zustandsgraph für $M$
$\Sigma^*$	$(\Sigma, \{z_0\}, \{\delta(z_0, a) = z_0, a \in \Sigma\}, z_0, \{z_0\})$	
$\{a\}, a \in \Sigma$	$(\Sigma, \{z_0, z_1\}, \{\delta(z_0, a) = z_1\}, z_0, \{z_1\})$	
$\{\lambda\}$	$(\Sigma, \{z_0\}, \emptyset, z_0, \{z_0\})$	
$\emptyset$	$(\Sigma, \{z_0\}, \emptyset, z_0, \emptyset)$	

**Satz 2.7** Jede von einem DFA akzeptierte Sprache ist regulär (d.h. vom Typ 3).

**Beweis.** Seien  $A \subseteq \Sigma^*$  eine Sprache und  $M = (\Sigma, Z, \delta, z_0, F)$  ein DFA mit  $L(M) = A$ . Definiere eine reguläre Grammatik  $G = (\Sigma, N, S, P)$  durch:

- $N = Z$ ,
- $S = z_0$ ,

- $P$  besteht aus genau den folgenden Regeln:
  - Gilt  $\delta(z, a) = z'$ , so ist  $z \rightarrow az'$  in  $P$ .
  - Ist dabei  $z' \in F$ , so ist zusätzlich  $z \rightarrow a$  in  $P$ .
  - Ist  $\lambda \in A$  (d.h.,  $z_0 \in F$ ), so ist auch  $z_0 \rightarrow \lambda$  in  $P$ , und die bisher konstruierte Grammatik wird gemäß  $\frac{1}{2}i; \frac{1}{2}i$  der Sonderregel für  $\lambda$  modifiziert.

Offenbar gilt  $\lambda \in A \iff \lambda \in L(G)$ . Für alle  $x = a_1a_2 \cdots a_n \in \Sigma^*$  mit  $n \geq 1$  gilt:

$$\begin{aligned}
 x \in A &\iff \text{es gibt eine Zustandsfolge } z_0, z_1, \dots, z_n, \text{ so dass } z_0 \text{ Startzustand} \\
 &\quad \text{und } z_n \in F \text{ ist, und } \delta(z_{i-1}, a_i) = z_i \text{ für jedes } i, 1 \leq i \leq n \\
 &\iff \text{es gibt eine Folge von Nichtterminalen } z_0, z_1, \dots, z_n \text{ mit} \\
 &\quad S = z_0 \vdash a_1z_1 \vdash a_1a_2z_2 \vdash \cdots \vdash a_1a_2 \cdots a_{n-1}z_{n-1} \vdash a_1a_2 \cdots a_{n-1}a_n \\
 &\iff x \in L(G).
 \end{aligned}$$

Somit ist  $A = L(G)$ . ■

**Beispiel 2.8 (Reguläre Grammatik zu gegebenem DFA)** Wir konstruieren eine reguläre Grammatik  $G = (\Sigma, N, S, P)$  für den DFA in Abbildung 2.2.

$$\begin{aligned}
 \Sigma &= \{0, 1\} \\
 N &= \{z_0, z_1, z_2, z_3\} \\
 S &= z_0 \\
 P &= \left\{ \begin{array}{l} z_0 \rightarrow 0z_1 \mid 1z_3 \\ z_1 \rightarrow 0z_3 \mid 1z_2 \mid 1 \\ z_2 \rightarrow 0z_2 \mid 1z_2 \mid 0 \mid 1 \\ z_3 \rightarrow 0z_3 \mid 1z_3 \end{array} \right\}.
 \end{aligned}$$

Nun erweitern wir die Überföhrungsfunktion in der Definition eines DFA und bezeichnen die neuen Automaten als NFA.

**Definition 2.9 (NFA)** Ein nichtdeterministischer endlicher Automat (kurz NFA) ist ein Quintupel  $M = (\Sigma, Z, \delta, S, F)$ , wobei

- $\Sigma$  ein Alphabet ist,
- $Z$  eine endliche Menge von Zuständen mit  $\Sigma \cap Z = \emptyset$ ,
- $\delta : Z \times \Sigma \rightarrow \mathfrak{P}(Z)$  die Überföhrungsfunktion (hier:  $\mathfrak{P}(Z)$  ist die Potenzmenge von  $Z$ , also die Menge aller Teilmengen von  $Z$ ),

- $S \subseteq Z$  die Menge der Startzustände und
- $F \subseteq Z$  die Menge der Endzustände (Finalzustände).

**Beispiel 2.10 (NFA)** Der folgende Automat  $M$  ist nichtdeterministisch (da  $\|\delta(z_0, 1)\| = \|\{z_0, z_1\}\| = 2 > 1$ ).

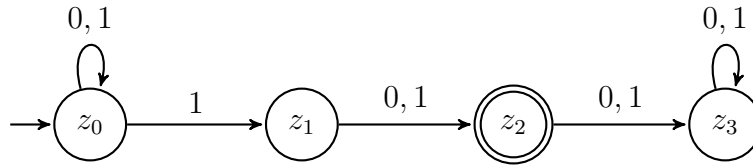


Abbildung 2.3: Ein nichtdeterministischer endlicher Automat

**Definition 2.11 (Sprache eines NFA)** Die erweiterte Überföhrungsfunktion  $\hat{\delta} : \mathfrak{P}(Z) \times \Sigma^* \rightarrow \mathfrak{P}(Z)$  von  $M$  ist induktiv definiert:

$$\begin{aligned}\hat{\delta}(Z', \lambda) &= Z' \\ \hat{\delta}(Z', ax) &= \bigcup_{z \in Z'} \hat{\delta}(\delta(z, a), x)\end{aligned}$$

für alle  $Z' \subseteq Z$ ,  $a \in \Sigma$  und  $x \in \Sigma^*$ .

Die vom NFA  $M$  akzeptierte Sprache ist definiert durch

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(S, w) \cap F \neq \emptyset\}.$$

**Beispiel 2.12 (Sprache eines NFA)** Wie arbeitet der NFA  $M$  in Abbildung 2.3 bei Eingabe von 0111?

$$\begin{aligned}\hat{\delta}(\{z_0\}, 0111) &= \hat{\delta}(\delta(z_0, 0), 111) \\ &= \hat{\delta}(\{z_0\}, 111) \\ &= \hat{\delta}(\delta(z_0, 1), 11) \\ &= \hat{\delta}(\{z_0, z_1\}, 11) \\ &= \hat{\delta}(\delta(z_0, 1), 1) \cup \hat{\delta}(\delta(z_1, 1), 1) \\ &= \hat{\delta}(\{z_0, z_1\}, 1) \cup \hat{\delta}(\{z_2\}, 1) \\ &= (\hat{\delta}(\delta(z_0, 1), \lambda) \cup \hat{\delta}(\delta(z_1, 1), \lambda)) \cup \hat{\delta}(\delta(z_2, 1), \lambda) \\ &= (\hat{\delta}(\{z_0, z_1\}, \lambda) \cup \hat{\delta}(\{z_2\}, \lambda)) \cup \hat{\delta}(\{z_3\}, \lambda) \\ &= \{z_0, z_1\} \cup \{z_2\} \cup \{z_3\}.\end{aligned}$$

$$\begin{aligned}
&\implies \widehat{\delta}(\{z_0\}, 0111) \cap \{z_2\} = (\{z_0, z_1\} \cup \{z_2\} \cup \{z_3\}) \cap \{z_2\} \neq \emptyset \\
&\implies 0111 \in L(M).
\end{aligned}$$

Wie arbeitet der NFA  $M$  in Abbildung 2.3 bei Eingabe von 01?

$$\begin{aligned}
\widehat{\delta}(\{z_0\}, 01) &= \widehat{\delta}(\delta(z_0, 0), 1) \\
&= \widehat{\delta}(\{z_0\}, 1) \\
&= \widehat{\delta}(\delta(z_0, 1), \lambda) \\
&= \widehat{\delta}(\{z_0, z_1\}, \lambda) \\
&= \{z_0, z_1\}.
\end{aligned}$$

$$\begin{aligned}
&\implies \widehat{\delta}(\{z_0\}, 01) \cap \{z_2\} = \{z_0, z_1\} \cap \{z_2\} = \emptyset \\
&\implies 01 \notin L(M).
\end{aligned}$$

Der NFA aus Beispiel 2.10 akzeptiert die Sprache aller Wörter über  $\{0, 1\}^*$ , die an vorletzter Stelle eine 1 haben.

Es scheint jedoch wesentlich schwieriger zu sein für die Sprache „alle Wörter über  $\{0, 1\}^*$ , die an vorletzter Stelle eine 1 haben“ einen DFA anzugeben. Deshalb stellt sich die Frage, ob NFAs mächtiger sind als DFAs. Die Antwort lautet: Nein.

**Satz 2.13 (Rabin und Scott)** *Jede von einem NFA akzeptierte Sprache kann auch von einem DFA akzeptiert werden.*

**Beweis.** Sei  $M = (\Sigma, Z, \delta, S, E)$  ein NFA. Konstruiere einen zu  $M$  äquivalenten DFA  $M' = (\Sigma, \mathfrak{P}(Z), \delta', z'_0, F)$  wie folgt:

- Zustandsmenge von  $M'$ : die Potenzmenge  $\mathfrak{P}(Z)$  von  $Z$ ,
- $\delta'(Z', a) = \bigcup_{z \in Z'} \delta(z, a) = \widehat{\delta}(Z', a)$  für alle  $Z' \subseteq Z$  und  $a \in \Sigma$ ,
- $z'_0 = S$ ,
- $F = \{Z' \subseteq Z \mid Z' \cap E \neq \emptyset\}$ .

Offenbar sind  $M'$  und  $M$  äquivalent, denn für alle  $x = a_1 a_2 \cdots a_n$  in  $\Sigma^*$  gilt:

$$\begin{aligned}
x \in L(M) &\iff \widehat{\delta}(S, x) \cap E \neq \emptyset \\
&\iff \text{es gibt eine Folge von Teilmengen } Z_1, Z_2, \dots, Z_n \subseteq Z \text{ mit} \\
&\quad \delta'(S, a_1) = Z_1, \delta'(Z_1, a_2) = Z_2, \dots, \delta'(Z_{n-1}, a_n) = Z_n \text{ und} \\
&\quad Z_n \cap E \neq \emptyset \\
&\iff Z_n = \widehat{\delta}(z'_0, x) \in F \\
&\iff x \in L(M').
\end{aligned}$$

Somit ist  $L(M') = L(M)$ . ■

**Beispiel 2.14 (DFA zu gegebenem NFA)** Wir modifizieren den NFA aus Beispiel 2.10 etwas und betrachten den folgenden um einen Zustand kleineren NFA  $M = (\Sigma, Z, \delta, S, E)$  in Abbildung 2.4.

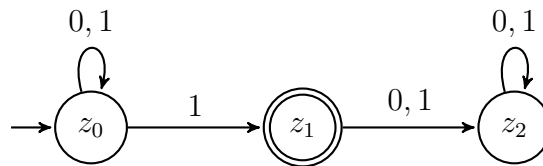


Abbildung 2.4: Ein nichtdeterministischer endlicher Automat

Der NFA aus Abbildung 2.4 akzeptiert die Sprache aller Wörter über  $\{0, 1\}^*$ , die an letzter Stelle eine 1 haben.

Wir konstruieren nun einen zu  $M$  äquivalenten DFA  $M' = (\Sigma, Z', \delta', S, F)$  gemäß dem Satz von Rabin und Scott:

$$\begin{aligned}
 \Sigma &= \{0, 1\}, \\
 Z' &= \{\{z_0, z_1, z_2\}, \{z_0, z_1\}, \{z_0, z_2\}, \{z_1, z_2\}, \{z_0\}, \{z_1\}, \{z_2\}, \emptyset\}, \\
 S &= \{z_0\}, \\
 F &= \{\{z_0, z_1, z_2\}, \{z_0, z_1\}, \{z_1, z_2\}, \{z_1\}\},
 \end{aligned}$$

$\delta'$	$\emptyset$	$\{z_0\}$	$\{z_1\}$	$\{z_2\}$	$\{z_0, z_1\}$	$\{z_0, z_2\}$	$\{z_1, z_2\}$	$\{z_0, z_1, z_2\}$
0	$\emptyset$	$\{z_0\}$	$\{z_2\}$	$\{z_2\}$	$\{z_0, z_2\}$	$\{z_0, z_2\}$	$\{z_2\}$	$\{z_0, z_2\}$
1	$\emptyset$	$\{z_0, z_1\}$	$\{z_2\}$	$\{z_2\}$	$\{z_0, z_1, z_2\}$	$\{z_0, z_1, z_2\}$	$\{z_2\}$	$\{z_0, z_1, z_2\}$

Offensichtlich können die Zustände  $\emptyset, \{z_1\}, \{z_2\}, \{z_1, z_2\}$  nicht vom Startzustand  $\{z_0\}$  erreicht werden und können zur Vereinfachung weggelassen werden.

Die Potenzmengenkonstruktion lässt vermuten, dass es Sprachen gibt, bei denen jeder DFA exponentiell mehr Zustände haben muss als der beste NFA. Dies trifft im folgenden Beispiel zu.

**Beispiel 2.15** Wir betrachten für  $n \geq 1$  die Sprache

$$\begin{aligned}
 L_n &= \{x \in \{0, 1\}^* \mid \text{der } n\text{-te Buchstabe von hinten in } x \text{ ist } 1\} \\
 &= \{x \in \{0, 1\}^* \mid x = u1v \text{ mit } u \in \{0, 1\}^* \text{ und } v \in \{0, 1\}^{n-1}\}.
 \end{aligned}$$

Es gilt:

1. Es gibt einen NFA für  $L_n$  mit  $n + 1$  Zuständen.

(Beweis: Einfache Modifikation der NFAs für  $L_2$  und  $L_1$  in Abbildung 2.3 und 2.4.)

2. Jeder DFA für  $L_n$  hat mindestens  $2^n$  Zustände.

(Beweis: s. Kapitel 2.5.1: Der Satz von Myhill und Nerode.)

Nun vollenden wir den Ringschluss.

**Satz 2.16** Für jede reguläre Grammatik  $G$  gibt es einen NFA  $M$  mit  $L(M) = L(G)$ .

**Beweis.** Sei  $G = (\Sigma, N, S, P)$  eine gegebene reguläre Grammatik. Konstruiere einen NFA  $M = (\Sigma, Z, \delta, S', F)$  so:

- $Z = N \cup \{X\}$ , wobei  $X \notin N \cup \Sigma$  ein neues Symbol ist,

- 

$$F = \begin{cases} \{S, X\} & \text{falls } S \rightarrow \lambda \text{ in } P \\ \{X\} & \text{falls } S \rightarrow \lambda \text{ nicht in } P, \end{cases}$$

- $S' = \{S\}$  und

- für alle  $A \in N$  und  $a \in \Sigma$  sei

$$\delta(A, a) = \left( \bigcup_{A \rightarrow aB \in P} \{B\} \right) \cup \bigcup_{A \rightarrow a \in P} \{X\}.$$

Es gilt:  $\lambda \in L(G) \iff \lambda \in L(M)$ . Außerdem gilt für alle  $n \geq 1$  und  $x = a_1 a_2 \cdots a_n$  in  $\Sigma^*$ :

$$\begin{aligned} x \in L(G) &\iff \text{es gibt eine Folge } A_1, A_2, \dots, A_{n-1} \in N \text{ mit} \\ &\quad S \vdash a_1 A_1 \vdash a_1 a_2 A_2 \vdash \cdots \vdash a_1 a_2 \cdots a_{n-1} A_{n-1} \vdash a_1 a_2 \cdots a_{n-1} a_n \\ &\iff \text{es gibt eine Folge } A_1, A_2, \dots, A_{n-1} \in Z \text{ mit } A_1 \in \delta(S, a_1), \\ &\quad A_2 \in \delta(A_1, a_2), \dots, A_{n-1} \in \delta(A_{n-2}, a_{n-1}), X \in \delta(A_{n-1}, a_n) \\ &\iff \widehat{\delta}(S', x) \cap F = \widehat{\delta}(\{S\}, x) \cap F \neq \emptyset \\ &\iff x \in L(M). \end{aligned}$$

Somit ist  $L(G) = L(M)$ . ■

**Beispiel 2.17 (NFA zu regulärer Grammatik)** Es sei  $G = (\Sigma, N, S, P)$  die folgende reguläre Grammatik:

$$\begin{aligned}\Sigma &= \{0, 1\}, \\ N &= \{S\}, \\ P &= \{S \rightarrow 0S \mid 1S \mid 1\}.\end{aligned}$$

Nach dem obigen Beweis konstruieren wir den NFA  $M = (\Sigma, Z, \delta, S', F)$  mit

$$\begin{aligned}\Sigma &= \{0, 1\}, \\ Z &= \{S, X\}, \\ F &= \{X\}, \\ S' &= \{S\}, \\ \delta &: \delta(S, 0) = \{S\}, \\ &\quad \delta(S, 1) = \{S, X\}.\end{aligned}$$

**Korollar 2.18**  $\text{REG} = \{L(M) \mid M \text{ ist DFA}\} = \{L(M) \mid M \text{ ist NFA}\}.$

Zu jeder regulären Sprache gibt es unendlich viele DFAs bzw. NFAs, die sie akzeptieren.

## 2.2 Reguläre Ausdrücke

**Definition 2.19 (Reguläre Ausdrücke)** Sei  $\Sigma$  ein Alphabet. Die Menge der regulären Ausdrücke (über  $\Sigma$ ) ist definiert durch:

- $\emptyset$  und  $\lambda$  sind reguläre Ausdrücke.
- Jedes  $a \in \Sigma$  ist ein regulärer Ausdruck.
- Sind  $\alpha$  und  $\beta$  reguläre Ausdrücke, so sind auch
  - $\alpha\beta$ ,
  - $(\alpha + \beta)$  und
  - $(\alpha)^*$

reguläre Ausdrücke.

- Nichts sonst ist ein regulärer Ausdruck.



**Definition 2.20 (Sprache regulärer Ausdrücke)** Sei  $\Sigma$  ein Alphabet. Jedem regulären Ausdruck  $\gamma$  ist in eindeutiger Weise eine Sprache  $L(\gamma) \subseteq \Sigma^*$  zugeordnet (wir sagen: „ $\gamma$  beschreibt  $L(\gamma)$ “), die so definiert ist:

$$L(\gamma) = \begin{cases} \emptyset & \text{falls } \gamma = \emptyset \\ \{\lambda\} & \text{falls } \gamma = \lambda \\ \{a\} & \text{falls } \gamma = a \text{ für ein } a \in \Sigma \\ L(\alpha)L(\beta) & \text{falls } \gamma = \alpha\beta \\ L(\alpha) \cup L(\beta) & \text{falls } \gamma = (\alpha + \beta) \\ L(\alpha)^* & \text{falls } \gamma = (\alpha)^*. \end{cases}$$

Zwei reguläre Ausdrücke  $\alpha_1$  und  $\alpha_2$  heißen äquivalent (kurz  $\alpha_1 \sim \alpha_2$ ), falls  $L(\alpha_1) = L(\alpha_2)$ .

**Beispiel 2.21** Es sei  $\Sigma = \{a, b\}$ .

1. Der reguläre Ausdruck  $\alpha_1 = (\lambda + a(a + b)^*)$  beschreibt die Sprache

$$L(\alpha_1) = \{\lambda\} \cup \{ax \mid x \in \{a, b\}^*\},$$

d.h. das leere Wort und alle Wörter über  $\Sigma$ , die mit  $a$  beginnen.

2. Der reguläre Ausdruck  $\alpha_2 = ((a + b)^*ab)$  beschreibt die Sprache

$$L(\alpha_2) = \{xab \mid x \in \{a, b\}^*\},$$

d.h. alle Wörter über  $\Sigma$ , die mit  $ab$  enden.

### Bemerkung 2.22

- Wegen  $\emptyset^* = \{\lambda\}$  hätte man den regulären Ausdruck  $\lambda$  in der Definition oben auch weglassen können.
- Formal korrekt hätte man die Menge der regulären Ausdrücke unabhängig vom Alphabet (wie Grammatiken und DFAs) definieren müssen, etwa so:

$$\begin{aligned} \text{RegAusdruck}(\Sigma) &= \{\alpha \mid \alpha \text{ ist regulärer Ausdruck über } \Sigma\} \\ \text{RegAusdruck} &= \bigcup_{\Sigma \text{ ist Alphabet}} \text{RegAusdruck}(\Sigma). \end{aligned}$$

Wir nehmen jedoch an, dass implizit immer ein Alphabet fixiert ist.

- Jede endliche Sprache  $A \subseteq \Sigma^*$  wird durch einen regulären Ausdruck beschrieben. Ist etwa  $A = \{a_1, a_2, \dots, a_k\}$ , so ist  $A = L(\alpha)$  mit

$$\alpha = (\dots((a_1 + a_2) + a_3) + \dots + a_k).$$

Somit ist jede endliche Sprache regulär.

- Jeder reguläre Ausdruck  $\alpha$  beschreibt genau eine Sprache, nämlich  $L(\alpha)$ , aber für jede reguläre Sprache  $A = L(\alpha)$  gibt es zum Beispiel wegen

$$L((\alpha + \alpha)) = L(\alpha) \cup L(\alpha) = L(\alpha)$$

unendlich viele reguläre Ausdrücke, die  $A$  beschreiben.

**Satz 2.23 (Kleene)**  $\text{REG} = \{L(\alpha) \mid \alpha \text{ ist regulärer Ausdruck}\}.$

**Beweis.**

( $\supseteq$ ) Sei  $A = L(\alpha)$  für einen regulären Ausdruck  $\alpha$ . Wir zeigen durch Induktion über den Aufbau von  $\alpha$ , dass  $A \in \text{REG}$  durch Angabe eines NFA  $M$  mit  $L(M) = A$ .

**Induktionsanfang:**  $\alpha = \emptyset$  oder  $\alpha = \lambda$  oder  $\alpha = a \in \Sigma$ .

Der NFA

$$M = \begin{cases} (\Sigma, \{z_0\}, \emptyset, \{z_0\}, \emptyset) & \text{falls } \alpha = \emptyset \\ (\Sigma, \{z_0\}, \emptyset, \{z_0\}, \{z_0\}) & \text{falls } \alpha = \lambda \\ (\Sigma, \{z_0, z_1\}, \{\delta(z_0, a) = \{z_1\}\}, \{z_0\}, \{z_1\}) & \text{falls } \alpha = a \in \Sigma \end{cases}$$

leistet je nach Fall das Gewünschte.

**Induktionsschritt:**  $\alpha \in \{\alpha_1\alpha_2, (\alpha_1 + \alpha_2), (\alpha_1)^*\}$ , für reguläre Ausdrücke  $\alpha_1$  und  $\alpha_2$ .

Nach Induktionsvoraussetzung existieren NFAs  $M_1$  und  $M_2$  mit  $L(M_i) = L(\alpha_i)$ , für  $i \in \{1, 2\}$ . Sei für  $i \in \{1, 2\}$ :

$$M_i = (\Sigma, Z_i, \delta_i, S_i, F_i),$$

wobei o.B.d.A.  $Z_1 \cap Z_2 = \emptyset$  gelte; andernfalls werden Zustände entsprechend umbenannt.

**Fall 1:**  $\alpha = \alpha_1\alpha_2$ .

Definiere  $M = (\Sigma, Z, \delta, S, F)$  als die „Hintereinanderschaltung“ von  $M_1$  und  $M_2$  durch:

$$\begin{aligned} Z &= Z_1 \cup Z_2 \\ S &= \begin{cases} S_1 & \text{falls } \lambda \notin L(\alpha_1) = L(M_1) \\ S_1 \cup S_2 & \text{falls } \lambda \in L(\alpha_1) = L(M_1) \end{cases} \\ F &= \begin{cases} F_2 & \text{falls } \lambda \notin L(\alpha_2) = L(M_2) \\ F_1 \cup F_2 & \text{falls } \lambda \in L(\alpha_2) = L(M_2) \end{cases} \\ \delta(z, a) &= \begin{cases} \delta_1(z, a) & \text{falls } z \in Z_1 \text{ und } \delta_1(z, a) \cap F_1 = \emptyset \\ \delta_1(z, a) \cup S_2 & \text{falls } z \in Z_1 \text{ und } \delta_1(z, a) \cap F_1 \neq \emptyset \\ \delta_2(z, a) & \text{falls } z \in Z_2 \end{cases} \end{aligned}$$

für alle  $a \in \Sigma$  und  $z \in Z$ .

**Fall 2:**  $\alpha = (\alpha_1 + \alpha_2)$ .

Der „Vereinigungsautomat“  $M = (\Sigma, Z_1 \cup Z_2, \delta_1 \cup \delta_2, S_1 \cup S_2, F_1 \cup F_2)$  von  $M_1$  und  $M_2$  leistet das Gewünschte.

**Fall 3:**  $\alpha = (\alpha_1)^*$ .

Der „Rückkopplungsautomat“  $M = (\Sigma, Z, \delta, S, F)$  von  $M_1$  leistet das Gewünschte, der folgendermaßen definiert ist:

1. Zunächst fügen wir  $\lambda$  hinzu, falls nötig:

$$\widehat{M} = \begin{cases} M_1 & \text{falls } \lambda \in L(M_1) \\ (\Sigma, \widehat{Z}, \delta, \widehat{S}, \widehat{F}) & \text{falls } \lambda \notin L(M_1) \end{cases}$$

mit  $\widehat{Z} = Z_1 \cup \{\hat{z}\}$ ,  $\widehat{S} = S_1 \cup \{\hat{z}\}$ ,  $\widehat{F} = F_1 \cup \{\hat{z}\}$ , wobei  $\hat{z} \notin Z_1$ .  
Offenbar ist  $L(\widehat{M}) = L(M_1) \cup \{\lambda\}$ .

2. Rückkopplung: Definiere  $M = (\Sigma, Z, \delta, S, F)$ , wobei  $Z$ ,  $S$  und  $F$  die Menge der Zustände, Anfangszustände und Endzustände von  $\widehat{M}$  sind, und für alle  $a \in \Sigma$  und  $z \in Z$ :

$$\delta(z, a) = \begin{cases} \delta_1(z, a) & \text{falls } \delta_1(z, a) \cap F = \emptyset \\ \delta_1(z, a) \cup S & \text{falls } \delta_1(z, a) \cap F \neq \emptyset. \end{cases}$$

Offenbar gilt  $L(M) = L(M_1)^* = L((\alpha_1)^*) = L(\alpha)$ .

( $\subseteq$ ) Sei  $A \in \text{REG}$ , und sei  $M$  ein DFA mit  $L(M) = A$ , wobei  $M = (\Sigma, Z, \delta, z_1, E)$  mit  $Z = \{z_1, z_2, \dots, z_n\}$ .

Konstruiere einen regulären Ausdruck  $\alpha$  mit  $L(\alpha) = A$ . Definiere dazu Sprachen  $R_{i,j}^k$  mit  $i, j \in \{1, 2, \dots, n\}$  und  $k \in \{0, 1, \dots, n\}$ , die sich durch reguläre Ausdrücke beschreiben lassen:

$$R_{i,j}^k = \left\{ x \in \Sigma^* \mid \begin{array}{l} \widehat{\delta}(z_i, x) = z_j \text{ und keiner der dabei durchlaufenen Zustände} \\ \text{(außer } z_i \text{ und } z_j \text{ selbst) hat einen Index größer als } k. \end{array} \right\}.$$

Wir zeigen durch Induktion über  $k$ : Für jedes  $k \in \{0, 1, \dots, n\}$  lässt sich jede Sprache  $R_{i,j}^k$  durch einen regulären Ausdruck beschreiben.

**Induktionsanfang:  $k = 0$ .** Ist  $i \neq j$ , so ist

$$R_{i,j}^0 = \{a \in \Sigma \mid \delta(z_i, a) = z_j\}.$$

Ist  $i = j$ , so ist

$$R_{i,i}^0 = \{\lambda\} \cup \{a \in \Sigma \mid \delta(z_i, a) = z_i\}.$$

In beiden Fällen ist  $R_{i,j}^0$  endlich und daher durch einen regulären Ausdruck beschreibbar.

**Induktionsschritt:  $k \mapsto k + 1$ .** Es gilt

$$R_{i,j}^{k+1} = R_{i,j}^k \cup R_{i,k+1}^k (R_{k+1,k+1}^k)^* R_{k+1,j}^k,$$

denn: Entweder braucht man den Zustand  $z_{k+1}$  gar nicht, um von  $z_i$  nach  $z_j$  zu kommen, oder  $z_{k+1}$  wird dabei einmal oder mehrmals in Schleifen durchlaufen, siehe Abbildung 2.5.

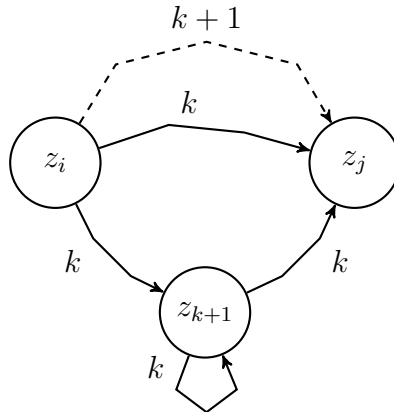


Abbildung 2.5: Skizze zum Beweis des Induktionsschritts

Nach Induktionsvoraussetzung gibt es reguläre Ausdrücke  $\alpha_{i,j}^k$  für die Mengen  $R_{i,j}^k$ . Somit ist

$$\alpha_{i,j}^{k+1} = (\alpha_{i,j}^k + \alpha_{i,k+1}^k (\alpha_{k+1,k+1}^k)^* \alpha_{k+1,j}^k)$$

ein regulärer Ausdruck für  $R_{i,j}^{k+1}$ . Ende der Induktion.

Ferner gilt:

$$A = L(M) = \bigcup_{z_i \in E} R_{1,i}^n.$$

Ist also  $E = \{z_{i_1}, z_{i_2}, \dots, z_{i_m}\}$ , so ist

$$\alpha = (\alpha_{1,i_1}^n + \alpha_{1,i_2}^n + \dots + \alpha_{1,i_m}^n)$$

ein regulärer Ausdruck für  $A$ .

■

## 2.3 Gleichungssysteme

**Ziel:** Bestimmung der von einem NFA (bzw. DFA) akzeptierten Sprache durch Lösung eines linearen Gleichungssystems.

**Gegeben:** NFA  $M = (\Sigma, Z, \delta, S, F)$  mit  $Z = \{z_1, z_2, \dots, z_n\}$ .

**Gesucht:**  $L(M)$ .

**Methode:** Bilde ein Gleichungssystem mit  $n$  Variablen und  $n$  Gleichungen wie folgt:

1. Jedes  $z_i \in Z$ ,  $1 \leq i \leq n$ , ist Variable auf der linken Seite einer Gleichung.
2. Gilt  $z_j \in \delta(z_i, a)$  für  $z_i, z_j \in Z$  und  $a \in \Sigma$ , so ist  $az_j$  Summand auf der rechten Seite der Gleichung „ $z_i = \dots$ “.
3. Gilt  $z_i \in F$ , so ist  $\emptyset^*$  Summand auf der rechten Seite der Gleichung „ $z_i = \dots$ “.

Die  $z_i$  werden als reguläre Sprachen (bzw. Ausdrücke) interpretiert und gemäß Lemma 2.24 und Satz 2.26 ausgerechnet. Es gilt dann:

$$L(M) = \bigcup_{z_i \in S} z_i$$

bzw.  $L(M) = L(\alpha)$  für den regulären Ausdruck  $\alpha = \sum_{z_i \in S} z_i$ .

**Lemma 2.24** Sind  $A, B \subseteq \Sigma^*$  reguläre Sprachen mit  $\lambda \notin A$ , so gibt es genau eine reguläre Sprache  $X$ , die Lösung der Gleichung

$$X = AX \cup B \tag{2.1}$$

ist.

**Beweis.** Setze  $X = A^*B$ . Offenbar ist  $X$  regulär, denn  $X = L(\gamma)$  für den regulären Ausdruck  $\gamma = (\alpha)^*\beta$ , wobei  $A = L(\alpha)$  und  $B = L(\beta)$ .

Es gilt:

1.  $X = A^*B$  ist kleinste (d.h. inklusionsminimale) Lösung von (2.1), denn:

$$\begin{aligned}
 A(A^*B) \cup B &= \left( A \bigcup_{n \geq 0} A^n \cup \{\lambda\} \right) B \\
 &= \left( A^0 \cup \bigcup_{n \geq 1} A^n \right) B \\
 &= \left( \bigcup_{n \geq 0} A^n \right) B \\
 &= A^*B.
 \end{aligned}$$

Ist  $Y$  irgendeine Lösung von (2.1), so gilt:

$$Y = AY \cup B.$$

Es folgt:

$$\begin{aligned}
 B &\subseteq AY \cup B = Y \\
 AB &\subseteq AY \cup B = Y \\
 A^2B &\subseteq AY \cup B = Y \\
 &\vdots \\
 A^*B &= \left( \bigcup_{n \geq 0} A^n \right) B \subseteq AY \cup B = Y.
 \end{aligned}$$

2.  $X = A^*B$  ist die einzige Lösung von (2.1), denn: Angenommen, es gäbe eine andere Lösung  $Y \not\subseteq A^*B$  von (2.1). Dann existiert ein *kürzestes* Wort  $y \in Y$  mit  $y \notin A^*B$ . Da  $Y$  Lösung von (2.1) ist, folgt aus  $y \in Y$ :

$$y \in AY \cup B.$$

**Fall 1:**  $y \in B$ . Dann ist  $y \in B = \{\lambda\}B \subseteq A^*B$ . Widerspruch.

**Fall 2:**  $y \notin B$ . Dann muss  $y \in AY$  gelten. Somit ist  $y = vw$  mit  $v \in A$  und  $w \in Y$ . Da  $\lambda \notin A$ , ist  $|v| \geq 1$ . Somit ist  $|w| < |y|$ .

**Unterfall 2.1:**  $w \in A^*B$ . Dann ist auch  $y = vw \in A^*B$ . Widerspruch.

**Unterfall 2.2:**  $w \notin A^*B$ . Dann ist  $y$  nicht das kürzeste Wort mit  $y \in Y$  und  $y \notin A^*B$ . Widerspruch zur Wahl von  $y$ .

Folglich gibt es genau eine Lösung von (2.1), nämlich die reguläre Sprache  $X = A^*B$ . ■

**Bemerkung 2.25** Gilt  $\lambda \in A$  in (2.1), so ist die reguläre Menge  $X = A^*B$  ebenfalls Lösung von (2.1). Allerdings muss diese dann nicht mehr eindeutig sein, da  $y = vw \in AY$  auch für  $v = \lambda$  gelten kann und der obige Widerspruch dann nicht auftreten muss.

Aus obiger Methode ergibt sich ein Gleichungssystem der Form:

$$\begin{aligned} z_1 &= A_{11}z_1 \cup A_{12}z_2 \cup \dots \cup A_{1n}z_n \cup B_1 \\ z_2 &= A_{21}z_1 \cup A_{22}z_2 \cup \dots \cup A_{2n}z_n \cup B_2 \\ &\vdots \\ z_n &= A_{n1}z_1 \cup A_{n2}z_2 \cup \dots \cup A_{nn}z_n \cup B_n \end{aligned} \tag{2.2}$$

**Satz 2.26** Gilt  $A_{ij}, B_k \in \text{REG}$  und  $\lambda \notin A_{ij}$  für alle  $i, j, k \in \{1, 2, \dots, n\}$  im obigen Gleichungssystem (2.2), dann hat es eine eindeutig bestimmte Lösung durch reguläre Mengen  $z_i$ ,  $1 \leq i \leq n$ .

**Beweis.** Gegeben sei ein Gleichungssystem der Form (2.2). Löse es sukzessive durch Anwendung der folgenden Schritte:

1. Gibt es Gleichungen „ $z_i = \dots$ “, deren linke Seite  $z_i$  nicht auf der rechten Seite vorkommt, so eliminiere diese Gleichung „ $z_i = \dots$ “ und die Variable  $z_i$  in jeder anderen Gleichung durch Einsetzen der rechten Seite von „ $z_i = \dots$ “.
2. Vereinfache nach dem Distributivgesetz

$$UX \cup VX = (U \cup V)X.$$

3. Das entstandene System hat wieder die Form (2.2), wobei nun jede linke Seite  $z_i$  auch rechts vorkommt. Das heißt, jede verbliebene Gleichung hat die Form (2.1):

$$z_i = Az_i \cup B,$$

wobei  $A$  und  $B$  sich aus den Koeffizienten  $A_{ij}, B_k \in \text{REG}$  und den Variablen  $z_j$ ,  $i \neq j$ , in (2.2) gemäß dem obigen Umformungsprozess ergeben. Nach Lemma 2.24 hat eine solche Gleichung die eindeutige Lösung  $A^*B$ , die eine reguläre Sprache ist.

4. Ersetze der Reihe nach alle noch vorhandenen Variablen durch die entsprechenden Lösungen. Stets entsteht dabei ein System von der Form (2.2) mit einer Variablen weniger, und in keiner der Koeffizientenmengen kommt  $\lambda$  vor.

Satz 2.26 ist bewiesen. ■

## 2.4 Das Pumping-Lemma

**Ziel:** Nachweis der Nichtregularität von Sprachen.

**Satz 2.27 (Pumping-Lemma für reguläre Sprachen)** Sei  $L \in \text{REG}$ . Dann existiert eine (von  $L$  abhängige) Zahl  $n \geq 1$ , so dass sich alle Wörter  $x \in L$  mit  $|x| \geq n$  zerlegen lassen in  $x = uvw$ , wobei gilt:

1.  $|uv| \leq n$ ,
2.  $|v| \geq 1$ ,
3.  $(\forall i \geq 0) [uv^i w \in L]$ .

**Beweis.** Sei  $L \in \text{REG}$ , und sei  $M$  ein DFA für  $L$ . Wähle als  $n$  die Zahl der Zustände von  $M$ . Sei  $x \in L$  beliebig mit  $|x| \geq n$ .

Beim Abarbeiten von  $x$  durchläuft  $M$  genau  $|x| + 1 \geq n + 1$  Zustände einschließlich des Startzustands. Folglich gibt es einen Zustand  $z$ , der zweimal durchlaufen wird. Das heißt,  $M$  hat bei Eingabe  $x$  eine Schleife durchlaufen. Wähle nun die Zerlegung  $x = uvw$  so, dass nach Lesen von  $u$  und von  $uv$  derselbe Zustand  $z$  erreicht ist und die Bedingungen (1) und (2) erfüllt sind. Es ist klar, dass das geht.

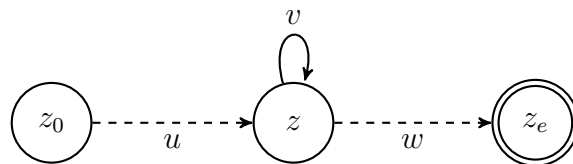


Abbildung 2.6: Beweis des Pumping-Lemmas für reguläre Sprachen

$M$  erreicht somit denselben Endzustand  $z_e$  bei Eingabe von

$$\begin{aligned}
 uv^0 w &= uw \\
 uv^1 w &= uvw = x \\
 uv^2 w & \\
 &\vdots \\
 uv^i w & \\
 &\vdots
 \end{aligned}$$

(siehe Abbildung 2.6). Somit sind alle diese Wörter in  $L$ , und auch Eigenschaft (3) ist bewiesen. ■



**Bemerkung 2.28** Man beachte, dass Satz 2.27 keine Charakterisierung von REG liefert, sondern lediglich eine Implikation:

$$L \in \text{REG} \Rightarrow (\exists n \geq 1) (\forall x \in L, |x| \geq n) (\exists u, v, w \in \Sigma^*) [x = uvw \wedge (1) \wedge (2) \wedge (3)].$$

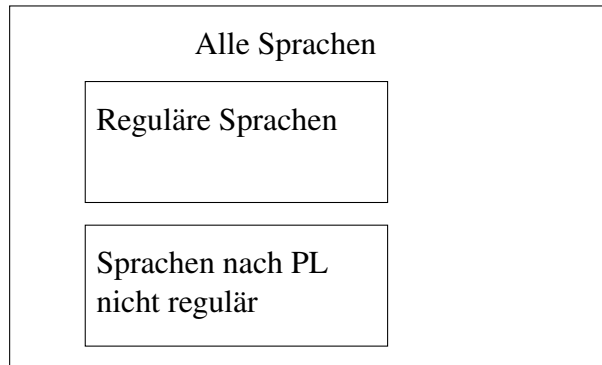


Abbildung 2.7: Anwendbarkeit des Pumping-Lemmas (PL)

## Anwendungsbeispiele

**Behauptung 2.29**  $L = \{a^m b^m \mid m \geq 1\}$  ist nicht regulär.

**Beweis.** Der Beweis wird indirekt geführt. Angenommen,  $L \in \text{REG}$ . Sei  $n$  die Zahl, die nach dem Pumping-Lemma (Satz 2.27) für  $L$  existiert. Betrachte das Wort  $x = a^n b^n$  mit  $|x| = 2n > n$ . Da  $x \in L$ , lässt sich  $x$  so in  $x = uvw = a^n b^n$  zerlegen, dass gilt:

1.  $|uv| \leq n$ , d.h.,  $uv = a^m$  für  $m \leq n$ ;
2.  $|v| \geq 1$ , d.h.,  $v = a^k$  mit  $k \geq 1$ ;
3.  $(\forall i \geq 0) [uv^i w \in L]$ .

Insbesondere gilt für  $i = 0$ :

$$uv^0 w = uw = a^{n-k} b^n \in L,$$

was ein Widerspruch zur Definition von  $L$  ist. Also ist die Annahme falsch und  $L$  nicht regulär. ■

**Behauptung 2.30**  $L = \{0^m \mid m \text{ ist Quadratzahl}\}$  ist nicht regulär.

**Beweis.** Der Beweis wird wieder indirekt geführt. Angenommen,  $L \in \text{REG}$ . Sei  $n$  die Zahl, die nach dem Pumping-Lemma für  $L$  existiert. Betrachte das Wort  $x = 0^{n^2}$  mit  $|x| = n^2 \geq n$ . Da  $x \in L$ , lässt sich  $x$  so in  $x = uvw = 0^{n^2}$  zerlegen, dass die Bedingungen (1), (2) und (3) aus Satz 2.27 gelten. Aus den Bedingungen (1) und (2) folgt:

$$1 \leq |v| \leq |uv| \leq n.$$

Aus Bedingung (3) folgt für  $i = 2$ , dass  $uv^2w$  ein Wort in  $L$  ist. Andererseits gilt:

$$n^2 = |x| = |uvw| < |uv^2w| \leq n^2 + n < n^2 + 2n + 1 = (n + 1)^2.$$

Folglich ist die Länge des Wortes  $uv^2w \in L$  keine Quadratzahl, was ein Widerspruch zur Definition von  $L$  ist. Also ist die Annahme falsch und  $L$  nicht regulär. ■

Hier folgt ein Beispiel für eine nicht reguläre Sprache, die jedoch trotzdem die Bedingungen des Pumping-Lemmas für reguläre Sprachen erfüllt.

**Beispiel 2.31 (Eine nicht reguläre Sprache, die die PL-Bedingungen dennoch erfüllt)**

*$L$  sei die Sprache aller Wörter, die entweder nur Einsen enthalten (wobei auch null Einsen möglich sind) oder mit mindestens einer Null beginnen, gefolgt von einer Quadratzahl von Einsen, d.h.,*

$$L = \{x \in \{0, 1\}^* \mid x = 1^k \text{ mit } k \geq 0 \text{ oder } x = 0^j 1^{k^2} \text{ mit } j \geq 1 \text{ und } k \geq 1\}.$$

*Wir versuchen wieder (wie sich hier zeigt: vergeblich!), einen Widerspruchsbeweis für  $L \notin \text{REG}$  zu führen, und nehmen also  $L \in \text{REG}$  an.*

*Sei  $x \in L$  beliebig mit  $|x| \geq n$ , wobei  $n$  die Pumping-Lemma-Zahl für  $L$  ist. Nach dem Pumping-Lemma gibt es eine Zerlegung  $x = uvw$ , die die Bedingungen (1), (2) und (3) erfüllt.*

**Fall 1:**  $x = 1^k$ . Für  $x = 1^k = uvw$  liefern diese drei Bedingungen (insbesondere  $uv^i w = 1^{k'} \in L$  für alle  $i \geq 0$ ) jedoch keinen Widerspruch.

**Fall 2:**  $x = 0^j 1^{k^2}$ . Auch für  $x = 0^j 1^{k^2} = uvw$  ergibt sich kein Widerspruch aus den drei Bedingungen; insbesondere könnte die Zerlegung  $x = uvw$  mit  $u = \lambda$  und  $v = 0$  (da  $j \geq 1$ ) gewählt worden sein, so dass  $uv^i w = 0^{j+i-1} 1^{k^2} \in L$  für alle  $i \geq 0$  tatsächlich gilt.

*Auch wenn das Pumping-Lemma hier nicht den gewünschten Widerspruch liefert, lässt sich aufgrund von Behauptung 2.30 vermuten, dass  $L$  nicht regulär ist, was sich mittels einer Verallgemeinerung des Pumping-Lemmas tatsächlich auch beweisen lässt.*

## 2.5 Satz von Myhill und Nerode und Minimalautomaten

### 2.5.1 Der Satz von Myhill und Nerode

Man kann einer Sprache  $L \subseteq \Sigma^*$  wie folgt eine Äquivalenzrelation  $R_L$  auf  $\Sigma^*$  zuordnen.

**Definition 2.32 (Myhill-Nerode-Relation)** Sei  $L \subseteq \Sigma^*$  gegeben. Für  $x, y \in \Sigma^*$  gelte  $xR_Ly$  genau dann, wenn

$$(\forall z \in \Sigma^*) [xz \in L \iff yz \in L].$$

Insbesondere gilt für Wörter  $x, y \in \Sigma^*$  mit  $xR_Ly$ :

$$x \in L \iff y \in L,$$

nämlich für  $z = \lambda$ .

Wie jede Äquivalenzrelation induziert  $R_L$  eine Zerlegung (d.h. eine disjunkte und vollständige Überdeckung) von  $\Sigma^*$  in Äquivalenzklassen:

$$[x] = \{y \in \Sigma^* \mid xR_Ly\},$$

die repräsentantenunabhängig ist. Die Anzahl der verschiedenen Äquivalenzklassen wird bezeichnet mit

$$\text{Index}(R_L) = \|\{[x] \mid x \in \Sigma^*\}\|.$$

**Satz 2.33 (Myhill und Nerode)** Es sei  $L$  eine Sprache über einem beliebigen Alphabet  $\Sigma$ .

$$L \in \text{REG} \iff \text{Index}(R_L) < \infty.$$

**Beweis.** ( $\Rightarrow$ ) Sei  $L \in \text{REG}$ , und sei  $M = (\Sigma, Z, \delta, z_0, E)$  ein DFA mit  $L(M) = L$ . Definiere eine Äquivalenzrelation  $R_M$  auf  $\Sigma^*$  wie folgt:  $xR_My$  gelte genau dann, wenn

$$\hat{\delta}(z_0, x) = \hat{\delta}(z_0, y),$$

d.h.,  $M$  ist nach dem Lesen von  $x$  im selben Zustand wie nach dem Lesen von  $y$ . Es gilt:

$$\text{Index}(R_M) = \text{Anzahl der von } z_0 \text{ aus erreichbaren Zustände.} \quad (2.3)$$

Wir zeigen nun:

$$(\forall x, y \in \Sigma^*) [xR_My \Rightarrow xR_Ly], \quad (2.4)$$

d.h.,  $R_M \subseteq R_L$  (also ist  $R_M$  eine Verfeinerung von  $R_L$ ).

Sei also  $xR_My$ , d.h.,  $\widehat{\delta}(z_0, x) = \widehat{\delta}(z_0, y)$ . Sei  $z \in \Sigma^*$  beliebig. Dann gilt:

$$\begin{aligned}
 xz \in L &\iff \widehat{\delta}(z_0, xz) \in E \\
 &\iff \widehat{\delta}(\widehat{\delta}(z_0, x), z) \in E \\
 &\iff \widehat{\delta}(\widehat{\delta}(z_0, y), z) \in E \\
 &\iff \widehat{\delta}(z_0, yz) \in E \\
 &\iff yz \in L.
 \end{aligned}$$

Folglich gilt  $xR_Ly$  und somit (2.4).

Aus (2.3) und (2.4) folgt:

$$\text{Index}(R_L) \leq \text{Index}(R_M) \leq \|Z\| < \infty.$$

( $\Leftarrow$ ) Sei  $\text{Index}(R_L) = k < \infty$  für ein  $k \in \mathbb{N}$ . Dann lässt sich  $\Sigma^*$  in  $k$  Äquivalenzklassen bezüglich  $R_L$  zerlegen:

$$[x_1] \cup [x_2] \cup \dots \cup [x_k] = \Sigma^*,$$

wobei  $x_1, x_2, \dots, x_k \in \Sigma^*$ . Der Äquivalenzklassen-Automat für  $L$  ist der DFA  $M = (\Sigma, Z, \delta, z_0, F)$ , der definiert ist durch:

$$\begin{aligned}
 Z &= \{[x_1], [x_2], \dots, [x_k]\}, \\
 \delta([x], a) &= [xa] \text{ für jedes } [x] \in Z \text{ und } a \in \Sigma, \\
 z_0 &= [\lambda], \\
 F &= \{[x] \mid x \in L\}.
 \end{aligned}$$

**Lemma 2.34** Für alle  $x' \in [x]$  und  $a \in \Sigma$  gilt  $[x'a] = [xa]$ . Insbesondere gilt  $\widehat{\delta}([\lambda], x) = [x]$  für jedes  $x \in \Sigma^*$ .

Der Beweis von Lemma 2.34 ist so offensichtlich, dass er weggelassen wird.

Es folgt:

$$\begin{aligned}
 x \in L(M) &\iff \widehat{\delta}(z_0, x) \in F \\
 &\iff \widehat{\delta}([\lambda], x) \in F \\
 &\iff [x] \in F \text{ (nach Lemma 2.34)} \\
 &\iff x \in L.
 \end{aligned}$$

Also ist  $L \in \text{REG}$ . ■

**Beispiel 2.35 (Anwendungsbeispiele – Satz von Myhill und Nerode)****1. Nachweis der Nichtregularität von Sprachen** (vgl. Behauptung 2.29).

Wir zeigen mittels Satz 2.33, dass  $L = \{a^m b^m \mid m \geq 1\}$  nicht regulär ist. Zu den Äquivalenzklassen von  $L$  bzgl.  $R_L$  gehören:

$$\begin{aligned} [ab] &= \{x \in \Sigma^* \mid ab R_L x\} = \{ab, a^2 b^2, a^3 b^3, \dots\} = L \\ [a^2 b] &= \{a^2 b, a^3 b^2, a^4 b^3, \dots\} \\ [a^3 b] &= \{a^3 b, a^4 b^2, a^5 b^3, \dots\} \\ &\vdots \\ [a^k b] &= \{a^k b, a^{k+1} b^2, a^{k+2} b^3, \dots\} \\ &\vdots \end{aligned}$$

Diese sind alle paarweise verschieden, d.h.,  $[a^i b] \neq [a^j b]$  für  $i \neq j$ . Denn für  $z = b^{i-1}$  gilt  $a^i b z \in L$ , aber  $a^j b z \notin L$ . Folglich ist  $\text{Index}(R_L) = \infty$ . Nach Satz 2.33 ist  $L \notin \text{REG}$ .

**2. Nachweis der Regularität von Sprachen**

Wir zeigen mittels Satz 2.33, dass die Sprache  $L = \{x \in \{a, b\}^* \mid x \text{ endet mit } aa\}$  regulär ist. Die Äquivalenzklassen von  $L$  bzgl.  $R_L$  sind:

$$\begin{aligned} [aa] &= \{x \in \{a, b\}^* \mid aa R_L x\} = \{a^2, a^3, a^4, \dots, ba^2, ba^3, ba^4, \dots\} = L \\ [a] &= \{a, ba, b^2 a, \dots\} = \{x \in \{a, b\}^* \mid x \text{ endet mit } a, \text{ aber nicht mit } aa\} \\ [\lambda] &= \{\lambda, b, ab, \dots\} = \{x \in \{a, b\}^* \mid x \text{ endet nicht mit } a\}. \end{aligned}$$

Da  $\{a, b\}^* = [aa] \cup [a] \cup [\lambda]$ , ist  $\text{Index}(R_L) = 3$ . Nach Satz 2.33 ist  $L \in \text{REG}$ .

**3. Untere Schranken für die Anzahl der Zustände eines DFA**

Es sei  $L$  eine reguläre Sprache. Für jeden DFA  $M = (\Sigma, Z, \delta, z_0, E)$  mit  $L(M) = L$  gilt nach der „ $(\Rightarrow)$ “-Richtung im Beweis von Satz 2.33:

$$\begin{aligned} \text{Index}(R_L) &\leq \text{Index}(R_M) \\ &= \text{Anzahl der in } M \text{ von } z_0 \text{ aus erreichbaren Zustände} \\ &\leq \|Z\|. \end{aligned}$$

Das heißt  $\frac{1}{2}t$ ,  $M$  hat mindestens so viele Zustände wie  $R_L$  Äquivalenzklassen. Wir wenden dieses Ergebnis an, um eine untere Schranke für die Anzahl der Zustände

der DFAs anzugeben, die die Sprache

$$\begin{aligned} L_n &= \{x \in \{0, 1\}^* \mid \text{der } n\text{-te Buchstabe von hinten in } x \text{ ist } 1\} \\ &= \{x \in \{0, 1\}^* \mid x = u1v \text{ mit } u \in \{0, 1\}^* \text{ und } v \in \{0, 1\}^{n-1}\} \end{aligned}$$

erkennen.

Wir betrachten die Wörter aus  $\{0, 1\}^n$ , d.h. die Wörter der Länge  $n$  über dem Alphabet  $\{0, 1\}$ . Es seien  $u, v \in \{0, 1\}^n$  mit  $u \neq v$ . Dann gibt es ein  $i$ ,  $1 \leq i \leq n$ , so dass sich  $u$  und  $v$  an der Position  $i$  unterscheiden. Wir nehmen o.B.d.A. an, dass  $u$  an der Position  $i$  eine 0 und  $v$  an der Position  $i$  eine 1 hat:

$$\begin{array}{ccccccc|l} u & = & & & 0 & & & 0^{i-1} & \notin L_n \\ v & = & & & 1 & & & 0^{i-1} & \in L_n \\ & & 1 & 2 & \dots & i & \dots & n & \end{array}$$

Es gilt nun:  $u0^{i-1} \notin L_n$ , da der  $n$ -te Buchstabe von hinten in  $u0^{i-1}$  eine 0 ist, und  $v0^{i-1} \in L_n$ , da der  $n$ -te Buchstabe von hinten in  $v0^{i-1}$  eine 1 ist. Somit sind  $u$  und  $v$  nicht äquivalent bezüglich  $R_{L_n}$ . Da  $u$  und  $v$  beliebig waren, sind alle Wörter aus  $\{0, 1\}^n$  nicht äquivalent bezüglich  $R_{L_n}$ , und es folgt:  $\text{Index}(R_{L_n}) \geq 2^n$ . Somit hat jeder DFA für  $L_n$  mindestens  $2^n$  Zustände. (Erinnerung: Es gibt einen NFA für  $L_n$  mit  $n + 1$  Zuständen.)

#### 4. Minimalautomaten. Siehe Abschnitt 2.5.2.

### 2.5.2 Minimalautomaten

**Definition 2.36 (Minimalautomat)** Ein DFA  $M$  mit totaler Überföhrungsfunktion heißt Minimalautomat, falls es keinen zu  $M$  äquivalenten DFA mit totaler Überföhrungsfunktion und weniger Zuständen gibt.

**Bemerkung 2.37** Es sei  $L \subseteq \Sigma^*$  eine reguläre Sprache.

- Der in der Rückrichtung im Beweis von Satz 2.33 bestimmte Äquivalenzklassen-Automat  $M_0$  ist der DFA mit einer kleinstmöglichen Anzahl von Zuständen, der  $L$  akzeptiert.

Begründung:

- die Anzahl der Zustände von  $M_0$  ist gleich  $\text{Index}(R_L)$  (Anzahl der Äquivalenzklassen der Myhill-Nerode-Relation) und

- für jeden DFA  $M = (\Sigma, Z, \delta, z_0, E)$  mit  $L(M) = L = L(M_0)$  gilt nach der „ $(\Rightarrow)$ “-Richtung im Beweis von Satz 2.33:

$$\begin{aligned} \text{Index}(R_L) &\leq \text{Index}(R_M) \\ &= \text{Anzahl der in } M \text{ von } z_0 \text{ aus erreichbaren Zustände} \\ &\leq \|Z\|. \end{aligned}$$

Folglich hat  $M_0$  höchstens so viele Zustände wie  $M$ .

- Alle totalen DFAs, die  $L$  akzeptieren und genau  $\text{Index}(R_L)$  Zustände haben, sind bis auf Umbenennung der Zustände äquivalent, d.h., Minimalautomaten sind (bis auf Isomorphie) eindeutig bestimmt. **ohne Beweis**

### Algorithmus Minimalautomat:

**Eingabe:** DFA  $M = (\Sigma, Z, \delta, z_0, F)$  mit totaler Überföhrungsfunktion (d.h.,  $\delta(z, a)$  ist für alle  $z \in Z$  und  $a \in \Sigma$  definiert).

**Ausgabe:** Ein zu  $M$  äquivalenter Minimalautomat.

### **Algorithmus:**

1. Entferne alle von  $z_0$  aus nicht erreichbaren Zustände aus  $Z$ .
2. Erstelle eine Tabelle aller (ungeordneten) Zustandspaare  $\{z, z'\}$  von  $M$  mit  $z \neq z'$ .
3. Markiere alle Paare  $\{z, z'\}$  mit

$$z \in F \iff z' \notin F.$$

4. Sei  $\{z, z'\}$  ein unmarkiertes Paar. Prüfe für jedes  $a \in \Sigma$ , ob

$$\{\delta(z, a), \delta(z', a)\}$$

bereits markiert ist. Ist mindestens ein Test erfolgreich, so markiere auch  $\{z, z'\}$ .

5. Wiederhole Schritt 4, bis keine Änderung mehr eintritt.
6. Bilde maximale Mengen paarweise nicht disjunkter unmarkierter Zustandspaare und verschmelze jeweils alle Zustände einer Menge zu einem neuen Zustand.

Der obige Algorithmus kann mit einer Laufzeit von  $\mathcal{O}(\|Z\|^2)$  implementiert werden.

**Beispiel 2.38 (Minimalautomaten zu gegebenem DFA)** Wir wollen einen Minimalautomaten zum DFA  $M = (\Sigma, Z, \delta, z_0, F)$  bestimmen, wobei

$$\begin{aligned}\Sigma &= \{0, 1\} \\ Z &= \{z_0, z_1, z_2, z_3, z_4\} \\ F &= \{z_3, z_4\}\end{aligned} \quad \delta :$$

$\Sigma \backslash Z$	0	1
$z_0$	$z_1$	$z_2$
$z_1$	$z_3$	$z_3$
$z_2$	$z_4$	$z_4$
$z_3$	$z_3$	$z_3$
$z_4$	$z_4$	$z_4$

(1.) Es sind alle Zustände von  $z_0$  aus erreichbar.

(2.)+(3.) Markiere alle Paare  $\{z, z'\}$  mit  $z \in F \iff z' \notin F$ .

	$z_0$	$z_1$	$z_2$	$z_3$
$z_4$	×	×	×	
$z_3$	×	×	×	
$z_2$				
$z_1$				

(4.) Da  $\{\delta(z_0, 0), \delta(z_2, 0)\}$  markiert ist, wird  $\{z_0, z_2\}$  markiert, und da  $\{\delta(z_0, 0), \delta(z_1, 0)\}$  markiert ist, wird  $\{z_0, z_1\}$  markiert:

	$z_0$	$z_1$	$z_2$	$z_3$
$z_4$	×	×	×	
$z_3$	×	×	×	
$z_2$	×			
$z_1$	×			

(5.) Es ergeben sich nun keine weiteren Änderungen mehr.

(6.) Wir können die Zustände  $z_1$  und  $z_2$  bzw.  $z_3$  und  $z_4$  zu einem Zustand  $z_{12}$  bzw.  $z_{34}$  zusammenfassen.

Damit erhalten wir einen zu  $M$  äquivalenten DFA  $M' = (\Sigma, Z', \delta', z_0, F')$  mit

$$\begin{aligned}\Sigma &= \{0, 1\} \\ Z' &= \{z_0, z_{12}, z_{34}\} \\ F' &= \{z_{34}\}\end{aligned} \quad \delta' :$$

$\Sigma \backslash Z'$	0	1
$z_0$	$z_{12}$	$z_{12}$
$z_{12}$	$z_{34}$	$z_{34}$
$z_{34}$	$z_{34}$	$z_{34}$

Der DFA akzeptiert offenbar die Sprache aller Worte über  $\{0, 1\}^*$  der Länge mindestens zwei.



## 2.6 Abschlusseigenschaften regulärer Sprachen

**Definition 2.39** Seien  $\Sigma$  ein Alphabet und  $\mathcal{C} \subseteq \mathfrak{P}(\Sigma^*)$  eine Sprachklasse über  $\Sigma$ .  $\mathcal{C}$  heißt abgeschlossen unter

- Vereinigung, falls  $(\forall A, B \subseteq \Sigma^*) [(A \in \mathcal{C} \wedge B \in \mathcal{C}) \Rightarrow A \cup B \in \mathcal{C}]$ ;
- Komplement, falls  $(\forall A \subseteq \Sigma^*) [A \in \mathcal{C} \Rightarrow \overline{A} \in \mathcal{C}]$ ;
- Schnitt, falls  $(\forall A, B \subseteq \Sigma^*) [(A \in \mathcal{C} \wedge B \in \mathcal{C}) \Rightarrow A \cap B \in \mathcal{C}]$ ;
- Differenz, falls  $(\forall A, B \subseteq \Sigma^*) [(A \in \mathcal{C} \wedge B \in \mathcal{C}) \Rightarrow A - B \in \mathcal{C}]$ ;
- Konkatenation, falls  $(\forall A, B \subseteq \Sigma^*) [(A \in \mathcal{C} \wedge B \in \mathcal{C}) \Rightarrow AB \in \mathcal{C}]$ ;
- Iteration (Kleene-Hülle), falls  $(\forall A \subseteq \Sigma^*) [A \in \mathcal{C} \Rightarrow A^* \in \mathcal{C}]$ ;
- Spiegelung, falls  $(\forall A \subseteq \Sigma^*) [A \in \mathcal{C} \Rightarrow sp(A) \in \mathcal{C}]$ .

**Satz 2.40** REG ist unter allen in Definition 2.39 angegebenen Operationen abgeschlossen.  
ohne Beweis

Abschlusseigenschaften regulärer Sprachen können genutzt werden, um die Regularität bzw. die Nichtregularität von Sprachen zu zeigen.

### Behauptung 2.41

1. Für jedes  $n \geq 1$  ist die Sprache

$$\begin{aligned} L'_n &= \{x \in \{0, 1\}^* \mid \text{der } n\text{-te Buchstabe in } x \text{ ist } 1\} \\ &= \{x \in \{0, 1\}^* \mid x = u1v \text{ mit } u \in \{0, 1\}^{n-1} \text{ und } v \in \{0, 1\}^*\} \end{aligned}$$

regulär.

2. Die Sprache

$$\{x \in \{0, 1\}^* \mid x \text{ enthält gleich viele 0en und 1en}\}$$

ist nicht regulär.

ohne Beweis

## 2.7 Charakterisierungen regulärer Sprachen

Wir fassen nun die verschiedenen oben gezeigten Charakterisierungen regulärer Sprachen zusammen.

**Korollar 2.42** *Es sei  $L \subseteq \Sigma^*$  eine Sprache. Dann sind die folgenden Aussagen äquivalent:*

1. *Es gibt eine rechtslineare Grammatik  $G$  mit  $L(G) = L$ .*
2. *Es gibt eine linkslineare Grammatik  $G$  mit  $L(G) = L$ .*
3. *Es gibt einen DFA  $M$  mit  $L(M) = L$ .*
4. *Es gibt einen NFA  $M$  mit  $L(M) = L$ .*
5. *Es gibt einen regulären Ausdruck  $\alpha$  mit  $L(\alpha) = L$ .*
6. *Für die Myhill-Nerode-Relation  $R_L$  gilt:  $\text{Index}(R_L) < \infty$ .*

# Kapitel 3

## Kontextfreie Sprachen

**Satz 3.1** REG ist echt in CF enthalten.

**Beweis.** Nach Behauptung 2.29 ist die Sprache  $L = \{a^m b^m \mid m \geq 1\}$  nicht regulär. Andererseits ist  $L$  kontextfrei, wie die einfache kontextfreie Grammatik  $G = (\{a, b\}, \{S\}, S, P)$  mit den Regeln

$$P = \{S \rightarrow aSb \mid ab\}$$

zeigt. Somit ist  $L \in \text{CF} - \text{REG}$ , also  $\text{REG} \subset \text{CF}$ . ■

### 3.1 Normalformen

**Ziel:** Vereinfachung kontextfreier Grammatiken (kurz kfGs).

**Ausnahmeregelung für das leere Wort:** Für kfGs, und nur für diese, erlauben wir  $\lambda$ -Regeln, d.h., Regeln der Form  $A \rightarrow \lambda$ , auch wenn  $A$  nicht das Startsymbol ist.

Dies kann manchmal wünschenswert sein. Für kfGs kann man dies o.B.d.A. zulassen, denn  $\lambda$ -Regeln können ungestraft wieder entfernt werden.

**Definition 3.2 ( $\lambda$ -freie Grammatik)** Eine kfG  $G = (\Sigma, N, S, P)$  heißt  $\lambda$ -frei, falls in  $P$  keine Regel  $A \rightarrow \lambda$  mit  $A \neq S$  auftritt.

**Satz 3.3** Zu jeder kfG  $G$  (mit  $\lambda$ -Regeln) gibt es eine  $\lambda$ -freie kfG  $G'$  mit  $L(G) = L(G')$ .

**Beweis.** Der Beweis beruht auf folgender Konstruktion:

**Gegeben:** kfG  $G = (\Sigma, N, S, P)$  mit  $\lambda$ -Regeln; o.B.d.A. sei  $\lambda \notin L(G)$  (andernfalls ist die Sonderregel für  $\lambda$  anzuwenden).

**Gesucht:**  $\lambda$ -freie kfG  $G'$  mit  $L(G) = L(G')$ .

**Konstruktion:**

1. Bestimme die Menge

$$N_\lambda = \{A \in N \mid A \vdash_G^* \lambda\}$$

sukzessive wie folgt:

- (a) Ist  $A \rightarrow \lambda$  eine Regel in  $P$ , so ist  $A \in N_\lambda$ .
  - (b) Ist  $A \rightarrow A_1 A_2 \cdots A_k$  eine Regel in  $P$  mit  $k \geq 1$  und  $A_i \in N_\lambda$  für alle  $i$ ,  $1 \leq i \leq k$ , so ist  $A \in N_\lambda$ .
2. Füge für jede Regel der Form

$$B \rightarrow uAv \quad \text{mit } B \in N, A \in N_\lambda \text{ und } uv \in (N \cup \Sigma)^+$$

zusätzlich die Regel  $B \rightarrow uv$  zu  $P$  hinzu.

3. Entferne alle Regeln  $A \rightarrow \lambda$  aus  $P$ .

Schritt 2 muss auch für neu generierte Regeln iterativ angewendet werden. Dies ergibt die gesuchte  $\lambda$ -freie kfG  $G'$  mit  $L(G) = L(G')$ . ■

**Beispiel 3.4** Wir betrachten die Grammatik  $G = (\Sigma, N, S, P)$  mit

- dem terminalen Alphabet  $\Sigma = \{0, 1\}$ ,
- dem nichtterminalen Alphabet  $N = \{S, A, B, C, D, E\}$  und
- der Regelmenge

$$P = \left\{ \begin{array}{l} S \rightarrow ABD, \\ A \rightarrow ED \mid BB, \\ B \rightarrow AC \mid \lambda, \\ C \rightarrow \lambda, \\ D \rightarrow 0, \\ E \rightarrow 1 \end{array} \right\}.$$

1. Wir bestimmen für die Grammatik  $G$  nun gemäß dem Beweis von Satz 3.3 die Menge  $N_\lambda$ :

- (a)  $N_\lambda = \{B, C\}$ , da  $B \rightarrow \lambda \in P$  und  $C \rightarrow \lambda \in P$ ;
- (b)  $N_\lambda = \{A, B, C\}$ , da  $A \rightarrow BB \in P$ .

2. Wir entfernen alle Regeln  $A \rightarrow \lambda$ ,  $A \in N$ , aus  $P$  und ergänzen die restlichen Regeln gemäß dem zweiten Schritt der Konstruktion im Beweis von Satz 3.3.

Wir erhalten eine kontextfreie Grammatik  $G' = (\Sigma, \{S, A, B, C, D, E\}, S, P')$  mit

$$P' = \left\{ \begin{array}{l} S \rightarrow ABD \mid BD \mid AD \mid D, \\ A \rightarrow ED \mid BB \mid B, \\ B \rightarrow AC \mid A \mid C, \\ D \rightarrow 0, \\ E \rightarrow 1 \end{array} \right\}$$

und  $L(G') = L(G)$ . (Die Regeln  $B \rightarrow AC$  und  $B \rightarrow C$  sind überflüssig, weil  $C$  auf keiner linken Seite einer Regel steht, und können wieder entfernt werden.)

Nun wollen wir Regeln der Form  $A \rightarrow B$  betrachten. Da hier nur Nichtterminale umbenannt werden, wollen wir solche Regeln vermeiden.

**Definition 3.5 (Einfache Regel)** Regeln  $A \rightarrow B$  heißen einfach, falls  $A$  und  $B$  Nichtterminale sind.

In kontextfreien Grammatiken können einfache Regeln eliminiert werden.

**Satz 3.6** Zu jeder kfG  $G$  gibt es eine kfG  $G'$  ohne einfache Regeln, so dass  $L(G) = L(G')$ .

**Beweis.** Der Beweis beruht auf folgender Konstruktion:

**Gegeben:** kfG  $G = (\Sigma, N, S, P)$  (mit einfachen Regeln).

**Gesucht:** kfG  $G'$  ohne einfache Regeln, so dass  $L(G) = L(G')$ .

**Konstruktion:**

1. Entferne alle Zyklen

$$B_1 \rightarrow B_2, B_2 \rightarrow B_3, \dots, B_{k-1} \rightarrow B_k, B_k \rightarrow B_1 \quad \text{mit } B_i \in N$$

und ersetze alle  $B_i$  (in den verbleibenden Regeln) durch ein neues Nichtterminal  $B$ .

2. Nummeriere die Nichtterminale als  $\{A_1, A_2, \dots, A_n\}$  so, dass aus  $A_i \rightarrow A_j$  folgt:  $i < j$ .
3. Für  $k = n - 1, n - 2, \dots, 1$  (rückwärts!) eliminiere die Regel  $A_k \rightarrow A_\ell$  mit  $k < \ell$  so: Sind die Regeln mit  $A_\ell$  als linker Seite gegeben durch

$$A_\ell \rightarrow u_1 \mid u_2 \mid \dots \mid u_m,$$

so entferne  $A_k \rightarrow A_\ell$  und füge die folgenden Regeln hinzu:

$$A_k \rightarrow u_1 \mid u_2 \mid \dots \mid u_m.$$

Dies liefert die gesuchte kfG  $G'$  ohne einfache Regeln mit  $L(G) = L(G')$ . ■

**Beispiel 3.7 (Entfernen einfacher Regeln)** Betrachte die Grammatik  $G = (\Sigma, N, S, P)$  mit

- dem terminalen Alphabet  $\Sigma = \{0, 1\}$ ,
- dem nichtterminalen Alphabet  $N = \{S, A, B, C, D\}$  und
- der Regelmenge

$$P = \{ \begin{array}{l} S \rightarrow A \mid 0C \mid 00 \mid 0000, \\ A \rightarrow B \mid 11, \\ B \rightarrow C \mid 1, \\ C \rightarrow D, \\ D \rightarrow B \end{array} \}.$$

1. Entferne alle Zyklen über Nichtterminalsymbole gemäß dem Beweis von Satz 3.6.

$B \rightarrow C, C \rightarrow D, D \rightarrow B$  werden entfernt und alle Vorkommen von  $B, C, D$  in den restlichen Regeln werden durch  $B$  ersetzt:

$$P_1 = \{ \begin{array}{l} S \rightarrow A \mid 0B \mid 00 \mid 0000, \\ A \rightarrow B \mid 11, \\ B \rightarrow 1 \end{array} \}.$$

2. Nummeriere die Nichtterminalsymbole:  $S$  (1),  $A$  (2),  $B$  (3).

3. Eliminiere Regeln wie folgt:

- (a)  $A \rightarrow B$  wird entfernt und dafür  $A \rightarrow 1$  hinzugefügt;
- (b)  $S \rightarrow A$  wird entfernt und dafür  $S \rightarrow 11$  und  $S \rightarrow 1$  hinzugefügt:

$$P' = \{ \begin{array}{l} S \rightarrow 1 \mid 11 \mid 0B \mid 00 \mid 0000, \\ A \rightarrow 1 \mid 11, \\ B \rightarrow 1 \end{array} \}.$$

Die so erhaltene Grammatik  $G' = (\Sigma, N', S, P')$  erfüllt  $L(G') = L(G)$ .

**Definition 3.8 (Chomsky-Normalform)** Eine kfG  $G = (\Sigma, N, S, P)$  mit  $\lambda \notin L(G)$  ist in Chomsky-Normalform (kurz CNF), falls alle Regeln in  $P$  eine der folgenden Formen haben:

- $A \rightarrow BC$  mit  $A, B, C \in N$ ;

- $A \rightarrow a$  mit  $A \in N$  und  $a \in \Sigma$ .

**Satz 3.9** Zu jeder kfG  $G$  mit  $\lambda \notin L(G)$  gibt es eine kfG  $G'$  in CNF, so dass  $L(G) = L(G')$ .

**Beweis.** Der Beweis beruht auf folgender Konstruktion:

**Gegeben:**  $\lambda$ -freie kfG  $G = (\Sigma, N, S, P)$  ohne einfache Regeln;  $\lambda \notin L(G)$ .

**Gesucht:** kfG  $G'$  in CNF mit  $L(G) = L(G')$ .

**Konstruktion:**

1. Regeln  $A \rightarrow a$  mit  $A \in N$  und  $a \in \Sigma$  sind in CNF und werden übernommen. Betrachte im Folgenden nur noch die restlichen Regeln; diese sind von der Form:  $A \rightarrow x$  mit  $x \in (N \cup \Sigma)^*$  und  $|x| \geq 2$ .
2. Füge für jedes  $a \in \Sigma$  ein neues Nichtterminal  $B_a$  zu  $N$  hinzu, ersetze jedes Vorkommen von  $a \in \Sigma$  durch  $B_a$  und füge zu  $P$  die Regel  $B_a \rightarrow a$  hinzu.
3. Nicht in CNF sind nun nur noch Regeln der Form

$$A \rightarrow B_1 B_2 \cdots B_k, \quad \text{wobei } k \geq 3 \text{ und jedes } B_i \text{ ein Nichtterminal ist.}$$

Jede solche Regel wird ersetzt durch die Regeln:

$$\begin{aligned} A &\rightarrow B_1 C_2, \\ C_2 &\rightarrow B_2 C_3, \\ &\vdots \\ C_{k-2} &\rightarrow B_{k-2} C_{k-1}, \\ C_{k-1} &\rightarrow B_{k-1} B_k, \end{aligned}$$

wobei  $C_2, C_3, \dots, C_{k-1}$  neue Nichtterminale sind.

Dies liefert die gesuchte Grammatik  $G'$  in CNF mit  $L(G) = L(G')$ . ■

**Beispiel 3.10 (Grammatik in Chomsky-Normalform)** Wir betrachten die transformierte Grammatik aus Beispiel 3.7. O.B.d.A. entfernen wir die Regeln  $A \rightarrow 1 \mid 11$  und das Nichtterminal  $A$  und erhalten die Grammatik  $G = (\Sigma, N, S, P)$  mit

- dem terminalen Alphabet  $\Sigma = \{0, 1\}$ ,
- dem nichtterminalen Alphabet  $N = \{S, B\}$  und

- der Regelmenge

$$P = \{ \begin{array}{l} S \rightarrow 1 \mid 11 \mid 0B \mid 00 \mid 0000, \\ B \rightarrow 1 \end{array} \}.$$

1. Regeln  $A \rightarrow a$  mit  $A \in N$  und  $a \in \Sigma$  sind in CNF und werden übernommen.

Das heißt,  $S \rightarrow 1$  und  $B \rightarrow 1$  werden übernommen.

2. Wir führen zwei neue Nichtterminalsymbole ein,  $X_1$  und  $X_0$ , und erhalten die folgende Regelmenge:

$$P_1 = \{ \begin{array}{l} S \rightarrow 1 \mid X_1X_1 \mid X_0B \mid X_0X_0 \mid X_0X_0X_0X_0, \\ B \rightarrow 1, \\ X_1 \rightarrow 1, \\ X_0 \rightarrow 0 \end{array} \}.$$

3. Noch nicht in Chomsky-Normalform ist die Regel

$$S \rightarrow X_0X_0X_0X_0.$$

Diese Regel ersetzen wir durch die drei Regeln

$$S \rightarrow X_0C_2, \quad C_2 \rightarrow X_0C_3, \quad C_3 \rightarrow X_0X_0,$$

wobei  $C_2$  und  $C_3$  neue Nichtterminalsymbole sind.

Wir erhalten die Grammatik  $G' = (\Sigma, N', S, P')$  mit

- $\Sigma = \{0, 1\}$ ,
- $N' = \{S, B, X_0, X_1, C_2, C_3\}$  und
- Regelmenge

$$P' = \{ \begin{array}{l} S \rightarrow 1 \mid X_1X_1 \mid X_0B \mid X_0X_0 \mid X_0C_2, \\ C_2 \rightarrow X_0C_3, \\ C_3 \rightarrow X_0X_0, \\ B \rightarrow 1, \\ X_1 \rightarrow 1, \\ X_0 \rightarrow 0 \end{array} \}$$

in CNF mit  $L(G) = L(G')$ .



**Bemerkung 3.11** Es sei  $G$  eine Grammatik in Chomsky-Normalform,  $w \in L(G)$  und  $w \neq \lambda$ .

- Jede Ableitung von  $w$  in  $G$  besteht aus genau  $2|w| - 1$  Schritten.  
(Es wird  $(|w| - 1)$ -mal eine Regel der Form  $A \rightarrow BC$  und  $|w|$ -mal eine Regel der Form  $A \rightarrow a$  angewandt.)
- Jeder Syntaxbaum für  $w$  in  $G$  ist ein Binärbaum.

**Definition 3.12 (Greibach-Normalform)** Eine kontextfreie Grammatik  $G = (\Sigma, N, S, P)$  mit  $\lambda \notin L(G)$  ist in Greibach-Normalform (kurz GNF), falls jede Regel in  $P$  die folgende Form hat:

$$A \rightarrow aB_1B_2 \cdots B_k \quad \text{mit } k \geq 0 \text{ und } A, B_i \in N \text{ und } a \in \Sigma.$$

**Satz 3.13** Zu jeder kfG  $G$  mit  $\lambda \notin L(G)$  gibt es eine kfG  $G'$  in GNF, so dass  $L(G) = L(G')$ .  
**ohne Beweis**

**Beispiel 3.14 (Grammatik in Greibach-Normalform)**

1. Die Grammatik  $G = (\Sigma, N, S, P)$  mit

- $\Sigma = \{0, 1\}$ ,
- $N = \{S, A\}$  und
- Regelmenge

$$P = \left\{ \begin{array}{l} S \rightarrow 0A \mid 0SA, \\ A \rightarrow 1 \end{array} \right\}$$

ist offensichtlich in Greibach-Normalform, und es gilt  $L(G) = \{0^n 1^n \mid n \geq 1\}$ .

2. Die Grammatik  $G = (\Sigma, N, S, P)$  mit

- $\Sigma = \{0, 1\}$ ,
- $N = \{S, A, B\}$  und
- Regelmenge

$$P = \left\{ \begin{array}{l} S \rightarrow 0A \mid 1B \mid 0SA \mid 1SB, \\ A \rightarrow 0, \\ B \rightarrow 1 \end{array} \right\}$$

ist offensichtlich in Greibach-Normalform, und es gilt  $L(G) = \{x \operatorname{sp}(x) \mid x \in \{0, 1\}^+\}$ .

**Bemerkung 3.15** Grammatiken in GNF unterscheidet man bezüglich der Längen der rechten Seiten der Produktionen.

- Man kann zeigen, dass sich jede kontextfreie Grammatik in eine äquivalente kontextfreie Grammatik in Greibach-Normalform transformieren lässt, so dass für alle Regeln  $A \rightarrow aB_1B_2 \cdots B_k$  stets  $k \leq 2$  gilt.
- Für den Spezialfall  $k \in \{0, 1\}$  erhalten wir gerade die Definition rechtslinearer Grammatiken.

**Bemerkung 3.16** Es sei  $G$  eine Grammatik in Greibach-Normalform,  $w \in L(G)$  und  $w \neq \lambda$ . Dann besteht jede Ableitung von  $w$  in  $G$  aus genau  $|w|$  Schritten. (Es wird in jedem Ableitungsschritt genau ein Terminalsymbol von  $w$  abgeleitet.)

## 3.2 Das Pumping-Lemma

**Ziel:** Nachweis, dass bestimmte Sprachen nicht kontextfrei sind.

**Satz 3.17 (Pumping-Lemma für kontextfreie Sprachen)** Sei  $L$  eine kontextfreie Sprache. Dann existiert eine (von  $L$  abhängige) Zahl  $n \geq 1$ , so dass sich alle Wörter  $z \in L$  mit  $|z| \geq n$  zerlegen lassen in  $z = uvwxy$ , wobei gilt:

1.  $|vx| \geq 1$ ,
2.  $|vwx| \leq n$ ,
3.  $(\forall i \geq 0) [uv^iwx^iy \in L]$ .

**Beweis.** Es sei  $L$  eine kontextfreie Sprache. Wir setzen voraus, dass  $\lambda \notin L$ . Es sei  $G = (\Sigma, N, S, P)$  eine kontextfreie Grammatik für  $L$  in Chomsky-Normalform mit  $k$  Nichtterminalen. Wir wählen  $n = 2^{k+1}$ .

Sei  $z \in L$  ein beliebiges Wort mit  $|z| \geq n$ . Der Syntaxbaum  $B$  der Ableitung  $S \vdash_G^* z$  ist (bis auf den letzten Ableitungsschritt) ein Binärbaum mit  $|z| \geq n = 2^{k+1}$  Blättern. Benutzt wird dann das folgende Lemma.

**Lemma 3.18** Jeder Binärbaum  $B_k$  mit mindestens  $2^k$  Blättern besitzt mindestens einen Pfad der Länge<sup>1</sup> mindestens  $k$ .

**Beweis von Lemma 3.18.** Der Beweis wird durch Induktion über  $k$  geführt.

**Induktionsanfang:  $k = 0$ .** Jeder Binärbaum  $B_0$  mit mindestens  $2^0 = 1$  Blatt besitzt mindestens einen Pfad der Länge mindestens 0.

---

<sup>1</sup>Die Länge eines Pfades ist die Anzahl seiner Kanten.

**Induktionsschritt:**  $k \mapsto k + 1$ . Die Behauptung gelte für  $k$ . Betrachte einen beliebigen Binärbaum  $B_{k+1}$  mit mindestens  $2^{k+1}$  Blättern. Mindestens einer seiner Teilbäume hat mindestens  $2^{k+1}/2$  Blätter (sonst hätte  $B_{k+1}$  weniger als  $2^{k+1}/2 + 2^{k+1}/2 = 2^{k+1}$  Blätter); sei  $B_k$  dieser Teilbaum. Nach Induktionsvoraussetzung gibt es in  $B_k$  einen Pfad  $\alpha$  der Länge mindestens  $k$ . Verlängert man  $\alpha$  zur Wurzel von  $B_{k+1}$ , so ergibt sich ein Pfad der Länge mindestens  $k + 1$  in  $B_{k+1}$ .

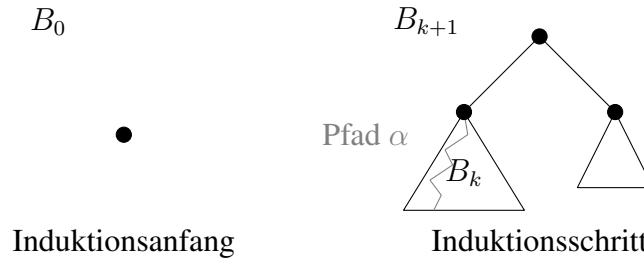


Abbildung 3.1: Beweis von Lemma 3.18

■ Lemma 3.18

Weiter im Beweis von Satz 3.17. Im Syntaxbaum  $B$  der Ableitung  $S \vdash_G^* z$  gibt es nach Lemma 3.18 einen Pfad der Länge mindestens  $k + 1$ . Fixiere einen solchen Pfad  $\alpha$  maximaler Länge.

Nach Wahl von  $n = 2^{k+1}$  muss es wegen  $\|N\| = k$  auf einem solchen Pfad  $\alpha$  maximaler Länge ein Nichtterminal  $A$  geben, das doppelt vorkommt. (Ein Pfad mit Länge  $k + 1$  hat  $k + 2$  Knoten, davon darf der letzte ein Blatt sein. Die restlichen  $k + 1$  Knoten sind mit Nichtterminalen beschriftet.) Diese beiden Vorkommen von  $A$  können so gewählt werden, dass die ersten beiden Eigenschaften von Satz 3.17 erfüllt sind. Dazu bestimmen wir dieses doppelte Vorkommen von  $A$  auf  $\alpha$  von unten nach oben so, dass das obere  $A$  höchstens  $k + 1$  Schritte von der Blattebene entfernt ist. Die Teilbäume unter diesen  $A$  induzieren eine Zerlegung von  $z = uvwxy$ .

Nun verifizieren wir die Aussagen (1), (2) und (3) des Satzes.

- (1) Da  $G$  in CNF ist, muss das obere  $A$  mittels einer Regel  $A \rightarrow BC$  weiter abgeleitet werden. Also ist  $|vx| \geq 1$ .
- (2) Da das obere  $A$  höchstens  $k + 1$  Schritte von der Blattebene entfernt ist, gilt  $|vwx| \leq 2^{k+1} = n$ . Dies folgt wieder aus Lemma 3.18: Hätte der Teilbaum unter dem oberen  $A$  mehr als  $2^{k+1}$  Blätter, so gäbe es unter ihm einen Pfad der Länge  $> k + 1$ , Widerspruch.

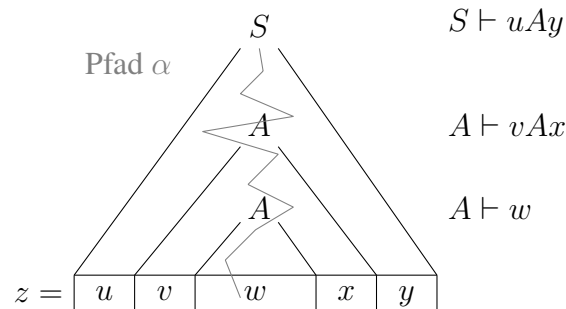


Abbildung 3.2: Auf einem Pfad  $\alpha$  einer Länge mindestens  $k + 1$  muss es ein Nichtterminal  $A$  geben, das doppelt vorkommt.

- (3) Die dritte Eigenschaft folgt daraus, dass stets ein Wort in  $L$  abgeleitet wird, wenn bei der Ableitung von  $z$  die Schritte zwischen den beiden Vorkommen von  $A$  entweder weggelassen ( $i = 0$ ) oder beliebig oft wiederholt ( $i \geq 1$ ) werden. Also gilt:

$$(\forall i \geq 0) [uv^iwx^iy \in L].$$

Satz 3.17 ist bewiesen. ■

**Bemerkung 3.19** Man beachte, dass Satz 3.17 keine Charakterisierung von CF liefert, sondern lediglich eine Implikation:

$$L \in \text{CF} \Rightarrow (\exists n \geq 1) (\forall z \in L, |z| \geq n) (\exists u, v, w, x, y \in \Sigma^*) [z = uvwxy \wedge (1) \wedge (2) \wedge (3)].$$

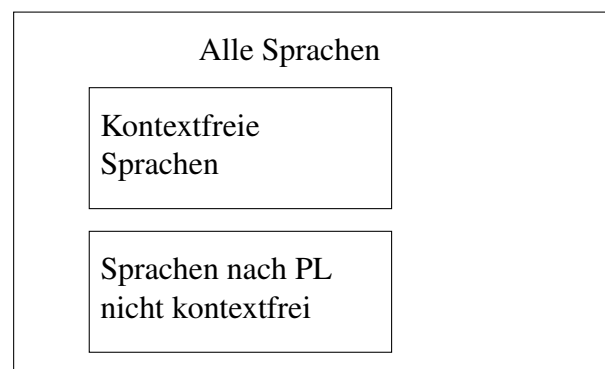


Abbildung 3.3: Anwendbarkeit des Pumping-Lemmas (PL) für kontextfreie Sprachen

## Anwendungsbeispiele

**Behauptung 3.20**  $L = \{a^m b^m c^m \mid m \geq 1\}$  ist nicht kontextfrei.

**Beweis.** Der Beweis wird indirekt geführt. Angenommen,  $L \in \text{CF}$ . Sei  $n$  die Zahl, die nach dem Pumping-Lemma (Satz 3.17) für  $L$  existiert. Betrachte das Wort  $z = a^n b^n c^n$  mit  $|z| = 3n > n$ . Da  $z \in L$ , lässt sich  $z$  so in  $z = uvwxy = a^n b^n c^n$  zerlegen, dass gilt:

1.  $|vwx| \leq n$ , d.h.,  $vx$  kann nicht aus  $a$ -,  $b$ - und  $c$ -Symbolen bestehen ( $vx$  kann also höchstens zwei der drei Symbole  $a$ ,  $b$ ,  $c$  enthalten);
2.  $|vx| \geq 1$ , d.h.,  $vx \neq \lambda$ ;
3.  $(\forall i \geq 0) [uv^i wx^i y \in L]$ .

Insbesondere gilt für  $i = 0$ :

$$uv^0 wx^0 y = uwy \in L,$$

was ein Widerspruch zur Definition von  $L$  ist, denn wegen der oben gezeigten Eigenschaften kann  $uwy$  nicht die Form  $a^m b^m c^m$  haben. Also ist die Annahme falsch und  $L$  nicht kontextfrei. ■

**Behauptung 3.21**  $L = \{0^p \mid p \text{ ist Primzahl}\}$  ist nicht kontextfrei.

**Beweis.** Der Beweis wird indirekt geführt. Angenommen,  $L \in \text{CF}$ . Sei  $n$  die Zahl, die nach dem Pumping-Lemma (Satz 3.17) für  $L$  existiert und  $p \geq n$  eine Primzahl. Betrachte das Wort  $z = 0^p$  mit  $|z| = p \geq n$ . Da  $z \in L$ , lässt sich  $z$  so in  $z = uvwxy = 0^p$  zerlegen, dass gilt:

1.  $|vwx| \leq n$ ,
2.  $|vx| \geq 1$  und
3.  $(\forall i \geq 0) [uv^i wx^i y \in L]$ .

Insbesondere gilt für  $i = p + 1$ :

$$|uv^{p+1} wx^{p+1} y| = |uvwxy| + |v^p| + |x^p| = p + p(|v| + |x|) = p + p(|vx|) = p(1 + |vx|).$$

Da  $|vx| \geq 1$ , ist  $|uv^{p+1} wx^{p+1} y|$  keine Primzahl, was ein Widerspruch zur Definition von  $L$  ist. Also ist die Annahme falsch und  $L$  nicht kontextfrei. ■

**Behauptung 3.22**  $L = \{0^m \mid m \text{ ist Quadratzahl}\}$  ist nicht kontextfrei.

**Beweis.** Siehe Übungen. ■

### 3.3 Der Satz von Parikh

Bisher haben wir uns mit Eigenschaften kontextfreier Grammatiken beschäftigt. Nun betrachten wir kontextfreie Sprachen als Mengen und charakterisieren deren Struktur.

**Definition 3.23 (Parikh-Abbildung für Wörter)** Es sei  $\Sigma = \{a_1, \dots, a_n\}$  ein Alphabet und  $L \subseteq \Sigma^*$  eine Sprache. Die Parikh-Abbildung

$$\Psi : L \rightarrow \mathbb{N}^n$$

für  $w \in L$  ist definiert durch

$$\Psi(w) = (|w|_{a_1}, |w|_{a_2}, \dots, |w|_{a_n}),$$

wobei für  $w \in L$  und  $a \in \Sigma$  der Wert  $|w|_a$  die Anzahl der Vorkommen des Zeichens  $a$  in  $w$  angibt.

**Bemerkung 3.24** Offensichtlich gilt:

- Für  $\Sigma = \{a, b, c, d\}$  ist  $\Psi(abbaac) = (2, 2, 1, 0)$  und  $\Psi(bbab) = (1, 3, 0, 0)$ .
- $\Psi(\lambda) = (\underbrace{0, 0, \dots, 0}_{\|\Sigma\|})$ .
- $\Psi(w_1 w_2) = \Psi(w_1) + \Psi(w_2)$ , unter Verwendung der üblichen Vektoraddition, da das Ergebnis der Parikh-Abbildung unabhängig von der Reihenfolge der Zeichen in Wörtern ist. (Allerdings hängt es von der Reihenfolge der Nummerierung der Zeichen in  $\Sigma$  ab.)

**Definition 3.25 (Parikh-Abbildung für Sprachen)** Für eine Sprache  $L \subseteq \Sigma^*$  mit  $\Sigma = \{a_1, \dots, a_n\}$  ist die Parikh-Abbildung

$$\Psi : \mathfrak{P}(\Sigma^*) \rightarrow \mathfrak{P}(\mathbb{N}^n)$$

definiert durch

$$\Psi(L) = \{\Psi(w) \mid w \in L\}.$$

**Beispiel 3.26** 1. Wir betrachten die folgenden beiden Sprachen über  $\Sigma = \{0, 1\}$ :

$$\begin{aligned} L_1 &= \{0^n 1^n \mid n \geq 0\} \quad \text{und} \\ L_2 &= \{(01)^n \mid n \geq 0\}. \end{aligned}$$

Es gilt  $\Psi(L_1) = \Psi(L_2) = \{(i, i) \mid i \geq 0\}$ . Da  $L_1$  kontextfrei und  $L_2$  regulär ist, unterscheidet die Parikh-Abbildung nicht zwischen kontextfreien und regulären Sprachen.

2. Wir betrachten die beiden Sprachen über  $\Sigma = \{0, 1, 2\}$

$$L_1 = \{0^n 1^n 2^m \mid n, m \geq 0\} \cup \{0^n 1^m 2^m \mid n, m \geq 0\}$$

und

$$L_2 = \{(01)^n 2^m \mid n, m \geq 0\} \cup \{0^n (12)^m \mid n, m \geq 0\}.$$

Es gilt  $\Psi(L_1) = \Psi(L_2) = \{(i, j, k) \mid i = j \text{ oder } j = k\}$ .  $L_1$  ist kontextfrei und  $L_2$  regulär.

**Definition 3.27 (lineare und semilineare Mengen)** Eine Menge  $M$  von Vektoren heißt linear, falls es eine natürliche Zahl  $k$  und Vektoren  $b_0, b_1, \dots, b_k$  gibt, so dass alle Vektoren  $b \in M$  die Form

$$b = b_0 + \sum_{i=1}^k \lambda_i b_i, \quad \lambda_i \in \mathbb{N},$$

haben (und umgekehrt).

Eine Menge  $M$  von Vektoren heißt semilinear, falls  $M$  eine endliche Vereinigung linearer Mengen ist.

**Beispiel 3.28 (lineare und semilineare Mengen)**

- Die Menge  $M_1 = \{(r, s) \mid r = s\}$  ist linear.

Mit  $b_0 = (0, 0)$  und  $b_1 = (1, 1)$  lassen sich alle Vektoren aus  $M_1$  darstellen.

- Die Menge  $M_2 = \{(r, s, t) \mid r = s \text{ oder } s = t\}$  ist die Vereinigung der zwei Mengen

$$\begin{aligned} M_{2,1} &= \{(r, s, t) \mid r = s\} \\ M_{2,2} &= \{(r, s, t) \mid s = t\}. \end{aligned}$$

Beide Mengen sind linear, da sie sich mittels der Vektoren

$$\begin{aligned} b_0 &= (0, 0, 0), & b_1 &= (1, 1, 0), & b_2 &= (0, 0, 1) & \text{bzw.} \\ b'_0 &= (0, 0, 0), & b'_1 &= (0, 1, 1), & b'_2 &= (1, 0, 0) \end{aligned}$$

darstellen lassen. Aus der Darstellung  $M_2 = M_{2,1} \cup M_{2,2}$  folgt, dass  $M_2$  semilinear ist.

Nun geben wir den Satz von Parikh an, dessen relativ umfangreichen Beweis wir hier nicht durchführen wollen.

**Satz 3.29 (Parikh)** Für jede kontextfreie Sprache  $L$  ist  $\Psi(L)$  semilinear. **ohne Beweis**

Der Satz von Parikh liefert somit (wie das Pumping-Lemma für kontextfreie Sprachen) für bestimmte Sprachen einen Nachweis, dass diese nicht kontextfrei sind.

**Behauptung 3.30** Wir betrachten die Sprache

$$L = \{x \in \{0, 1\}^* \mid x = 1^k \text{ mit } k \geq 0 \text{ oder } x = 0^j 1^{k^2} \text{ mit } j \geq 1 \text{ und } k \geq 1\}.$$

über  $\Sigma = \{0, 1\}$  aus Beispiel 2.31.

Die Menge

$$\Psi(L) = \{(i, j) \mid (i = 0 \text{ und } j \geq 0) \text{ oder } (i \geq 1 \text{ und } j = k^2 \text{ und } k \geq 1)\}$$

ist nicht semilinear und  $L$  somit nicht kontextfrei.

**Bemerkung 3.31** Die Umkehrung von Satz 3.29 gilt nicht. Dies zeigt z.B. die Sprache

$$L = \{a^m b^m c^m \mid m \geq 1\},$$

die nach Behauptung 3.20 nicht kontextfrei ist. Das Parikh-Bild

$$\Psi(L) = \{(i, i, i) \mid i \geq 1\}$$

ist jedoch offensichtlich linear und somit auch semilinear.

In Beispiel 3.26 haben wir gesehen, dass es kontextfreie Sprachen gibt, die das gleiche Parikh-Bild haben wie eine reguläre Sprache. Diese Beobachtung gilt sogar immer.

**Satz 3.32** Für jede kontextfreie Sprache  $L$  gibt es eine reguläre Sprache  $L'$  mit  $\Psi(L) = \Psi(L')$ .

**Beweis.** Es sei  $L$  eine kontextfreie Sprache über  $\Sigma = \{a_1, \dots, a_n\}$ . Da nach dem Satz von Parikh  $\Psi(L)$  semilinear ist, gibt es eine endliche Zerlegung von  $\Psi(L)$  in lineare Mengen  $M_1, M_2, \dots, M_\ell$ . Jede dieser Mengen  $M_j$ ,  $1 \leq j \leq \ell$ , ist von der Form

$$M_j = \{b_{j,0} + \sum_{i=1}^k \lambda_{j,i} b_{j,i} \mid \lambda_{j,i} \in \mathbb{N}\}.$$

Die Vektoren seien von der folgenden Form:

$$b_{j,i} = (x_{1,j,i}, \dots, x_{n,j,i}), \quad 0 \leq i \leq k.$$

Definiere für  $1 \leq j \leq \ell$  die regulären Ausdrücke

$$\alpha_j = a_1^{x_{1,j,0}} \dots a_n^{x_{n,j,0}} (a_1^{x_{1,j,1}} \dots a_n^{x_{n,j,1}})^* \dots (a_1^{x_{1,j,k}} \dots a_n^{x_{n,j,k}})^*.$$

Dann erhalten wir mit  $L' = \bigcup_{j=1}^{\ell} L(\alpha_j)$  eine reguläre Sprache mit  $\Psi(L) = \Psi(L')$ . ■



**Beispiel 3.33** Wir betrachten die kontextfreie Sprache

$$L = \{a^n b^n c^m \mid n, m \geq 0\} \cup \{a^n b^m c^m \mid n, m \geq 0\}$$

über  $\Sigma = \{a, b, c\}$ .  $\Psi(L)$  ist die Vereinigung zweier linearer Mengen  $M_1$  und  $M_2$ , die sich durch die Vektoren

$$\begin{aligned} b_{1,0} &= (0, 0, 0), & b_{1,1} &= (1, 1, 0), & b_{1,2} &= (0, 0, 1) & \text{ bzw.} \\ b_{2,0} &= (0, 0, 0), & b_{2,1} &= (0, 1, 1), & b_{2,2} &= (1, 0, 0) \end{aligned}$$

darstellen lassen. Die regulären Ausdrücke  $\alpha_1$  und  $\alpha_2$  aus dem Beweis von Satz 3.32 sind dann:

$$\begin{aligned} \alpha_1 &= a^0 b^0 c^0 (a^1 b^1 c^0)^* (a^0 b^0 c^1)^* \sim (ab)^* c^* \quad \text{und} \\ \alpha_2 &= a^0 b^0 c^0 (a^0 b^1 c^1)^* (a^1 b^0 c^0)^* \sim (bc)^* a^* \end{aligned}$$

Somit ist  $L' = L((ab)^* c^* + (bc)^* a^*)$  eine reguläre Sprache mit  $\Psi(L) = \Psi(L')$ .

Eine sehr interessante Folgerung aus dem Satz von Parikh ist der folgende Satz.

**Satz 3.34** Eine Sprache über  $\Sigma$  mit  $\|\Sigma\| = 1$  ist genau dann kontextfrei, wenn sie regulär ist.

**Beweis.** Offensichtlich sind zwei Sprachen  $L_1, L_2$  über  $\Sigma$  mit  $\|\Sigma\| = 1$  genau dann äquivalent, wenn  $\Psi(L_1) = \Psi(L_2)$ .

Es sei  $L \subseteq \Sigma^*$  mit  $\|\Sigma\| = 1$  eine kontextfreie Sprache. Nach Satz 3.32 existiert eine reguläre Sprache  $L'$  mit  $\Psi(L) = \Psi(L')$ . Nach der Vorbemerkung gilt  $L = L'$  und  $L$  ist regulär. ■

### 3.4 Abschlusseigenschaften kontextfreier Sprachen

**Satz 3.35** CF ist abgeschlossen unter Vereinigung, Konkatenation, Iteration und Spiegelung ist jedoch nicht abgeschlossen unter Schnitt, Komplement und Differenz.

**Beweis.** Seien  $L_1$  und  $L_2$  kontextfreie Sprachen und  $G_i = (\Sigma, N_i, S_i, P_i)$ , wobei  $i \in \{1, 2\}$ , zwei kontextfreie Grammatiken mit  $N_1 \cap N_2 = \emptyset$  und  $L(G_i) = L_i$ . Im Folgenden seien  $S, S' \notin N_1 \cup N_2$  neue Nichtterminale.

- **Vereinigung:** Die Grammatik  $G = (\Sigma, N_1 \cup N_2 \cup \{S\}, S, P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\})$  leistet  $L(G) = L_1 \cup L_2$ .
- **Konkatenation:** Die Grammatik  $G = (\Sigma, N_1 \cup N_2 \cup \{S\}, S, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\})$  leistet  $L(G) = L_1 L_2$ .

- *Iteration*: Die Grammatik  $G = (\Sigma, N_1 \cup \{S, S'\}, S, P)$  mit  $P = (P_1 - \{S_1 \rightarrow \lambda\}) \cup \{S \rightarrow \lambda \mid S', S' \rightarrow S'S' \mid S_1\}$  leistet  $L(G) = (L_1)^*$ .
- *Spiegelung*: Es sei  $G = (\Sigma, N, S, P)$  eine kontextfreie Grammatik in CNF, d.h., alle Regeln in  $G$  sind von der Form  $A \rightarrow BC$  oder  $A \rightarrow a$ , wobei  $A, B, C \in N$  und  $a \in \Sigma$ . Um die gespiegelten Wörter zu erzeugen, drehen wir die rechten Seiten in Regeln der Form  $A \rightarrow BC$  um.

Wir erhalten durch  $G' = (\Sigma, N, S, P')$  mit

$$P' = \{A \rightarrow CB \mid A \rightarrow BC \in P\} \cup \{A \rightarrow a \mid A \rightarrow a \in P\}$$

eine kontextfreie Grammatik in CNF mit  $L(G') = sp(L(G))$ .

- *Schnitt*: Die Sprachen

$$A = \{a^i b^i c^j \mid i, j \geq 1\} \quad \text{und} \quad B = \{a^i b^j c^j \mid i, j \geq 1\}$$

sind beide kontextfrei, jedoch (nach Behauptung 3.20) nicht ihr Durchschnitt

$$A \cap B = \{a^i b^i c^i \mid i \geq 1\}.$$

- *Komplement*: folgt nach de Morgan ( $A \cap B = \overline{\overline{A} \cup \overline{B}}$ ) aus dem Abschluss unter Vereinigung und dem Nicht-Abschluss unter Schnitt.
- *Differenz*: folgt nach  $\overline{A} = \Sigma^* - A$  aus dem Nicht-Abschluss unter Komplement.



In Satz 3.35 wurde bewiesen, dass kontextfreie Sprachen nicht abgeschlossen bezüglich Schnittbildung sind. Der folgende Satz besagt, dass kontextfreie Sprachen abgeschlossen bezüglich Schnittbildung mit regulären Sprachen sind.

**Satz 3.36** *Es seien  $L_1$  eine kontextfreie Sprache und  $L_2$  eine reguläre Sprache. Dann ist  $L_1 \cap L_2$  eine kontextfreie Sprache.*

**Beweis.** Es sei  $G = (\Sigma, N, S, P)$  eine kontextfreie Grammatik in Chomsky-Normalform mit  $L_1 = L(G)$ .  $M = (\Sigma, Z, \delta, z_0, F)$  sei ein DFA mit  $L_2 = L(M)$ . Es sei  $Z = \{z_0, \dots, z_n\}$ .

Wir definieren die Grammatik  $\hat{G} = (\Sigma, \hat{N}, \hat{S}, \hat{P})$  mit

- $\hat{N} = \{A_{ij} \mid \text{für alle } A \in N, 0 \leq i, j \leq n\} \cup \{\hat{S}\}$  und

- der Regelmenge

$$\begin{aligned} \hat{P} = \{ & A_{ij} \rightarrow B_{ik}C_{kj} \quad \text{für alle } A \rightarrow BC \in P, 0 \leq i, j, k \leq n \\ & A_{ij} \rightarrow a \quad \text{für alle } A \rightarrow a \in P, 0 \leq i, j \leq n \\ & \quad \text{und } \delta(z_i, a) = z_j \\ & \hat{S} \rightarrow S_{0j} \quad \text{für alle } z_j \in F \end{aligned} \quad \}$$

Es gilt:  $L(\hat{G}) = L_1 \cap L_2$ .

$\subseteq$  Es sei  $x \in L(\hat{G})$ , d.h.,  $\hat{S} \vdash_{\hat{G}} S_{0j} \vdash_{\hat{G}}^* x$ .

Durch Ignorieren der Indizierung der Nichtterminale in der Ableitung  $S_{0j} \vdash_{\hat{G}}^* x$ , erhalten wir eine Ableitung für  $x$  in der Grammatik  $G$ , also  $x \in L_1$ .

Die Indizierung impliziert, dass  $\hat{\delta}(z_0, x) \in F$ , also  $x \in L_2$ .

$\supseteq$  Nach Konstruktion von  $\hat{P}$ .

■

### 3.5 Der Algorithmus von Cocke, Younger und Kasami

**Ziel:** Ein effizienter Algorithmus für das Wortproblem für kontextfreie Grammatiken.

**Definition 3.37 (Wortproblem)** Für  $i \in \{0, 1, 2, 3\}$  definieren wir das Wortproblem für Typ- $i$ -Grammatiken wie folgt:

$$\text{Wort}_i = \{(G, x) \mid G \text{ ist Typ-}i\text{-Grammatik und } x \in L(G)\}.$$

Wir werden später sehen, dass das Wortproblem für Typ-1-Grammatiken algorithmisch lösbar ist, allerdings nur mit einem exponentiellen Aufwand. Für Typ-2-Grammatiken ist das Wortproblem sogar effizient lösbar, sofern die entsprechende kfG in Chomsky-Normalform gegeben ist. Der Algorithmus von Cocke, Younger und Kasami beruht auf der Idee des dynamischen Programmierens.

Zur Erinnerung: Dynamische Programmierung ist eine algorithmische Umsetzung des *Bellmanschen Optimalitätsprinzips*. Dieses sagt, dass sich die optimale Lösung eines Problems der Größe  $n$  zusammensetzt aus den optimalen Teillösungen der kleineren Teilprobleme. Auch wenn das Wortproblem eigentlich nicht ein Optimierungsproblem ist, lässt sich dieses Prinzip hier anwenden.

**Algorithmenentwurf mit dynamischer Programmierung:**

1. Charakterisiere den Lösungsraum und die Struktur einer erwünschten optimalen Lösung.
2. Definiere rekursiv, wie sich eine optimale Lösung (und der ihr zugeordnete Wert) aus kleineren optimalen Lösungen (und deren Werten) zusammensetzt. Dabei wird das Bellmansche Optimalitätsprinzip angewandt.
3. Konzipiere den Algorithmus bottom-up so, dass für  $n = 1, 2, 3, \dots$  tabellarisch optimale Teillösungen (und deren Werte) gefunden werden. Beim Finden einer optimalen Teillösung  $k > 1$  hilft dabei, dass bereits alle optimalen Teillösungen der Größe  $< k$  bereitstehen.

**Idee:** Gegeben seien eine kfG  $G = (\Sigma, N, S, P)$  in CNF und ein Wort  $x \in \Sigma^*$ .

- Hat  $x = a \in \Sigma$  die Länge 1, so kann es aus einem Nichtterminal  $A$  nur abgeleitet werden, indem eine Regel der Form  $A \rightarrow a$  angewandt wird.
- Ist dagegen  $x = a_1 a_2 \cdots a_n$  ein Wort der Länge  $n \geq 2$ , wobei  $a_i \in \Sigma$ , so ist  $x$  aus  $A$  nur ableitbar, weil zunächst eine Regel der Form  $A \rightarrow BC$  angewandt wurde, aus  $B$  dann das Anfangsstück von  $x$  und aus  $C$  das Endstück von  $x$  abgeleitet wurde.
- Es gibt also ein  $k$  mit  $1 \leq k < n$ , so dass gilt: die Regel  $A \rightarrow BC$  ist in  $P$  und  $B \vdash_G^* a_1 a_2 \cdots a_k$  und  $C \vdash_G^* a_{k+1} a_{k+2} \cdots a_n$ .
- Folglich kann das Wortproblem für ein Wort  $x$  der Länge  $n$  zurückgeführt werden auf die Lösungen des Wortproblems für zwei kleinere Wörter der Länge  $k$  und  $n - k$ . Der Wert von  $k$  steht dabei nicht fest, sondern es ist jeder Wert mit  $1 \leq k < n$  möglich.
- Mit dynamischer Programmierung beginnen wir also bei der Länge 1 und untersuchen systematisch alle Teilwörter von  $x$  auf ihre mögliche Ableitbarkeit aus einem Nichtterminal von  $G$ . All diese Information werden in einer Tabelle gespeichert. Wird nun ein Teilwort der Länge  $m \leq n$  von  $x$  untersucht, so stehen die Teilinformationen über alle kürzeren Teilwörter bereits zur Verfügung.

### Ansatz mit dynamischer Programmierung:

#### **Schritt 1: Lösungsraum und Struktur der Lösung.**

Sei  $G = (\Sigma, N, S, P)$  eine gegebene kfG in CNF, und sei  $x = a_1 a_2 \cdots a_n$  ein gegebenes Wort. Der Algorithmus von Cocke, Younger und Kasami baut eine zweidimensionale Tabelle  $T$  der Größe  $n \times n$  auf, so dass  $T(i, j)$  genau die Nichtterminale  $A$  von  $G$  enthält, so dass

$$A \vdash_G^* a_i a_{i+1} \cdots a_{i+j}$$

gilt. Dabei werden nicht alle Matrixelemente von  $T$  benötigt; es genügt eine untere Dreiecksmatrix.

### Schritt 2: Herleitung der Rekursion.

- $T(i, 0) = \{A \in N \mid A \rightarrow a_i \text{ ist Regel in } P\}$ , für  $1 \leq i \leq n$ .
- Die weiteren Einträge in die Tabelle werden Zeile für Zeile, von unten nach oben, bestimmt. Für  $j = 1, 2, \dots, n-1$  enthält  $T(i, j)$  genau die Nichtterminale  $A \in N$ , für die es eine Regel  $A \rightarrow BC$  in  $P$  und ein  $k \in \{0, 1, \dots, j-1\}$  (also in den darunterliegenden Zeilen von  $T$ ) gibt mit  $B \in T(i, k)$  und  $C \in T(i+k+1, j-k-1)$ . Inhaltlich heißt das, dass es eine Ableitung  $A \vdash_G BC$  gibt sowie darauf folgende Ableitungen  $B \vdash_G^* a_i a_{i+1} \dots a_{i+k}$  und  $C \vdash_G^* a_{i+k+1} a_{i+k+2} \dots a_{i+j}$ , insgesamt also

$$A \vdash_G BC \vdash_G^* a_i a_{i+1} \dots a_{i+k} C \vdash_G^* a_i a_{i+1} \dots a_{i+k} a_{i+k+1} a_{i+k+2} \dots a_{i+j}.$$

- In der obersten Tabellenposition  $T(1, n-1)$  steht nach Definition das Startsymbol  $S$  genau dann, wenn gilt

$$S \vdash_G^* a_1 a_2 \dots a_{1+(n-1)} = a_1 a_2 \dots a_n = x.$$

Also kann man die Entscheidung, ob  $x \in L(G)$ , daran ablesen, ob  $S$  in der Position  $T(1, n-1)$  der Tabelle enthalten ist.

### Schritt 3: Algorithmus in Pseudocode (Abbildung 3.4).

Die Komplexität des Algorithmus von Cocke, Younger und Kasami in Abbildung 3.4 ist wegen der drei ineinander verschachtelten for-Schleifen  $\mathcal{O}(n^3)$ .

**Beispiel 3.38 (Algorithmus von Cocke, Younger und Kasami)** Gegeben sei die Grammatik  $G'$  mit den folgenden drei Regeln:

$$S \rightarrow ab \mid aSb \mid aSbb.$$

Umformen in CNF ergibt zunächst die Grammatik  $G''$  mit den folgenden Regeln:

$$S \rightarrow AB \mid ASB \mid ASBB, \quad A \rightarrow a, \quad B \rightarrow b$$

und schließlich die Grammatik  $G$  in CNF mit den folgenden Regeln:

$$S \rightarrow AB \mid AC \mid DE, \quad C \rightarrow SB, \quad D \rightarrow AS, \quad E \rightarrow BB, \quad A \rightarrow a, \quad B \rightarrow b.$$

Offenbar ist  $L(G) = \{a^m b^n \mid 1 \leq m \leq n < 2m\}$ . Betrachte das Wort  $x = aaabbbb$  in  $\Sigma^*$ , wobei  $\Sigma = \{a, b\}$ . Dieses Wort  $x$  gehört zu  $L(G)$ , und eine Ableitung (nämlich die

```

CYK( $G, x$ ) {
  //  $G = (\Sigma, N, S, P)$  ist kfG in CNF und  $x = a_1 a_2 \cdots a_n$  ein Wort in  $\Sigma^*$ 
  for ( $i = 1, 2, \dots, n$ ) {
     $T(i, 0) = \{A \in N \mid A \rightarrow a_i \text{ ist Regel in } P\}$ ;
  }
  for ( $j = 1, 2, \dots, n - 1$ ) {
    for ( $i = 1, 2, \dots, n - j$ ) {
       $T(i, j) = \emptyset$ ;
      for ( $k = 0, 1, \dots, j - 1$ ) {
         $T(i, j) = T(i, j) \cup \{A \in N \mid \text{es gibt eine Regel } A \rightarrow BC \text{ in } P$ 
          und  $B \in T(i, k) \text{ und } C \in T(i + k + 1, j - k - 1)\}$ ;
      }
    }
  }
  if ( $S \in T(1, n - 1)$ ) return " $x \in L(G)$ ";
  else return " $x \notin L(G)$ ";
}

```

Abbildung 3.4: Der Algorithmus von Cocke, Younger und Kasami

so genannte *Linksableitung*, da stets das am weitesten links stehende Nichtterminal ersetzt wird) ist:

$$\begin{aligned}
 S &\vdash_G DE \vdash_G ASE \vdash_G aSE \vdash_G aACE \vdash_G aaCE \vdash_G aaSBE \vdash_G aaABBE \\
 &\vdash_G aaaBBE \vdash_G aaabBE \vdash_G aaabbE \vdash_G aaabbBB \vdash_G aaabbbB \vdash_G aaabbbb.
 \end{aligned}$$

Der Algorithmus von Cocke, Younger und Kasami in Abbildung 3.4 erzeugt dann Zeile für Zeile, von unten nach oben, die folgende Tabelle:

$i$	1	2	3	4	5	6	7
6	$S, C$						
5	$S, D$	$C$					
4	$D$	$S, C$					
3		$S$					
2		$D$	$C$				
1			$S$	$E$	$E$	$E$	
0	$A$	$A$	$A$	$B$	$B$	$B$	$B$
$j$	$a$	$a$	$a$	$b$	$b$	$b$	$b$

Der Eintrag  $S$  an der Position  $T(1, n - 1)$  signalisiert, dass  $x = aaabbbb$  in  $L(G)$  ist.

Indem man rückverfolgt, weshalb das Startsymbol  $S$  schließlich in die Tabellenposition  $T(1, n - 1)$  kommt, erhält man den Syntaxbaum der entsprechenden Ableitung.

### 3.6 Kellerautomaten

**Ziel:** Automatenmodell zur Charakterisierung von CF. Dazu erweitern wir das NFA-Modell um einen Speicher (Keller bzw. Stack), der nach dem Lifo-Prinzip („Last in – first out“) arbeitet.

**Definition 3.39 (Kellerautomat (PDA))** Ein (nichtdeterministischer) Kellerautomat (kurz PDA für push-down automaton) ist ein 6-Tupel  $M = (\Sigma, \Gamma, Z, \delta, z_0, \#)$ , wobei

- $\Sigma$  das Eingabe-Alphabet ist,
- $\Gamma$  das Kelleralphabet,
- $Z$  eine endliche Menge von Zuständen,
- $\delta : Z \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow \mathfrak{P}_e(Z \times \Gamma^*)$  die Überföhrungsfunktion (hier:  $\mathfrak{P}_e(Z \times \Gamma^*)$  ist die Menge aller endlichen Teilmengen von  $Z \times \Gamma^*$ ),
- $z_0 \in Z$  der Startzustand,
- $\# \in \Gamma$  das Bottom-Symbol im Keller.

$\delta$ -Übergänge  $(z', B_1 B_2 \cdots B_k) \in \delta(z, a, A)$  schreiben wir kurz auch so:

$$zaA \rightarrow z' B_1 B_2 \cdots B_k.$$

#### Arbeitsweise eines PDA

- $zaA \rightarrow z' B_1 B_2 \cdots B_k$  heißt: Wird im Zustand  $z \in Z$  das Eingabesymbol  $a \in \Sigma$  gelesen und ist  $A \in \Gamma$  das Top-Symbol im Keller, so geht  $M$  über in den Zustand  $z'$ , der Lesekopf rückt ein Feld nach rechts auf dem Eingabeband und das Top-Symbol  $A$  im Keller wird ersetzt durch die Kellersymbole  $B_1 B_2 \cdots B_k$ , wobei  $B_1$  das neue Top-Symbol ist.
- POP-Operation: Ist  $k = 0$ , so wird  $A$  aus dem Keller „gePOPt“.
- PUSH-Operation: Ist  $k = 2$  und  $B_1 B_2 = BA$ , so wird  $B$  in den Keller „gePUSHt“.
- $z\lambda A \rightarrow z' B_1 B_2 \cdots B_k$  heißt dasselbe wie oben, nur dass hier kein Eingabesymbol gelesen wird und der Lesekopf stehen bleibt.

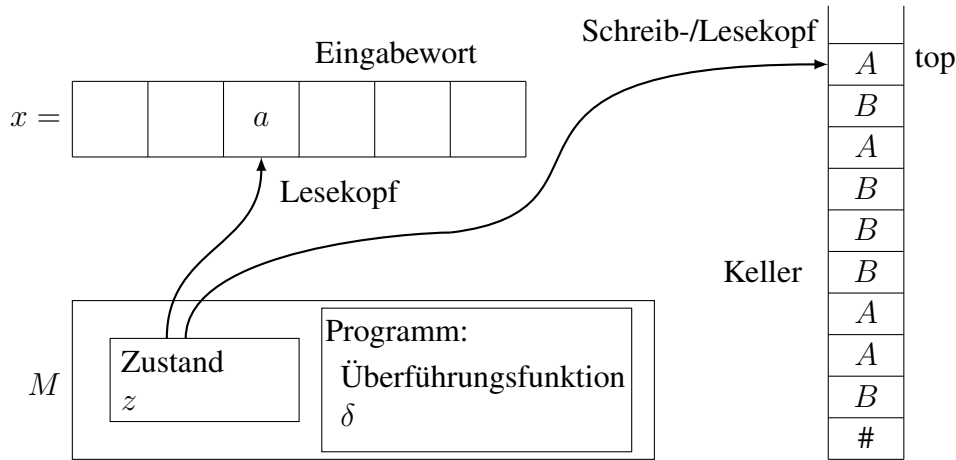


Abbildung 3.5: Arbeitsweise eines Kellerautomaten

## Unterschiede zwischen PDA und NFA

- Akzeptierung erfolgt durch *leeren Keller* statt durch Endzustand (obwohl dies äquivalent auch möglich wäre).
- Takte (d.h. Rechenschritte) ohne Lesen eines Eingabesymbols sind beim PDA mit Regeln der Form  $z\lambda A \rightarrow z'B_1B_2 \cdots B_k$  möglich. (Für NFAs gibt es auch ein äquivalentes Modell mit  $\lambda$ -Übergängen, so genannte  $\lambda$ -NFAs.)
- Daher ist auch nur ein Startzustand nötig (man kann, wenn gewünscht, von diesem in jeden anderen Zustand gelangen, ohne ein Symbol der Eingabe zu verarbeiten).

**Definition 3.40 (Sprache eines Kellerautomaten)** Sei  $M = (\Sigma, \Gamma, Z, \delta, z_0, \#)$  ein PDA.

- $\mathfrak{K}_M = Z \times \Sigma^* \times \Gamma^*$  ist die Menge aller Konfigurationen von  $M$ .
- Ist  $k = (z, \alpha, \gamma)$  eine Konfiguration aus  $\mathfrak{K}_M$ , so ist im aktuellen Takt der Rechnung von  $M$ :
  - $z \in Z$  der aktuelle Zustand von  $M$ ;
  - $\alpha \in \Sigma^*$  der noch zu lesende Teil des Eingabeworts;
  - $\gamma \in \Gamma^*$  der aktuelle Kellerinhalt.

Für jedes Eingabewort  $w \in \Sigma^*$  ist  $(z_0, w, \#)$  die entsprechende Startkonfiguration von  $M$ .

- Auf  $\mathfrak{K}_M$  definieren wir eine binäre Relation  $\vdash_M \subseteq \mathfrak{K}_M \times \mathfrak{K}_M$  wie folgt.



- Intuitiv: Für  $k, k' \in \mathfrak{K}_M$  gilt  $k \vdash_M k'$  genau dann, wenn  $k'$  aus  $k$  durch eine Anwendung von  $\delta$  hervorgeht.
- Formal: Für Zustände  $z, z' \in Z$ , Symbole  $a_1, a_2, \dots, a_n \in \Sigma$  und Kellersymbole  $A_1, A_2, \dots, A_m, B_1, B_2, \dots, B_k \in \Gamma$  ist

$$(z, a_1 a_2 \dots a_n, A_1 A_2 \dots A_m) \vdash_M \begin{cases} (z', a_2 \dots a_n, B_1 B_2 \dots B_k A_2 \dots A_m) & \text{falls } (z', B_1 B_2 \dots B_k) \in \delta(z, a_1, A_1) \\ (z', a_1 \dots a_n, B_1 B_2 \dots B_k A_2 \dots A_m) & \text{falls } (z', B_1 B_2 \dots B_k) \in \delta(z, \lambda, A_1). \end{cases}$$

- Sei  $\vdash_M^*$  die reflexive und transitive Hülle von  $\vdash_M$ , d.h., für  $k, k' \in \mathfrak{K}_M$  gilt  $k \vdash_M^* k'$  genau dann, wenn  $k'$  aus  $k$  durch endlich-fache Anwendung von  $\delta$  hervorgeht.
- Die vom PDA  $M$  akzeptierte Sprache ist definiert durch

$$L(M) = \{w \in \Sigma^* \mid (z_0, w, \#) \vdash_M^* (z, \lambda, \lambda) \text{ für ein } z \in Z\}.$$

Wir bezeichnen für jedes  $z \in Z$  die Konfiguration  $(z, \lambda, \lambda)$  als Endkonfiguration für den PDA  $M$ .

### Beispiel 3.41 (Kellerautomat)

1. Die Sprache  $L = \{a^m b^m \mid m \geq 1\}$  ist kontextfrei. Ein Kellerautomat, der  $L$  erkennt, ist definiert durch

$$M = (\{a, b\}, \{A, \#\}, \{z_0, z_1\}, \delta, z_0, \#),$$

wobei die Überföhrungsfunktion  $\delta$  durch die folgenden Regeln gegeben ist:

$z_0 a \# \rightarrow z_0 A \#$	$z_1 \lambda \# \rightarrow z_1 \lambda$
$z_0 a A \rightarrow z_0 A A$	$z_1 b A \rightarrow z_1 \lambda$
$z_0 b A \rightarrow z_1 \lambda$	

Beispielsweise ist  $aabb \in L(M) = L$ , denn:

$$\begin{aligned} (z_0, aabb, \#) &\vdash_M (z_0, abb, A\#) \\ &\vdash_M (z_0, bb, AA\#) \\ &\vdash_M (z_1, b, A\#) \\ &\vdash_M (z_1, \lambda, \#) \\ &\vdash_M (z_1, \lambda, \lambda) \end{aligned}$$

Andererseits ist  $abb \notin L(M) = L$ , denn:

$$\begin{aligned} (z_0, abb, \#) &\vdash_M (z_0, bb, A\#) \\ &\vdash_M (z_1, b, \#) \\ &\vdash_M (z_1, b, \lambda) \end{aligned}$$

und keine weitere Regel ist anwendbar, da der Keller leer ist, also kein Top-Symbol existiert. Da die Eingabe jedoch nicht vollständig verarbeitet wurde, wird nicht akzeptiert.

Die Überföhrungsfunktion von  $M$  ist sogar „deterministisch“ (die formale Definition kommt später), denn jede Konfiguration hat nur eine mögliche Folgekonfiguration.

2. Der folgende Kellerautomat ist jedoch „echt nichtdeterministisch“:

$$M' = (\{a, b\}, \{A, B, \#\}, \{z_0, z_1\}, \delta, z_0, \#),$$

wobei die Überföhrungsfunktion  $\delta$  durch die folgenden Regeln gegeben ist:

$z_0a\# \rightarrow z_0A\#$	$z_0aA \rightarrow z_1\lambda$
$z_0aA \rightarrow z_0AA$	$z_0bB \rightarrow z_1\lambda$
$z_0aB \rightarrow z_0AB$	$z_0\lambda\# \rightarrow z_1\lambda$
$z_0b\# \rightarrow z_0B\#$	$z_1aA \rightarrow z_1\lambda$
$z_0bA \rightarrow z_0BA$	$z_1bB \rightarrow z_1\lambda$
$z_0bB \rightarrow z_0BB$	$z_1\lambda\# \rightarrow z_1\lambda$

Es gilt  $L(M') = \{w \, sp(w) \mid w \in \{a, b\}^*\}$ , wobei  $sp(w)$  die Spiegelung des Wortes  $w$  ist.

$M'$  hat nichtdeterministische Übergänge:

$$\begin{array}{ccc} & \rightarrow z_0AA & \rightarrow z_0BB \\ z_0aA & & \text{und } z_0bB \\ & \rightarrow z_1\lambda & \rightarrow z_1\lambda \end{array}$$

Drei mögliche Folgen von Konfigurationen bei Eingabe  $abba$  sind:

$$\begin{aligned} &\vdash (z_1, abba, \lambda) \\ (z_0, abba, \#) &\vdash (z_0, bba, A\#) \vdash (z_0, ba, BA\#) \vdash (z_0, a, BBA\#) \vdash (z_0, \lambda, ABBA\#) \\ &\vdash (z_1, a, A\#) \vdash (z_1, \lambda, \#) \vdash (z_1, \lambda, \lambda) \end{aligned}$$

- Der obere Pfad führt zur Nicht-Akzeptierung, da das Eingabeband nicht leer, aber ohne Top-Symbol im leeren Keller keine weitere Regel anwendbar ist.

- Der mittlere Pfad führt zur Nicht-Akzeptierung, da der Keller nicht leer, aber keine weitere Regel anwendbar ist.
- Der untere Pfad führt zur Akzeptierung. ( $M$  wechselt nach Abarbeitung der Hälfte des Wortes in den Zustand  $z_1$  und vergleicht den Kellerinhalt zeichenweise mit der zweiten Hälfte der Eingabe.)

Der Nichtdeterminismus wird hier benötigt, um die Wortmitte zu raten. Es gibt keinen deterministischen PDA, der  $L(M')$  erkennt.

Nun zeigen wir, dass PDA ein geeignetes Automatenmodell zur Charakterisierung der Klasse der kontextfreien Sprachen ist.

**Satz 3.42**  $L$  ist kontextfrei  $\iff$  es gibt einen Kellerautomaten  $M$  mit  $L(M) = L$ .

**Beweis.**

( $\Rightarrow$ ) Sei  $L \in \text{CF}$ , und sei  $G = (\Sigma, N, S, P)$  eine kfG mit  $L(G) = L$ .

Idee: Wir konstruieren einen Kellerautomaten  $M$  mit  $L(M) = L$ . Der Kellerautomat  $M$  simuliert bei Eingabe  $x$  auf dem Keller eine Linksableitung (d.h., es wird stets das am weitesten links stehende Nichtterminal ersetzt) von  $x$  in der Grammatik  $G$ .

$$M = (\Sigma, N \cup \Sigma, \{z\}, \delta, z, S)$$

Die Überföhrungsfunktion  $\delta$  ist wie folgt definiert:

- Ist  $A \rightarrow q$  eine Regel in  $P$  mit  $A \in N$  und  $q \in (N \cup \Sigma)^*$ , so sei  $(z, q) \in \delta(z, \lambda, A)$ .  
Das heit, lsst sich eine Regel der Grammatik auf das Top-Symbol im Keller anwenden, so tue dies, ohne ein Eingabesymbol zu lesen.
- Fr jedes  $a \in \Sigma$  sei  $(z, \lambda) \in \delta(z, a, a)$ .  
Das heit, ist das Top-Symbol im Keller ein Terminalzeichen  $a$ , das auch links in der Eingabe steht, so wird  $a$  aus dem Keller „gePOPt“.

Es gilt fr alle  $x \in \Sigma^*$ :

$$\begin{aligned} x \in L(G) &\iff S \vdash_G^* x \\ &\iff \text{es gibt eine Folge von Konfigurationen von } M \text{ mit} \\ &\quad (z, x, S) \vdash_M \cdots \vdash_M (z, \lambda, \lambda) \\ &\iff (z, x, S) \vdash_M^* (z, \lambda, \lambda) \\ &\iff x \in L(M). \end{aligned}$$

Somit ist  $L(M) = L(G) = L$ .

( $\Leftarrow$ ) Im Beweis der Rückrichtung dieser Äquivalenz wird eine kfG  $G = (\Sigma, N, S, P)$  aus einem gegebenen PDA

$$M = (\Sigma, \Gamma, Z, \delta, z_0, \#)$$

konstruiert. O.B.d.A. gelte  $k \leq 2$  für alle  $\delta$ -Regeln der Form

$$zaA \rightarrow z' B_1 B_2 \cdots B_k.$$

Denn gilt etwa  $k > 2$ , dann wähle neue Zustände  $z_1, z_2, \dots, z_{k-2}$  und ersetze diese  $\delta$ -Regel durch die folgenden neuen  $\delta$ -Regeln:

$$\begin{aligned} zaA &\rightarrow z_1 B_{k-1} B_k \\ z_1 \lambda B_{k-1} &\rightarrow z_2 B_{k-2} B_{k-1} \\ &\vdots \\ z_{k-2} \lambda B_2 &\rightarrow z' B_1 B_2. \end{aligned}$$

Um die Arbeitsweise des PDA  $M$  auf einem Wort  $x$  mittels einer Grammatik  $G$  zu simulieren, verwenden wir in  $G$  Variablen (Nichtterminale), die (bis auf das Startsymbol  $S$ ) Tripel aus  $Z \times \Gamma \times Z$  sind, d.h.,

$$N = \{S\} \cup Z \times \Gamma \times Z.$$

Ist etwa  $(z_\ell, \gamma, z_r) \in N$  mit  $z_\ell, z_r \in Z$  und  $\gamma \in \Gamma$ , so ist

- $z_\ell \in Z$  der Zustand vor einer Folge von Rechenschritten des PDA  $M$ ,
- $\gamma \in \Gamma$  das dabei verarbeitete Kellersymbol,
- $z_r \in Z$  der Zustand, wenn  $\gamma$  aus dem Keller „gePOPt“ wird.

Aus der Variablen  $(z_\ell, \gamma, z_r)$  sollen alle Zeichenreihen erzeugt werden können, die bewirken, dass  $\gamma$  vom Stack entfernt wird, während der Automat vom Zustand  $z_\ell$  in den Zustand  $z_r$  übergeht. Das heißt, wir simulieren Zustands- und Kelleränderung von  $M$  in den Nichtterminalen von  $G$ .

$P$  besteht aus genau den folgenden Regeln:

1.  $S \rightarrow (z_0, \#, z)$  für jedes  $z \in Z$ .

Vom Startsymbol in  $G$  wollen wir alle Nichtterminale erreichen, die bedeuten, dass der PDA  $M$  seinen Keller leert, wenn er mit der Startkonfiguration beginnt. (Der dabei erreichte Zustand  $z$  ist beliebig.)

2.  $(z, A, z') \rightarrow a$ , falls  $(z', \lambda) \in \delta(z, a, A)$ .

Falls der Kellerautomat die Möglichkeit hat, im Zustand  $z$  das Zeichen  $a$  aus der Eingabe zu lesen,  $A$  aus dem Keller zu entfernen und in Zustand  $z'$  überzugehen, so realisieren wir dies durch die Regel  $(z, A, z') \rightarrow a$  in der Grammatik.

3.  $(z, A, z') \rightarrow a(z_1, B, z')$ , falls  $(z_1, B) \in \delta(z, a, A)$ .

Falls der Kellerautomat die Möglichkeit hat, im Zustand  $z$  das Zeichen  $a$  aus der Eingabe zu lesen,  $A$  aus dem Keller zu entfernen,  $B$  auf den Keller zu schreiben und in Zustand  $z_1$  überzugehen, so realisieren wir dies durch die Regeln  $(z, A, z') \rightarrow a(z_1, B, z')$  in der Grammatik.

4.  $(z, A, z') \rightarrow a(z_1, B, z_2)(z_2, C, z')$ , falls  $(z_1, BC) \in \delta(z, a, A)$ .

Falls der Kellerautomat die Möglichkeit hat, im Zustand  $z$  das Zeichen  $a$  aus der Eingabe zu lesen,  $A$  aus dem Keller zu entfernen,  $BC$  auf den Keller zu schreiben und in Zustand  $z_1$  überzugehen, so realisieren wir dies durch die Regel  $(z, A, z') \rightarrow a(z_1, B, z_2)(z_2, C, z')$  in der Grammatik.

Hierbei ist stets  $z, z', z_1, z_2 \in Z$ ,  $A, B, C \in \Gamma$  und  $a \in \Sigma \cup \{\lambda\}$ .

Die Regeln von  $G$  sind so beschaffen, dass eine Rechnung von  $M$  bei Eingabe  $x$  durch eine Linksableitung von  $x$  simuliert wird. Die Korrektheit der Regeln liefert das folgende Lemma:

**Lemma 3.43** Für alle  $(z, A, z') \in N$  und alle  $x \in \Sigma^*$  gilt:

$$(z, A, z') \vdash_G^* x \iff (z, x, A) \vdash_M^* (z', \lambda, \lambda).$$

**Beweis von Lemma 3.43.** ( $\Leftarrow$ ) Der Beweis wird durch Induktion über die Anzahl  $n$  der Rechenschritte von  $M$  geführt.

**Induktionsanfang:  $n = 1$ .** Es gilt

$$\begin{aligned} (z, A, z') \vdash_G a &\iff (z, A, z') \rightarrow a \text{ ist Regel in } P \\ &\iff (z', \lambda) \in \delta(z, a, A) \quad \text{wegen (2) in der Konstruktion von } P \\ &\iff (z, a, A) \vdash_M (z', \lambda, \lambda). \end{aligned}$$

**Induktionsschritt:  $(n - 1) \mapsto n$ .** Sei  $n > 1$ , und die Behauptung gelte für  $n - 1$ . Folglich ist  $x = ay$  mit  $a \in \Sigma \cup \{\lambda\}$ , und es gilt:

$$(z, ay, A) \vdash_M (z_1, y, \alpha) \vdash_M^{n-1} (z', \lambda, \lambda)$$

für einen geeigneten Zustand  $z_1 \in Z$  und Kellerinhalt  $\alpha \in \{\lambda, B, BC\}$ . Unterscheide die folgenden drei Fälle.

**Fall 1:  $\alpha = \lambda$ .** Dieser Fall kann nicht eintreten, da  $(z_1, y, \lambda)$  keine Folgekonfiguration hat.

**Fall 2:**  $\alpha = B$ . Nach Induktionsvoraussetzung gilt:

$$(z_1, y, \alpha) \vdash_M^{n-1} (z', \lambda, \lambda),$$

was  $(z_1, B, z') \vdash_G^* y$  impliziert. Außerdem muss es wegen

$$(z, ay, A) \vdash_M (z_1, y, \alpha)$$

eine  $\delta$ -Regel der Form  $(z_1, B) \in \delta(z, a, A)$  geben. Nach (3) in der Konstruktion von  $P$  gibt es in  $P$  also eine Regel der Form

$$(z, A, z') \rightarrow a(z_1, B, z').$$

Somit ergibt sich insgesamt:

$$(z, A, z') \vdash_G a(z_1, B, z') \vdash_G^* ay = x,$$

also  $(z, A, z') \vdash_G^* x$ .

**Fall 3:**  $\alpha = BC$ . Die Konfigurationsfolge

$$(z_1, y, BC) \vdash_M^* (z', \lambda, \lambda)$$

kann in zwei Teile zerlegt werden:

$$(z_1, y, BC) \vdash_M^* (z_2, y_2, C) \vdash_M^* (z', \lambda, \lambda),$$

wobei  $y = y_1 y_2$  für ein geeignetes  $y_1$  und  $(z_2, y_2, C)$  der erste Zustand ist, wo  $C$  auf dem Stapel liegt. Für dieses  $y_1$  gilt:

$$(z_1, y_1, B) \vdash_M^* (z_2, \lambda, \lambda).$$

Außerdem muss es wegen des ersten Schritts

$$(z, ay, A) \vdash_M (z_1, y, BC)$$

eine  $\delta$ -Regel der Form  $(z_1, BC) \in \delta(z, a, A)$  geben. Nach (4) in der Konstruktion von  $P$  gibt es in  $P$  also eine Regel der Form

$$(z, A, z') \rightarrow a(z_1, B, z_2)(z_2, C, z').$$

Insgesamt ergibt sich:

$$\begin{aligned} (z, A, z') &\vdash_G a(z_1, B, z_2)(z_2, C, z') \\ &\vdash_G^* ay_1(z_2, C, z') \\ &\vdash_G^* ay_1 y_2 = ay = x, \end{aligned}$$

also  $(z, A, z') \vdash_G^* x$ .

( $\Rightarrow$ ) Der Beweis wird durch Induktion über  $k$  geführt, die Länge der Linksableitung von  $x$ .

**Induktionsanfang:**  $k = 1$ . Siehe den Induktionsanfang ( $n = 1$ ) im Beweis von ( $\Leftarrow$ ) von Lemma 3.43.

**Induktionsschritt:**  $(k - 1) \mapsto k$ . Sei  $k > 1$ , und die Behauptung gelte für  $k - 1$ . Wir unterscheiden wieder drei Fälle.

**Fall 1:**  $(z, A, z') \vdash_G a \vdash_G^* x$ . Dann gilt  $x = a$ , im Widerspruch zu  $k > 1$ .

**Fall 2:**  $(z, A, z') \vdash_G a(z_1, B, z') \vdash_G^{k-1} ay = x$ .

Dann ist  $(z_1, B) \in \delta(z, a, A)$ . Nach Induktionsvoraussetzung gilt:

$$(z_1, y, B) \vdash_M^* (z', \lambda, \lambda).$$

Somit ergibt sich:

$$(z, x, A) = (z, ay, A) \vdash_M (z_1, y, B) \vdash_M^* (z', \lambda, \lambda).$$

**Fall 3:**  $(z, A, z') \vdash_G a(z_1, B, z_2)(z_2, C, z') \vdash_G^{k-1} ay = x$ .

Dann ist  $(z_1, BC) \in \delta(z, a, A)$ . Nach Induktionsvoraussetzung gilt für  $y = y_1y_2$ :

$$\begin{aligned} (z_1, y_1, B) &\vdash_M^* (z_2, \lambda, \lambda); \\ (z_2, y_2, C) &\vdash_M^* (z', \lambda, \lambda). \end{aligned}$$

Somit ergibt sich:

$$\begin{aligned} (z, x, A) = (z, ay_1y_2, A) &\vdash_M (z_1, y_1y_2, BC) \\ &\vdash_M^* (z_2, y_2, C) \\ &\vdash_M^* (z', \lambda, \lambda). \end{aligned}$$

Das Lemma ist bewiesen. ■

Lemma 3.43

Aus Lemma 3.43 ergibt sich  $L(G) = L(M)$  so:

$$\begin{aligned} x \in L(M) &\iff (z_0, x, \#) \vdash_M^* (z, \lambda, \lambda) \text{ für ein } z \in Z \\ &\iff (z_0, \#, z) \vdash_G^* x \text{ für ein } z \in Z, \quad \text{nach Lemma 3.43} \\ &\iff S \vdash_G^* x \quad \text{wegen (1) in der Konstruktion von } P \\ &\iff x \in L(G). \end{aligned}$$

Satz 3.42 ist bewiesen. ■

Wir veranschaulichen die beiden Beweisrichtungen jeweils durch ein Beispiel.

**Beispiel 3.44 (kontextfreie Grammatik  $\Rightarrow$  Kellerautomat)** Wir betrachten die Grammatik  $G = (\Sigma, N, S, R)$  aus Beispiel 1.5(3) mit

- $\Sigma = \{*, +, (, ), a\}$ ,
- $N = \{S\}$  und
- Regeln

$$R = \{S \rightarrow S + S \mid S * S \mid (S) \mid a\}.$$

$G$  erzeugt die Sprache aller verschachtelten Klammerausdrücke mit den Operationen  $+$  und  $*$  und einem Zeichen  $a$ .

Konstruiere einen PDA  $M$  mit  $L(M) = L(G)$  wie folgt:

$$M = (\Sigma, N \cup \Sigma, \{z\}, \delta, z, S)$$

mit der folgenden Überföhrungsfunktion  $\delta$ :

$z\lambda S \rightarrow zS + S$	$z(( \rightarrow z\lambda$
$z\lambda S \rightarrow zS * S$	$z)) \rightarrow z\lambda$
$z\lambda S \rightarrow z(S)$	$z * * \rightarrow z\lambda$
$z\lambda S \rightarrow za$	$z + + \rightarrow z\lambda$
	$zaa \rightarrow z\lambda$

Eine Ableitung von  $w = (a * a) + a$  in  $G$  und  $M$ :

$kfG \ G$	$PDA \ M$
$S \vdash_G S + S$	$(z, (a * a) + a, S) \vdash_M (z, (a * a) + a, S + S)$
$\vdash_G (S) + S$	$\vdash_M (z, (a * a) + a, (S) + S)$
	$\vdash_M (z, a * a) + a, S) + S)$
$\vdash_G (S * S) + S$	$\vdash_M (z, a * a) + a, S * S) + S)$
$\vdash_G (a * S) + S$	$\vdash_M (z, a * a) + a, a * S) + S)$
	$\vdash_M (z, *a) + a, *S) + S)$
	$\vdash_M (z, a) + a, S) + S)$
$\vdash_G (a * a) + S$	$\vdash_M (z, a) + a, a) + S)$
	$\vdash_M (z, ) + a, ) + S)$
	$\vdash_M (z, +a, +S)$
	$\vdash_M (z, a, S)$
$\vdash_G (a * a) + a$	$\vdash_M (z, a, a)$
	$\vdash_M (z, \lambda, \lambda)$



**Beispiel 3.45 (Kellerautomat  $\Rightarrow$  kontextfreie Grammatik)** Wir betrachten den Kellerautomaten

$$M = (\Sigma, \Gamma, Z, \delta, z_0, \#)$$

mit

- Eingabealphabet  $\Sigma = \{a, b\}$ ,
- Kelleralphabet  $\Gamma = \{A, B, \#\}$ ,
- Zustandsmenge  $Z = \{z_0, z_1\}$  und
- Überföhrungsfunktion  $\delta$ , die durch die folgenden Regeln gegeben ist:

$z_0 a \# \rightarrow z_0 A \#$	$z_0 a A \rightarrow z_1 \lambda$
$z_0 a A \rightarrow z_0 A A$	$z_0 b B \rightarrow z_1 \lambda$
$z_0 a B \rightarrow z_0 A B$	$z_0 \lambda \# \rightarrow z_1 \lambda$
$z_0 b \# \rightarrow z_0 B \#$	$z_1 a A \rightarrow z_1 \lambda$
$z_0 b A \rightarrow z_0 B A$	$z_1 b B \rightarrow z_1 \lambda$
$z_0 b B \rightarrow z_0 B B$	$z_1 \lambda \# \rightarrow z_1 \lambda$

Es gilt  $L(M) = \{w \text{ sp}(w) \mid w \in \{a, b\}^*\}$ .

Konstruiere eine Grammatik  $G = (\Sigma, \{S\} \cup Z \times \Gamma \times Z, S, P)$  mit  $L(G) = L(M)$  wie folgt:

$P$  besteht aus genau den folgenden Regeln:

1.  $S \rightarrow (z_0, \#, z)$  für jedes  $z \in Z$ ; d.h.,  
 $S \rightarrow (z_0, \#, z_0)$   
 $S \rightarrow (z_0, \#, z_1)$
2.  $(z, A, z') \rightarrow a$ , falls  $(z', \lambda) \in \delta(z, a, A)$ ; d.h.,  
 $(z_0, A, z_1) \rightarrow a$ , da  $(z_1, \lambda) \in \delta(z_0, a, A)$   
 $(z_0, B, z_1) \rightarrow b$ , da  $(z_1, \lambda) \in \delta(z_0, b, B)$   
 $(z_0, \#, z_1) \rightarrow \lambda$ , da  $(z_1, \lambda) \in \delta(z_0, \lambda, \#)$   
 $(z_1, A, z_1) \rightarrow a$ , da  $(z_1, \lambda) \in \delta(z_1, a, A)$   
 $(z_1, B, z_1) \rightarrow b$ , da  $(z_1, \lambda) \in \delta(z_1, b, B)$   
 $(z_1, \#, z_1) \rightarrow \lambda$ , da  $(z_1, \lambda) \in \delta(z_1, \lambda, \#)$
3.  $(z, A, z') \rightarrow a(z_1, B, z')$ , falls  $(z_1, B) \in \delta(z, a, A)$ ; d.h.,  
hier keine

4.  $(z, A, z') \rightarrow a(z_1, B, z_2)(z_2, C, z')$ , falls  $(z_1, BC) \in \delta(z, a, A)$ ; d.h.,
- $(z_0, \#, z') \rightarrow a(z_0, A, z_2)(z_2, \#, z')$ ,  $z', z_2 \in \{z_0, z_1\}$ , da  $(z_0, A\#) \in \delta(z_0, a, \#)$
  - $(z_0, A, z') \rightarrow a(z_0, A, z_2)(z_2, A, z')$ ,  $z', z_2 \in \{z_0, z_1\}$ , da  $(z_0, AA) \in \delta(z_0, a, A)$
  - $(z_0, B, z') \rightarrow a(z_0, A, z_2)(z_2, B, z')$ ,  $z', z_2 \in \{z_0, z_1\}$ , da  $(z_0, AB) \in \delta(z_0, a, B)$
  - $(z_0, \#, z') \rightarrow b(z_0, B, z_2)(z_2, \#, z')$ ,  $z', z_2 \in \{z_0, z_1\}$ , da  $(z_0, B\#) \in \delta(z_0, b, \#)$
  - $(z_0, A, z') \rightarrow b(z_0, B, z_2)(z_2, A, z')$ ,  $z', z_2 \in \{z_0, z_1\}$ , da  $(z_0, BA) \in \delta(z_0, b, A)$
  - $(z_0, B, z') \rightarrow b(z_0, B, z_2)(z_2, B, z')$ ,  $z', z_2 \in \{z_0, z_1\}$ , da  $(z_0, BB) \in \delta(z_0, b, B)$
- Da  $z', z_2 \in \{z_0, z_1\}$ , sind hier  $6 \cdot 4 = 24$  Regeln angegeben.

Eine Ableitung von  $w = abba$  in  $M$  und  $G$ :

<i>PDA M</i>	<i>kfG G</i>
	$S \vdash_G (z_0, \#, z_1)$
$(z_0, abba, \#) \vdash_M (z_0, bba, A\#)$	$\vdash_G a(z_0, A, z_1)(z_1, \#, z_1)$
$\vdash_M (z_0, ba, BA\#)$	$\vdash_G ab(z_0, B, z_1)(z_1, A, z_1)(z_1, \#, z_1)$
$\vdash_M (z_1, a, A\#)$	$\vdash_G abb(z_1, A, z_1)(z_1, \#, z_1)$
$\vdash_M (z_1, \lambda, \#)$	$\vdash_G abba(z_1, \#, z_1)$
$\vdash_M (z_1, \lambda, \lambda)$	$\vdash_G abba$

# Kapitel 4

## Deterministisch kontextfreie Sprachen

### 4.1 Deterministische Kellerautomaten

**Definition 4.1 (Deterministischer Kellerautomat (DPDA))** Ein Septupel

$$M = (\Sigma, \Gamma, Z, \delta, z_0, \#, F)$$

heißt deterministischer Kellerautomat (kurz DPDA), falls gilt:

1.  $M' = (\Sigma, \Gamma, Z, \delta, z_0, \#)$  ist ein PDA;
2.  $(\forall a \in \Sigma) (\forall A \in \Gamma) (\forall z \in Z) [\|\delta(z, a, A)\| + \|\delta(z, \lambda, A)\| \leq 1]$ ;
3.  $F \subseteq Z$  ist eine ausgezeichnete Teilmenge von Endzuständen ( $M$  akzeptiert per Endzustand, nicht per leerem Keller).

**Beispiel 4.2 (Deterministischer Kellerautomat)**

1. Der PDA  $M$  aus Beispiel 3.41(1) hat nur deterministische  $\delta$ -Übergänge, ist aber aufgrund der Akzeptanz mittels leerem Keller nach obiger Definition kein DPDA. Um einen deterministischen Kellerautomaten  $M_e$  für die Sprache  $L = \{a^m b^m \mid m \geq 1\}$  zu definieren, müssen wir noch einen Endzustand  $z_e$  einführen:

$$M_e = (\{a, b\}, \{A, \#\}, \{z_0, z_1, z_e\}, \delta_e, z_0, \#, \{z_e\})$$

mit der Überföhrungsfunktion  $\delta_e$ :

$z_0 a \# \rightarrow z_0 A \#$	$z_1 \lambda \# \rightarrow z_e \lambda$
$z_0 a A \rightarrow z_0 A A$	$z_1 b A \rightarrow z_1 \lambda$
$z_0 b A \rightarrow z_1 \lambda$	

Beispielsweise ist  $aabb \in L(M_e) = L$ , denn:

$$\begin{aligned}
 (z_0, aabb, \#) &\vdash_{M_e} (z_0, abb, A\#) \\
 &\vdash_{M_e} (z_0, bb, AA\#) \\
 &\vdash_{M_e} (z_1, b, A\#) \\
 &\vdash_{M_e} (z_1, \lambda, \#) \\
 &\vdash_{M_e} (z_e, \lambda, \lambda)
 \end{aligned}$$

2. Der PDA  $M'$  aus Beispiel 3.41(2) ist kein DPDA, da  $M'$  nichtdeterministische  $\delta$ -Übergänge hat:

- $\delta(z_0, a, A) = \{(z_0, AA), (z_1, \lambda)\}$ , also  $\|\delta(z_0, a, A)\| = 2 > 1$ , und
- $\delta(z_0, b, B) = \{(z_0, BB), (z_1, \lambda)\}$ , also  $\|\delta(z_0, b, B)\| = 2 > 1$ .

**Definition 4.3 (Sprache eines deterministischen Kellerautomaten)** Die von einem deterministischen Kellerautomaten  $M = (\Sigma, \Gamma, Z, \delta, z_0, \#, F)$  akzeptierte Sprache ist definiert durch

$$L(M) = \{x \in \Sigma^* \mid (z_0, x, \#) \vdash_M^* (z, \lambda, \gamma) \text{ für ein } z \in F \text{ und } \gamma \in \Gamma^*\}.$$

Für jedes Eingabewort  $w \in \Sigma^*$  ist  $(z_0, w, \#)$  die entsprechende Startkonfiguration von  $M$ . Für jedes  $\gamma \in \Gamma^*$  und  $z \in F$  ist die Konfiguration  $(z, \lambda, \gamma)$  eine Endkonfiguration des DPDA  $M$ .

**Definition 4.4 (Deterministisch kontextfreie Sprache)** Eine Sprache  $A$  heißt deterministisch kontextfrei, falls es einen deterministischen Kellerautomaten  $M$  gibt mit

$$A = L(M).$$

Mit  $\text{DCF} = \{L(M) \mid M \text{ ist DPDA}\}$  bezeichnen wir die Klasse der deterministisch kontextfreien Sprachen.

**Bemerkung 4.5** • Im Gegensatz zu (nichtdeterministischen) Kellerautomaten macht es bei DPDA einen Unterschied, ob man sie mittels Akzeptierung per Endzustand oder mittels Akzeptierung per leerem Keller definiert.

- DCF stimmt genau mit der Klasse der so genannten  $\text{LR}(k)$ -Sprachen überein, die bei der Syntaxanalyse im Compilerbau eine Rolle spielen.
- Für deterministisch kontextfreie Sprachen ist das Wortproblem in linearer Zeit lösbar.

**Satz 4.6**  $\text{REG} \subset \text{DCF} \subset \text{CF}$ .

**Beweis.** Die Inklusionen  $\text{REG} \subseteq \text{DCF} \subseteq \text{CF}$  sind klar. Die Ungleichheiten werden nicht bewiesen, sondern nur skizziert:

- Die Sprache  $L = \{w \$ sp(w) \mid w \in \{a, b\}^*\}$  ist deterministisch kontextfrei. Andererseits kann man mit dem Pumping-Lemma für reguläre Sprachen zeigen, dass  $L \notin \text{REG}$ .
- Die Sprache  $L' = \{w sp(w) \mid w \in \{a, b\}^*\}$  aus Beispiel 3.41(2) ist jedoch *nicht* deterministisch kontextfrei. Andererseits ist  $L'$  in CF.

Daraus folgt  $\text{REG} \neq \text{DCF} \neq \text{CF}$ , also  $\text{REG} \subset \text{DCF} \subset \text{CF}$ . ■

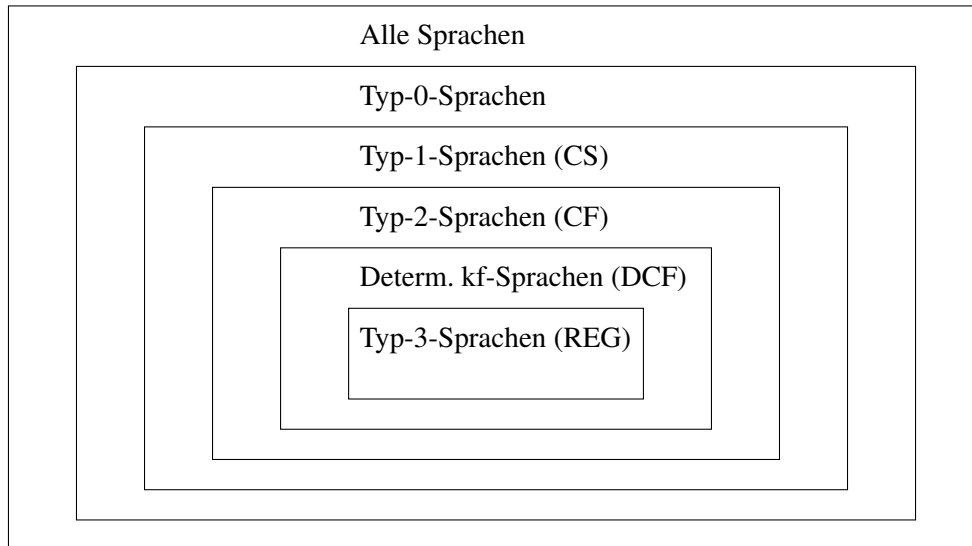


Abbildung 4.1: Einordnung von DCF in die Chomsky-Hierarchie

**Satz 4.7** DCF ist abgeschlossen unter Komplement, aber weder unter Schnitt noch unter Vereinigung noch unter Konkatenation noch unter Iteration. **ohne Beweis**

**Behauptung 4.8** Die Sprache

$$\begin{aligned} L &= \{a^n b^n c^m \mid n, m \geq 0\} \cup \{a^n b^m c^m \mid n, m \geq 0\} \\ &= \{a^i b^j c^k \mid i = j \text{ oder } j = k\} \end{aligned}$$

ist kontextfrei und inhärent mehrdeutig.

**ohne Beweis**

## 4.2 LR( $k$ )- und LL( $k$ )-Grammatiken

**Ziel:** Kontextfreie Sprachen, die eindeutige (d.h. nicht mehrdeutige) Sprachen definieren und Grammatiken für deterministisch kontextfreie Sprachen.

Zunächst definieren wir spezielle Ableitungsrelationen (vgl. Definition 1.4):

**Definition 4.9 (Rechtsableitung, Linksableitung)** Sei  $G = (\Sigma, N, S, P)$  eine kontextfreie Grammatik. Wir definieren

$$u \vdash_{G,r} v \iff u = xpz, v = xqz,$$

wobei  $x \in (N \cup \Sigma)^*$  und  $p \rightarrow q \in P$  und  $z \in \Sigma^*$  und

$$u \vdash_{G,l} v \iff u = xpz, v = xqz,$$

wobei  $z \in (N \cup \Sigma)^*$  und  $p \rightarrow q \in P$  und  $x \in \Sigma^*$ .

Eine Folge  $(x_0, x_1, \dots, x_n)$  mit  $x_i \in (\Sigma \cup N)^*$ ,  $x_0 = S$  und  $x_n \in \Sigma^*$  heißt Rechtsableitung bzw. Linksableitung von  $x_n$  in  $G$ , falls

$$x_0 \vdash_{G,r} x_1 \vdash_{G,r} \dots \vdash_{G,r} x_n$$

bzw.

$$x_0 \vdash_{G,l} x_1 \vdash_{G,l} \dots \vdash_{G,l} x_n.$$

Analog zu Definition 1.4 verwenden wir  $\vdash_{G,r}^n$  bzw.  $\vdash_{G,r}^*$  und  $\vdash_{G,l}^n$  bzw.  $\vdash_{G,l}^*$ .

**Beispiel 4.10 (Rechtsableitung, Linksableitung)** Wir betrachten die kontextfreie Grammatik  $G = (\{*, +, (, ), a\}, \{S\}, S, \{S \rightarrow S + S \mid S * S \mid (S) \mid a\})$  aus Beispiel 1.5.

- Eine Linksableitung für  $(a + a) * a + a$  in  $G$ :

$$\begin{aligned} S &\vdash_{G,l} S + S \\ &\vdash_{G,l} S * S + S \\ &\vdash_{G,l} (S) * S + S \\ &\vdash_{G,l} (S + S) * S + S \\ &\vdash_{G,l} (a + S) * S + S \\ &\vdash_{G,l} (a + a) * S + S \\ &\vdash_{G,l} (a + a) * a + S \\ &\vdash_{G,l} (a + a) * a + a \end{aligned}$$

- Eine Rechtsableitung für  $(a + a) * a + a$  in  $G$ :

$$\begin{array}{l}
S \vdash_{G,r} S + S \\
\vdash_{G,r} S + a \\
\vdash_{G,r} S * S + a \\
\vdash_{G,r} S * a + a \\
\vdash_{G,r} (S) * a + a \\
\vdash_{G,r} (S + S) * a + a \\
\vdash_{G,r} (S + a) * a + a \\
\vdash_{G,r} (a + a) * a + a
\end{array}$$

**Definition 4.11** ( $\text{FIRST}_k$ ) Seien  $\Sigma$  ein Alphabet und  $k \geq 0$ . Definiere die Funktion

$$\text{FIRST}_k : \Sigma^* \rightarrow \Sigma^*$$

durch

$$\text{FIRST}_k(x) = \begin{cases} x & \text{falls } |x| \leq k \\ y & \text{falls } x = yz, |y| = k, \text{ für } y, z \in \Sigma^*. \end{cases}$$

Das heißt,  $\text{FIRST}_k(x)$  ist das Wort  $x$ , falls  $|x| \leq k$ , bzw. das Anfangswort von  $x$  der Länge  $k$ , falls  $|x| \geq k$ .

**Beispiel 4.12** ( $\text{FIRST}_k$ ) Es seien  $\Sigma = \{0, 1\}$  und  $x = 0110111 \in \Sigma^*$ .

$k$	0	1	2	3	4	5	6	7	8
$\text{FIRST}_k(x)$	$\lambda$	0	01	011	0110	01101	011011	0110111	0110111

## LR( $k$ )-Grammatiken

Um deterministisch kontextfreie Sprachen durch Grammatiken zu charakterisieren, betrachten wir LR( $k$ )-Grammatiken.

**Definition 4.13** (**LR( $k$ )-Grammatik**) Es sei  $k \geq 0$ . Eine kontextfreie Grammatik  $G = (\Sigma, N, S, P)$  heißt LR( $k$ )-Grammatik, falls die Ableitung  $S \vdash_G^n S$  für kein  $n > 0$  möglich ist und für alle Wörter  $P_1, P'_1, P_2, P'_2 \in (\Sigma \cup N)^*$ , alle Wörter  $P_3, P'_3 \in \Sigma^*$  und alle  $X, X' \in N$  aus den Bedingungen

1.  $S \vdash_{G,r}^* P_1 X P_3 \vdash_{G,r} P_1 P_2 P_3$
2.  $S \vdash_{G,r}^* P'_1 X' P'_3 \vdash_{G,r} P'_1 P'_2 P'_3$
3.  $\text{FIRST}_{|P_1|+|P_2|+k}(P_1 P_2 P_3) = \text{FIRST}_{|P'_1|+|P'_2|+k}(P'_1 P'_2 P'_3)$

folgt, dass  $P_1 = P'_1$ ,  $X = X'$  und  $P_2 = P'_2$  gilt.

In  $LR(k)$ -Grammatiken muss man also höchstens  $k$  Stellen von Links in das terminale Wort  $P_3$  hineinschauen, um die angewandte Produktion  $X \rightarrow \dots$  bei einer Rechtsableitung eindeutig zu erkennen. Die Zahl  $k$  wird auch als *Look-ahead* (Vorausblick) bezeichnet.

**Beispiel 4.14 (LR( $k$ )-Grammatik)** Wir betrachten Grammatiken für die Sprache  $L = \{ab^{2n+1}c \mid n \geq 0\}$ .

1.  $G_1 = (\{a, b, c\}, \{S, X\}, S, \{S \rightarrow aXc, X \rightarrow bXb \mid b\})$ .

Betrachte z.B. die Ableitungen

$$S \vdash_{G,r}^{k+1} ab^k X b^k c \quad \vdash_{G,r} ab^{2k+1} c$$

$$S \vdash_{G,r}^{k+2} ab^{k+1} X b^{k+1} c \quad \vdash_{G,r} ab^{2k+3} c$$

Die drei Bedingungen der Definition sind erfüllt für:

- (a)  $P_1 = ab^k$ ,  $P_2 = b$ ,  $P_3 = b^k c$ .
- (b)  $P'_1 = ab^{k+1}$ ,  $P'_2 = b$ ,  $P'_3 = b^{k+1} c$ .
- (c)  $\text{FIRST}_{|P_1|+|P_2|+k}(P_1 P_2 P_3) = \text{FIRST}_{2k+2}(ab^{2k+1} c) = ab^{2k+1}$  und  
 $\text{FIRST}_{|P'_1|+|P'_2|+k}(P'_1 P'_2 P'_3) = \text{FIRST}_{2k+2}(ab^{2k+3} c) = ab^{2k+1}$ .

Aber  $P_1 = P'_1$  gilt für kein  $k$ . Also ist  $G_1$  keine  $LR(k)$ -Grammatik.

2.  $G_2 = (\{a, b, c\}, \{S, X\}, S, \{S \rightarrow aXc, X \rightarrow Xbb \mid b\})$ .

Betrachte z.B. die Ableitungen

$$S \vdash_{G,r}^{k+1} aXb^{2k} c \quad \vdash_{G,r} ab^{2k+1} c$$

$$S \vdash_{G,r}^{k+2} aXb^{2k+2} c \quad \vdash_{G,r} ab^{2k+3} c$$

Die drei Bedingungen der Definition sind erfüllt für:

- (a)  $P_1 = a$ ,  $P_2 = b$ ,  $P_3 = b^{2k} c$ .
- (b)  $P'_1 = a$ ,  $P'_2 = b$ ,  $P'_3 = b^{2k+2} c$ .
- (c)  $\text{FIRST}_{|P_1|+|P_2|+k}(P_1 P_2 P_3) = \text{FIRST}_2(ab^{2k+1} c) = ab$  für  $k = 0$  und  
 $\text{FIRST}_{|P'_1|+|P'_2|+k}(P'_1 P'_2 P'_3) = \text{FIRST}_2(ab^{2k+3} c) = ab$  für  $k = 0$ .



Es gilt  $P_1 = P'_1$ ,  $X = X'$  und  $P_2 = P'_2$ .

Allgemein kann man zeigen:  $G_2$  ist eine  $LR(0)$ -Grammatik.

3.  $G_3 = (\{a, b, c\}, \{S, X\}, S, \{S \rightarrow aXc, X \rightarrow bbX \mid b\})$

$G_3$  ist keine  $LR(0)$ -Grammatik.

Betrachte z.B. die Ableitungen

$$S \vdash_{G,r}^1 aXc \quad \vdash_{G,r} abc$$

$$S \vdash_{G,r}^2 ab^2Xc \quad \vdash_{G,r} ab^3c$$

$G_3$  ist aber eine  $LR(1)$ -Grammatik (s. Übungen).

**Definition 4.15 (LR(k)-Sprache)** Eine Sprache  $L$  ist eine  $LR(k)$ -Sprache, falls es eine  $LR(k)$ -Grammatik  $G$  gibt mit  $L(G) = L$ . Es sei  $LR(k)$  die Klasse aller  $LR(k)$ -Sprachen.

**Beispiel 4.16 (LR(k)-Sprache)**

1.  $L = \{ab^{2n+1}c \mid n \geq 0\}$  ist eine  $LR(0)$ -Sprache (s. Beispiel 4.14(2)).
2.  $L = \{w\$sp(w) \mid w \in \{a, b\}^*\}$  ist eine  $LR(0)$ -Sprache (s. Übungen).

Den Zusammenhang zwischen  $LR(0)$ -Sprachen und deterministisch kontextfreien Sprachen fassen wir hier ohne Beweise zusammen.

**Satz 4.17**

1. Jede  $LR(k)$ -Grammatik ist eindeutig (d.h. nicht mehrdeutig).
2. Jede  $LR(k)$ -Sprache ist deterministisch kontextfrei.
3. Jede deterministisch kontextfreie Sprache kann von einer  $LR(1)$ -Grammatik erzeugt werden.
4. Es gilt die Inklusionskette

$$LR(0) \subset LR(1) = LR(2) = \dots = LR(k) = DCF \subset CF.$$

## LL( $k$ )-Grammatiken

Eng mit LR( $k$ )-Grammatiken verwandt sind die LL( $k$ )-Grammatiken, die im Compilerbau eine wichtige Rolle spielen.

**Definition 4.18 (LL( $k$ )-Grammatik)** Es sei  $k \geq 0$ . Eine kontextfreie Grammatik  $G = (\Sigma, N, S, P)$  heißt LL( $k$ )-Grammatik, falls die Ableitung  $S \vdash_G^n S$  für kein  $n > 0$  möglich ist und für alle Wörter  $P_2, P'_2, P_3, P'_3 \in (\Sigma \cup N)^*$ , alle Wörter  $P_1, P_4, P'_4 \in \Sigma^*$  und alle  $X \in N$  aus den Bedingungen

1.  $S \vdash_{G,l}^* P_1 X P_3 \vdash_{G,l} P_1 P_2 P_3 \vdash_{G,l}^* P_1 P_4$
2.  $S \vdash_{G,l}^* P_1 X P'_3 \vdash_{G,l} P_1 P'_2 P'_3 \vdash_{G,l}^* P_1 P'_4$
3.  $\text{FIRST}_k(P_4) = \text{FIRST}_k(P'_4)$

folgt, dass  $P_2 = P'_2$  gilt.

In LL( $k$ )-Grammatiken muss man also höchstens  $k$  Stellen von Links in das terminale Wort  $P_4$  hineinschauen, um die angewandte Produktion  $X \rightarrow \dots$  bei einer Linksableitung eindeutig zu erkennen.

**Definition 4.19 (LL( $k$ )-Sprache)** Eine Sprache  $L$  ist eine LL( $k$ )-Sprache, falls es eine LL( $k$ )-Grammatik  $G$  gibt mit  $L(G) = L$ . Es sei LL( $k$ ) die Klasse aller LL( $k$ )-Sprachen.

### Definition 4.20 (Starke LL( $k$ )-Grammatik)

- Eine kfG  $G = (\Sigma, N, S, P)$  heißt starke LL( $k$ )-Grammatik, falls für je zwei Linksableitungen

$$S \vdash_{G,l}^* \begin{cases} wA\gamma \vdash_{G,l} w\alpha\gamma \vdash_{G,l}^* wy \\ xA\delta \vdash_{G,l} x\beta\delta \vdash_{G,l}^* xz \end{cases} \quad \text{mit } w, x, y, z \in \Sigma^*, A \in N, \alpha, \beta, \gamma, \delta \in (N \cup \Sigma)^*$$

aus  $\text{FIRST}_k(y) = \text{FIRST}_k(z)$  stets folgt:  $\alpha = \beta$ .

- $G$  heißt starke LL-Grammatik, falls  $G$  starke LL( $k$ )-Grammatik für ein  $k$  ist.
- Eine Sprache  $L$  heißt stark LL( $k$ ) (bzw. stark LL), falls es eine starke LL( $k$ )-Grammatik (bzw. eine starke LL-Grammatik)  $G$  gibt mit  $L(G) = L$ .

**Bemerkung 4.21** Nach Definition ist jede starke LL( $k$ )-Grammatik auch eine LL( $k$ )-Grammatik. Für  $k > 1$  gilt die Umkehrung i.A. nicht, aber für  $k = 1$  sind beide Begriffe äquivalent.

Es ist einfach zu entscheiden, ob eine Grammatik eine LL(1)-Grammatik ist.

**Satz 4.22** Eine kfG  $G$  ist genau dann eine LL(1)-Grammatik, wenn  $G$  eine starke LL(1)-Grammatik ist.

**Beweis.** ( $\Leftarrow$ ) Dies gilt nach Definition.

( $\Rightarrow$ ) Für einen Widerspruch nehmen wir an, dass die kfG  $G = (\Sigma, N, S, P)$  ein Gegenbeispiel für die Aussage sei, also ist  $G$  zwar LL(1), aber nicht stark LL(1). Dann gibt es in  $P$  zwei Regeln  $A \rightarrow \alpha$  und  $A \rightarrow \beta$  mit  $\alpha \neq \beta$ , und es gibt zwei Linksableitungen

$$S \vdash_{G,l}^* \begin{cases} wA\gamma \vdash_{G,l} w\alpha\gamma \vdash_{G,l}^* wy_1\gamma \vdash_{G,l}^* wy_1y_2 \in \Sigma^* \\ xA\delta \vdash_{G,l} x\beta\delta \vdash_{G,l}^* xz_1\delta \vdash_{G,l}^* xz_1z_2 \in \Sigma^*, \end{cases}$$

so dass

$$\text{FIRST}_1(y_1y_2) = \text{FIRST}_1(z_1z_2). \quad (4.1)$$

Kann  $G$  dann wirklich LL(1) sein?

**Fall 1:**  $y_1 = z_1 = \lambda$ . Dann gibt es Linksableitungen

$$S \vdash_{G,l}^* wA\gamma \vdash_{G,l} \begin{cases} w\alpha\gamma \vdash_{G,l}^* w\gamma \vdash_{G,l}^* wy_2 \\ w\beta\gamma \vdash_{G,l}^* w\gamma \vdash_{G,l}^* wy_2. \end{cases}$$

Da natürlich  $\text{FIRST}_1(y_2) = \text{FIRST}_1(y_2)$  gilt, würde aus der Annahme, dass  $G$  LL(1) ist,  $\alpha = \beta$  folgen, im Widerspruch zu  $\alpha \neq \beta$ .

( $G$  ist in diesem Fall nicht nur nicht LL(1), sondern sogar mehrdeutig.)

**Fall 2:**  $y_1 \neq \lambda$  oder  $z_1 \neq \lambda$ . Sei o.B.d.A.  $y_1 \neq \lambda$ . Aus der Gleichheit (4.1) folgt dann:

$$\text{FIRST}_1(y_1y_2) = \text{FIRST}_1(y_1) = \text{FIRST}_1(z_1z_2).$$

Daher gibt es die folgenden beiden Linksableitungen:

$$S \vdash_{G,l}^* xA\delta \vdash_{G,l} \begin{cases} x\alpha\delta \vdash_{G,l}^* xy_1\delta \vdash_{G,l}^* xy_1z_2 \\ x\beta\delta \vdash_{G,l}^* xz_1\delta \vdash_{G,l}^* xz_1z_2. \end{cases}$$

Nun gilt:

$$\text{FIRST}_1(y_1z_2) = \text{FIRST}_1(y_1) = \text{FIRST}_1(z_1z_2),$$

aber  $\alpha \neq \beta$ , ein Widerspruch dazu, dass  $G$  eine LL(1)-Grammatik ist.

Satz 4.22 ist bewiesen. ■

**Beispiel 4.23 (LL( $k$ )-Grammatik)** Die Grammatiken aus Beispiel 4.14 sind keine LL(1)-Grammatiken.

Erinnerung an Definition 1.9: Eine kfG  $G$  heißt *mehrdeutig*, falls es ein Wort  $w$  in  $L(G)$  gibt, das auf verschiedene Weise abgeleitet werden kann, also zwei verschiedene Syntaxbäume hat. Äquivalent dazu kann man fordern, dass  $w$  verschiedene Linksableitungen hat.  $LL(k)$ -Grammatiken erzwingen die Eindeutigkeit der Ableitungen. Somit erzwingt die  $LL(k)$ -Bedingung die Nichtmehrdeutigkeit der Grammatik.

Einige Aussagen zu  $LL(k)$ -Grammatiken fassen wir hier ohne Beweise zusammen.

#### Satz 4.24

1. Jede  $LL(k)$ -Grammatik ist eindeutig (d.h. nicht mehrdeutig).
2. Jede  $LL(k)$ -Grammatik ist eine  $LR(k)$ -Grammatik.
3. Es gilt eine echte Inklusionskette

$$LL(0) \subset LL(1) \subset LL(2) \subset \dots \subset LL(k) \subset LR(1) = DCF \subset CF.$$

**Bemerkung 4.25**  $RL(k)$ -Grammatiken und  $RR(k)$ -Grammatiken gibt es auch.

Wir wollen nun noch ein Kriterium für die  $LL(k)$ -Bedingung angeben. Dazu brauchen wir die folgende Definition.

**Definition 4.26** Sei  $G = (\Sigma, N, S, P)$  eine kfG.

- Für jedes  $\alpha \in (N \cup \Sigma)^*$  definieren wir

$$\text{First}_k(\alpha) = \left\{ x \in \Sigma^* \left| \begin{array}{l} \text{es gibt ein } y \in \Sigma^* \text{ und eine Linksableitung } \alpha \vdash_{G,l}^* xy \\ \text{mit } |x| = k \text{ oder } (|x| < k \text{ und } y = \lambda) \end{array} \right. \right\}.$$

- $A \in N$  heißt *nützlich* in  $G$ , falls es Wörter  $\alpha, \beta \in (\Sigma \cup N)^*$  und  $w \in \Sigma^*$  gibt mit

$$S \vdash_G^* \alpha A \beta \vdash_G^* w.$$

- $G$  heißt *reduziert*, falls jedes  $A \in N$  nützlich in  $G$  ist.

**Lemma 4.27** Zu jeder kfG  $G$  gibt es eine reduzierte kfG  $G'$  mit  $L(G) = L(G')$ . **ohne Beweis**

**Satz 4.28** Eine kfG  $G = (\Sigma, N, S, P)$  ist genau dann eine  $LL(k)$ -Grammatik, wenn für alle  $\alpha \in (\Sigma \cup N)^*$ ,  $A \in N$ ,  $w \in \Sigma^*$  mit  $S \vdash_{G,l}^* wA\alpha$  und für alle Regeln  $A \rightarrow \beta$  und  $A \rightarrow \gamma$  in  $P$  mit  $\beta \neq \gamma$  gilt:

$$\text{First}_k(\beta\alpha) \cap \text{First}_k(\gamma\alpha) = \emptyset.$$

**Beweis. ( $\Rightarrow$ )** Wir zeigen die Kontraposition. Sei  $\text{First}_k(\beta\alpha) \cap \text{First}_k(\gamma\alpha) \neq \emptyset$ . Betrachte ein  $x$  in diesem Durchschnitt. Nach Lemma 4.27 können wir annehmen, dass  $G$  reduziert ist. Nach Definition 4.26 gibt es Linksableitungen

$$S \vdash_{G,l}^* wA\alpha \vdash_{G,l} \begin{cases} w\beta\alpha \vdash_{G,l}^* wxy \in \Sigma^* \\ w\gamma\alpha \vdash_{G,l}^* wxz \in \Sigma^* \end{cases}$$

Falls  $|x| < k$ , dann ist  $y = \lambda = z$ .

Wegen  $\beta \neq \gamma$  und  $\text{FIRST}_k(x) = \text{FIRST}_k(x)$  ist  $G$  keine  $\text{LL}(k)$ -Grammatik.

( $\Leftarrow$ ) Wieder zeigen wir die Kontraposition. Angenommen,  $G$  ist keine  $\text{LL}(k)$ -Grammatik. Dann existieren Linksableitungen

$$S \vdash_{G,l}^* wA\alpha \vdash_{G,l} \begin{cases} w\beta\alpha \vdash_{G,l}^* wx \in \Sigma^* \\ w\gamma\alpha \vdash_{G,l}^* wy \in \Sigma^* \end{cases}$$

mit  $\text{FIRST}_k(x) = \text{FIRST}_k(y)$  und  $\beta \neq \gamma$ . Also gibt es in  $P$  verschiedene Regeln  $A \rightarrow \beta$  und  $A \rightarrow \gamma$  mit

$$\text{FIRST}_k(x) = \text{FIRST}_k(y) \in \text{First}_k(\beta\alpha) \cap \text{First}_k(\gamma\alpha).$$

Folglich ist die Schnittmenge  $\text{First}_k(\beta\alpha) \cap \text{First}_k(\gamma\alpha)$  nicht leer. ■

**Korollar 4.29** Sei  $G = (\Sigma, N, S, P)$  eine kfG ohne Regeln der Form  $A \rightarrow \lambda$ . Dann ist  $G$  eine  $\text{LL}(1)$ -Grammatik genau dann, wenn für alle Regeln  $A \rightarrow \beta$  und  $A \rightarrow \gamma$  in  $P$  mit  $\beta \neq \gamma$  gilt:

$$\text{First}_1(\beta) \cap \text{First}_1(\gamma) = \emptyset.$$

ohne Beweis

Für kfGs mit  $\lambda$ -Regeln ist die Situation komplizierter; dies sei einer Spezialvorlesung zum Compilerbau bzw. zur Syntaxanalyse vorbehalten.

## 4.3 Anwendung: Syntaxanalyse durch $\text{LL}(k)$ -Parser

**Ziel der Syntaxanalyse:** Programmiersprachen sind meist kontextfreie Sprachen, deren Wörter die syntaktisch korrekten Programme sind. Für eine durch eine kfG  $G$  gegebene Programmiersprache  $L = L(G)$  und ein gegebenes Programm  $w$  soll getestet werden, ob  $w$  syntaktisch korrekt ist, d.h., ob gilt:  $w \in L(G)$ . Das ist gerade das Wortproblem.

- Falls ja: Angabe eines Syntaxbaums der Ableitung  $S \vdash_G^* w$ .
- Falls nein: Angabe möglichst vieler Syntaxfehler.

Wir beschränken uns auf den Fall „ja“.

### **Aufgaben eines Compilers (Übersetzers):**

Übersetzen eines Programms aus einer höheren Programmiersprache in Maschinensprache.

1. Die *lexikalische Analyse* erkennt mittels endlicher Automaten und bestimmter Hashing-Techniken Schlüsselwörter (*while*, *for*, *if*, *then*, etc.) und vereinbarte Namen.
2. Die *Syntaxanalyse* überprüft die syntaktische Korrektheit des Programmtextes.
3. Die *semantische Analyse* überprüft die „semantische“ Korrektheit des Programms.
4. Die *Code-Erzeugung* (Assembler Code, Maschinensprache) und *Optimierung* ermöglichen die Synthese des erfolgreich kompilierten Programms.

Der Compilerbau und insbesondere auch die Syntaxanalyse sind umfangreiche Gebiete, die Inhalt einer selbstständigen Vorlesung sind. Hier beschränken wir uns auf einige Teilaspekte, um die allgemeinen Prinzipien und Strategien kurz vorzustellen.

### **Strategien der Syntaxanalyse (Parsing):**

- allgemeine Verfahren für bestimmte Teilklassen;
- Tabellenstrategien (z.B. der Algorithmus von Cocke, Younger und Kasami);
- ableitungsorientierte Strategien.

Unter den vielen Strategien für die Syntaxanalyse interessieren wir uns nur für ein spezielles Top-down-Verfahren, das v.l.n.r. vorgeht: das  $LL(k)$ -Parsing.

### **Grundidee bei ableitungsorientierten Strategien:**

- Sei  $G = (\Sigma, N, S, P)$  eine kfG,  $w \in L(G)$  sei ein gegebenes Wort (d.h. Programm), und  $T$  sei ein Syntaxbaum für die Ableitung  $S \vdash_G^* w$ .
- Betrachte eine Verzweigung in  $T$ , die durch Anwendung der Regel

$$A \rightarrow \alpha = a_1 a_2 \cdots a_n$$

entsteht, wobei  $A \in N$  und jedes  $a_i \in \Sigma \cup N$ . Abbildung 4.2 zeigt die Struktur des Syntaxbaums  $T$  beim Parsing. Dabei ist  $w = xyz$ , wobei  $x$  der terminale Linkskontext,  $y$  der terminale Nachkontext und  $z$  der terminale Rechtskontext von  $A$  sind.

- Die wesentlichen Strategien sind:

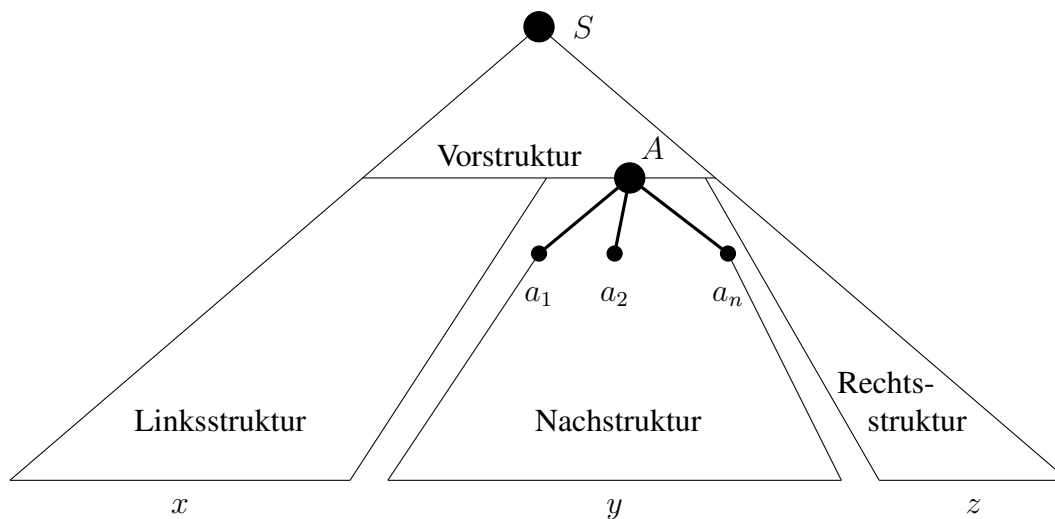
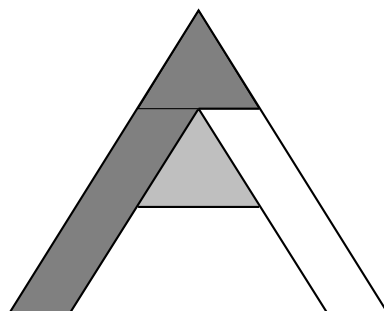


Abbildung 4.2: Struktur eines Syntaxbaums beim Parsing

**Top-down-Analyse:** Fortschreiten, wenn die Vorstruktur bekannt und die Nachstruktur unbekannt ist, d.h. von oben nach unten. Dabei geht man von links nach rechts vor, erweitert also einen schon bekannten Präfix des abzuleitenden Wortes.

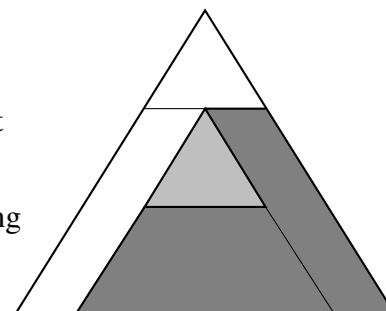
**Bottom-up-Analyse:** Fortschreiten, wenn die Nachstruktur bekannt und die Vorstruktur unbekannt ist, d.h. von unten nach oben. Dabei geht man von rechts nach links vor, erweitert also einen schon bekannten Suffix des abzuleitenden Wortes.

Top-down von links nach rechts



Präfix erweitern

Bottom-up von rechts nach links



Suffix erweitern

Abbildung 4.3: Top-down-Strategie und Bottom-up-Strategie

Abbildung 4.3 veranschaulicht beide Strategien.

Will man z.B. bei einem Top-down-Verfahren einen schon erkannten Präfix erweitern und dazu die richtige anzuwendende Regel erkennen, kann es nötig sein, einige Symbole vorausblicken zu können.

**Beispiel 4.30 (Look-ahead)** Betrachte das Wort  $w = abba\$$  und die kfG  $G$  mit den folgenden Regeln:

$$S \rightarrow C\$, \quad C \rightarrow bA \mid aB, \quad A \rightarrow a \mid aC, \quad B \rightarrow b \mid bC,$$

wobei das Terminalzeichen  $\$$  als eine Endmarke dient. Die einzige mögliche Startmöglichkeit für eine Ableitung  $S \vdash_{G,l}^* w$  ist die Regel  $S \rightarrow C\$$ . Danach gibt es zwei Möglichkeiten,  $C$  zu ersetzen:  $C \rightarrow bA$  und  $C \rightarrow aB$ . Die erste Möglichkeit entfällt aber, da der erste Buchstabe von  $w$  ein  $a$  ist; also muss  $C \rightarrow aB$  angewandt werden. Es reicht also ein Look-ahead von einem Symbol aus, um die richtige Entscheidung zu treffen. Bisher gilt:

$$S \vdash_{G,l} C\$ \vdash_{G,l} aB\$.$$

Nun gibt es zwei Möglichkeiten,  $B$  zu ersetzen:  $B \rightarrow b$  und  $B \rightarrow bC$ . Aber diesmal reicht ein Look-ahead von einem Symbol nicht mehr aus, da beide Regeln mit  $b$  beginnen. Wir brauchen also einen Look-ahead von zwei Symbolen. Damit sehen wir, dass wieder die erste Möglichkeit entfällt, da  $w \neq ab\$$ . Es wird also  $B \rightarrow bC$  angewandt, und wir erhalten:

$$S \vdash_{G,l} C\$ \vdash_{G,l} aB\$ \vdash_{G,l} abC\$.$$

Es gibt zwei Möglichkeiten,  $C$  zu ersetzen. Da der dritte Buchstabe von  $w$  ein  $b$  ist (ein Look-ahead von einem Symbol genügt), wird  $C \rightarrow bA$  angewandt:

$$S \vdash_{G,l} C\$ \vdash_{G,l} aB\$ \vdash_{G,l} abC\$ \vdash_{G,l} abbA\$.$$

Nun ist nur die Regel  $A \rightarrow a$  möglich:

$$S \vdash_{G,l} C\$ \vdash_{G,l} aB\$ \vdash_{G,l} abC\$ \vdash_{G,l} abba\$,$$

und der Syntaxbaum für die Ableitung  $S \vdash_{G,l}^* w$  ist gefunden. Insgesamt genügt immer ein Look-ahead von zwei Symbolen.  $G$  ist ein Beispiel für eine (starke) LL(2)-Grammatik (siehe Abschnitt 4.2).



# Kapitel 5

## Kontextsensitive und $\mathcal{L}_0$ -Sprachen

**Satz 5.1** *CF ist echt in CS enthalten.*

**Beweis.** Nach Behauptung 2.29 ist die Sprache  $L = \{a^m b^m c^m \mid m \geq 1\}$  nicht kontextfrei. Andererseits ist  $L$  kontextsensitiv, wie die Grammatik aus Beispiel 1.5 (2.) zeigt. Somit ist  $L \in \text{CS} - \text{CF}$ , also  $\text{CF} \subset \text{CS}$ . ■

**Ziel:** Automatenmodelle für CS und für  $\mathcal{L}_0$ .

### 5.1 Turingmaschinen

Ein grundlegendes Algorithmenmodell ist die Turingmaschine, die 1936 von Alan Turing (1912 bis 1954) in seiner bahnbrechenden Arbeit “*On computable numbers, with an application to the Entscheidungsproblem*” (*Proceedings of the London Mathematical Society*, ser. 2, 42:230–265, 1936; Correction, *ibid*, vol. 43, pp. 544–546, 1937) eingeführt wurde. Die Turingmaschine ist ein sehr einfaches, abstraktes Modell eines Computers. Im Folgenden definieren wir dieses Modell durch Angabe seiner Syntax und Semantik, wobei wir wieder zwei verschiedene Berechnungsparadigma berücksichtigen: Determinismus und Nichtdeterminismus. Es ist zweckmäßig, zuerst das allgemeinere Modell der nichtdeterministischen Turingmaschine zu beschreiben. Deterministische Turingmaschinen ergeben sich dann sofort als ein Spezialfall.

#### Modell und Arbeitsweise:

- Eine Turingmaschine ist mit  $k$  beidseitig unendlichen Arbeitsbändern ausgestattet, die in Felder unterteilt sind, in denen Buchstaben stehen können.
- Enthält ein Feld keinen Buchstaben, so wird dies durch ein spezielles Leerzeichen, das  $\square$ -Symbol, signalisiert.

- Auf den Arbeitsbändern findet die eigentliche Rechnung statt. Zu Beginn einer Rechnung steht das Eingabewort auf einem bestimmten Band, dem Eingabeband, und alle anderen Felder enthalten das  $\square$ -Zeichen. Am Ende der Rechnung erscheint das Ergebnis der Rechnung auf einem bestimmten Band, dem Ausgabeband.<sup>1</sup>
- Auf jedes Band kann je ein Schreib-Lese-Kopf zugreifen. Dieser kann in einem Takt der Maschine den aktuell gelesenen Buchstaben überschreiben und anschließend eine Bewegung um ein Feld nach rechts oder links ausführen oder aber auf dem aktuellen Feld stehenbleiben. Gleichzeitig kann sich der aktuelle Zustand der Maschine ändern, den sie sich in ihrem inneren Gedächtnis (“*finite control*”) merkt. Abbildung 5.1 zeigt eine Turingmaschine mit zwei Bändern.
- PDAs und somit auch NFAs und DFAs sind spezielle TMs.

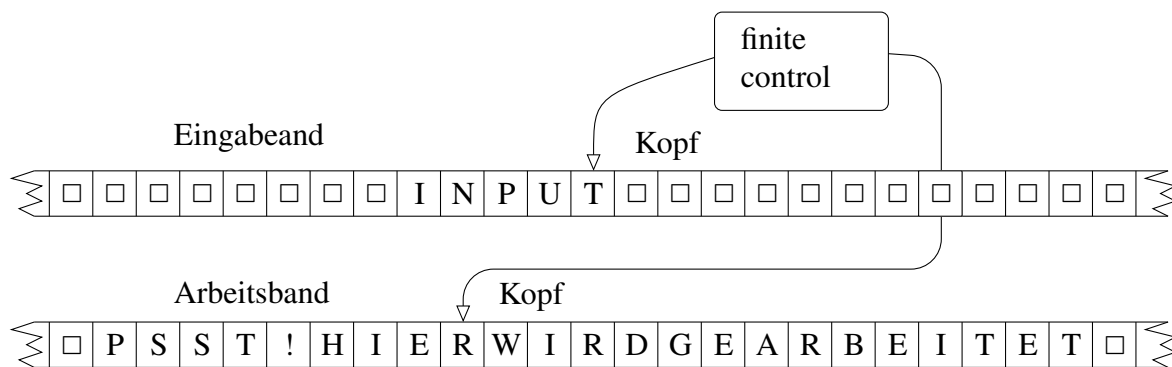


Abbildung 5.1: Eine Turingmaschine

**Definition 5.2 (Syntax von Turingmaschinen)** Eine (nichtdeterministische) Turingmaschine mit  $k$  Bändern (kurz  $k$ -Band-TM) ist ein 7-Tupel  $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$ , wobei

- $\Sigma$  das Eingabe-Alphabet ist,
- $\Gamma$  das Arbeitsalphabet mit  $\Sigma \subseteq \Gamma$ ,
- $Z$  eine endliche Menge von Zuständen mit  $Z \cap \Gamma = \emptyset$ ,
- $\delta : Z \times \Gamma^k \rightarrow \mathfrak{P}(Z \times \Gamma^k \times \{L, R, N\}^k)$  die Überföhrungsfunktion,

<sup>1</sup>Man kann z.B. festlegen, dass auf dem Eingabeband nur gelesen und auf dem Ausgabeband nur geschrieben werden darf. Ebenso kann man eine Vielzahl weiterer Variationen der technischen Details festlegen. Zum Beispiel könnte man verlangen, dass bestimmte Köpfe nur in einer Richtung wandern dürfen oder dass die Bänder halbseitig unendlich sind und so weiter.

- $z_0 \in Z$  der Startzustand,
- $\square \in \Gamma - \Sigma$  das „Blank“-Symbol (oder Leerzeichen) und
- $F \subseteq Z$  die Menge der Endzustände.

Der Spezialfall der deterministischen Turingmaschine mit  $k$  Bändern ergibt sich, wenn die Überföhrungsfunktion  $\delta$  von  $Z \times \Gamma^k$  nach  $Z \times \Gamma^k \times \{L, R, N\}^k$  abbildet.

Für  $k = 1$  ergibt sich die 1-Band-Turingmaschine, die wir einfach mit TM abkürzen. Jede  $k$ -Band-TM kann durch eine entsprechende Maschine mit nur einem Band simuliert werden, wobei sich die Rechenzeit höchstens quadriert. Spielt die Effizienz eine Rolle, kann es dennoch sinnvoll sein, mehrere Bänder zu haben. Wir werden uns im Folgenden auf 1-Band-Turingmaschinen beschränken.

### Weiter zur Arbeitsweise:

- Statt  $(z', b, x) \in \delta(z, a)$  mit  $z, z' \in Z, x \in \{L, R, N\}, a, b \in \Gamma$  schreiben wir kurz:  $(z, a) \rightarrow (z', b, x)$ , oder (wenn es keine Verwechslungen geben kann):  $za \rightarrow z'bx$ .
- Dieser Turingbefehl bedeutet: Ist im Zustand  $z$  der Kopf auf einem Feld mit aktueller Inschrift  $a$ , so wird:
  - $a$  durch  $b$  überschrieben,
  - der neue Zustand  $z'$  angenommen und
  - eine Kopfbewegung gemäß  $x \in \{L, R, N\}$  ausgeführt (d.h., ein Feld nach links, ein Feld nach rechts oder neutral, also Stehenbleiben).

Turingmaschinen kann man sowohl als Akzeptoren auffassen, die Sprachen (also Wortmengen) akzeptieren, als auch zur Berechnung von Funktionen benutzen. In diesem Abschnitt betrachten wir Turingmaschinen als Akzeptoren; die Funktionsberechnung einer Turingmaschine wird später definiert.

### **Definition 5.3 (Semantik von Turingmaschinen)**

- Eine Konfiguration einer TM  $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$  ist ein Wort  $k \in \Gamma^* Z \Gamma^*$ . Dabei bedeutet  $k = \alpha z \beta$ , dass  $\alpha \beta$  die aktuelle Bandinschrift ist (also das Wort, in das die Eingabe bisher transformiert wurde), der Kopf auf dem ersten Symbol von  $\beta$  steht und  $z$  der aktuelle Zustand von  $M$  ist.
- Auf der Menge  $\mathfrak{K}_M = \Gamma^* Z \Gamma^*$  aller Konfigurationen von  $M$  definieren wir eine binäre Relation  $\vdash_M$  wie folgt. Intuitiv gilt  $k \vdash_M k'$  für  $k, k' \in \mathfrak{K}_M$  genau dann, wenn  $k'$  aus  $k$  durch eine Anwendung von  $\delta$  hervorgeht.

*Formal: Für alle  $\alpha = a_1a_2 \cdots a_m$  und  $\beta = b_1b_2 \cdots b_n$  in  $\Gamma^*$  ( $m \geq 0, n \geq 1$ ) und für alle  $z \in Z$  sei*

$$\alpha z \beta \vdash_M \begin{cases} a_1a_2 \cdots a_m z' c b_2 \cdots b_n & \text{falls } (z, b_1) \rightarrow (z', c, N) \text{ und } m \geq 0, n \geq 1 \\ a_1a_2 \cdots a_m c z' b_2 \cdots b_n & \text{falls } (z, b_1) \rightarrow (z', c, R) \text{ und } m \geq 0, n \geq 2 \\ a_1a_2 \cdots a_{m-1} z' a_m c b_2 \cdots b_n & \text{falls } (z, b_1) \rightarrow (z', c, L) \text{ und } m \geq 1, n \geq 1. \end{cases}$$

*Sonderfälle:*

1.  $n = 1$  und  $(z, b_1) \rightarrow (z', c, R)$  (d.h.,  $M$  läuft nach rechts und trifft auf ein  $\square$ ):

$$a_1a_2 \cdots a_m z b_1 \vdash_M a_1a_2 \cdots a_m c z' \square.$$

2.  $m = 0$  und  $(z, b_1) \rightarrow (z', c, L)$  (d.h.,  $M$  läuft nach links und trifft auf ein  $\square$ ):

$$z b_1 b_2 \cdots b_n \vdash_M z' \square c b_2 \cdots b_n.$$

- Die Startkonfiguration von  $M$  bei Eingabe  $x$  ist stets  $z_0x$ . Die Endkonfigurationen von  $M$  bei Eingabe  $x$  haben die Form  $\alpha z \beta$  mit  $z \in F$  und  $\alpha, \beta \in \Gamma^*$ .  $M$  hält an, falls eine Endkonfiguration erreicht wird, oder falls kein Turingbefehl mehr auf die aktuelle Konfiguration von  $M$  anwendbar ist.
- Sei  $\vdash_M^*$  die reflexive, transitive Hülle von  $\vdash_M$ .
- Die von der TM  $M$  akzeptierte Sprache ist definiert durch

$$L(M) = \{x \in \Sigma^* \mid z_0x \vdash_M^* \alpha z \beta \text{ mit } z \in F \text{ und } \alpha, \beta \in \Gamma^*\}.$$

#### Bemerkung 5.4

- Da im Falle einer nichtdeterministischen TM jede Konfiguration mehrere Folgekonfigurationen haben kann, ergibt sich ein Berechnungsbaum, dessen Wurzel die Startkonfiguration und dessen Blätter die Endkonfigurationen sind.
- Die Knoten des Berechnungsbaums von  $M(x)$  sind die Konfigurationen von  $M$  bei Eingabe  $x$ .
- Für zwei Konfigurationen  $k$  und  $k'$  aus  $\mathfrak{K}_M$  gibt es genau dann eine gerichtete Kante von  $k$  nach  $k'$ , wenn  $k \vdash_M k'$  gilt.
- Ein Pfad im Berechnungsbaum von  $M(x)$  ist eine Folge von Konfigurationen  $k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_t \vdash_M \cdots$ , also eine Rechnung von  $M(x)$ .
- Der Berechnungsbaum einer nichtdeterministischen TM kann unendliche Pfade haben.

- Im Falle einer deterministischen TM wird jede Konfiguration außer der Startkonfiguration eindeutig (deterministisch) durch ihre Vorgängerkonfiguration bestimmt. Deshalb entartet der Berechnungsbaum einer deterministischen TM zu einer linearen Kette, die mit der Startkonfiguration beginnt und mit einer Endkonfiguration endet, falls die Maschine bei dieser Eingabe hält; andernfalls geht die Kette ins Unendliche.

**Beispiel 5.5 (Turingmaschine)** Betrachte die Sprache  $L = \{a^n b^n c^n \mid n \geq 1\}$ . Eine Turingmaschine, die  $L$  akzeptiert, ist definiert durch

$$M = (\{a, b, c\}, \{a, b, c, \$, \square\}, \{z_0, z_1, \dots, z_6\}, \delta, z_0, \square, \{z_6\}),$$

wobei die Liste der Turingbefehle gemäß der Überföhrungsfunktion  $\delta$  in Tabelle 5.1 angegeben ist. Tabelle 5.2 gibt die Bedeutung der einzelnen Zustände von  $M$  sowie die mit den einzelnen Zuständen verbundene Absicht an.

$(z_0, a) \mapsto (z_1, \$, R)$	$(z_2, \$) \mapsto (z_2, \$, R)$	$(z_5, c) \mapsto (z_5, c, L)$
$(z_1, a) \mapsto (z_1, a, R)$	$(z_3, c) \mapsto (z_3, c, R)$	$(z_5, \$) \mapsto (z_5, \$, L)$
$(z_1, b) \mapsto (z_2, \$, R)$	$(z_3, \square) \mapsto (z_4, \square, L)$	$(z_5, b) \mapsto (z_5, b, L)$
$(z_1, \$) \mapsto (z_1, \$, R)$	$(z_4, \$) \mapsto (z_4, \$, L)$	$(z_5, a) \mapsto (z_5, a, L)$
$(z_2, b) \mapsto (z_2, b, R)$	$(z_4, \square) \mapsto (z_6, \square, R)$	$(z_5, \square) \mapsto (z_0, \square, R)$
$(z_2, c) \mapsto (z_3, \$, R)$	$(z_4, c) \mapsto (z_5, c, L)$	$(z_0, \$) \mapsto (z_0, \$, R)$

Tabelle 5.1: Liste  $\delta$  der Turingbefehle von  $M$  für die Sprache  $L = \{a^n b^n c^n \mid n \geq 1\}$

$Z$	Bedeutung	Absicht
$z_0$	Anfangszustand	neuer Zyklus
$z_1$	ein $a$ gemerkt	nächstes $b$ suchen
$z_2$	je ein $a, b$ gemerkt	nächstes $c$ suchen
$z_3$	je ein $a, b, c$ getilgt	rechten Rand suchen
$z_4$	rechter Rand erreicht	Zurücklaufen und Test, ob alle $a, b, c$ getilgt
$z_5$	Test nicht erfolgreich	Zurücklaufen zum linken Rand und neuer Zyklus
$z_6$	Test erfolgreich	Akzeptieren

Tabelle 5.2: Interpretation der Zustände von  $M$

Die (deterministische) Konfigurationenfolge von  $M$  bei Eingabe von  $aabbcc$  ist:

$z_0 a a b b c c \quad \vdash_M \$ z_1 a b b c c \quad \vdash_M \$ a z_1 b b c c \quad \vdash_M \$ a \$ z_2 b c c \quad \vdash_M \$ a \$ b z_2 c c \quad \vdash_M$   
 $\$ a \$ b \$ z_3 c \quad \vdash_M \$ a \$ b \$ c z_3 \square \vdash_M \$ a \$ b \$ z_4 c \quad \vdash_M \$ a \$ b z_5 \$ c \quad \vdash_M \dots \quad \vdash_M$   
 $\$ z_5 a \$ b \$ c \quad \vdash_M z_5 \$ a \$ b \$ c \quad \vdash_M z_5 \square \$ a \$ b \$ c \vdash_M z_0 \$ a \$ b \$ c \quad \vdash_M \$ z_0 a \$ b \$ c \quad \vdash_M$   
 $\$ \$ z_1 \$ b \$ c \quad \vdash_M \$ \$ \$ z_1 b \$ c \quad \vdash_M \$ \$ \$ \$ z_2 \$ c \quad \vdash_M \$ \$ \$ \$ \$ z_2 c \quad \vdash_M \$ \$ \$ \$ \$ \$ z_3 \square \vdash_M$   
 $\$ \$ \$ \$ \$ z_4 \$ \quad \vdash_M \dots \quad \vdash_M z_4 \$ \$ \$ \$ \$ \$ \quad \vdash_M z_4 \square \$ \$ \$ \$ \$ \$ \vdash_M z_6 \$ \$ \$ \$ \$ \$$   
 (Akzeptieren)

## 5.2 Linear beschränkte Automaten

Linear beschränkte Automaten sind spezielle Turingmaschinen, die nie den Bereich des Bandes verlassen, auf dem die Eingabe steht. Dazu ist es zweckmäßig, den rechten Rand der Eingabe wie folgt zu markieren (auf dem linken Rand steht der Kopf zu Beginn der Berechnung sowieso und kann diesen im ersten Takt markieren):

1. Verdoppele das Eingabe-Alphabet  $\Sigma$  zu  $\widehat{\Sigma} = \Sigma \cup \{\widehat{a} \mid a \in \Sigma\}$ .
2. Repräsentiere die Eingabe  $a_1 a_2 \cdots a_n \in \Sigma^+$  durch das Wort  $a_1 a_2 \cdots a_{n-1} \widehat{a}_n$  über  $\widehat{\Sigma}$ .

### Definition 5.6 (LBA)

- Eine nichtdeterministische TM  $M$  heißt linear beschränkter Automat (kurz LBA), falls für alle Konfigurationen  $\alpha z \beta$  und

– für alle Wörter  $x = a_1 a_2 \cdots a_{n-1} a_n \in \Sigma^+$  mit

$$z_0 a_1 a_2 \cdots a_{n-1} \widehat{a}_n \vdash_M^* \alpha z \beta$$

gilt:  $|\alpha \beta| = n$ , und

– für  $x = \lambda$  mit  $z_0 \sqcap \vdash_M^* \alpha z \beta$  gilt:  $\alpha \beta = \sqcap$ .

- Die vom LBA  $M$  akzeptierte Sprache ist definiert durch

$$L(M) = \left\{ a_1 a_2 \cdots a_{n-1} a_n \in \Sigma^* \mid \begin{array}{l} z_0 a_1 a_2 \cdots a_{n-1} \widehat{a}_n \vdash_M^* \alpha z \beta \\ \text{mit } z \in F \text{ und } \alpha, \beta \in \Gamma^* \end{array} \right\}.$$

**Satz 5.7**  $L \in \text{CS} \iff L = L(M)$  für einen LBA  $M$ .

### Beweis.

( $\Rightarrow$ ) Es sei  $L$  eine kontextsensitive Sprache und  $G = (\Sigma, N, S, P)$  eine Typ-1-Grammatik mit  $L(G) = L$ . Wir beschreiben den gesuchten LBA  $M$  für  $L$  informal wie folgt:

1. Eingabe  $x = a_1 a_2 \cdots a_n$ .
2. Wähle nichtdeterministisch eine Regel  $u \rightarrow v$  aus  $P$  und suche eine beliebiges Vorkommen von  $v$  in der aktuellen Bandinschrift von  $M$ .
3. Ersetze  $v$  durch  $u$ . Ist dabei  $|u| < |v|$ , so verschiebe entsprechend alle Symbole rechts der Lücke, um diese zu schließen.
4. Ist die aktuelle Bandinschrift nur noch das Startsymbol  $S$ , so halte im Endzustand und akzeptiere; andernfalls gehe zu (2) und wiederhole.

Die so konstruierte Turingmaschine  $M$  ist ein LBA, weil alle Regeln in  $P$  nicht-verkürzend sind. Es gilt:

$$\begin{aligned}
 x \in L(G) &\iff \text{es gibt eine Ableitung } S \vdash_G^* x \\
 &\iff \text{es gibt eine Rechnung von } M, \text{ die diese} \\
 &\quad \text{Ableitung in umgekehrter Reihenfolge simuliert} \\
 &\iff x \in L(M).
 \end{aligned}$$

( $\Leftarrow$ ) Sei  $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$  ein LBA mit  $L(M) = L$ . Ist  $x$  die Eingabe und ist  $k \in \mathfrak{K}_M = \Gamma^* Z \Gamma^*$  eine Konfiguration mit  $z_0 x \vdash_M^* k$ , so müssen wir sichern, dass  $|k| \leq |x|$  gilt. Um Konfigurationen durch Wörter der Länge  $|x|$  darzustellen, verwenden wir für die zu konstruierende Typ-1-Grammatik  $G$  das Alphabet

$$\Delta = \Gamma \cup (Z \times \Gamma).$$

Beispielsweise hat die Konfiguration  $k = azbcd$  mit  $z \in Z$  und  $a, b, c, d \in \Gamma$  über dem Alphabet  $\Delta$  die Darstellung  $k' = a(z, b)cd$  und somit die Länge  $4 = |abcd|$ , siehe Abbildung 5.2.

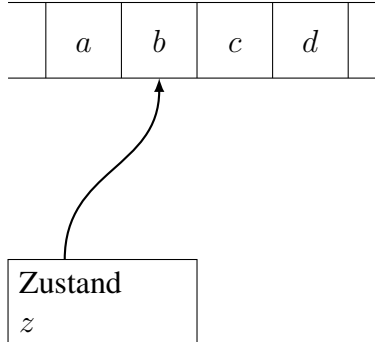


Abbildung 5.2: Darstellung von Konfigurationen durch Wörter.

Einen  $\delta$ -Übergang von  $M$  wie etwa

$$(z, a) \rightarrow (z', b, L)$$

kann man durch nichtverkürzende Regeln der Form

$$c(z, a) \rightarrow (z', c)b$$

für alle  $c \in \Gamma$  beschreiben. Die Menge aller solcher Regeln der Grammatik heiße  $P'$ . Es gilt für alle  $k_1, k_2 \in \mathfrak{K}_M$ :

$$k_1 \vdash_M^* k_2 \iff k'_1 \vdash_{G'}^* k'_2$$

wobei  $k'_i, i \in \{1, 2\}$ , die obige Darstellung der Konfiguration  $k_i$  bezeichnet und  $G'$  die Grammatik mit Regelmenge  $P'$  ist.

Definiere  $G = (\Sigma, N, S, P)$  so

$$N = \{S, A\} \cup (\Delta \times \Sigma);$$

$$P = \{S \rightarrow A(\hat{a}, a) \mid a \in \Sigma\} \cup \quad (5.1)$$

$$\{A \rightarrow A(a, a) \mid a \in \Sigma\} \cup \quad (5.2)$$

$$\{A \rightarrow ((z_0, a), a) \mid a \in \Sigma\} \cup \quad (5.3)$$

$$\{(\alpha_1, a)(\alpha_2, b) \rightarrow (\beta_1, a)(\beta_2, b) \mid \alpha_1\alpha_2 \rightarrow \beta_1\beta_2 \in P', a, b \in \Sigma\} \cup \quad (5.4)$$

$$\{(\alpha_1, a) \rightarrow (\beta_1, a) \mid \alpha_1 \rightarrow \beta_1 \in P' \text{ für } a \in \Sigma\} \cup \quad (5.5)$$

$$\{((z, a), b) \rightarrow b \mid z \in F, a \in \Gamma, b \in \Sigma\} \cup \quad (5.6)$$

$$\{(a, b) \rightarrow b \mid a \in \Gamma, b \in \Sigma\}. \quad (5.7)$$

Offenbar ist  $G$  eine Typ-1-Grammatik.

Die Idee hinter dieser Konstruktion von  $G$  ist die folgende:

- Regeln der Form (5.1), (5.2) und (5.3) ermöglichen Ableitungen der Form

$$S \vdash_G^* ((z_0, a_1), a_1)(a_2, a_2) \cdots (a_{n-1}, a_{n-1})(\hat{a}_n, a_n).$$

Dabei ist  $((z_0, a_1), a_1)(a_2, a_2) \cdots (a_{n-1}, a_{n-1})(\hat{a}_n, a_n)$  ein Wort mit  $n$  Buchstaben über dem Alphabet  $\Delta \times \Sigma \subseteq N$ . Jeder Buchstabe ist ein Paar. Dabei stellen die ersten Komponenten die Startkonfiguration

$$z_0 a_1 a_2 \cdots a_{n-1} \hat{a}_n = (z_0, a_1) a_2 \cdots a_{n-1} \hat{a}_n$$

dar und die zweiten Komponenten das Eingabewort

$$x = a_1 a_2 \cdots a_{n-1} a_n.$$

- Mit den Regeln der Form (5.4) und (5.5) simuliert  $G$  dann die Rechnung von  $M$  bei Eingabe  $x = a_1 a_2 \cdots a_{n-1} a_n$ , wobei die Regeln aus  $P'$  auf die ersten Komponenten der Paare angewandt werden und die zweiten Komponenten unverändert bleiben. Die Simulation der Rechnung von  $M(x)$  ist beendet, sobald eine Endkonfiguration erreicht ist.
- Regeln der Form (5.6) und (5.7) löschen schließlich die ersten Komponenten der Paare weg. Übrig bleiben die zweiten Komponenten, also das akzeptierte Eingabewort  $x$ .



- Wird nie eine Endkonfiguration von  $M(x)$  erreicht, so kommen die Löschregeln (5.6) der Grammatik  $G$  nie zur Anwendung, es bleiben im abgeleiteten Wort immer Nichtterminale enthalten und  $x$  wird also nicht aus dem Startsymbol  $S$  abgeleitet.

Zusammengefasst folgt aus der obigen Idee formal die Äquivalenzkette:

$$\begin{aligned}
 x \in L(M) &\iff S \vdash_G^* ((z_0, a_1), a_1)(a_2, a_2) \cdots (a_{n-1}, a_{n-1})(\hat{a}_n, a_n) \\
 &\quad \text{mit (5.1), (5.2) und (5.3)} \\
 &\quad \vdash_G^* (\gamma_1, a_1) \cdots (\gamma_{k-1}, a_{k-1})((z, \gamma_k), a_k)(\gamma_{k+1}, a_{k+1}) \cdots (\gamma_n, a_n) \\
 &\quad \text{mit (5.4), wobei } z \in F, \gamma_i \in \Gamma, a_i \in \Sigma \\
 &\quad \vdash_G^* a_1 a_2 \cdots a_n = x \\
 &\quad \text{mit (5.6) und (5.7)} \\
 &\iff x \in L(G).
 \end{aligned}$$

■

Analog zum Beweis von Satz 5.7 kann man den folgenden Satz beweisen.

**Satz 5.8**  $L \in \mathfrak{L}_0 \iff L = L(M)$  für eine Turingmaschine  $M$ .

**Beweis.**

- ( $\Rightarrow$ ) Wie im ersten Teil des Beweises von Satz 5.7. Da  $G$  nun nicht nur nichtverkürzende Regeln enthält, erhält man i. A. keinen LBA, sondern eine TM.
- ( $\Leftarrow$ ) Man kann die gleiche Konstruktion wie im zweiten Teil des Beweises von Satz 5.7 verwenden. Da eine TM keine lineare Beschränkung auf dem Band hat, können in Regeln vom Typ (5.4) Konfigurationen  $k$  mit  $|k| > |x|$  aufgebaut werden. Solche Konfigurationen entsprechen Wörtern in  $G$ , die Nichtterminale der Form  $(\alpha, a)$  mit  $a = \lambda$  enthalten. Um solche Nichtterminale wieder zu löschen, müssen diese durch  $\lambda$  ersetzt werden, d.h., es werden verkürzende Regeln benötigt. Somit erhalten wir i. A. keine kontextsensitive, sondern eine Grammatik vom Typ 0.

■

Satz 5.8 liefert eine Charakterisierung der Klasse  $\mathfrak{L}_0$  durch das Automatenmodell der Turingmaschine. Somit haben wir nun alle Klassen der Chomsky-Hierarchie durch geeignete Automaten charakterisiert.

**Bemerkung 5.9**

- In Satz 5.8 ist es dabei gleichgültig, ob die TM  $M$  deterministisch oder nichtdeterministisch ist. Da man jede nichtdeterministische TM durch deterministische TM simulieren kann, folgt

$$\begin{aligned}\mathcal{L}_0 &= \{L(M) \mid M \text{ ist eine deterministische TM}\} \\ &= \{L(M) \mid M \text{ ist eine nichtdeterministische TM}\}.\end{aligned}$$

- Im Gegensatz dazu ist es in Satz 5.7 wesentlich, dass der LBA  $M$  eine nichtdeterministische TM ist.
- Bis heute offen ist das

**Erste LBA-Problem:** Sind deterministische und nichtdeterministische LBAs äquivalent?

- Hingegen ist das ebenfalls 1964 von Kuroda gestellte

**Zweite LBA-Problem:** Ist die Klasse der durch nichtdeterministische LBAs akzeptierbaren Sprachen komplementabgeschlossen?

inzwischen gelöst worden, und zwar unabhängig und etwa zeitgleich 1988 von Neil Immerman und Robert Szelepcsényi. Aus Satz 5.7 folgt damit, dass die Klasse CS komplementabgeschlossen ist.

## 5.3 Zusammenfassung

In diesem Abschnitt fassen wir einige der bisher erzielten Resultat in übersichtlicher Tabellenform zusammen. Einige der dabei aufgeführten Ergebnisse wurden hier nicht bewiesen, können aber in der einschlägigen Literatur gefunden werden.

### Charakterisierungen durch Automaten und Grammatiken

Tabelle 5.3 listet für die Klassen der Chomsky-Hierarchie die Beschreibungen durch Grammatiken bzw. die entsprechenden Charakterisierungen durch Automaten auf.

### Determinismus vs. Nichtdeterminismus

Tabelle 5.4 gibt an, für welche Automatenmodelle Determinismus und Nichtdeterminismus übereinstimmen.

Typ 3	reguläre Grammatik deterministischer endlicher Automat (DFA) nichtdeterministischer endlicher Automat (NFA) regulärer Ausdruck
deterministisch kontextfrei	LR(1)-Grammatik deterministischer Kellerautomat (DPDA)
Typ 2	kontextfreie Grammatik Kellerautomat (PDA)
Typ 1	kontextsensitive Grammatik linear beschränkter Automat (LBA)
Typ 0	Typ-0-Grammatik Turingmaschine (NTM bzw. DTM)

Tabelle 5.3: Charakterisierungen durch Automaten und Grammatiken

Deterministischer Automat	Nichtdeterministischer Automat	äquivalent?
DFA	NFA	ja
DPDA	PDA	nein
DLBA	LBA	?
DTM	NTM	ja

Tabelle 5.4: Determinismus versus Nichtdeterminismus

## Abschlusseigenschaften

Tabelle 5.5 gibt einige der betrachteten Abschlusseigenschaften für die untersuchten Sprachklassen an.

	Typ 3	det. kf.	Typ 2	Typ 1	Typ 0
Schnitt	ja	nein	nein	ja	ja
Vereinigung	ja	nein	ja	ja	ja
Komplement	ja	ja	nein	ja	nein
Konkatenation	ja	nein	ja	ja	ja
Iteration	ja	nein	ja	ja	ja
Spiegelung	ja	nein	ja	ja	ja

Tabelle 5.5: Abschlusseigenschaften

## Komplexität des Wortproblems

**Definition 5.10 (Wortproblem)** Für  $i \in \{0, 1, 2, 3\}$  definieren wir das Wortproblem für Typ- $i$ -Grammatiken wie folgt:

$$\text{Wort}_i = \{(G, x) \mid G \text{ ist Typ-}i\text{-Grammatik und } x \in L(G)\}$$

Tabelle 5.6 gibt die Zeitkomplexität für dieses Problem in den einzelnen Stufen der Chomsky-Hierarchie an, sofern es entscheidbar ist. Man beachte, dass hierbei wichtig ist, in welcher Form das Problem gegeben ist.

Typ 3 (DFA gegeben)	lineare Komplexität
det. kf.	lineare Komplexität
Typ 2 (CNF gegeben)	Komplexität $\mathcal{O}(n^3)$ (CYK-Algorithmus)
Typ 1	exponentielle Komplexität
Typ 0	unentscheidbar (d.h. algorithmisch nicht lösbar)

Tabelle 5.6: Komplexität des Wortproblems gemessen in der Länge des Eingabewortes

# **Teil II**

## **Berechenbarkeit**



# Kapitel 6

## Intuitiver Berechenbarkeitsbegriff und die These von Church

**Ziel:** Wir suchen Antworten auf die folgenden Fragen:

- Was ist Berechenbarkeit? Wie kann man das intuitiv Berechenbare formal fassen?
- Was ist ein Algorithmus?
- Welche Indizien hat man dafür, dass ein formaler Algorithmenbegriff tatsächlich den Begriff des intuitiv Berechenbaren exakt einfängt?

Intuitiv versteht man unter „Berechenbarkeit“ alles, was sich algorithmisch lösen lässt. Wir beschränken uns auf die Berechenbarkeit von Funktionen  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  bzw. von Wortfunktionen  $f : \Sigma^* \rightarrow \Sigma^*$ . Zahlen  $n \in \mathbb{N} = \{0, 1, 2, 3, \dots\}$  können beispielsweise durch ihre Binärdarstellung als Wörter über dem Alphabet  $\Sigma = \{0, 1\}$  dargestellt werden. Zu einer natürlichen Zahl  $n$  bezeichnen wir mit  $\text{bin}(n)$  die *Binärdarstellung von  $n$  ohne führende Nullen*, z.B.  $\text{bin}(19) = 10011$  und  $\text{bin}(5) = 101$ . Damit kann man durch  $\text{bin} : \mathbb{N} \rightarrow \{0, 1\}^*$  in natürlicher Weise eine Injektion zwischen  $\mathbb{N}$  und  $\{0, 1\}^*$  angeben. Die Berechenbarkeit z.B. reellwertiger Funktionen wird hier nicht betrachtet.

**Definition 6.1** *Es sei  $f : A \rightarrow B$  eine Funktion.*

- *Die Menge  $A$  heißt Eingabemenge oder Urbildbereich und die Menge  $B$  Ausgabemenge oder Bildbereich von  $f$ .*
- *Der Definitionsbereich  $D_f$  der Funktion  $f$  ist die Teilmenge von  $A$ , auf der  $f$  definiert ist.*
- *Der Wertebereich  $W_f$  der Funktion  $f$  ist die Menge  $\{f(x) \mid x \in D_f\}$ .*

- $f$  ist stets eine partielle Funktion.
- $f$  heißt total, falls sie überall auf ihrer Eingabemenge definiert ist, d.h., falls  $A = D_f$  gilt.

Insbesondere ist also jede totale Funktion eine partielle Funktion.

Die Funktion  $f_1 : \mathbb{N} \rightarrow \mathbb{N}$  mit  $f_1(n) = n + 1$  ist total (und partiell), da  $D_{f_1} = \mathbb{N}$ .

Die Funktion  $f_2 : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit  $f_2(n_1, n_2) = n_1 \text{ div } n_2$  ist nicht-total (und partiell), da  $D_{f_2} = \mathbb{N}^2 - \{(n, 0) \mid n \in \mathbb{N}\}$ .

Intuitiv sagt man, ein „Algorithmus“ ist eine endliche Folge von Befehlen oder Anweisungen, die eine Eingabe in eine bestimmte Ausgabe in endlich vielen Rechenschritten transformieren. Das heißt, ein *Algorithmus*  $A$  berechnet eine Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  genau dann, wenn gilt:

1. Für alle  $(n_1, n_2, \dots, n_k) \in D_f$  hält der Algorithmus  $A$  bei Eingabe  $(n_1, n_2, \dots, n_k)$  nach endlich vielen Schritten mit der Ausgabe  $f(n_1, n_2, \dots, n_k)$  an.
2. Für alle  $(n_1, n_2, \dots, n_k) \notin D_f$  hält  $A$  bei Eingabe  $(n_1, n_2, \dots, n_k)$  nie an (ist also z.B. in einer Endlosschleife). (Alternativ hierzu könnte man verlangen, dass der Algorithmus zwar hält, aber die Eingabe verwirft.)

Nur intuitiv zu argumentieren, dass eine bestimmte Funktion  $f$  nicht berechenbar ist, ist unmöglich. Für einen solchen Nachweis ist es unabdingbar, dass man den Algorithmusbegriff im mathematischen Sinne formalisiert und zeigt, dass  $f$  durch *keinen* Algorithmus dieser formal definierten Klasse berechnet werden kann.

Beispiele von formalisierten Algorithmeklassen, die wir schon kennen, sind:

- Programme in einer fixierten Programmiersprache, z.B. Java;
- endliche Automaten (DFAs bzw. NFAs);
- Kellerautomaten (PDAs);
- deterministische Kellerautomaten (DPDAs);
- linear beschränkte Automaten (LBAs);
- Turingmaschinen (TMs).

Alle diese Modelle können so modifiziert werden, dass sie Funktionen berechnen, nicht Sprachen entscheiden. Beispielsweise gibt es Funktionen, die sich zwar durch Turingmaschinen, nicht aber durch endliche Automaten berechnen lassen. Hat man eine Algorithmeklasse fixiert, so tritt das neue Problem auf, festzustellen, ob sie wirklich genau den Begriff des intuitiv Berechenbaren erfasst. Dies kann nur intuitiv begründet, nicht aber formal bewiesen werden.



**Beispiel 6.2** Sei  $\tilde{\pi} = 1415 \dots$  die Folge der Nachkommaziffern der Zahl  $\pi = 3,1415 \dots$ . Die Anfangswortrelation und Teilwortrelation wird mit  $\sqsubseteq_a$  bzw.  $\sqsubseteq$  bezeichnet, siehe Definition 1.2. Betrachte die folgenden Funktionen und überlege, ob sie intuitiv berechenbar sind oder nicht:

1. Definiere die Funktion  $f : \{0, 1, \dots, 9\}^* \rightarrow \{0, 1\}$  durch

$$f(n) = \begin{cases} 1 & \text{falls } n \sqsubseteq_a \tilde{\pi} = 1415 \dots \\ 0 & \text{sonst.} \end{cases}$$

Offenbar ist  $f$  berechenbar, denn es gibt Näherungsverfahren für  $\pi$ , die nur bis zur durch die Länge von  $n$  vorgegebenen Genauigkeit laufen müssen, also in endlicher Zeit zum Ergebnis kommen.

2. Definiere die Funktion  $g : \{0, 1, \dots, 9\}^* \rightarrow \{0, 1\}$  durch

$$g(n) = \begin{cases} 1 & \text{falls } n \sqsubseteq \tilde{\pi} = 1415 \dots \\ 0 & \text{sonst.} \end{cases}$$

Es ist offen, ob  $g$  berechenbar ist. Wäre die Zahl  $\pi$  so „zufällig“, dass jede endliche Ziffernfolge als ein Teilwort in  $\tilde{\pi}$  erscheint, dann wäre  $g \equiv 1$  (d.h.,  $g(n) = 1$  für alle  $n$ ) und somit berechenbar.

3. Definiere die Funktion  $h : \mathbb{N} \rightarrow \{0, 1\}$  durch

$$h(n) = \begin{cases} 1 & \text{falls in } \tilde{\pi} = 1415 \dots \text{ mindestens } n\text{-mal} \\ & \text{hintereinander eine 7 vorkommt} \\ 0 & \text{sonst.} \end{cases}$$

Obwohl wir nicht wissen, ob  $h(n) = 1$  für jedes  $n$  ist, ist die Funktion  $h$  berechenbar! Begründung:

**Fall 1:** Es gibt in  $\tilde{\pi}$  beliebig lange Blöcke  $77 \dots 7$ . Dann ist  $h(n) = 1$  für alle  $n$  und  $h$  somit berechenbar.

**Fall 2:** Es gibt ein  $n_0$ , so dass in  $\tilde{\pi}$  Blöcke der Form  $77 \dots 7$  bis zur Länge  $n_0$  vorkommen, aber keine der Länge  $n_0 + 1$ . Dann gilt

$$h(n) = \begin{cases} 1 & \text{falls } n \leq n_0 \\ 0 & \text{sonst.} \end{cases}$$

Folglich ist  $h$  berechenbar, auch wenn wir  $n_0$  nicht kennen und nicht wissen, welcher Fall vorliegt.

*Das heißt, „Berechenbarkeit“ ist nicht-konstruktiv definiert: Es genügt, die Existenz eines Algorithmus für  $h$  zu beweisen; wir müssen ihn nicht explizit angeben können.*

4. Definiere die Funktion  $i : \mathbb{N} \rightarrow \{0, 1\}$  durch

$$i(n) = \begin{cases} 1 & \text{falls das 1. LBA Problem eine positive Lösung hat} \\ 0 & \text{sonst.} \end{cases}$$

*Die Funktion  $i$  ist berechenbar, auch wenn wir das 1. LBA Problem derzeit nicht lösen können. Denn entweder ist  $i \equiv 1$  oder  $i \equiv 0$ , und beide konstante Funktionen sind selbstverständlich berechenbar.*

Im ersten Beispiel oben ordneten wir der reellen Zahl  $\pi$  die Funktion  $f = f_\pi$  zu. Dies kann man entsprechend für jede reelle Zahl  $r$  tun und erhält so die Funktion  $f_r$ . Ist  $f_r$  für jede reelle Zahl  $r$  berechenbar?

Nein! Denn:

- Die Menge  $\mathbb{R}$  der reellen Zahlen ist überabzählbar. Genauso ist die Menge  $[0, 1)$  der reellen Zahlen im Intervall von 0 bis 1 überabzählbar.
- Sind  $r$  und  $r'$  reelle Zahlen in  $[0, 1)$  mit  $r \neq r'$ , so gilt  $f_r \neq f_{r'}$ . Somit haben  $f_r$  und  $f_{r'}$  verschiedene Berechnungsalgorithmen.
- Die Menge aller Algorithmen (egal in welcher fest gewählten Formalisierung) ist aber nur abzählbar unendlich, da sich ein jeder Algorithmus durch einen endlichen Text beschreiben lassen muss.

Es sind seit den 30er Jahren des 20. Jahrhunderts eine Reihe von Formalisierungen des Algorithmusbegriffs vorgeschlagen worden, beispielsweise:

- die Turing-Berechenbarkeit von Turing;
- der  $\lambda$ -Kalkül von Church und Rosser;
- die Markov-Berechenbarkeit von Markov;
- der Gleichungskalkül von Gödel und Herbrand;
- die partiell rekursiven Funktionen von Kleene,
- weitere Formalisierungen von Post und anderen.

Da sich all diese formalen Algorithmusbegriffe als äquivalent herausgestellt haben (siehe den Hauptsatz der Berechenbarkeitstheorie, Satz 9.12), nimmt man dies als ein starkes Indiz dafür, dass jeder einzelne dieser Begriffe tatsächlich exakt den intuitiven Berechenbarkeitsbegriff charakterisiert. Diese Behauptung ist bekannt als die These von Church.

**These 6.3 (Churchsche These)** *Die durch die formale Definition*

1. *der Turing-Berechenbarkeit,*
2. *der WHILE-Berechenbarkeit,*
3. *der GOTO-Berechenbarkeit,*
4. *der  $\mu$ -Rekursivität,*
5. *der Markov-Berechenbarkeit,*
6. *des  $\lambda$ -Kalküls*
7. *und einer Reihe von anderen Formalisierungen des Algorithmienbegriffs*

*beschriebenen Klassen von Funktionen stimmen jeweils genau mit der Klasse der im intuitiven Sinne berechenbaren Funktionen überein.*

Die These ist natürlich unmöglich zu beweisen, da der intuitive Begriff der Berechenbarkeit nicht formal definierbar ist. Die These ist allgemein akzeptiert.

Einige dieser Formalisierungen des Algorithmienbegriffs werden wir im Weiteren kennenlernen.

Historischer Kontext:

- um 1880: Georg Cantor begründet seine axiomatische Mengentheorie.
- 1900: David Hilbert unternimmt den Versuch, die gesamte Mathematik auf ein einheitliches axiomatisches Fundament zu stellen. Forschungsprogramm:
  - Vollständigkeit der Mathematik.
  - Widerspruchsfreiheit der Mathematik.
- 1901: Bertrand Russells Paradoxon: „Enthält die Menge  $R = \{X \mid X \notin X\}$  sich selbst als Element?“ verursacht zunächst viel Verwirrung, trägt aber dann dazu bei, die Axiome der Mengentheorie korrekt zu formulieren.
- 1902: Gottlieb Frege veröffentlicht sein einflussreiches Werk „Grundgesetze der Arithmetik“.
- 1931: Kurt Gödels Unvollständigkeitssätze beschäftigen sich ebenfalls mit dem Begriff des mathematischen Beweises und mit den Grundlagen der Logik. Sinngemäß und informal sagen sie das Folgende.

**1. Satz:** Wenn die axiomatische Mengentheorie widerspruchsfrei ist, dann gibt es in ihr Sätze, die innerhalb der Theorie weder bewiesen noch widerlegt werden können. Das heißt, die Widerspruchsfreiheit (Konsistenz) einer Theorie impliziert ihre Unvollständigkeit. In diesem Sinne ist jede hinreichend ausdrucksstarke Theorie unvollständig.

**2. Satz:** Es gibt kein konstruktives Verfahren, das die Konsistenz einer axiomatischen Theorie beweisen könnte.

Dies zeigt, dass Hilberts Programm scheitern muss.

- 1936: Alan Turing knüpft an Gödels Arbeit an und fragt nach der *algorithmischen* Entscheidbarkeit von Problemen. Er führt dazu das grundlegende Modell der Turingmaschine ein, das nach der Churchschen These genau das intuitiv Berechenbare erfasst.

# Kapitel 7

## Turing-Berechenbarkeit

Bisher wurden Turingmaschinen als Akzeptoren definiert. Nun wollen wir Turingmaschinen als Maschinen zur Berechnung von Funktionen auffassen.

**Definition 7.1 (Turing-Berechenbarkeit)** Eine Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  heißt Turing-berechenbar, falls es eine deterministische Turingmaschine  $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$  gibt, so dass für alle  $n_1, n_2, \dots, n_k, m \in \mathbb{N}$ :

$$f(n_1, n_2, \dots, n_k) = m \iff$$

$$\exists z \in F : z_0 \text{bin}(n_1) \# \text{bin}(n_2) \# \dots \# \text{bin}(n_k) \vdash_M^* z \text{bin}(m)$$

Falls  $f(n_1, n_2, \dots, n_k)$  nicht definiert ist, läuft  $M$  in eine unendliche Schleife oder stoppt in einem Zustand  $z \notin F$ .

### Beispiel 7.2 (Turing-Berechenbarkeit)

1. Die (total definierte) Nachfolgerfunktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  mit  $f : n \mapsto n + 1$  ist Turing-berechenbar.

Eine Turingmaschine, die  $f$  berechnet, ist definiert durch

$$M = (\{0, 1\}, \{0, 1, \square\}, \{z_0, z_1, z_2, z_e\}, \delta, z_0, \square, \{z_e\}),$$

wobei die Liste der Turingbefehle gemäß der Überföhrungsfunktion  $\delta$  in Tabelle 7.1 angegeben ist.

Tabelle 7.2 gibt die Bedeutung der einzelnen Zustände von  $M$  sowie die mit den einzelnen Zuständen verbundene Absicht an.

$(z_0, 0) \mapsto (z_0, 0, R)$	$(z_1, 0) \mapsto (z_2, 1, L)$	$(z_2, 0) \mapsto (z_2, 0, L)$
$(z_0, 1) \mapsto (z_0, 1, R)$	$(z_1, 1) \mapsto (z_1, 0, L)$	$(z_2, 1) \mapsto (z_2, 1, L)$
$(z_0, \square) \mapsto (z_1, \square, L)$	$(z_1, \square) \mapsto (z_e, 1, N)$	$(z_2, \square) \mapsto (z_e, \square, R)$

Tabelle 7.1: Liste  $\delta$  der Turingbefehle von  $M$  für die Funktion  $f(n) = n + 1$ 

$Z$	Bedeutung	Absicht
$z_0$	Anfangszustand	gehe bis zum Wortende und wechsele in Zustand $z_1$
$z_1$	addiere 1	mache eine 0 zu 1 bzw. alle 1en zu 0en
$z_2$	nach links laufen	gehe an den Wortanfang und wechsele in Zustand $z_e$
$z_e$	Endzustand	Akzeptieren

Tabelle 7.2: Interpretation der Zustände von  $M$ 

Für  $n = 5$  ergibt sich:

$$\begin{aligned}
z_0 101 &\vdash_M 1z_0 01 \\
&\vdash_M 10z_0 1 \\
&\vdash_M 101z_0 \square \\
&\vdash_M 10z_1 1 \\
&\vdash_M 1z_1 00 \\
&\vdash_M z_2 110 \\
&\vdash_M z_2 \square 110 \\
&\vdash_M z_e 110
\end{aligned}$$

Für  $n = 3$  ergibt sich:

$$\begin{aligned}
z_0 11 &\vdash_M 1z_0 1 \\
&\vdash_M 11z_0 \square \\
&\vdash_M 1z_1 1 \\
&\vdash_M z_1 10 \\
&\vdash_M z_1 \square 00 \\
&\vdash_M z_e 100
\end{aligned}$$

2. Die (total definierte) Funktion  $\text{div}_2 : \mathbb{N} \rightarrow \mathbb{N}$  mit  $\text{div}_2(n) = \lfloor \frac{n}{2} \rfloor$  ist Turing-berechenbar.

Idee: Falls  $n \geq 2$ , letzte Ziffer in  $\text{bin}(n)$  löschen.

Für  $n = 0$  ist nichts zu tun, für  $n = 1$  muss eine 0 aufs Band geschrieben werden.

Eine Turingmaschine, die  $\text{div}_2$  berechnet, ist definiert durch

$$M = (\{0, 1\}, \{0, 1, \square\}, \{z_0, z_1, z_2, z_3, z_4, z_5, z_e\}, \delta, z_0, \square, \{z_e\}),$$

$(z_0, 0) \mapsto (z_e, 0, N)$ $(z_0, 1) \mapsto (z_1, 1, R)$	$(z_1, 0) \mapsto (z_3, 0, R)$ $(z_1, 1) \mapsto (z_3, 1, R)$ $(z_1, \square) \mapsto (z_2, \square, L)$	$(z_2, 1) \mapsto (z_e, 0, N)$
$(z_3, 0) \mapsto (z_3, 0, R)$ $(z_3, 1) \mapsto (z_3, 1, R)$ $(z_3, \square) \mapsto (z_4, \square, L)$	$(z_4, 0) \mapsto (z_5, \square, L)$ $(z_4, 1) \mapsto (z_5, \square, L)$	$(z_5, 0) \mapsto (z_5, 0, L)$ $(z_5, 1) \mapsto (z_5, 1, L)$ $(z_5, \square) \mapsto (z_e, \square, R)$

Tabelle 7.3: Liste  $\delta$  der Turingbefehle von  $M$  für die Funktion  $\text{div}_2 = \lfloor \frac{n}{2} \rfloor$ 

wobei die Liste der Turingbefehle gemäß der Überföhrungsfunktion  $\delta$  in Tabelle 7.3 angegeben ist.

Tabelle 7.4 gibt die Bedeutung der einzelnen Zustände von  $M$  sowie die mit den einzelnen Zuständen verbundene Absicht an.

$Z$	Bedeutung	Absicht
$z_0$	Anfangszustand	falls Eingabe = 0, dann wechsele in $z_e$ , sonst in $z_1$
$z_1$	Eingabe = 1 testen	falls Eingabe = 1, dann wechsele in $z_2$ , sonst in $z_3$
$z_2$	Eingabe 1 zu 0 machen	(hier konnte nichts gelöscht werden)
$z_3$	nach rechts laufen	Wortende suchen und in $z_4$ wechseln
$z_4$	letztes Zeichen löschen	letztes Zeichen durch $\square$ ersetzen und in $z_5$ wechseln
$z_5$	nach links laufen	Wortanfang suchen und in $z_e$ wechseln
$z_e$	Endzustand	Akzeptieren

Tabelle 7.4: Interpretation der Zustände von  $M$ 

Für  $n = 0$  ergibt sich:

$$z_0 0 \vdash_M z_e 0$$

Für  $n = 1$  ergibt sich:

$$\begin{aligned} z_0 1 &\vdash_M 1 z_1 \square \\ &\vdash_M z_2 1 \\ &\vdash_M z_e 0 \end{aligned}$$

Für  $n = 2$  ergibt sich:

$$\begin{aligned} z_0 10 &\vdash_M 1 z_1 0 \square \\ &\vdash_M 10 z_3 \square \\ &\vdash_M 1 z_4 0 \\ &\vdash_M z_5 1 \square \\ &\vdash_M z_5 \square 1 \\ &\vdash_M z_e 1 \end{aligned}$$

Für  $n = 5$  ergibt sich:

$$\begin{aligned}
 z_0 101 &\vdash_M 1z_1 01 \\
 &\vdash_M 10z_3 1 \\
 &\vdash_M 101z_3 \square \\
 &\vdash_M 10z_4 1 \\
 &\vdash_M 1z_5 0\square \\
 &\vdash_M z_5 10 \\
 &\vdash_M z_5 \square 10 \\
 &\vdash_M z_e 10
 \end{aligned}$$

3. Die (partiell definierte) Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  mit  $f : n \rightarrow n - 1$  ist Turing-berechenbar. (s. Übungen)

Für Wortfunktionen  $f : \Sigma^* \rightarrow \Sigma^*$  definiert man den Begriff der Turing-Berechenbarkeit wie folgt.

**Definition 7.3 (Turing-Berechenbarkeit)** Eine Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  heißt Turing-berechenbar, falls es eine deterministische Turingmaschine  $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$  gibt, so dass für alle  $x, y \in \Sigma^*$ :

$$f(x) = y \iff \exists z \in F : z_0 x \vdash_M^* zy.$$

Falls  $f(x)$  nicht definiert ist, dann läuft  $M$  in eine unendliche Schleife oder stoppt in einem Zustand  $z \notin F$ .

**Beispiel 7.4 (Turing-Berechenbarkeit)** Die Wortfunktion  $\text{FIRST}_k : \Sigma^* \rightarrow \Sigma^*$  aus Definition 4.11 ist für jedes feste  $k$  Turing-berechenbar. (s. Übungen)



# Kapitel 8

## LOOP-, WHILE- und GOTO-Berechenbarkeit

### 8.1 LOOP-Berechenbarkeit

Im Gegensatz zu Turingmaschinen, wo die Formalisierung eines intuitiven Berechenbarkeitsbegriffes zugrunde liegt, werden wir im Folgenden Konzepte von prozeduralen Programmiersprachen kennenlernen. Als erste einfache Programmiersprache betrachten wir LOOP.

**Definition 8.1 (Syntax von LOOP-Programmen)** *LOOP-Programme bestehen aus:*

- *Variablen:*  $x_0, x_1, x_2, x_3, \dots$
- *Konstanten:*  $0, 1, 2, 3, \dots$
- *Trennsymbolen:*  $;$  und  $:=$
- *Operationen:*  $+$  und  $-$
- *Befehlen:* LOOP, DO, END

*Wir definieren die Syntax von LOOP-Programmen induktiv wie folgt:*

1.  $x_i := x_j + c$ ,  $x_i := x_j - c$  und  $x_i := c$ , für Konstanten  $c \in \mathbb{N}$ , sind LOOP-Programme.
2. Falls  $P_1$  und  $P_2$  LOOP-Programme sind, so ist auch  $P_1; P_2$  ein LOOP-Programm.
3. Falls  $P$  ein LOOP-Programm ist, so ist auch

LOOP  $x_i$  DO  $P$  END

ein LOOP-Programm. Dabei darf in der LOOP-Anweisung die Wiederholvariable  $x_i$  nicht im Schleifenkörper  $P$  vorkommen.

Um die Semantik von LOOP-Programmen zu beschreiben, stellen wir uns die Werte der Variablen in Registern gespeichert vor. Abstraktion: Es gibt unendlich viele Register unendlicher Kapazität, d.h., beliebig große Zahlen können in einem Register gespeichert werden.

**Definition 8.2 (Semantik von LOOP-Programmen)** In einem LOOP-Programm, das eine  $k$ -stellige Funktion  $f$  berechnen soll, gehen wir davon aus, dass dieses mit den Startwerten  $n_1, \dots, n_k \in \mathbb{N}$  in den Variablen  $x_1, \dots, x_k$  (und 0 in den restlichen) gestartet wird.

1. Die Zuweisungen  $x_i := x_j + c$  und  $x_i := c$  werden wie üblich interpretiert. In  $x_i := x_j - c$  wird  $x_i$  auf 0 gesetzt, falls  $c \geq x_j$  ist.
2. Das Programm  $P_1; P_2$  wird so interpretiert, dass zuerst  $P_1$  und dann  $P_2$  ausgeführt wird.
3. Das Programm  $P$  in

LOOP  $x_i$  DO  $P$  END

wird so oft ausgeführt, wie der Wert der Variablen  $x_i$  zu Beginn angibt. Da  $x_i$  nicht in  $P$  vorkommen darf, steht die Anzahl der Iterationen vor der ersten Ausführung der Schleife fest.

**Definition 8.3 (LOOP-Berechenbarkeit)** Eine Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  heißt LOOP-berechenbar, falls es ein LOOP-Programm  $P$  gibt, das gestartet mit  $n_1, \dots, n_k$  in den Variablen  $x_1, \dots, x_k$  (und 0 in den restlichen) stoppt mit dem Wert  $f(n_1, \dots, n_k)$  in der Variablen  $x_0$ .

#### Bemerkung 8.4

- Die Anweisung

IF  $x_1 = 0$  THEN  $P$  ELSE  $P'$  END

lässt sich wie folgt mit dem obigen Befehlssatz ausdrücken:

```

 $x_2 := 1; x_3 := 1;$ 
LOOP  $x_1$  DO  $x_2 := 0$  END;
LOOP  $x_2$  DO  $P; x_3 := 0$  END;
LOOP  $x_3$  DO  $P'$  END

```

- Die Anweisung

$$\text{IF } x_1 = c \text{ THEN } P \text{ ELSE } P' \text{ END}$$

lässt sich ähnlich mit LOOP-Befehlen ausdrücken (s. Übungen).

- Es ist nicht möglich, mit einem LOOP-Programm unendliche Schleifen zu programmieren. Das heißt, jedes LOOP-Programm stoppt nach endlich vielen Schritten. Somit sind LOOP-berechenbare Funktionen stets total.
- Da es nicht totale, aber intuitiv berechenbare Funktionen gibt, z.B.  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit  $f(n_1, n_2) = n_1 \text{ div } n_2$ , kann die Menge der LOOP-berechenbaren Funktionen nicht die Menge aller intuitiv berechenbaren Funktionen umfassen.
- Es gibt sogar total definierte Funktionen, die zwar intuitiv berechenbar, aber nicht LOOP-berechenbar sind (z.B. die so genannte Ackermann-Funktion, s. Beispiel 9.7).

### Beispiel 8.5 (LOOP-Berechenbarkeit)

1. Die Addition  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit  $f(n_1, n_2) = n_1 + n_2$  ist LOOP-berechenbar.

Idee: Berechne  $n_1 + n_2 = n_1 + \underbrace{1 + 1 + \dots + 1}_{n_2}$ .

```

 $x_0 := x_1 + 0;$ 
LOOP  $x_2$  DO
 $x_0 := x_0 + 1$ 
END

```

2. Die Multiplikation  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit  $f(n_1, n_2) = n_1 \cdot n_2$  ist LOOP-berechenbar.

Idee: Berechne  $n_1 \cdot n_2 = 0 + \underbrace{n_1 + n_1 + \dots + n_1}_{n_2}$ .

```

 $x_0 := 0;$ 
LOOP  $x_2$  DO
 $x_0 := x_0 + x_1$ 
END

```

Dabei wird  $x_0 := x_0 + x_1$  durch ein Unterprogramm gemäß (1) berechnet, d.h., hier entstehen zwei ineinander geschachtelte LOOP-Schleifen.

## 8.2 WHILE-Berechenbarkeit

Wir erweitern LOOP-Programme um eine WHILE-Schleife und nennen die so definierbaren Programme WHILE-Programme.

**Definition 8.6 (Syntax von WHILE-Programmen)** *WHILE-Programme sind wie folgt definiert:*

1. Jedes LOOP-Programm ist ein WHILE-Programm.
2. Falls  $P$  ein WHILE-Programm ist, so ist auch

$$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$$

*ein WHILE-Programm.*

3. Falls  $P_1$  und  $P_2$  WHILE-Programme sind, so ist auch  $P_1; P_2$  ein WHILE-Programm.

Zur Definition der Semantik von WHILE-Programmen verweisen wir auf Definition 8.2 und geben hier die Semantik der WHILE-Schleife an.

**Definition 8.7 (Semantik von WHILE-Programmen)** 1. Die Semantik von LOOP-Programmen wurde bereits in Definition 8.2 definiert.

2. Das Programm  $P$  in einer WHILE-Anweisung wird wiederholt, solange der Wert von  $x_i$  ungleich 0 ist.

**Bemerkung 8.8** Offensichtlich kann man

$$\text{LOOP } x_i \text{ DO } P \text{ END}$$

durch

$$\begin{aligned} &x_j := x_i + 0; \\ &\text{WHILE } x_j \neq 0 \text{ DO } x_j := x_j - 1; P \text{ END} \end{aligned}$$

simulieren, wobei  $P$  die Variable  $x_j$  nicht verwenden darf. Somit kann man in WHILE-Programmen auf LOOP-Schleifen verzichten.

**Definition 8.9 (WHILE-Berechenbarkeit)** Eine Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  heißt WHILE-berechenbar, falls es ein WHILE-Programm  $P$  gibt, das gestartet mit  $n_1, \dots, n_k$  in den Variablen  $x_1, \dots, x_k$  (und 0 in den restlichen) stoppt mit dem Wert  $f(n_1, \dots, n_k)$  in der Variablen  $x_0$  – sofern  $f(n_1, \dots, n_k)$  definiert ist, andernfalls stoppt  $P$  nicht.

**Korollar 8.10** Jede LOOP-berechenbare Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  ist WHILE-berechenbar.

**Beispiel 8.11 (WHILE-Berechenbarkeit)**

1. Die Potenz  $f : \mathbb{N} \rightarrow \mathbb{N}$  mit  $f(n) = 2^n$  ist WHILE-berechenbar.

Idee: Berechne  $2^n = 1 \cdot \underbrace{2 \cdot 2 \cdot \dots \cdot 2}_n$ .

```

 $x_0 := 1;$ 
WHILE  $x_1 \neq 0$  DO
 $x_0 := x_0 + x_0;$ 
 $x_1 := x_1 - 1$ 
END

```

2. Das WHILE-Programm

```

 $x_3 := x_1 - 4;$ 
WHILE  $x_3 \neq 0$  DO  $x_1 := x_1 + 1$  END;
LOOP  $x_1$  DO  $x_0 := x_0 + 1$  END;
LOOP  $x_2$  DO  $x_0 := x_0 + 1$  END

```

berechnet die Funktion  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit

$$f(n_1, n_2) = \begin{cases} n_1 + n_2 & \text{falls } n_1 \leq 4 \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Die Funktion  $f$  ist aber nicht LOOP-berechenbar, da  $f$  nicht total ist. Es gibt also WHILE-berechenbare Funktionen, die nicht LOOP-berechenbar sind.

**Satz 8.12** Jedes WHILE-Programm kann durch eine Turingmaschine simuliert werden, d.h., jede WHILE-berechenbare Funktion ist auch Turing-berechenbar. **ohne Beweis**

## 8.3 GOTO-Berechenbarkeit

**Definition 8.13 (Syntax von GOTO-Programmen)** GOTO-Programme bestehen aus Folgen von markierten Anweisungen:

$$M_1 : A_1; M_2 : A_2; \dots; M_m : A_m;$$

Anweisungen  $A_i$  dürfen dabei sein:

- *Zuweisung*:  $x_i := x_j + c$ ,  $x_i := x_j - c$  und  $x_i := c$ , für Konstanten  $c \in \mathbb{N}$
- *unbedingter Sprung*: GOTO  $M_i$
- *bedingter Sprung*: IF  $x_i = c$  THEN GOTO  $M_j$
- *Abbruchanweisung*: HALT

**Definition 8.14 (Semantik von GOTO-Programmen)**

- Mit der GOTO Anweisung springt man zu der Anweisung mit der angegebenen Marke.
- Die HALT Anweisung beendet ein GOTO Programm, d.h., die letzte Anweisung sollte entweder GOTO oder HALT sein.

**Bemerkung 8.15** GOTO-Programme können auch unendliche Schleifen enthalten, z.B.:

$M_1 : \text{GOTO } M_1$

**Definition 8.16 (GOTO-Berechenbarkeit)** Eine Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  heißt GOTO-berechenbar, falls es ein GOTO-Programm  $P$  gibt, das gestartet mit  $n_1, \dots, n_k$  in den Variablen  $x_1, \dots, x_k$  (und 0 in den restlichen) stoppt mit dem Wert  $f(n_1, \dots, n_k)$  in der Variablen  $x_0$  – sofern  $f(n_1, \dots, n_k)$  definiert ist, andernfalls stoppt  $P$  nicht.

**Beispiel 8.17 (GOTO-Berechenbarkeit)**

1. Die Addition  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit  $f(n_1, n_2) = n_1 + n_2$  ist GOTO-berechenbar.

Idee: Berechne  $n_1 + n_2 = n_1 + \underbrace{1 + 1 + \dots + 1}_{n_2}$ .

```

M1  :  x0 := x1 + 0;
M2  :  IF x2 = 0 THEN GOTO M6;
M3  :  x0 := x0 + 1;
M4  :  x2 := x2 - 1;
M5  :  GOTO M2;
M6  :  HALT;

```

2. Die Multiplikation  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit  $f(n_1, n_2) = n_1 \cdot n_2$  ist GOTO-berechenbar (s. Übungen).

**Satz 8.18** Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden, d.h., jede WHILE-berechenbare Funktion ist auch GOTO-berechenbar.

**Beweis.** Wir simulieren die Schleife

WHILE  $x_i \neq 0$  DO  $P$  END

durch

$$\begin{array}{ll} M_1 & : \text{ IF } x_i = 0 \text{ THEN GOTO } M_2; \\ \dots & P; \\ \dots & \text{ GOTO } M_1; \\ M_2 & : \dots \end{array}$$

■

**Satz 8.19** *Jedes GOTO-Programm kann durch ein WHILE-Programm mit einer WHILE-Schleife simuliert werden, d.h., jede GOTO-berechenbare Funktion ist auch WHILE-berechenbar.*

**Beweis.** Wir betrachten das GOTO-Programm  $P$ :

$$M_1 : A_1; M_2 : A_2; \dots; M_k : A_k;$$

Wir simulieren  $P$  durch folgendes WHILE-Programm mit einer zusätzlichen Variablen  $x_{\text{Sprung}}$  zur Simulation der Sprungmarken:

$$\begin{array}{l} x_{\text{Sprung}} := 1; \\ \text{WHILE } x_{\text{Sprung}} \neq 0 \text{ DO} \\ \quad \text{IF } x_{\text{Sprung}} = 1 \text{ THEN } B_1 \text{ END;} \\ \quad \dots \\ \quad \dots \\ \quad \text{IF } x_{\text{Sprung}} = k \text{ THEN } B_k \text{ END} \\ \text{END} \end{array}$$

Dabei ist  $B_i$ ,  $1 \leq i \leq k$ , wie folgt definiert:

$$B_i = \begin{cases} A_i; x_{\text{Sprung}} := x_{\text{Sprung}} + 1 & \text{falls } A_i \text{ eine Zuweisung ist} \\ x_{\text{Sprung}} := n & \text{falls } A_i = \text{GOTO } M_n \text{ ist} \\ \text{IF } x_j = c \text{ THEN } x_{\text{Sprung}} := n & \\ \text{ELSE } x_{\text{Sprung}} := x_{\text{Sprung}} + 1 \text{ END} & \text{falls } A_i = \text{IF } x_j = c \text{ THEN GOTO } M_n \text{ ist} \\ x_{\text{Sprung}} := 0 & \text{falls } A_i = \text{HALT ist} \end{cases}$$

Die Anweisung  $\text{IF } x = c \text{ THEN } P \text{ ELSE } P' \text{ END}$  kann nach Bemerkung 8.4 durch LOOP-Schleifen und somit auch durch WHILE-Schleifen simuliert werden. ■

**Korollar 8.20 (Kleenesche Normalform für WHILE-Programme)** *Jede WHILE-berechenbare Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  kann durch ein WHILE-Programm mit nur einer WHILE-Schleife (und mehreren IF-THEN-ELSE-Anweisungen) berechnet werden.*

**Beweis.** Es sei  $P$  ein beliebiges WHILE-Programm, das eine Funktion  $f$  berechnet. Nach Satz 8.18 gibt es ein GOTO-Programm  $P'$ , das  $f$  berechnet. Nach Satz 8.19 gibt es ein WHILE-Programm  $P''$  mit nur einer WHILE-Schleife und mehreren IF-THEN-ELSE-Anweisungen, das  $f$  berechnet. ■

**Satz 8.21** *Jede Turingmaschine kann durch ein GOTO-Programm simuliert werden, d.h., jede Turing-berechenbare Funktion ist auch GOTO-berechenbar.* **ohne Beweis**

### Bemerkung 8.22

- Aus Korollar 8.10 und den Sätzen 8.12, 8.19, 8.18, 8.21 folgen die in der Abbildung dargestellten Beziehungen zwischen den Mengen der Turing-, WHILE-, GOTO- und LOOP-berechenbaren Funktionen.

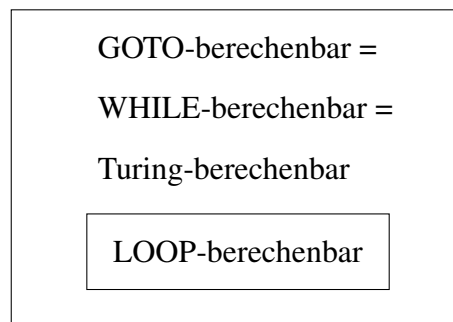


Abbildung 8.1: Beziehung zwischen den Mengen der Turing-, WHILE-, GOTO- und LOOP-berechenbaren Funktionen

- Es gibt WHILE-berechenbare Funktionen, die nicht LOOP-berechenbar sind (z.B. die Ackermann-Funktion, s. Beispiel 9.7).



# Kapitel 9

## Primitiv rekursive und partiell rekursive Funktionen

### 9.1 Primitiv rekursive Funktionen

Die Einführung der primitiven Rekursivität war, historisch gesehen, ein erster (und erfolgloser) Versuch, den Begriff der „Berechenbarkeit“ (oft synonym mit „Rekursivität“ verwendet) zu definieren. Erfolglos deshalb, weil die primitive Rekursion offensichtlich unterhalb dessen bleibt, was man intuitiv unter „berechenbar“ versteht. Dennoch ist es ein wichtiger Ausgangspunkt.

Man beginnt mit einfachsten Basisfunktionen und gibt Prinzipien an, wie aus diesen neue, kompliziertere („zusammengesetzte“) Funktionen konstruiert werden können. Da diese Prinzipien rekursiver Natur und die Basisfunktionen (intuitiv) berechenbar sind, müssen die so konstruierten komplizierteren Funktionen ebenfalls berechenbar sein. Die Hoffnung, auf diese Art irgendwann *alles* Berechenbare zu erfassen, hat sich allerdings nicht nur nicht bestätigt, sondern kann sogar widerlegt werden.

**Definition 9.1 (Primitiv rekursive Funktionen)** Die Klasse  $\mathbb{P}r$  der primitiv rekursiven Funktionen ist induktiv so definiert:

**(1) Induktionsbasis:**

- (1a)** Alle konstanten Funktionen (wie etwa die einstelligen konstanten Funktionen  $c \in \{0, 1, 2, \dots\}$ ,  $c : \mathbb{N} \rightarrow \mathbb{N}$  mit  $c(n) = c$  für alle  $n \in \mathbb{N}$ ) sind primitiv rekursiv.
- (1b)** Die Nachfolgerfunktion  $s : \mathbb{N} \rightarrow \mathbb{N}$ , definiert durch  $s(n) = n + 1$ , ist primitiv rekursiv.
- (1c)** Alle Identitäten (Projektionen)  $\text{id}_k^m : \mathbb{N}^m \rightarrow \mathbb{N}$ ,  $m \geq 0$ ,  $k \leq m$ , definiert durch  $\text{id}_k^m(n_1, n_2, \dots, n_m) = n_k$ , sind primitiv rekursiv.

**(2) Induktionsschritt:**

**(2a) Substitution (Komposition von Funktionen):** Sind die Funktionen  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  und  $g_1, g_2, \dots, g_k : \mathbb{N}^m \rightarrow \mathbb{N}$  für  $k, m \in \mathbb{N}$  in  $\mathbb{Pr}$ , so ist auch die Funktion  $h : \mathbb{N}^m \rightarrow \mathbb{N}$  in  $\mathbb{Pr}$ , die definiert ist durch

$$h(n_1, n_2, \dots, n_m) = f(g_1(n_1, n_2, \dots, n_m), g_2(n_1, n_2, \dots, n_m), \dots, g_k(n_1, n_2, \dots, n_m)).$$

**(2b) Primitive Rekursion:** Sind die Funktionen  $g$  und  $h$  in  $\mathbb{Pr}$ , wobei  $g : \mathbb{N}^m \rightarrow \mathbb{N}$  und  $h : \mathbb{N}^{m+2} \rightarrow \mathbb{N}$  für  $m \in \mathbb{N}$ , so ist auch die Funktion  $f : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$  in  $\mathbb{Pr}$ , die rekursiv definiert ist durch

$$\begin{aligned} f(0, x_1, x_2, \dots, x_m) &= g(x_1, x_2, \dots, x_m) \\ f(n+1, x_1, x_2, \dots, x_m) &= h(n, f(n, x_1, x_2, \dots, x_m), x_1, x_2, \dots, x_m). \end{aligned}$$

Dabei ist  $n$  die Rekursionsvariable, und die  $x_1, x_2, \dots, x_m$  heißen Parameter.

Anders gesagt ist die Klasse der primitiv rekursiven Funktionen also der Abschluss der in (1a), (1b) und (1c) definierten Basisfunktionen unter den in (2a) und (2b) definierten Operationen der Substitution und der primitiven Rekursion (die Funktionen in Funktionen überführen). Da die Basisfunktionen total und berechenbar (vorerst noch im intuitiven Sinne) sind und da das Normalschema der Substitution (2a) und das Normalschema der primitiven Rekursion (2b) sowohl die Totalität als auch die Berechenbarkeit von Funktionen erhalten, ist jede primitiv rekursive Funktion automatisch total und berechenbar.

**Beispiel 9.2 (Primitiv rekursive Funktionen)**

**Addition** Die Additionsfunktion  $\text{add} : \mathbb{N}^2 \rightarrow \mathbb{N}$ , die durch  $\text{add}(x, y) = x + y$  definiert ist, ist primitiv rekursiv, denn mit dem Normalschema der primitiven Rekursion lässt sie sich wie folgt beschreiben:

$$\begin{aligned} \text{add}(0, x) &= x &= \text{id}_1^1(x) \\ \text{add}(n+1, x) &= s(\text{add}(n, x)) &= s(\text{id}_2^3(n, \text{add}(n, x), x)) \end{aligned}$$

Die identische Funktion  $\text{id}_1^1$  entspricht dabei der Funktion  $g$  aus obiger Definition 9.1 (2b) und die Funktion  $s \circ \text{id}_2^3$  entspricht dabei der Funktion  $h$ .

Der Rekursionsanfang beruht auf der Tatsache, dass  $x + 0 = x$  für alle  $x \in \mathbb{N}$  gilt, und der Rekursionsschritt bedeutet  $x + (n+1) = (x+n) + 1$ . Also ist  $\text{add} \in \mathbb{Pr}$ .

**Multiplikation** Die Multiplikationsfunktion  $\text{mult} : \mathbb{N}^2 \rightarrow \mathbb{N}$ , definiert durch  $\text{mult}(x, y) = x \cdot y$ , ist primitiv rekursiv, denn mit dem Normalschema der primitiven Rekursion lässt sie sich wie folgt beschreiben:

$$\begin{aligned} \text{mult}(0, x) &= 0 \\ \text{mult}(n+1, x) &= \text{add}(\text{mult}(n, x), x) \\ &= f'(n, \text{mult}(n, x), x), \end{aligned}$$

wobei  $f'(a, b, c) = \text{add}(\text{id}_2^3(a, b, c), \text{id}_3^3(a, b, c))$ . Die konstante Funktion 0 entspricht dabei der Funktion  $g$  aus obiger Definition 9.1 (2b) und die Funktion  $f'$  entspricht dabei der Funktion  $h$ .

Die Funktion  $\text{add}$  ist primitiv rekursiv nach dem ersten Beispiel. Nach dem Normalschema der Substitution 9.1 (2a) ist die Funktion  $f'$  somit primitiv rekursiv. Der Rekursionsanfang beruht auf der Tatsache, dass  $x \cdot 0 = 0$  für alle  $x \in \mathbb{N}$  gilt, und der Rekursionsschritt bedeutet  $x \cdot (n + 1) = x \cdot n + x$ . Also ist  $\text{mult}$  eine primitiv rekursive Funktion.

**Exponentialfunktion** Die Exponentialfunktion  $\exp : \mathbb{N}^2 \rightarrow \mathbb{N}$  ist durch  $\exp(y, x) = x^y$  definiert, wobei wir in Abweichung vom mathematischen Standard  $x^0 = 1$  auch für  $x = 0$  setzen, um die Totalität der Funktion zu erreichen. Diese Funktion ist primitiv rekursiv, denn mit dem Normalschema der primitiven Rekursion lässt sie sich wie folgt beschreiben:

$$\begin{aligned} \exp(0, x) &= 1 \\ \exp(n + 1, x) &= \text{mult}(\exp(n, x), x) \\ &= f'(n, \exp(n, x), x), \end{aligned}$$

wobei  $f'(a, b, c) = \text{mult}(\text{id}_2^3(a, b, c), \text{id}_3^3(a, b, c))$ . Die konstante Funktion 1 entspricht dabei der Funktion  $g$  aus obiger Definition 9.1 (2b) und die Funktion  $f'$  entspricht dabei der Funktion  $h$ .

Die Funktion  $\text{mult}$  ist primitiv rekursiv nach dem zweiten Beispiel. Nach dem Normalschema der Substitution 9.1 (2a) ist die Funktion  $f'$  somit primitiv rekursiv. Der Rekursionsanfang beruht auf der Tatsache, dass  $x^0 = 1$  für alle  $x \in \mathbb{N}$  gilt, und der Rekursionsschritt bedeutet  $x^{n+1} = x^n \cdot x$ . Also ist  $\exp$  eine primitiv rekursive Funktion.

**Fakultät** Die Fakultätsfunktion  $\text{fa} : \mathbb{N} \rightarrow \mathbb{N}$  ist definiert durch

$$\text{fa}(x) = \begin{cases} 1 & \text{falls } x = 0 \\ 1 \cdot 2 \cdot \dots \cdot x & \text{falls } x \geq 1. \end{cases}$$

Diese Funktion ist primitiv rekursiv, denn mit dem Normalschema der primitiven Rekursion lässt sie sich wie folgt beschreiben:

$$\begin{aligned} \text{fa}(0) &= 1 \\ \text{fa}(n + 1) &= \text{mult}(\text{fa}(n), s(n)) \\ &= f'(n, \text{fa}(n)), \end{aligned}$$

wobei  $f'(a, b) = \text{mult}(\text{id}_2^2(a, b), s(\text{id}_1^2(a, b)))$ . Die konstante Funktion 1 entspricht dabei der Funktion  $g$  aus obiger Definition 9.1 (2b) und die Funktion  $f'$  entspricht dabei der Funktion  $h$ .

Die Funktion  $\text{mult}$  ist primitiv rekursiv nach dem zweiten Beispiel. Nach dem Normalschema der Substitution 9.1 (2a) ist die Funktion  $f'$  somit primitiv rekursiv. Der Rekursionsanfang beruht auf der Tatsache, dass  $\text{fa}(0) = 1$  gilt, und der Rekursionsschritt bedeutet  $(x+1)! = x! \cdot (x+1)$ . Also ist  $\text{fa}$  eine primitiv rekursive Funktion.

Die folgenden Funktionen sind ebenfalls primitiv rekursiv:

**Vorgängerfunktion** Die Vorgängerfunktion  $V : \mathbb{N} \rightarrow \mathbb{N}$  ist definiert durch

$$V(x) = \begin{cases} 0 & \text{falls } x = 0 \\ x - 1 & \text{falls } x \geq 1. \end{cases}$$

**modifizierte Differenz** Die modifizierte Differenz  $\text{md} : \mathbb{N}^2 \rightarrow \mathbb{N}$  ist definiert durch

$$\text{md}(x, y) = x \dot{-} y = \begin{cases} 0 & \text{falls } x < y \\ x - y & \text{falls } x \geq y. \end{cases}$$

**Abstand** Die Abstandsfunktion  $A : \mathbb{N}^2 \rightarrow \mathbb{N}$  ist definiert durch

$$A(x, y) = |x - y| = \begin{cases} x - y & \text{falls } y \leq x \\ y - x & \text{falls } x < y. \end{cases}$$

**Signumfunktion** Die Signumfunktion  $S : \mathbb{N} \rightarrow \mathbb{N}$  ist definiert durch

$$S(x) = \begin{cases} 0 & \text{falls } x = 0 \\ 1 & \text{falls } x \geq 1. \end{cases}$$

**Satz 9.3 (Dedekindscher Rechtfertigungssatz)** Es seien für  $m \geq 0$  die Funktionen  $g : \mathbb{N}^m \rightarrow \mathbb{N}$  und  $h : \mathbb{N}^{m+2} \rightarrow \mathbb{N}$  gegeben. Dann existiert genau eine Funktion  $f : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ , die Lösung des Normalschemas der primitiven Rekursion (2b) in Definition 9.1) ist. **ohne Beweis**

Da jede primitiv rekursive Funktion total ist, es aber natürlich nicht-totale Funktionen gibt, die (im intuitiven Sinne) berechenbar sind (z.B.  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ ,  $f(n_1, n_2) = n_1 \text{ div } n_2$ ), umfasst die Klasse  $\mathbb{P}r$  trivialerweise nicht alles intuitiv Berechenbare.

**Satz 9.4** Die Klasse der primitiv rekursiven Funktionen stimmt mit der Klasse der LOOP-berechenbaren Funktionen überein. **ohne Beweis**

## 9.2 Die Ackermann-Funktion

Interessanter ist die Frage, ob es *totale* berechenbare Funktionen gibt, die aber außerhalb von  $\mathbb{P}r$  liegen. Auch hier ist die Antwort positiv. Um dies zu beweisen, brauchen wir als technische Vorbereitung die so genannte *Gödelisierung* von  $\mathbb{P}r$ .

Eine solche Gödelisierung, auch *Gödelsches Wörterbuch* genannt, lässt sich analog für jede Klasse von Maschinen (TMs, PDAs, NFAs usw.) angeben, die eine syntaktische Beschreibung bzw. Formalisierung bestimmter Klassen von Algorithmen liefern.

Wir suchen eine Funktion  $G : \mathbb{P}r \rightarrow N$ , wobei  $N = \Sigma^*$  oder  $N$  die Menge der natürlichen Zahlen ist und

- $G$  injektiv und berechenbar,
- die Bildmenge  $G[\mathbb{P}r]$  „algorithmisch entscheidbar“ (formale Definition kommt später) und
- die Umkehrfunktion von  $G$  berechenbar ist.

$\mathbb{P}r$  steht hier für die Menge aller primitiv rekursiven Funktionsdefinitionen.

### Gödelisierung von $\mathbb{P}r$ :

#### 1. Über dem Alphabet

$$\Sigma = \{x, |, (, ), [, ], , , ;, *, s, 0, \text{id}, \text{SUB}, \text{PR}\}$$

lassen sich die Funktionen in  $\mathbb{P}r$  wie folgt darstellen:

- **Variablen:**  $x_1, x_2, \dots, x_i, \dots$  werden repräsentiert durch  $x|, x||, \dots, x \underbrace{|| \dots |}_{i\text{-mal}}, \dots$
- **Trennsymbole:**  $( ) [ ] , ; *$
- **Basisfunktionen:**  $s$  und  $0$ . Mit diesen beiden Funktionen lässt sich jede konstante Funktion  $c \in \{0, 1, 2, \dots\}$  darstellen. Zum Beispiel ist  $G(2) = s(s(0))$ . Wir haben auch  $G(s) = s$ .
- **Identitäten:**  $\text{id}_k^m$  wird dargestellt als  $\underbrace{\text{id} || \dots |}_{m\text{-mal}} * \underbrace{|| \dots |}_{k\text{-mal}}$ .
- **Substitutionen:** Werden die Funktionen  $g_1, g_2, \dots, g_k : \mathbb{N}^m \rightarrow \mathbb{N}$  in die Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  substituiert, so wird die resultierende Funktion  $h : \mathbb{N}^m \rightarrow \mathbb{N}$  dargestellt als

$$G(h) = \text{SUB}[G(f); G(g_1), G(g_2), \dots, G(g_k)](G(x_1), G(x_2), \dots, G(x_m)).$$

- **Primitive Rekursion:** Ergibt sich die Funktion  $f : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$  aus  $g : \mathbb{N}^m \rightarrow \mathbb{N}$  und  $h : \mathbb{N}^{m+2} \rightarrow \mathbb{N}$  durch primitive Rekursion, so stellen wir  $f$  dar als

$$G(f) = \text{PR}[G(g), G(h)](G(x_1), G(x_2), \dots, G(x_{m+1})).$$

Jedes  $f \in \mathbb{P}r$  lässt sich als ein Wort über dem Alphabet  $\Sigma$  darstellen. Beispielsweise kann man die Additionsfunktion  $\text{add} : \mathbb{N}^2 \rightarrow \mathbb{N}$  aus Beispiel 9.2 so als ein Wort über  $\Sigma$  darstellen.

### Beispiel 9.5 Betrachte

$$\begin{aligned} \text{add}(0, x) &= \text{id}_1^1(x) \\ \text{add}(n+1, x) &= h(n, \text{add}(n, x), x) \end{aligned}$$

mit  $h(n, y, z) = s(\text{id}_2^3(n, y, z))$ . Dann erhalten wir:

$$G(\text{add}) = \text{PR}[\text{id} \mid * \mid, \text{SUB}[s; \text{id} \mid \mid * \mid \mid](x \mid, x \mid \mid, x \mid \mid \mid)](x \mid, x \mid \mid).$$

Umgekehrt ist nicht jedes Wort  $w \in \Sigma^*$  eine syntaktisch korrekte Funktion in  $\mathbb{P}r$ .

2. Legen wir eine lineare Ordnung der Symbole in  $\Sigma$  fest, ergibt sich eine quasilexikographische Ordnung aller Wörter in  $\Sigma^*$ . Entfernen wir alle syntaktisch inkorrekten Wörter, so erhalten wir die Gödelisierung von  $\mathbb{P}r$ :

$$\psi_0, \psi_1, \psi_2, \dots$$

Jedes  $f \in \mathbb{P}r$  hat unendlich viele Gödelnummern, d.h., es gibt unendlich viele verschiedene  $i \in \mathbb{N}$ , so dass  $f = \psi_i$ . Beispielsweise gilt:

$$f = f + 0 = f + 0 + 0 = \dots,$$

und alle diese Funktionen sind syntaktisch verschieden und haben daher verschiedene Gödelnummern.

### Wichtige Eigenschaften der Gödelisierung:

1. Es gibt ein algorithmisches Verfahren, das zu gegebener Gödelnummer  $i$  die Funktion  $\psi_i$  (besser: das Wort, das diese Funktion beschreibt) bestimmt.
2. Es gibt ein algorithmisches Verfahren, das zu gegebenem Wort  $w \in \Sigma^*$  feststellt, ob  $w$  eine syntaktisch korrekte Funktion aus  $\mathbb{P}r$  beschreibt, und wenn ja, die Nr.  $i$  bestimmt, so dass  $\psi_i = w$ .

(Man braucht nämlich nur entsprechend obiger Vorschrift das Gödelsche Wörterbuch bis zur gegebenen Nr.  $i$  bzw. bis zum gegebenen Wort  $w$  aufzubauen.)

**Satz 9.6** *Es gibt eine totale, (intuitiv) berechenbare Funktion  $f$  mit  $f \notin \mathbb{P}r$ .*

**Beweis.** Sei  $\psi_0, \psi_1, \psi_2, \dots$  eine Gödelisierung aller einstelligen Funktionen in  $\mathbb{P}r$ . Setze  $f(i) = \psi_i(i) + 1$ . Aus der Annahme, dass  $f \in \mathbb{P}r$  ist, folgt, dass  $f = \psi_j$  für ein festes  $j$  gilt. Daraus ergibt sich der Widerspruch:  $\psi_j(j) = f(j) = \psi_j(j) + 1$ . ■

Nun geben wir ein konkretes „natürliches“ Beispiel für eine totale, (intuitiv) berechenbare Funktion an, die nicht primitiv rekursiv ist.

**Beispiel 9.7 (Ackermann-Funktion)** *Die Ackermann-Funktion  $\alpha : \mathbb{N}^2 \rightarrow \mathbb{N}$  ist definiert durch*

$$\alpha(m, n) = \begin{cases} n + 1 & \text{falls } m = 0 \text{ und } n \geq 0 & (1) \\ \alpha(m - 1, 1) & \text{falls } m > 0 \text{ und } n = 0 & (2) \\ \alpha(m - 1, \alpha(m, n - 1)) & \text{falls } m > 0 \text{ und } n > 0. & (3) \end{cases}$$

*Die Ackermann-Funktion ist eine totale und berechenbare Funktion, die nicht primitiv rekursiv ist. Sie war historisch die erste bekannte solche Funktion. Der Beweis, dass  $\alpha$  nicht primitiv rekursiv ist, beruht darauf, dass man zeigen kann, dass  $\alpha$  schneller wächst als jede primitiv rekursive Funktion. Beispielsweise ist*

$$\begin{aligned} \alpha(1, 2) &= \alpha(0, \alpha(1, 1)) & (3) \\ &= \alpha(1, 1) + 1 & (1) \\ &= \alpha(0, \alpha(1, 0)) + 1 & (3) \\ &= \alpha(1, 0) + 1 + 1 & (1) \\ &= \alpha(0, 1) + 2 & (2) \\ &= 1 + 1 + 2 & (1) \\ &= 4. \end{aligned}$$

Aber schon  $\alpha(3, 3) = 61$  und  $\alpha(4, 4) = 2^{2^{2^{16}}} - 3$ .

*Für alle  $n \in \mathbb{N}$  gilt:*

1.  $\alpha(0, n) = n + 1$
2.  $\alpha(1, n) = n + 2$
3.  $\alpha(2, n) = 2 \cdot n + 3$
4.  $\alpha(3, n) = 2^{n+3} - 3$

*Übung:* Füllen Sie die folgende Tabelle mit den Werten der Ackermannfunktion  $\alpha(m, n)$ ,  $0 \leq m \leq 3$ ,  $0 \leq n \leq 4$ :

$m \backslash n$	0	1	2	3	4
0					
1					
2					
3					

### 9.3 Allgemein und partiell rekursive Funktionen

**Frage:** Was fehlt den primitiv rekursiven Funktionen, um das intuitiv Berechenbare vollständig zu umfassen?

**Antwort:** Der  $\mu$ -Operator (Minimalisierung).

**Definition 9.8 ( $\mu$ -Operator)** Sei  $k \geq 0$ , und sei  $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$  eine Funktion. Durch Anwendung des  $\mu$ -Operators auf  $f$  entsteht die Funktion  $g : \mathbb{N}^k \rightarrow \mathbb{N}$ , die definiert ist als

$$g(x_1, x_2, \dots, x_k) = \min \left\{ n \in \mathbb{N} \mid \begin{array}{l} f(n, x_1, x_2, \dots, x_k) = 0 \text{ und für alle} \\ m < n \text{ ist } f(m, x_1, x_2, \dots, x_k) \text{ definiert} \end{array} \right\}.$$

Hier ist  $\min \emptyset$  nicht definiert, d.h., ist die Menge, über die minimiert wird, leer, so ist  $g$  an dieser Stelle nicht definiert.

Schreibweise:  $\mu f = g$  oder ausführlicher  $\mu n[f(n, \dots)] = g(\dots)$ .

#### Definition 9.9 (Partiell und allgemein rekursive Funktionen)

- Die Klasse  $\mathbb{P}$  der partiell rekursiven Funktionen (der  $\mu$ -rekursiven Funktionen) ist definiert als die kleinste Klasse von Funktionen, die die Basisfunktionen aus Definition 9.1 (die Konstanten, die Nachfolgerfunktion und die Identitäten) enthält und abgeschlossen ist unter Substitution, primitiver Rekursion und dem  $\mu$ -Operator.
- Die Klasse  $\mathbb{R}$  der allgemein rekursiven Funktionen ist definiert als

$$\mathbb{R} = \{f \mid f \in \mathbb{P} \text{ und } f \text{ ist total}\}.$$

Das folgende Beispiel gibt einige partiell und allgemein rekursive Funktionen an.



**Beispiel 9.10 (Partiell und allgemein rekursive Funktionen)** 1. Definiere die Funktion  $f$  wie folgt:

$$\begin{aligned} f(0, 0) &= 1 \\ f(1, 0) &= 1 \\ f(2, 0) &= \text{nicht definiert} \\ f(3, 0) &= 0 \\ f(m, 0) &= 1 \quad \text{für alle } m > 3. \end{aligned}$$

Ist  $g = \mu f$ , so ist  $g(0)$  nicht etwa gleich 3, sondern  $g(0)$  ist nicht definiert, weil  $f(2, 0)$  nicht definiert ist und die entsprechende Menge  $\{n \in \mathbb{N} \mid \dots\}$  somit leer ist.

2. Ist  $f(x, y) = x + y + 1$ , so gilt für alle  $x, y \in \mathbb{N}$ :  $f(x, y) \neq 0$ . Also ist  $g(y)$  für kein  $y \in \mathbb{N}$  definiert, also ist  $g = \mu f$  die nirgends definierte Funktion  $\Omega$ .

Somit können durch Anwendung des  $\mu$ -Operators partielle Funktionen entstehen.

3. Ist dagegen  $f(x, y) = x + y$ , so ist  $g = \mu f$  definiert durch

$$g(y) = \begin{cases} 0 & \text{falls } y = 0 \\ \text{undefiniert} & \text{falls } y > 0. \end{cases}$$

4. Die ganzzahlige Divisionsfunktion  $g : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit

$$g(y, z) = \begin{cases} \frac{z}{y} & \text{falls } y \neq 0 \text{ und } y \text{ teilt } z \\ \text{undefiniert} & \text{sonst} \end{cases}$$

ist partiell rekursiv. Dies kann man zeigen, indem man eine geeignete Funktion  $f : \mathbb{N}^3 \rightarrow \mathbb{N}$  angibt, so dass  $\mu f = g$  gilt.

Nach Beispiel 9.2 sind die Multiplikationsfunktion  $\text{mult}$ , die Signumfunktionen  $S$ , die Exponentialfunktion  $\text{exp}$  und die Abstandsfunktion  $A$  primitiv rekursiv. Somit ist die Funktion  $f : \mathbb{N}^3 \rightarrow \mathbb{N}$  mit

$$f(x, y, z) = |x \cdot y - z|^{S(y)} = \begin{cases} |x \cdot y - z| & \text{falls } y \neq 0 \\ 1 & \text{falls } y = 0 \end{cases}$$

ebenfalls primitiv rekursiv. Wann nimmt  $f$  den Wert 0 an? Dies ist genau dann der Fall, wenn  $y \neq 0$  und  $x \cdot y = z$  gilt. Wir wenden nun den  $\mu$ -Operator auf  $f$  an und

erhalten:

$$\begin{aligned}
 \mu x[f(x, y, z)] &= \begin{cases} \text{das kleinste } x \in \mathbb{N} \text{ mit } f(x, y, z) = 0 & \text{falls es existiert} \\ \text{undefiniert} & \text{sonst} \end{cases} \\
 &= \begin{cases} \text{das kleinste } x \in \mathbb{N} \text{ mit } x \cdot y = z \ (y \neq 0) & \text{falls es existiert} \\ \text{undefiniert} & \text{sonst} \end{cases} \\
 &= \begin{cases} \frac{z}{y} & \text{falls } y \neq 0 \text{ und } y \text{ teilt } z \\ \text{undefiniert} & \text{sonst} \end{cases} \\
 &= g(y, z)
 \end{aligned}$$

5. Die Funktion  $g : \mathbb{N} \rightarrow \mathbb{N}$  mit

$$g(m) = \begin{cases} \sqrt{m} & \text{falls } \sqrt{m} \in \mathbb{N} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

ist partiell rekursiv mittels  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit  $f(n, m) = |n^2 - m|$ , denn es gilt  $\mu n[f(n, m) = g(m)]$  (s. Übungen).

**Satz 9.11**  $\mathbb{P}r \subset \mathbb{R} \subset \mathbb{P}$ .

Der Beweis von Satz 9.11 ist trivial bis auf  $\mathbb{P}r \neq \mathbb{R}$ , was unmittelbar aus Satz 9.6 folgt. Wir werden jetzt zeigen, weshalb der Widerspruchsbeweis von Satz 9.6 für  $\mathbb{P}$  scheitert.

Sei  $\varphi_i$  die  $i$ -te einstellige Funktion in einer Gödelisierung von  $\mathbb{P}$ . Definiere die Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  durch  $f(i) = \varphi_i(i) + 1$ . Angenommen,  $f$  ist in  $\mathbb{P}$ . Dann existiert ein  $j$ , so dass  $f = \varphi_j$ . Bezeichnet  $D_h$  den Definitionsbereich einer Funktion  $h$ , so gilt also:

- $D_f = D_{\varphi_j}$  und
- $f(n) = \varphi_j(n)$  für alle  $n \in D_f$ .

Nun ist aber die Aussage

$$\varphi_j(j) = f(j) = \varphi_j(j) + 1 \tag{9.1}$$

kein Widerspruch mehr, denn an der Stelle  $j$  können  $f$  und  $\varphi_j$  undefiniert sein; dann ist natürlich auch  $\varphi_j(j) + 1$  nicht definiert. Die Gleichung (9.1) sagt also:

„nicht definiert = nicht definiert“.

Man überlege sich, weshalb der Widerspruchsbeweis von Satz 9.6 auch für  $\mathbb{R}$  scheitert. Kann man  $\mathbb{R}$  gödelisieren?

## 9.4 Der Hauptsatz der Berechenbarkeitstheorie

Der Hauptsatz der Berechenbarkeitstheorie, sagt dass alle eingangs genannten Algorithmenmodelle äquivalent sind. Nach der Churchschen These erfassen sie damit genau das im intuitiven Sinne Berechenbare.

**Satz 9.12 (Hauptsatz der Berechenbarkeitstheorie)** *Die folgenden Algorithmenbegriffe sind äquivalent in dem Sinne, dass die Klasse der von ihnen berechneten Funktionen mit der Klasse  $\mathbb{P}$  der partiell rekursiven Funktionen übereinstimmt:*

- Turingmaschinen,
- WHILE-Programme,
- GOTO-Programme,
- ...

Wir nennen nun Turing-, WHILE-, GOTO-, ...-berechenbare und partiell rekursive Funktionen im Folgenden kurz *berechenbare* Funktionen.

Aus Satz 9.4 und dem Hauptsatz 9.12 folgen die in Abbildung 9.1 dargestellten Beziehungen zwischen den Klassen der Turing-, WHILE-, GOTO- und LOOP-berechenbaren sowie der primitiv und partiell rekursiven Funktionen.

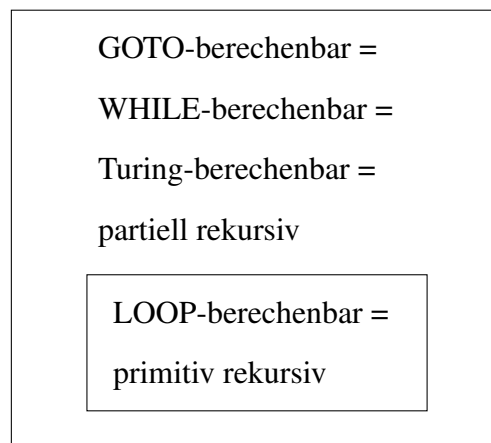


Abbildung 9.1: Beziehungen zwischen den Mengen der Turing-, WHILE-, GOTO- und LOOP-berechenbaren sowie der primitiv und partiell rekursiven Funktionen



# Kapitel 10

## Entscheidbarkeit und Aufzählbarkeit

### 10.1 Einige grundlegende Sätze

Es sei eine Gödelisierung  $\varphi_0, \varphi_1, \varphi_2, \dots$  der Klasse  $\mathbb{P}$  fixiert. Diese erhält man etwa durch eine Erweiterung der Gödelisierung  $\psi_0, \psi_1, \psi_2, \dots$  der Klasse  $\mathbb{Pr}$  um den  $\mu$ -Operator. Dazu äquivalent kann man eine Gödelisierung  $M_0, M_1, M_2, \dots$  aller Turingmaschinen angeben, wobei  $M_i$  die Funktion  $\varphi_i$  berechnet. Für jedes  $k \geq 0$  bezeichne  $\varphi_0^{(k)}, \varphi_1^{(k)}, \varphi_2^{(k)}, \dots$  eine Gödelisierung aller  $k$ -stelligen Funktionen in  $\mathbb{P}$ .

#### **Satz 10.1 (Aufzählbarkeitssatz; Satz von der universellen Funktion)**

*Es gibt eine zweistellige Funktion  $u \in \mathbb{P}$  (die so genannte „universelle Funktion“), so dass für alle  $i, x \in \mathbb{N}$ :  $u(i, x) = \varphi_i^{(1)}(x)$ .*

**Beweis.** Setze  $u(i, x) = \varphi_i^{(1)}(x)$ . Ist  $u$  berechenbar?

Ja, nämlich mit dem folgenden Algorithmus: Bei Eingabe von  $i$  und  $x$

1. berechne das Programm (d.h. die Turingmaschine  $M_i$ ) von  $\varphi_i$ ,
2. wende es auf die Eingabe  $x$  an und
3. gib den Funktionswert  $\varphi_i^{(1)}(x) = u(i, x)$  aus.

Es folgt, dass  $u \in \mathbb{P}$ . ■

**Satz 10.2 (Iterationssatz;  $s$ - $m$ - $n$ -Theorem)** *Für alle  $m, n \in \mathbb{N}$  gibt es eine  $(m + 1)$ -stellige Funktion  $s \in \mathbb{R}$ , so dass für alle  $i, x_1, \dots, x_n, y_1, \dots, y_m \in \mathbb{N}$ :*

$$\varphi_i^{(m+n)}(x_1, \dots, x_n, y_1, \dots, y_m) = \varphi_{s(i, y_1, \dots, y_m)}^{(n)}(x_1, \dots, x_n).$$

*(Hierbei fasst man die  $x_i$  als echte Variablen und die  $y_j$  als fixierte Parameter auf.)*

**Beweis.** Seien  $i \in \mathbb{N}$  und  $y_1, \dots, y_m \in \mathbb{N}$  gegeben. Betrachte

$$f(x_1, \dots, x_n) = \varphi_i^{(m+n)}(x_1, \dots, x_n, y_1, \dots, y_m)$$

als Funktion von  $x_1, \dots, x_n$ . Dann gibt es eine TM  $M$ , die  $f$  berechnet (und in deren Programm die Werte  $i$  und  $y_1, \dots, y_m$  hart codiert sind).

$M$  arbeitet bei Eingabe  $x_1, \dots, x_n$  so:

1. berechne das Programm (d.h. die Turingmaschine  $M_i$ ) von  $\varphi_i$ ,
2. simuliere  $M_i(x_1, \dots, x_n, y_1, \dots, y_m)$ .

$M$  hat natürlich das Ergebnis  $\varphi_i^{(m+n)}(x_1, \dots, x_n, y_1, \dots, y_m)$ .

Ist zum Beispiel  $f(x_1, \dots, x_n)$  nicht definiert, so hält  $M$  nie an. Es folgt  $f \in \mathbb{P}$ .

$M$  hat selbst eine Gödelnummer, sagen wir  $j$ , die von  $i$  und  $y_1, \dots, y_m$  abhängt. Für gegebenes  $i$  und  $y_1, \dots, y_m$  kann diese Nummer  $j$  algorithmisch bestimmt werden. Setzen wir

$$s(i, y_1, \dots, y_m) = j,$$

so gibt es also ein algorithmisches Verfahren zur Berechnung von  $s$ , d.h.,  $s \in \mathbb{R}$ , und es gilt:

$$f \equiv \varphi_j^{(n)} \equiv \varphi_{s(i, y_1, \dots, y_m)}^{(n)}.$$

Satz 10.2 ist bewiesen. ■

**Satz 10.3 (Kleenescher Fixpunktsatz)** Ist  $h \in \mathbb{R}$ , so existiert ein Fixpunkt  $a \in \mathbb{N}$  mit

$$(\forall x \in \mathbb{N}) [\varphi_{h(a)}(x) = \varphi_a(x)].$$

**Beweis.** Sei  $h \in \mathbb{R}$ . Da  $h$  somit auch in  $\mathbb{P}$  ist, gibt es eine Gödelnummer für  $h$ , etwa  $h = \varphi_i$ . Wir wenden den Iterationssatz auf zweistellige Funktionen in  $\mathbb{P}$  an. Nach Satz 10.2 existiert eine Funktion  $\sigma \in \mathbb{R}$  mit

$$\varphi_k(x, y) = \varphi_{\sigma(k, x)}(y). \tag{10.1}$$

Der Trick ist der folgende:

$$\begin{aligned} \varphi_{h(\sigma(x, x))}(y) &= v(i, x, y) && \text{mit } v \in \mathbb{P} \text{ und } v = \varphi_m \\ &= \varphi_m(i, x, y) \\ &= \varphi_{g(m, i)}(x, y) && \text{mit } g \in \mathbb{R} \text{ nach Satz 10.2} \\ &= \varphi_{s(i)}(x, y) && \text{denn } m \text{ ist ja fest, } s \in \mathbb{R} \\ &= \varphi_{\sigma(s(i), x)}(y) && \text{nach (10.1) mit } k = s(i). \end{aligned}$$

Also existieren Funktionen  $s, \sigma \in \mathbb{R}$ , so dass für alle  $x \in \mathbb{N}$  gilt:

$$\varphi_{h(\sigma(x,x))} \equiv \varphi_{\sigma(s(i),x)}. \quad (10.2)$$

Setze  $a = \sigma(s(i), s(i))$ . Da  $s, \sigma \in \mathbb{R}$ , existiert dieser Fixpunkt  $a$  stets. Aus (10.2) folgt mit  $x = s(i)$ :

$$\varphi_{h(a)} \equiv \varphi_{h(\sigma(s(i),s(i)))} \equiv \varphi_{\sigma(s(i),s(i))} \equiv \varphi_a.$$

Satz 10.3 ist bewiesen. ■

## 10.2 Entscheidbarkeit

Der Begriff Berechenbarkeit ist für Funktionen definiert. Für Sprachen wollen wir einen entsprechenden Begriff einführen.

**Definition 10.4 (Entscheidbarkeit)** Es sei  $A \subseteq \Sigma^*$  eine Menge (analog für  $A \subseteq \mathbb{N}$ ). Die charakteristische Funktion von  $A$  ist definiert durch:

$$\chi_A(x) = \begin{cases} 1 & \text{falls } x \in A \\ 0 & \text{falls } x \notin A. \end{cases}$$

$A$  heißt entscheidbar, falls  $\chi_A : \Sigma^* \rightarrow \{0, 1\}$  berechenbar ist. REC bezeichne die Klasse aller entscheidbaren Mengen.

Das heißt, eine Sprache  $A$  ist entscheidbar, falls es eine deterministische Turingmaschine gibt, die für jedes  $x \in \Sigma^*$  entscheidet, ob  $x \in A$  oder  $x \notin A$ .

**Definition 10.5 (Semi-Entscheidbarkeit)** Es sei  $A \subseteq \Sigma^*$  eine Menge (analog für  $A \subseteq \mathbb{N}$ ). Die partielle charakteristische Funktion von  $A$  ist definiert durch:

$$\chi'_A(x) = \begin{cases} 1 & \text{falls } x \in A \\ \text{nicht definiert} & \text{falls } x \notin A \end{cases}$$

$A$  heißt semi-entscheidbar, falls  $\chi'_A$  berechenbar ist.

Das heißt, eine Sprache  $A$  heißt semi-entscheidbar, falls es eine deterministische Turingmaschine  $M$  gibt, die  $A$  akzeptiert, d.h.,  $L(M) = A$ . Für  $x \in A$  stoppt die Maschine nach endlich vielen Schritten in einem Endzustand. Für  $x \notin A$  braucht die Maschine nicht zu stoppen – jedenfalls nicht in einem Endzustand. Hat die Maschine noch nicht gestoppt, so ist unklar, ob die Maschine noch stoppen wird ( $x \in A$ ) oder nicht ( $x \notin A$ ).

**Beispiel 10.6 (Entscheidbarkeit)** Die folgenden Mengen sind entscheidbar (und damit natürlich auch semi-entscheidbar), da sich leicht Algorithmen zur Berechnung ihrer charakteristischen Funktion angeben lassen:

1. die Menge der Quadratzahlen:  $A_1 = \{n^2 \mid n \in \mathbb{N}\} \in \text{REC}$ ,
2. die Menge der Zweierpotenzen:  $A_2 = \{2^n \mid n \in \mathbb{N}\} \in \text{REC}$ ,
3. die Menge der Primzahlen:  $A_3 = \{p \mid p \text{ Primzahl}\} \in \text{REC}$ .

**Satz 10.7**  $A$  ist entscheidbar  $\iff A$  und  $\bar{A} = \Sigma^* - A$  sind semi-entscheidbar.

**Beweis.**

( $\Rightarrow$ ) Eine Turingmaschine, die  $A$  entscheidet, kann leicht zu einer Turingmaschine modifiziert werden, die  $A$  bzw.  $\bar{A}$  akzeptiert.

( $\Leftarrow$ ) Nach Voraussetzung gibt es zwei Turingmaschinen  $M_A$  und  $M_{\bar{A}}$ , die die Sprachen  $A$  bzw.  $\bar{A}$  akzeptieren. Diese beiden Maschinen können wie folgt zu einer Maschine kombiniert werden, die für jedes  $x \in \Sigma^*$  entscheidet, ob  $x \in A$  oder  $x \notin A$ .

```

INPUT( $x$ );
FOR  $i = 1, 2, 3, \dots$  DO
    IF  $M_A$  hält bei Eingabe von  $x$  nach  $i$  Schritten THEN OUTPUT(1);
    IF  $M_{\bar{A}}$  hält bei Eingabe von  $x$  nach  $i$  Schritten THEN OUTPUT(0);
END

```

■

**Satz 10.8** REC ist abgeschlossen unter Schnitt, Vereinigung und Komplement, Konkatenation und Iteration.

**Beweis.**

1. Schnitt:  $\chi_{A \cap B}(x) = \min\{\chi_A(x), \chi_B(x)\} = \chi_A(x) \cdot \chi_B(x)$ .
2. Vereinigung:  $\chi_{A \cup B}(x) = \max\{\chi_A(x), \chi_B(x)\}$ .
3. Komplement:  $\chi_{\bar{A}}(x) = 1 - \chi_A(x)$ .
4. Konkatenation:  $\chi_{AB}(x) = \max_{x_1, x_2 \in \Sigma^*, x = x_1 x_2} \chi_A(x_1) \cdot \chi_B(x_2)$ .
5. Iteration:  $\chi_{A^n}(x) = \max_{x_1, \dots, x_n \in \Sigma^*, x = x_1 \dots x_n} \chi_A(x_1) \cdot \dots \cdot \chi_A(x_n)$ .



Satz 10.8 ist bewiesen. ■

Wie ordnet sich REC in die Chomsky-Hierarchie ein?

**Satz 10.9**  $CS \subset REC \subset \mathcal{L}_0$ .

**Beweis.**

$REC \subseteq \mathcal{L}_0$ . Es sei  $L$  entscheidbar. Dann gibt es eine Turingmaschine  $M$ , die  $\chi_L$  berechnet und somit entscheidet, ob  $x \in L$  oder  $x \notin L$  gilt.

Also ist  $L \in \{L(M) \mid M \text{ ist eine Turingmaschine}\} = \mathcal{L}_0$ .

$CS \subseteq REC$ . Sei  $L \in CS$ , und sei  $G = (\Sigma, N, S, P)$  eine Grammatik für  $L$  mit nur nicht-verkürzenden Regeln. Sei  $x \in \Sigma^*$  ein gegebenes Wort mit  $|x| = n$ .

**Idee:** Die „Zwischenergebnisse“  $x_i$  in einer beliebigen Ableitung

$$S \vdash_G x_1 \vdash_G x_2 \vdash_G \cdots \vdash_G x_k = x$$

haben alle die Länge  $|x_i| \leq n$ . Da es in  $(\Sigma \cup N)^*$  nur endlich viele Wörter der Länge  $\leq n$  gibt, kann man durch systematisches Durchprobieren entscheiden, ob  $x \in L$  oder  $x \notin L$  gilt, d.h.,  $L$  ist entscheidbar. (Somit ist das Wortproblem für Typ-1 Sprachen entscheidbar.)

**Formal:** Wir geben in Abbildung 10.1 einen Algorithmus an, der diese Entscheidung trifft. Dieser kann z.B. durch eine TM oder sonstwie implementiert werden.

Definiere für  $m, n \in \mathbb{N}$  die Mengen

$$T_m^n = \{w \in (\Sigma \cup N)^* \mid |w| \leq n \text{ und } S \vdash_G^m w\}.$$

Diese lassen sich, für festes  $n \geq 1$ , wie folgt induktiv über  $m$  definieren:

$$\begin{aligned} T_0^n &= \{S\} \\ T_{m+1}^n &= \text{Abl}^n(T_m^n), \end{aligned}$$

wobei für eine beliebige Menge  $X$  der Hüllenoperator  $\text{Abl}^n$  definiert ist durch

$$\text{Abl}^n(X) = X \cup \{w \in (\Sigma \cup N)^* \mid |w| \leq n \text{ und es existiert ein } v \in X \text{ mit } v \vdash_G w\}.$$

(Dies ist nur für Typ-1-Grammatiken korrekt. Bei Typ-0-Grammatiken könnte ein  $w$  mit  $|w| \leq n$  aus  $v$  mit  $|v| > n$  ableitbar sein.)

Der in Abbildung 10.1 dargestellte Algorithmus liefert die Entscheidung des Wortproblems für Typ-1-Grammatiken. Offenbar läuft dieser Algorithmus in Exponentialzeit.

```

Algorithmus-Typ-1( $G, x$ ) {
    //  $G$  ist Typ-1-Grammatik und
    //  $x \in \Sigma^*$  ein Wort mit  $|x| = n$ 

     $T := \{S\}$ ;
     $T_1 := \emptyset$ ;
    while ( $x \notin T$  und  $T \neq T_1$ ) {
         $T_1 := T$ ;
         $T := \text{Abl}^n(T_1)$ ;
    }
    if ( $x \in T$ ) return „ $x \in L$ “
    else return „ $x \notin L$ “
}

```

Abbildung 10.1: Algorithmus zur Entscheidung des Wortproblems für Typ-1-Grammatiken

Da es in  $(\Sigma \cup N)^*$  nur endlich viele Wörter der Länge  $\leq n$  gibt, folgt für jedes  $n$ , dass  $\bigcup_{m \geq 0} T_m^n$  eine endliche Menge ist (nämlich mit  $2^{c \cdot n}$  Elementen für ein von  $G$  abhängiges  $c$ ). Folglich existiert ein  $m_0$  mit

$$T_{m_0}^n = T_{m_0+1}^n = T_{m_0+2}^n = \dots = \bigcup_{m \geq 0} T_m^n.$$

Ist  $x \in L$ , so ist  $x \in \bigcup_{m \geq 0} T_m^n = T_{m_0}^n$ . Ist jedoch  $x \notin L$ , so ist  $x \notin T_{m_0}^n$ .

(CS  $\neq$  REC) Sei  $\Sigma = \{a, b\}$ . Wie definieren eine Sprache  $L \subseteq \Sigma^*$  mit  $L \in \text{REC}$ , aber  $L \notin \text{CS}$ . Dazu brauchen wir eine

### Gödelisierung aller Typ-1-Grammatiken mit dem Terminalalphabet $\Sigma = \{a, b\}$ :

- Die Nichtterminale seien durchnummeriert:  $X_0, X_1, X_2, \dots$ , wobei das Nichtterminal  $X_i$  dargestellt werde als  $X \underbrace{|| \dots ||}_{i\text{-mal}}$ .
- $X_0$  sei das Startsymbol.
- Regeln der Form

$$p_1 \rightarrow q_1, \quad p_2 \rightarrow q_2, \quad \dots, \quad p_n \rightarrow q_n$$

können durch das Wort

$$p_1 \rightarrow q_1 \# p_2 \rightarrow q_2 \# \cdots \# p_n \rightarrow q_n$$

über dem Alphabet  $\Gamma = \{X, |, a, b, \rightarrow, \#\}$  dargestellt werden.

- Nun codieren wir Wörter, die solche Typ-1-Grammatiken beschreiben, durch den Homomorphismus  $\varphi : \Gamma^* \rightarrow \Sigma^*$ :

$$\begin{array}{ll} \varphi(\lambda) &= \lambda \\ \varphi(a) &= bab \\ \varphi(b) &= ba^2b \end{array} \qquad \begin{array}{ll} \varphi(\rightarrow) &= ba^3b \\ \varphi(\#) &= ba^4b \\ \varphi(\underbrace{X || \cdots ||}_{i\text{-mal}}) &= ba^{5+i}b. \end{array}$$

(Ein Homomorphismus ist eine Abbildung  $h : \Gamma^* \rightarrow \Sigma^*$  mit  $h(xy) = h(x)h(y)$  und  $h(\lambda) = \lambda$ .)

Sind etwa die Regeln der Grammatik gegeben durch

$$X_0 \rightarrow \lambda, \quad X_0 \rightarrow X_1 X_2, \quad X_2 \rightarrow a, \quad X_1 X_2 \rightarrow b X_2,$$

so wird sie codiert durch das Wort

$$ba^5 bba^3 bba^4 bba^5 bba^3 bba^6 bba^7 bba^4 bba^7 bba^3 bba bba^4 bba^6 bba^7 bba^3 bba^2 bba^7 b.$$

- Eine Ordnung auf  $\Sigma$  (z.B.  $a < b$ ) induziert eine quasilexikographische Ordnung  $w_0, w_1, w_2, \dots$  auf  $\Sigma^*$ , wobei  $w_i$  mit  $i \in \mathbb{N}$  das  $i$ -te Wort ist:

$\Sigma^*$	$\lambda$	$a$	$b$	$aa$	$ab$	$ba$	$bb$	$aaa$	$\dots$
$i$ -tes Wort	$w_0$	$w_1$	$w_2$	$w_3$	$w_4$	$w_5$	$w_6$	$w_7$	$\dots$

- Entfernen wir nun alle Wörter, die keine syntaktisch korrekte Typ-1-Grammatik codieren, so erhalten wir die gesuchte Gödelisierung von CS:  $G_0, G_1, G_2, \dots$

Es ist dabei möglich, dass  $G_i = G_j$  für  $i \neq j$ , da eine Permutation der Regeln verschiedene Wörter  $w_i$  und  $w_j$  induziert.

Wie bei jeder Gödelisierung gilt:

- Jeder Typ-1-Grammatik  $G$  entspricht ein Wort  $w_G \in \Sigma^*$ .
- Für jedes Wort  $w \in \Sigma^*$  können wir algorithmisch entscheiden, ob es eine (syntaktisch korrekte) Typ-1-Grammatik codiert, und wenn ja, welche.

Nun definieren wir die Sprache  $L \subseteq \Sigma^*$  durch

$$L = \{w_i \mid i \in \mathbb{N} \text{ und } w_i \notin L(G_i)\}.$$

Beispielsweise ist  $\lambda \notin L$ , denn  $w_0 = \lambda$ , aber  $G_0$  ist gegeben durch die eine Regel  $X_0 \rightarrow \lambda$ , so dass  $\lambda \in L(G_0)$  gilt.

Dass  $L \in \text{REC}$  gilt, folgt unmittelbar aus der oben gezeigten Inklusion  $\text{CS} \subseteq \text{REC}$ . Für gegebenes  $x \in \Sigma^*$ :

- berechne das  $i$  mit  $x = w_i$ ;
- berechne die Grammatik  $G_i$  durch den Aufbau der Gödelisierung bis zur Nr.  $i$ ;
- entscheide mit dem Algorithmus aus Abbildung 10.1, ob  $x = w_i \notin L(G_i)$ .

Um zu zeigen, dass  $L \notin \text{CS}$ , nehmen wir für einen Widerspruch an, dass  $L \in \text{CS}$ . Dann existiert ein  $j \in \mathbb{N}$  mit  $L = L(G_j)$ . Daraus folgt

$$w_j \in L \iff w_j \notin L(G_j) = L,$$

ein Widerspruch.

( $\text{REC} \neq \mathcal{L}_0$ ) Diese Aussage zeigen wir spi $_{\mathcal{L}_0}^{\frac{1}{2}}$ ter. ■

Aus dem letzten Satz folgt insbesondere, dass die letzte (hier noch nicht betrachtete) Inklusion aus Fakt 1.7 echt ist.

Abbildung 10.2 zeigt wie sich die Menge aller entscheidbaren Sprachen in die Chomsky-Hierarchie einordnen lässt.

## 10.3 Rekursiv aufzählbare Mengen

### Definition 10.10 (Rekursive Aufzählbarkeit)

- Eine Menge  $A$  heißt rekursiv aufzählbar (kurz:  $A$  ist r.e., nach dem englischen recursively enumerable), falls entweder  $A = \emptyset$  oder  $A = W_f$  für ein  $f \in \mathbb{R}$ . Dabei bezeichnet  $W_f$  den Wertebereich von  $f$ , und  $f$  heißt Aufzählfunktion von  $A$ .
- RE bezeichne die Klasse aller rekursiv aufzählbaren Mengen.

**Bemerkung 10.11** • Eine Aufzählfunktion  $f$  für eine Menge  $A \in \text{RE}$  schöpft also den ganzen Wertebereich  $A$  aus, d.h.,  $A = \{f(i) \mid i \in \mathbb{N}\}$ .

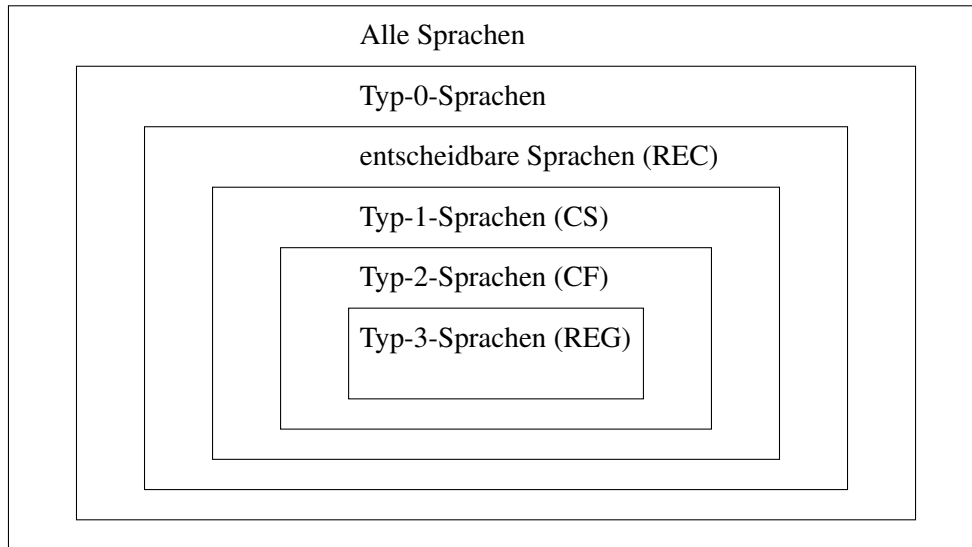


Abbildung 10.2: Einordnung der Menge aller entscheidbaren Sprachen in die Chomsky-Hierarchie

- *Rekursive Aufzählbarkeit darf nicht verwechselt werden mit dem Begriff der Abzählbarkeit. Diese verlangt keine effektive (algorithmische) Machbarkeit.*
- *Umgekehrt verlangt die rekursive Aufzählbarkeit keine 1-1-Zuordnung. Es ist möglich, dass  $f(i) = f(j)$  für  $i \neq j$ .*
- *Jede endliche Menge ist rekursiv aufzählbar.*

Eine Aufzählfunktion von  $A = \{a_0, \dots, a_n\}$  ist gegeben durch

$$f(i) = \begin{cases} a_i & \text{falls } 0 \leq i \leq n \\ a_n & \text{falls } i > n. \end{cases}$$

- *Jede Teilmenge einer abzählbaren Menge ist abzählbar.*
- *Jedoch ist nicht jede Teilmenge einer rekursiv aufzählbaren Menge auch rekursiv aufzählbar.*

**Satz 10.12**  $\text{REC} \subseteq \text{RE}$ .

**Beweis.** Sei  $A \in \text{REC}$ . Ist  $A = \emptyset$ , so ist  $A \in \text{RE}$  bereits gezeigt. Ist  $A \neq \emptyset$ , so wählen wir ein festes  $a \in A$  und konstruieren die Aufzählfunktion  $f$  für  $A$  so:

$$f(i) = \begin{cases} i & \text{falls } \chi_A(i) = 1 \text{ (d.h., } i \in A) \\ a & \text{sonst.} \end{cases}$$

Offenbar ist  $f$  total und berechenbar: Da  $A \in \text{REC}$ , ist  $\chi_A \in \mathbb{R}$  und somit auch  $f \in \mathbb{R}$ .

Es gilt  $A = W_f$ , denn:

- $A \subseteq W_f$ : Ist  $j \in A$ , so ist  $\chi_A(j) = 1$ , also  $f(j) = j$ , woraus  $j \in W_f$  folgt.
- $W_f \subseteq A$ : Ist  $j \in W_f$ , so ist entweder  $\chi_A(j) = 1$ , also  $j \in A$ , oder  $\chi_A(j) = 0$ , also  $j = a \in A$ .

Satz 10.12 ist bewiesen. ■

Wir werden später sehen, dass die Umkehrung „ $\text{RE} \subseteq \text{REC}$ “ *nicht* gilt.

**Satz 10.13**  $A \in \text{REC} \iff (A \in \text{RE} \wedge \overline{A} \in \text{RE})$ .

**Beweis.** ( $\Rightarrow$ ) Sei  $A \in \text{REC}$ . Nach Satz 10.12 ist  $A \in \text{RE}$ . Nach Satz 10.8 ist  $\text{REC}$  abgeschlossen unter Komplement, also  $\overline{A} \in \text{REC}$ . Wieder nach Satz 10.12 ist  $\overline{A} \in \text{RE}$ .

( $\Leftarrow$ ) Seien  $A$  und  $\overline{A}$  in  $\text{RE}$ . Ist  $A = \emptyset$  oder  $\overline{A} = \emptyset$  (d.h.,  $A = \Sigma^*$ ), so ist  $A \in \text{REC}$ .

Sei also  $\emptyset \neq A \neq \Sigma^*$ , und seien  $f, g \in \mathbb{R}$  Aufzählfunktionen mit  $A = W_f$  und  $\overline{A} = W_g$ . Der folgende Algorithmus berechnet  $\chi_A$  bei Eingabe  $x$ :

- Berechne  $f(0), g(0), f(1), g(1), f(2), \dots$  solange, bis  $x = f(i)$  oder  $x = g(i)$  für ein  $i$  gilt. (Alle diese Berechnungen terminieren wegen  $f, g \in \mathbb{R}$ .)
- Gilt  $x = f(i)$  für ein  $i$ , so gib 1 aus; gilt  $x = g(i)$  für ein  $i$ , so gib 0 aus.

Da entweder  $x \in A = W_f$  oder  $x \in \overline{A} = W_g$ , kann sich  $x$  nicht beliebig lange vor dieser Entscheidung drücken, sondern muss sich irgendwann „outen“. Folglich terminiert diese Prozedur in endlicher Zeit, und es gilt  $\chi_A \in \mathbb{R}$ . Es folgt  $A \in \text{REC}$ . ■

Mit ähnlichen Argumenten zeigt man den folgenden Satz. Wir werden später sehen, dass  $\text{RE}$  nicht komplementabgeschlossen ist.

**Satz 10.14**  $\text{RE}$  ist abgeschlossen unter Schnitt und Vereinigung.

**ohne Beweis**

Wir geben nun eine Reihe von weiteren Charakterisierungen der Klasse  $\text{RE}$  an.

**Satz 10.15**  $A \in \text{RE} \iff A$  ist semi-entscheidbar.

**ohne Beweis**

**Satz 10.16** Sei  $\varphi_0, \varphi_1, \varphi_2, \dots$  eine fixierte Gödelisierung der Klasse  $\mathbb{P}$ , und  $D_i = D_{\varphi_i}$  bzw.  $W_i = W_{\varphi_i}$  bezeichne den Definitions- bzw. den Wertebereich der  $i$ -ten Funktion  $\varphi_i \in \mathbb{P}$ . Dann gilt:

$$\text{RE} = \{D_i \mid i \in \mathbb{N}\} = \{W_i \mid i \in \mathbb{N}\}.$$

**Beweis.** 1. Sei  $A \in \text{RE}$ . Wir zeigen  $A = D_i$  und  $A = W_j$  für geeignete  $i, j \in \mathbb{N}$ .

Da  $A \in \text{RE}$ , gilt  $\chi'_A \in \mathbb{P}$  nach Satz 10.15. Genauer:

**Fall 1:**  $A = \emptyset$ . Dann ist  $\chi'_A$  die nirgends definierte Funktion, die natürlich in  $\mathbb{P}$  liegt.

**Fall 2:**  $A \neq \emptyset$ . Dann gibt es ein  $f \in \mathbb{R}$  mit  $W_f = A$ . Der folgende Algorithmus berechnet  $\chi'_A$  bei Eingabe  $x$ :

- Berechne  $f(0), f(1), f(2), \dots$  (also die Aufzählung von  $A$ ) solange, bis  $x = f(i)$  für ein  $i$  gilt. (Alle diese Berechnungen terminieren wegen  $f \in \mathbb{R}$ .)
- Gilt  $x = f(i)$  für ein  $i$ , so gib 1 aus.

Falls  $x$  nie vorkommt, weil  $x \notin A$ , so terminiert der obige Algorithmus nie.

Da  $\chi'_A \in \mathbb{P}$ , existiert eine Nr.  $i$  mit  $\chi'_A = \varphi_i$ . Somit ist  $A = D_{\chi'_A} = D_i$  gezeigt.

Definiere  $g(x) = x \cdot \chi'_A(x)$ . Es ist  $g(x) = x$ , falls  $x \in A$ , und  $g(x)$  ist nicht definiert, falls  $x \notin A$ . Da  $\chi'_A \in \mathbb{P}$ , ist auch  $g \in \mathbb{P}$ , und es gibt ein  $j$  mit  $\varphi_j = g$ . Somit gilt  $A = W_g = W_j$ . Wir haben also gezeigt, dass  $A \in \{D_i \mid i \in \mathbb{N}\}$  und  $A \in \{W_i \mid i \in \mathbb{N}\}$ . Somit:  $\text{RE} \subseteq \{D_i \mid i \in \mathbb{N}\}$  und  $\text{RE} \subseteq \{W_i \mid i \in \mathbb{N}\}$ .

2. Wir zeigen  $D_i \in \text{RE}$  für jedes  $i \in \mathbb{N}$ ; der Fall  $W_i \in \text{RE}$  wird analog bewiesen. Ist  $D_i = \emptyset$ , so gilt trivialerweise  $D_i \in \text{RE}$ .

Sei also  $D_i \neq \emptyset$ . Wähle ein festes  $a \in D_i$ . Definiere eine injektive Paarungsfunktion  $\pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  mit den folgenden Eigenschaften<sup>1</sup>:

- $\pi$  ist berechenbar.
- Die Umkehrfunktionen  $\pi_1, \pi_2 : \mathbb{N} \rightarrow \mathbb{N}$ , die für  $\pi(x, y) = n$  definiert sind durch  $\pi_1(n) = x$  und  $\pi_2(n) = y$ , sind ebenfalls berechenbar.
- $W_\pi \in \text{REC}$ .

Ein solches  $\pi$  kann so definiert werden:  $\pi(x, y) = 2^{x+y} + x$ . Anschaulich bedeutet dies:

$x$	0	1	2	3	4	...
$y$						
0	$2^0 = 1$	3	6	11	20	...
1	$2^1 = 2$	5	10	19	...	...
2	$2^2 = 4$	9	18	...	...	...
3	$2^3 = 8$	17	...	...	...	...
4	$2^4 = 16$	...	...	...	...	...
$\vdots$	$\vdots$	...	...	...	...	...

<sup>1</sup>Man kann auch bijektive Paarungsfunktionen mit diesen Eigenschaften angeben, z.B. die Funktion  $\tilde{\pi}(x, y) = \frac{1}{2}(x + y - 1)(x + y) + x + 2y$ .

Klar ist, dass  $\pi \in \mathbb{R}$  und  $W_\pi \in \text{REC}$ , denn es gilt

$$n \in W_\pi \iff 2^k + k \geq n,$$

wobei  $k$  die größte Zahl in  $\mathbb{N}$  mit  $2^k \leq n$  ist.

Wir zeigen  $\pi_1, \pi_2 \in \mathbb{P}$ : Für  $n \in W_\pi$  sei  $\pi(x, y) = n$ . Bestimme das größte  $k \in \mathbb{N}$  mit  $2^k \leq n$  und berechne  $x = n - 2^k$  und  $y = k - x$ .

Wir suchen nun ein  $f \in \mathbb{R}$  mit  $W_f = D_i$ . Dieses  $f$  werde durch den folgenden Algorithmus bei Eingabe  $n \in \mathbb{N}$  berechnet:

- Berechne  $x = \pi_1(n)$  und  $y = \pi_2(n)$ , falls  $n \in W_\pi$ . Andernfalls setze  $x = y = 0$ .
- Berechne aus der fixierten Nr.  $i$  das Programm der  $i$ -ten TM  $M_i$  in der fixierten Gödelisierung von  $\mathbb{P}$ .
- Simuliere die Berechnung von  $M_i$  auf Eingabe  $x$  für  $y$  Takte.
- Terminiert die Simulation, so gib  $x$  aus, sonst das fest gewählte Element  $a \in D_i$ .

Da der Algorithmus nur Elemente von  $D_i$  ausgibt, gilt  $W_f \subseteq D_i$ . Umgekehrt gilt auch  $D_i \subseteq W_f$ , denn wenn  $j \in D_i$ , so ist  $\varphi_i(j)$  definiert und  $M_i(j)$  hält nach  $t$  Takten an, für ein geeignetes  $t$ . Setze  $n = \pi(j, t)$ . Dann ist  $f(n) = j$  für ein geeignetes  $n$ , also  $j \in W_f$ . Somit gilt  $W_f = D_i$  für  $f \in \mathbb{R}$ . Es folgt  $D_i \in \text{RE}$ . ■

**Erinnerung:**  $\mathcal{L}_0$  ist die Klasse aller Sprachen, die durch Grammatiken erzeugt werden können, die keinerlei Einschränkung unterliegen.

**Satz 10.17**  $A \in \mathcal{L}_0 \iff A$  ist semi-entscheidbar.

ohne Beweis

Wir fassen die wichtigsten Charakterisierungen der Klasse RE zusammen.

**Korollar 10.18** Die folgenden Aussagen sind paarweise äquivalent.

1.  $A \in \text{RE}$ .
2.  $A$  ist semi-entscheidbar.
3.  $A$  ist vom Typ 0, d.h.,  $A \in \mathcal{L}_0$ .
4.  $\chi'_A \in \mathbb{P}$ .
5.  $A = L(M)$  für eine deterministische TM  $M$ .
6.  $A = L(M)$  für eine nichtdeterministische TM  $M$ .



7.  $A = D_f$  für ein  $f \in \mathbb{P}$ .

8.  $A = W_f$  für ein  $f \in \mathbb{P}$ .

**ohne Beweis**

Nun wollen wir das Verhältnis von RE und REC klären. Nach Satz 10.12 wissen wir bereits, dass  $\text{REC} \subseteq \text{RE}$ . Um zu zeigen, dass diese Inklusion echt ist, definieren wir nun das Halteproblem.

Bei diesem Problem kommen Turingmaschinen als Eingaben vor. Dazu erläutern wir zunächst, wie man Turingmaschinen  $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$  als ein Wort über dem Alphabet  $\{0, 1\}$  schreiben kann (Gödelisierung).

Wir nummerieren zunächst die Elemente aus  $Z$  und  $\Gamma$ , also

$$Z = \{z_0, z_1, z_2, \dots, z_n\}$$

und

$$\Gamma = \{a_0, a_1, a_2, \dots, a_k\}.$$

Jeder  $\delta$ -Regel

$$\delta(z_i, a_j) = (z_{i'}, a_{j'}, r)$$

ordnen wir ein Wort

$$w_{i,j,i',j',r} = \#\#\text{bin}(i)\#\text{bin}(j)\#\text{bin}(i')\#\text{bin}(j')\#\text{bin}(r)$$

zu, wobei

$$r = \begin{cases} 0 & \text{falls } r = L \\ 1 & \text{falls } r = R \\ 2 & \text{falls } r = N. \end{cases}$$

Für alle Regeln aus  $\delta$  schreiben wir diese Wörter in beliebiger Reihenfolge hintereinander und erhalten so eine Codierung von  $M$  über dem Alphabet  $\{0, 1, \#\}$ .

Um  $M$  über dem Alphabet  $\{0, 1\}$  zu codieren, nehmen wir die folgende Ersetzung vor:

$$\begin{array}{ll} 0 & \mapsto 00 \\ 1 & \mapsto 01 \\ \# & \mapsto 11 \end{array}$$

Das so erhaltene Wort zu Turingmaschine  $M$  bezeichnen wir mit  $\text{code}(M)$ .

**Beispiel 10.19** Wir geben eine Codierung der Turingmaschine

$$M = (\{0, 1\}, \{0, 1, \square\}, \{z_0, z_1, z_2, z_e\}, \delta, z_0, \square, \{z_e\})$$

aus Beispiel 7.2 an.

*Nummerierung der Zustände*

$z_0$	$z_1$	$z_2$	$z_e$
$z_0$	$z_1$	$z_2$	$z_3$

*und des Arbeitsalphabets*

0	1	$\square$
$a_0$	$a_1$	$a_2$

ergibt die folgende Codierung der Funktion  $\delta$

$\delta(z_i, a_j) = (z_{i'}, a_{j'}, r)$	$w_{i,j,i',j',r}$
$\delta(z_0, 0) = (z_0, 0, R)$	##0#0#0#0#1
$\delta(z_0, 1) = (z_0, 1, R)$	##0#1#0#1#1
$\delta(z_0, \square) = (z_1, \square, L)$	##0#10#1#10#0
$\delta(z_1, 0) = (z_2, 1, L)$	##1#0#10#1#0
$\delta(z_1, 1) = (z_1, 0, L)$	##1#1#1#0#0
$\delta(z_1, \square) = (z_e, 1, N)$	##1#10#11#1#10
$\delta(z_2, 0) = (z_2, 0, L)$	##10#0#10#0#0
$\delta(z_2, 1) = (z_2, 1, L)$	##10#1#10#1#0
$\delta(z_2, \square) = (z_e, \square, R)$	##10#10#11#10#1

Dies ergibt die Codierung

##0#0#0#0#1##0#1#0#1#1##0#10#1#10#0##1#0#10#1#0  
##1#1#1#0#0##1#10#11#1#10##10#0#10#0#0##10#1#10#1#0  
##10#10#11#10#1

von  $M$  über dem Alphabet  $\{0, 1, \#\}$ .

Mit der Ersetzung

0	$\mapsto$	00
1	$\mapsto$	01
#	$\mapsto$	11

erhalten wir  $\text{code}(M) = 1111001100110011001101 \dots$ .

Offensichtlich ist nicht jedes Wort über dem Alphabet  $\{0, 1\}$  eine so definierte Codierung einer Turingmaschine. Um die Umkehrabbildung der obigen Codierung anzugeben, sei  $M_0$  eine beliebige feste Turingmaschine. Dann ist für jedes Wort  $w \in \{0, 1\}^*$  eine Turingmaschine  $M_w$  definiert durch:

$$w \in \{0, 1\}^* \mapsto M_w = \begin{cases} M & \text{falls } w = \text{code}(M) \\ M_0 & \text{sonst} \end{cases}$$

**Definition 10.20 (spezielles Halteproblem)** Die Sprache

$$\begin{aligned} K &= \{w \in \{0, 1\}^* \mid M_w \text{ angesetzt auf } w \text{ hält nach endlich vielen Schritten}\} \\ &= \{i \in \mathbb{N} \mid i \in D_i\} \end{aligned}$$

wird als das spezielle Halteproblem bezeichnet.

**Satz 10.21** Das spezielle Halteproblem ist rekursiv aufzählbar, aber nicht entscheidbar, d.h.,  $K \in \text{RE}$  und  $K \notin \text{REC}$ .

**Beweis.** Der Beweis von  $K \in \text{RE}$  ist analog zum Beweis von Satz 10.16.

Um zu zeigen, dass  $K \notin \text{REC}$ , nehmen wir für einen Widerspruch  $K \in \text{REC}$  an. Dann ist die charakteristische Funktion  $\chi_K$  berechenbar mittels einer Turingmaschine  $M$  (s. Definition 10.4). Wir modifizieren  $M$  wie folgt zu einer Turingmaschine  $M'$ .

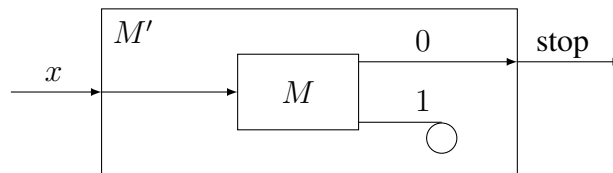


Abbildung 10.3: Zum Beweis von Satz 10.21

$M'$  hält, falls  $M$  eine 0 ausgibt, und  $M'$  geht in eine Endlosschleife, falls  $M$  eine 1 ausgibt, siehe Abbildung 10.3.

Es sei nun  $w' \in \{0, 1\}^*$  mit  $M_{w'} = M'$ , d.h.,  $w'$  sei das Codewort der Turingmaschine  $M'$ .

Dann gilt:

$$\begin{aligned} M' \text{ angesetzt auf } w' \text{ hält} &\Leftrightarrow M \text{ angesetzt auf } w' \text{ gibt 0 aus} && \text{(nach Def. von } M') \\ &\Leftrightarrow \chi_K(w') = 0 && \text{(nach Def. von } M) \\ &\Leftrightarrow w' \notin K && \text{(nach Def. von } \chi_K) \\ &\Leftrightarrow M_{w'} \text{ angesetzt auf } w' \text{ hält nicht} && \text{(nach Def. von } K) \\ &\Leftrightarrow M' \text{ angesetzt auf } w' \text{ hält nicht} && \text{(nach Def. von } M_{w'}) . \end{aligned}$$

Damit haben wir die Aussage

$$M' \text{ angesetzt auf } w' \text{ hält} \Leftrightarrow M' \text{ angesetzt auf } w' \text{ hält nicht}$$

hergeleitet, die offenbar einen Widerspruch darstellt. Damit ist die Annahme, dass  $K$  entscheidbar ist, falsch. ■

**Korollar 10.22** REC ist echt in RE enthalten. Insgesamt haben wir damit gezeigt, dass

$$\text{REG} \subset \text{DCF} \subset \text{CF} \subset \text{CS} \subset \text{REC} \subset \text{RE}$$

und die Chomsky-Hierarchie echt ist:  $\mathfrak{L}_3 \subset \mathfrak{L}_2 \subset \mathfrak{L}_1 \subset \mathfrak{L}_0$ .

**Korollar 10.23** RE ist nicht komplementabgeschlossen.

**Beweis.** Nach Satz 10.21 ist  $K \in \text{RE}$ , aber  $\overline{K} \notin \text{RE}$ , denn sonst wäre  $K \in \text{REC}$ . ■

**Behauptung 10.24** Es gibt Funktionen in  $\mathbb{P}$ , die nicht zu Funktionen in  $\mathbb{R}$  fortgesetzt werden können. **ohne Beweis**

Die obige Behauptung kann mittels einer Diagonalisierung bewiesen werden. Das heißt, man betrachtet die charakteristische Funktion des Halteproblems,  $\chi'_K$ , die natürlich in  $\mathbb{P}$  liegt. Dann füllt man die Lücken im Definitionsbereich von  $\chi'_K$  so, dass man an der  $i$ -ten Stelle gegen die  $i$ -te Funktion in  $\mathbb{P}$  diagonalisiert. Es folgt, dass die so definierte totale Funktion nicht berechenbar ist.

Abschließend geben wir den Projektionssatz an, der ebenfalls eine Charakterisierung von RE liefert. Projektionen sind geometrische Veranschaulichungen des Existenzquantors.

**Definition 10.25** Seien  $A \subseteq \mathbb{N}$  und  $B \subseteq \mathbb{N} \times \mathbb{N}$  Mengen.  $A$  ist Projektion von  $B$ , falls für alle  $x \in \mathbb{N}$  gilt:

$$x \in A \iff (\exists y \in \mathbb{N}) [(x, y) \in B].$$

**Satz 10.26 (Projektionssatz)**  $A \in \text{RE} \iff A$  ist Projektion einer Menge  $B \in \text{REC}$ .

**Beweis.** ( $\Rightarrow$ ) Sei  $A \in \text{RE}$ . Nach Satz 10.16 gibt es ein  $i \in \mathbb{N}$  mit  $A = D_i$ . Daraus folgt:

$$\begin{aligned} x \in A &\iff x \in D_i \\ &\iff \varphi_i(x) \text{ ist definiert, d.h., } (\exists y) [\varphi_i(x) = y] \\ &\iff M_i(x) \text{ hält} \\ &\iff (\exists y) [M_i(x) \text{ hält mit Ausgabe } y] \\ &\iff (\exists y) (\exists t) [M_i(x) \text{ hält nach } t \text{ Takten mit Ausgabe } y] \\ &\iff (\exists z) [z = \pi(y, t) \text{ und } T_i(x, z) \text{ ist erfüllt}] \\ &\iff (\exists z) [(x, z) \in B_i], \end{aligned}$$

wobei das so genannte Turingprädikat  $T_i(x, z)$  genau dann erfüllt ist, wenn  $M_i(x)$  nach  $t$  Takten mit Ausgabe  $y$  hält, wobei  $z = \pi(y, t)$ . Die Menge  $B_i$  ist für festes  $i$  so definiert:

$$B_i = \{(x, z) \mid z = \pi(y, t) \text{ und } T_i(x, z) \text{ ist erfüllt}\}.$$

Noch zu zeigen ist  $B_i \in \text{REC}$ . Dies leistet der folgende Algorithmus bei Eingabe  $(x, z)$ :

- Berechne  $y = \pi_1(z)$  und  $t = \pi_2(z)$ ; falls  $z \notin W_\pi$ , so setze  $y = t = 0$ ;
- berechne aus der bekannten Nr.  $i$  das Programm der  $i$ -ten TM  $M_i$ ;
- simuliere  $M_i(x)$  für  $\leq t$  Takte und teste, ob sie mit Ausgabe  $y$  anhält;
- falls ja, gib 1 aus; andernfalls gib 0 aus.

( $\Leftarrow$ ) Sei  $A$  Projektion einer Menge  $B \in \text{REC}$ . Das heißt, für alle  $x \in \mathbb{N}$  gilt:

$$x \in A \iff (\exists y \in \mathbb{N}) [(x, y) \in B].$$

Wir suchen eine Aufzählfunktion  $f \in \mathbb{R}$  für  $A$ , d.h.,  $W_f = A$ . Man muss von jedem  $x$  argwöhnen, dass es in  $A$  liegt. Ist das der Fall, so wird dies durch ein  $y \in \mathbb{N}$  mit  $(x, y) \in B$  bezeugt. Die Entscheidung, ob  $(x, y) \in B$  oder  $(x, y) \notin B$ , lässt sich mit dem Algorithmus für  $B$  treffen, das ja entscheidbar ist.

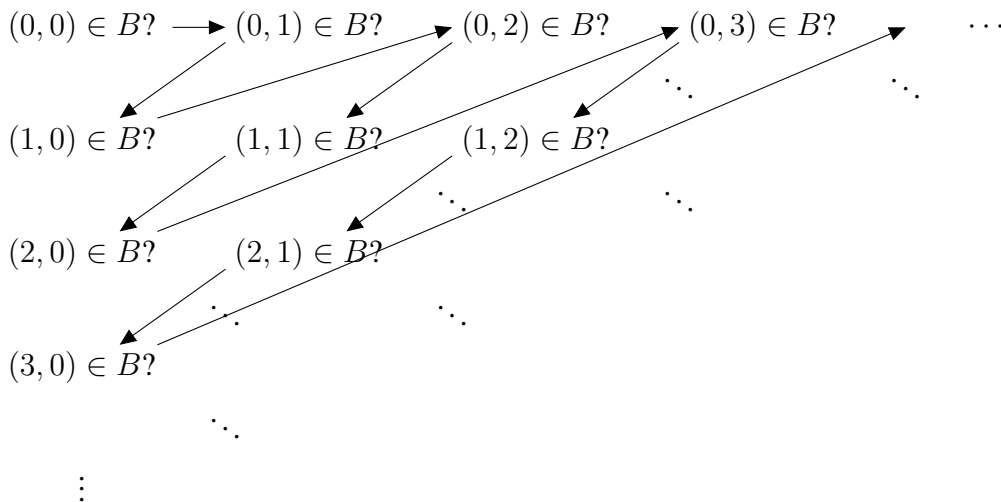


Abbildung 10.4: Rekursive Aufzählfunktion im Beweis des Projektionssatzes

Wie findet man für jedes  $x \in A$  einen Zeugen? Abbildung 10.4 zeigt, in welcher Reihenfolge die rekursive Aufzählfunktion  $f$  durch alle Paare geht und jeweils den Entscheidungsalgorithmus für  $B$  simuliert. Sagt dieser „ja“ für ein Paar  $(x, y)$ , so gibt  $f$  das Argument  $x$  aus, andernfalls einen fest gewählten Lückenbüßer  $a \in A$ . Dieser existiert unter der Annahme, dass  $A \neq \emptyset$ . Somit ist  $A \in \text{RE}$  via  $f \in \mathbb{R}$ . Ist  $A = \emptyset$ , so ist  $A$  nach Definition sowieso in RE. ■

**Anwendung des Projektionssatzes:** Nachweis der rekursiven Aufzählbarkeit, z.B.:

- $K$  ist in RE:

$$\begin{aligned}
 i \in K &\iff \varphi_i(i) \text{ ist definiert} \\
 &\iff (\exists t) [M_i(i) \text{ hält nach } t \text{ Takten}] \\
 &\iff (\exists t) [(i, t) \in B],
 \end{aligned}$$

wobei die Menge  $B \in \text{REC}$  so definiert ist:

$$B = \{(i, t) \mid M_i(i) \text{ hält nach } t \text{ Takten}\}.$$

- $X = \{i \in \mathbb{N} \mid D_i \neq \emptyset\}$  ist in RE:

$$\begin{aligned}
 i \in X &\iff D_i \neq \emptyset \\
 &\iff (\exists j) [j \in D_i] \\
 &\iff (\exists j) [\varphi_i(j) \text{ ist definiert}] \\
 &\iff (\exists j) (\exists t) [M_i(j) \text{ hält nach } t \text{ Takten}] \\
 &\iff (\exists z) [(i, z) \in B],
 \end{aligned}$$

wobei die Menge  $B \in \text{REC}$  so definiert ist:

$$B = \{(i, z) \mid z = \pi(j, t) \text{ und } M_i(j) \text{ hält nach } t \text{ Takten}\}.$$

- $Y = \{i \in \mathbb{N} \mid 17 \in W_i\}$  ist ebenso in RE.
- ...

Abbildung 10.5 zeigt wie sich die Klasse RE der rekursiv aufzählbaren (d.h. der semi-entscheidbaren) Sprachen und die Klasse REC der entscheidbaren Sprachen in die Chomsky-Hierarchie einordnen lassen.

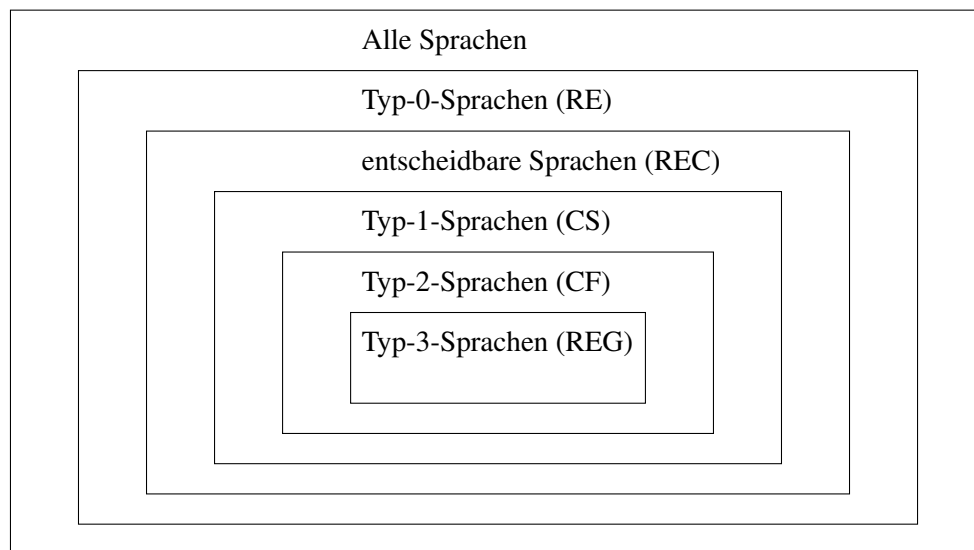


Abbildung 10.5: Einordnung von REC und RE in die Chomsky-Hierarchie





# Kapitel 11

## Unentscheidbarkeit

### 11.1 Der Satz von Rice

**Ziel:** Ein einfaches Kriterium für die Unentscheidbarkeit von Problemen.

**Definition 11.1** Sei  $G = (\varphi_0, \varphi_1, \varphi_2, \dots)$  eine fixierte Gödelisierung von  $\mathbb{P}$ .

- Jede Teilmenge  $F \subseteq \mathbb{P}$  ist eine Eigenschaft partiell rekursiver Funktionen.
- Die Nummernmenge (oder Indexmenge) von  $F \subseteq \mathbb{P}$  bzgl.  $G$  ist

$$N_F = \{i \in \mathbb{N} \mid \varphi_i \in F\}.$$

- Eine Eigenschaft  $F \subseteq \mathbb{P}$  heißt nichttrivial, falls  $N_F \neq \emptyset$  und  $N_F \neq \mathbb{N}$ .

**Satz 11.2 (Satz von Rice)** Die Nummernmenge einer jeden nichttrivialen Eigenschaft partiell rekursiver Funktionen ist unentscheidbar. Das heißt: Für jedes nichttriviale  $F \subseteq \mathbb{P}$  ist  $N_F \notin \text{REC}$ .

**Alternative Formulierung:** Ist  $\mathcal{A} \subseteq \text{RE}$  eine nichttriviale Teilmenge von  $\text{RE}$  (d.h.,  $\emptyset \neq \mathcal{A} \neq \text{RE}$ ), dann ist das folgende Problem unentscheidbar: Gegeben eine Turingmaschine  $M$  (als Wort, das  $M$  syntaktisch beschreibt), ist  $L(M) \in \mathcal{A}$ ?

**Interpretation:** Aus der Syntax von Programmen kann nichts Nichttriviales über ihre Semantik algorithmisch geschlossen werden!

**Beweis von Satz 11.2.** Sei  $F \subseteq \mathbb{P}$  eine nichttriviale Eigenschaft. Für einen Widerspruch nehmen wir an, die Nummernmenge  $N_F$  wäre in  $\text{REC}$ . Da  $\emptyset \neq N_F \neq \mathbb{N}$ , können wir feste Zahlen  $a \in N_F$  und  $b \notin N_F$  wählen.

Definiere die zweistellige Funktion

$$\alpha(i, x) = \begin{cases} \varphi_b(x) & \text{falls } i \in N_F \\ \varphi_a(x) & \text{falls } i \notin N_F. \end{cases}$$

Dann ist  $\alpha \in \mathbb{P}$  mit dem folgenden Algorithmus. Für gegebene  $i$  und  $x$ :

- entscheide, ob  $i \in N_F$  oder  $i \notin N_F$  (das geht nach Annahme  $N_F \in \text{REC}$ );
- berechne die Programme der Turingmaschinen  $M_a$  und  $M_b$ ;
- simuliere  $M_b(x)$ , falls  $i \in N_F$ ;
- simuliere  $M_a(x)$ , falls  $i \notin N_F$ .

Da  $\alpha \in \mathbb{P}$ , existiert ein  $m \in \mathbb{N}$  mit

$$\begin{aligned} \alpha(i, x) &= \varphi_m(i, x) \\ &= \varphi_{s(m, i)}(x) && \text{nach dem Iterationssatz (Satz 10.2), } s \in \mathbb{R} \\ &= \varphi_{g(i)}(x) && \text{da } m \text{ fest, } g \in \mathbb{R} \end{aligned}$$

Nach dem Kleeneschen Fixpunktsatz (Satz 10.3) existiert ein  $j \in \mathbb{N}$  mit  $\varphi_j = \varphi_{g(j)}$ .

Daraus folgt für alle  $x \in \mathbb{N}$ :

$$\varphi_j(x) = \varphi_{g(j)}(x) = \alpha(j, x) = \begin{cases} \varphi_b(x) & \text{falls } j \in N_F \\ \varphi_a(x) & \text{falls } j \notin N_F. \end{cases}$$

Damit erhalten wir den folgenden Widerspruch:

- Ist  $\varphi_j \in F$ , so ist  $j \in N_F$ , also gilt  $\varphi_j = \varphi_b$ , woraus  $\varphi_j \notin F$  mit  $b \notin N_F$  folgt.
- Ist  $\varphi_j \notin F$ , so ist  $j \notin N_F$ , also gilt  $\varphi_j = \varphi_a$ , woraus  $\varphi_j \in F$  mit  $a \in N_F$  folgt.

Zusammengefasst gilt also:

$$\varphi_j \in F \iff \varphi_j \notin F,$$

ein Widerspruch. Die Annahme war falsch, und es gilt  $N_F \notin \text{REC}$ . ■

Satz 11.2

### **Anwendung des Satzes von Rice:**

Nachweis der Unentscheidbarkeit von Problemen, z.B.:

- $A = \{x \mid \varphi_x \text{ ist total}\} \notin \text{REC}$ .

Denn es gibt *totale* ebenso wie *nicht-totale* partiell rekursive Funktionen. Also ist  $A$  die Nummernmenge einer nichttrivialen Eigenschaft von  $\mathbb{P}$  und nach dem Satz von Rice nicht entscheidbar.

- $B = \{x \mid \|W_x\| = \infty\} \notin \text{REC}.$

Denn es gibt partiell rekursive Funktionen mit *endlichem*, aber ebenso welche mit *unendlichem* Wertebereich. Also ist  $B$  die Nummernmenge einer nichttrivialen Eigenschaft von  $\mathbb{P}$  und nach dem Satz von Rice nicht entscheidbar.

- $C = \{(x, y) \mid y \in W_x\} \notin \text{REC}.$
- $D = \{(x, y) \mid \varphi_x \equiv \varphi_y\} \notin \text{REC}.$
- $E = \{(x, y, z) \mid \varphi_x(y) = z\} \notin \text{REC}.$
- $K = \{x \mid x \in D_x\} \notin \text{REC}.$
- $\overline{K} = \{x \mid x \notin D_x\} \notin \text{REC}.$
- $X = \{i \in \mathbb{N} \mid D_i \neq \emptyset\} \notin \text{REC}.$
- $Y = \{x \mid 17 \in W_x\} \notin \text{REC}.$
- ...

## 11.2 Reduzierbarkeit

**Ziel:** Durch (berechenbare) Reduktionen wollen wir ungelöste Probleme auf bereits gelöste Probleme zurückführen. Damit erhalten wir ein weiteres Unentscheidbarkeitskriterium.

**Definition 11.3 (Reduzierbarkeit)** Seien  $A, B \subseteq \mathbb{N}$  Mengen.  $A$  ist many-one-reduzierbar auf  $B$  (symbolisch:  $A \leq_m B$ ), falls gilt:

$$(\exists f \in \mathbb{R}) (\forall x \in \mathbb{N}) [x \in A \iff f(x) \in B].$$

**Lemma 11.4** Die folgenden Aussagen sind paarweise äquivalent:

1.  $A \leq_m B$  via  $f \in \mathbb{R}.$
2.  $A = f^{-1}(B)$ , wobei  $f^{-1}(B) = \{x \in \mathbb{N} \mid f(x) \in B\}.$
3.  $f(A) \subseteq B$  und  $f(\overline{A}) \subseteq \overline{B}$ , wobei  $f(A) = \{f(x) \in \mathbb{N} \mid x \in A\}.$
4.  $\chi_A = \chi_B \circ f.$

**ohne Beweis**

Das folgende Lemma gibt einige einfach zu beweisende, aber grundlegende Eigenschaften der many-one-Reduzierbarkeit an.

**Lemma 11.5** Seien  $A, B \subseteq \mathbb{N}$  Mengen.

1. Die Relation  $\leq_m$  ist reflexiv und transitiv, aber nicht antisymmetrisch.
2. Gilt  $A \leq_m B$  und ist  $B$  entscheidbar, so ist auch  $A$  entscheidbar.
3. Gilt  $A \leq_m B$  und ist  $B$  semi-entscheidbar, so ist auch  $A$  semi-entscheidbar.
4. Im Allgemeinen gilt nicht:  $\overline{A} \leq_m A$ .
5.  $A \leq_m B \implies \overline{A} \leq_m \overline{B}$ . ohne Beweis

Die Aussage 2 von Lemma 11.5 sagt insbesondere, dass sich Entscheidbarkeit bzgl. der  $\leq_m$ -Reduzierbarkeit nach unten vererbt. Wir benutzen diese Eigenschaft in ihrer Kontraposition als das

**Lemma 11.6 (Unentscheidbarkeitskriterium)** Unentscheidbarkeit vererbt sich bzgl. der  $\leq_m$ -Reduzierbarkeit nach oben:

$$(A \leq_m B \wedge A \notin \text{REC}) \implies B \notin \text{REC}.$$

Diese Kriterium kann man nutzen um die Unentscheidbarkeit des folgenden Problems zu zeigen.

**Definition 11.7 (allgemeines Halteproblem)** Die Sprache

$$H = \{w\#x \in \{0, 1, \#\}^* \mid M_w \text{ angesetzt auf } x \text{ h\u00e4lt nach endlich vielen Schritten}\}$$

wird als das allgemeine Halteproblem bezeichnet.

Alternative Definition:

$$H = \{(i, j) \in \mathbb{N}^2 \mid i \in D_j\}$$

**Satz 11.8** Das Halteproblem ist nicht entscheidbar, d.h.,  $H \notin \text{REC}$ .

**Beweis.** Wenn wir zeigen, dass  $K \leq_m H$  gilt, folgt die Aussage des Satzes aus Satz 10.21 und Lemma 11.6.

Die Funktion  $f : \{0, 1\}^* \rightarrow \{0, 1, \#\}^*$  mit  $f(w) = w\#w$  ist total und berechenbar. Es gilt f\u00fcr alle  $w \in \{0, 1\}^*$ :

$$\begin{aligned} w \in K &\iff M_w \text{ angesetzt auf } w \text{ h\u00e4lt} && \text{(nach Def. von } K) \\ &\iff w\#w \in H && \text{(nach Def. von } H) \\ &\iff f(w) \in H && \text{(nach Def. von } f) \end{aligned}$$

Damit gilt  $w \in K \Leftrightarrow f(w) \in H$ , also  $K \leq_m H$ . ■

Es gibt somit keine algorithmische Methode, die entscheidet, ob ein Programm bei einer Eingabe hält.

Alternativ kann Satz 11.8 für  $\frac{1}{2}$  für  $H = \{(i, j) \in \mathbb{N}^2 \mid i \in D_j\}$  so bewiesen werden: Definiere die Reduktion  $f(i) = (i, i)$ . Dann gilt:

$$\begin{aligned} i \in K &\iff i \in D_i \\ &\iff (i, i) = f(i) \in H. \end{aligned}$$

Da offenbar  $f \in \mathbb{R}$ , ist  $K \leq_m H$  via  $f$ . Mit Satz 10.21 und Lemma 11.6 folgt  $H \notin \text{REC}$ .

**Definition 11.9 (Halteproblem auf leerem Band)** Die Sprache

$$H_0 = \{w \mid M_w \text{ angesetzt auf } \lambda \text{ hält nach endlich vielen Schritten}\}$$

wird als das Halteproblem auf leerem Band bezeichnet.

**Satz 11.10** Das Halteproblem auf leerem Band ist nicht entscheidbar, d.h.,  $H_0 \notin \text{REC}$ .

**Beweis.** Es genügt zu zeigen, dass  $H \leq_m H_0$  gilt.

Wir ordnen jedem Wort  $z$  aus  $\{0, 1, \#\}^*$  wie folgt eine deterministische Turingmaschine  $M$  zu.

- Falls  $z$  nicht von der Form  $w\#x$  mit  $w, x \in \{0, 1\}^*$  ist, so geht  $M$  in eine Endlosschleife.
- Falls  $z$  von der Form  $w\#x$  mit  $w, x \in \{0, 1\}^*$  ist, dann schreibt  $M$  das Wort  $x$  auf das Band und arbeitet dann, angesetzt auf  $x$ , wie  $M_w$ .

Wir definieren die berechenbare totale Funktion  $f : \{0, 1, \#\}^* \rightarrow \{0, 1\}^*$  durch

$$f(z) = \text{code}(M).$$

Es gilt für alle  $z \in \{0, 1, \#\}^*$ :

$$\begin{aligned} z \in H &\iff z = w\#x \text{ und } M_w \text{ angesetzt auf } x \text{ hält} && \text{(nach Def. von } H) \\ &\iff f(z) = \text{code}(M) \text{ und } M \text{ angesetzt auf } \lambda \text{ hält} && \text{(nach Def. von } M \text{ und } f) \\ &\iff f(z) = \text{code}(M) \in H_0 && \text{(nach Def. von } H_0) \end{aligned}$$

Damit gilt  $z \in H \Leftrightarrow f(z) \in H_0$ . ■

**Behauptung 11.11** *Das Problem*

$$X = \{i \in \mathbb{N} \mid D_i \neq \emptyset\}$$

ist nicht entscheidbar.

**Beweis.** Wir zeigen:  $K \leq_m X$ . Da  $K \notin \text{REC}$  nach Satz 10.21, folgt aus dem Unentscheidbarkeitskriterium  $X \notin \text{REC}$ .

Sei die Funktion  $\alpha : \mathbb{N}^2 \rightarrow \mathbb{N}$  definiert durch

$$\alpha(i, x) = \begin{cases} 1 & \text{falls } i \in K \\ \text{nicht definiert} & \text{sonst.} \end{cases}$$

Da  $\alpha \in \mathbf{P}$ , gibt es ein  $j \in \mathbb{N}$  mit  $\varphi_j = \alpha$ .

Nach dem Iterationssatz (Satz 10.2) existiert eine Funktion  $s \in \mathbf{R}$ , so dass gilt:

$$\alpha(i, x) = \varphi_j(i, x) = \varphi_{s(j,i)}(x) = \varphi_{g(i)}(x),$$

wobei die letzte Gleichheit für ein geeignetes  $g \in \mathbf{R}$  gilt, da  $j$  fest ist. Es folgt:

$$D_{g(i)} = \begin{cases} \mathbb{N} & \text{falls } i \in K \\ \emptyset & \text{falls } i \notin K. \end{cases}$$

Somit gilt für alle  $i \in \mathbb{N}$ :

$$D_{g(i)} = \mathbb{N} \iff D_{g(i)} \neq \emptyset.$$

Also gilt für alle  $i \in \mathbb{N}$ :

$$\begin{aligned} i \in K &\iff D_{g(i)} \neq \emptyset \\ &\iff g(i) \in X. \end{aligned}$$

Folglich ist  $K \leq_m X$  via  $g$ . ■

Ebenso gilt:  $H \leq_m K$  und  $X \leq_m K$ . Wir zeigen die erste Behauptung:

Sei die Funktion  $\alpha : \mathbb{N}^3 \rightarrow \mathbb{N}$  definiert durch

$$\alpha(i, j, x) = \begin{cases} 1 & \text{falls } (i, j) \in H \\ \text{nicht definiert} & \text{sonst.} \end{cases}$$

Da  $\alpha \in \mathbf{P}$ , existiert nach dem Iterationssatz (Satz 10.2) eine Funktion  $h \in \mathbf{R}$ , so dass gilt:

$$\alpha(i, j, x) = \varphi_{h(i,j)}(x).$$

Also gilt für alle  $(i, j) \in \mathbb{N}^2$ :

$$\begin{aligned}
 (i, j) \in H &\iff \varphi_{h(i,j)}(x) \text{ ist für alle } x \text{ definiert} \\
 &\iff \varphi_{h(i,j)}(h(i, j)) \text{ ist definiert} \\
 &\iff h(i, j) \in D_{h(i,j)} \\
 &\iff h(i, j) \in K.
 \end{aligned}$$

Folglich ist  $H \leq_m K$  via  $h$ .

Die drei Probleme  $K$ ,  $H$  und  $X$  sind also alle „ $\leq_m$ -äquivalent“. Ebenso sind sie alle rekursiv aufzählbar. Sie sind Beispiele für „ $\leq_m$ -vollständige“ Probleme in RE. Das heißt, jede Menge in RE lässt sich auf ein solches vollständiges Problem reduzieren.

**Definition 11.12** Eine Menge  $A$  heißt  $\leq_m$ -vollständig in RE, falls gilt:

1.  $A \in \text{RE}$  und
2.  $(\forall B \in \text{RE}) [B \leq_m A]$ .

**Satz 11.13** Die Probleme  $K$ ,  $H$  und  $X$  sind  $\leq_m$ -vollständig in RE.

ohne Beweis

**Satz 11.14** Alle Mengen, die  $\leq_m$ -vollständig in RE sind, sind unentscheidbar.

**Beweis.** Sei  $A$  ein  $\leq_m$ -vollständiges Problem in RE. Das heißt, jedes Problem aus RE lässt sich auf  $A$  reduzieren. Insbesondere gilt  $K \leq_m A$ . Da das Halteproblem  $K$  unentscheidbar ist, folgt  $A \notin \text{REC}$  mit dem Unentscheidbarkeitskriterium. ■

Der Beweis der folgenden Behauptung ergibt sich unmittelbar aus der Transitivität der  $\leq_m$ -Reduzierbarkeit.

**Behauptung 11.15** Ist  $X \leq_m Y$  und  $X$   $\leq_m$ -vollständig in RE und  $Y \in \text{RE}$ , so ist  $Y$   $\leq_m$ -vollständig in RE.

## 11.3 Das Postsche Korrespondenzproblem

Nun wollen wir die Unentscheidbarkeit einiger natürlicher Probleme zeigen.

**Definition 11.16 (Postsches Korrespondenzproblem)** Sei  $\Sigma$  ein Alphabet.

- Das Postsche Korrespondenzproblem (über  $\Sigma$ ) ist definiert durch

$$\text{PCP}_\Sigma = \left\{ ((x_1, y_1), \dots, (x_k, y_k)) \left| \begin{array}{l} k \in \mathbb{N} \text{ und } x_i, y_i \in \Sigma^+ \text{ für } 1 \leq i \leq k \\ \text{und es gibt } i_1, i_2, \dots, i_n \in \{1, \dots, k\}, \\ \text{so dass } x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n} \end{array} \right. \right\}.$$

- Eine solche Indexfolge  $(i_1, i_2, \dots, i_n)$  mit  $x_{i_1}x_{i_2} \cdots x_{i_n} = y_{i_1}y_{i_2} \cdots y_{i_n}$  heißt Lösung der Problem Instanz  $((x_1, y_1), \dots, (x_k, y_k))$ .
- Das „modifizierte“ Postsche Korrespondenzproblem (über  $\Sigma$ ) ist definiert durch

$$\text{MPCP}_\Sigma = \left\{ ((x_1, y_1), \dots, (x_k, y_k)) \left| \begin{array}{l} k \in \mathbb{N} \text{ und } x_i, y_i \in \Sigma^+ \text{ für } 1 \leq i \leq k \\ \text{und es gibt } i_2, \dots, i_n \in \{1, \dots, k\} \\ \text{so dass } x_1x_{i_2} \cdots x_{i_n} = y_1y_{i_2} \cdots y_{i_n} \end{array} \right. \right\}.$$

Das heißt, das  $\text{MPCP}_\Sigma$  ist die Einschränkung des  $\text{PCP}_\Sigma$  auf diejenigen Eingaben  $((x_1, y_1), \dots, (x_k, y_k))$ , die eine mit dem Index 1 beginnende Lösung haben.

- $\text{PCP} = \bigcup_{\Sigma \text{ ist Alphabet}} \text{PCP}_\Sigma$  und  $\text{MPCP} = \bigcup_{\Sigma \text{ ist Alphabet}} \text{MPCP}_\Sigma$ .

**Beispiel 11.17 (Postsches Korrespondenzproblem)** Sei  $\Sigma = \{0, 1\}$ .

- Die Problem Instanz

$$\begin{pmatrix} x_1 = 1 & x_2 = 01 & x_3 = 010 \\ y_1 = 10 & y_2 = 1 & y_3 = 100 \end{pmatrix}$$

hat die Lösung  $(1, 3, 3, 2)$ , denn

$$\begin{array}{cccccccccccc} x_1 & x_3 & x_3 & x_2 & = & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ y_1 & y_3 & y_3 & y_2 & = & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}$$

und ist somit in  $\text{PCP}_\Sigma$  und in  $\text{MPCP}_\Sigma$ .

- Andererseits hat die Problem Instanz

$$\begin{pmatrix} x_1 = 01 & x_2 = 010 & x_3 = 1 \\ y_1 = 1 & y_2 = 100 & y_3 = 10 \end{pmatrix}$$

zwar die Lösung  $(3, 2, 2, 1)$  und ist damit in  $\text{PCP}_\Sigma$ , aber sie hat keine Lösung in  $\text{MPCP}_\Sigma$ , da  $x_1 = 01$  und  $y_1 = 1$ .

- Die Problem Instanz

$$\begin{pmatrix} x_1 = 001 & x_2 = 01 & x_3 = 01 & x_4 = 10 \\ y_1 = 0 & y_2 = 011 & y_3 = 101 & y_4 = 001 \end{pmatrix}$$

ist lösbar, aber die kleinstmögliche lösende Indexfolge besteht aus  $n = 66$  Elementen.

**Lemma 11.18**  $\text{MPCP} \leq_m \text{PCP}$ .



**Beweis.** Wir zeigen  $\text{MPCP}_\Sigma \leq_m \text{PCP}_{\Sigma \cup \{\$, \#\}}$ , wobei  $\$, \# \notin \Sigma$  neue Zeichen sind. Für jedes Wort  $w \in \Sigma^+$  mit  $w = a_1 a_2 \cdots a_m$  definieren wir:

$$\begin{aligned} \overleftrightarrow{w} &= \# a_1 \# a_2 \# \cdots \# a_m \# \\ \vec{w} &= a_1 \# a_2 \# \cdots \# a_m \# \\ \overleftarrow{w} &= \# a_1 \# a_2 \# \cdots \# a_m \end{aligned}$$

Jeder Eingabe  $z = ((x_1, y_1), \dots, (x_k, y_k))$  des  $\text{MPCP}_\Sigma$  ordnen wir diese Eingabe des  $\text{PCP}_{\Sigma \cup \{\$, \#\}}$  mit  $k + 2$  Komponenten zu:

$$f(z) = ((\overleftrightarrow{x_1}, \overleftarrow{y_1}), (\vec{x_1}, \overleftarrow{y_1}), \dots, (\vec{x_k}, \overleftarrow{y_k}), (\$, \#\$)).$$

Offenbar ist die so definierte Funktion  $f$  in  $\mathbb{R}$ . Zu zeigen ist, dass für alle  $z$  gilt:

$$z \in \text{MPCP}_\Sigma \iff f(z) \in \text{PCP}_{\Sigma \cup \{\$, \#\}}.$$

( $\Rightarrow$ ) Die Eingabe  $z \in \text{MPCP}_\Sigma$  habe die Lösung  $(1, i_2, i_3, \dots, i_n)$ , d.h.,

$$x_1 x_{i_2} \cdots x_{i_n} = y_1 y_{i_2} \cdots y_{i_n}.$$

Bezeichnen wir das  $r$ -te Symbol von  $x_{i_j}$  mit  $x_{i_j}^r$  bzw. das  $r$ -te Symbol von  $y_{i_j}$  mit  $y_{i_j}^r$ , dann gilt für die transformierte Instanz  $f(z) = ((\overleftrightarrow{x_1}, \overleftarrow{y_1}), (\vec{x_1}, \overleftarrow{y_1}), \dots, (\vec{x_k}, \overleftarrow{y_k}), (\$, \#\$))$ :

$$\begin{aligned} & \overbrace{\# x_1^1 \# \cdots \# x_1^{r_1} \#}^{\overleftrightarrow{x_1}} \overbrace{x_{i_2}^1 \# \cdots \# x_{i_2}^{r_2} \#}^{\vec{x_{i_2}}} \cdots \overbrace{\# x_{i_n}^1 \# \cdots \# x_{i_n}^{r_n} \#}^{\vec{x_{i_n}}} \$ \\ = & \underbrace{\# y_1^1 \# \cdots \# y_1^{s_1} \#}_{\overleftarrow{y_1}} \underbrace{\# y_{i_2}^1 \# \cdots \# y_{i_2}^{s_2} \#}_{\overleftarrow{y_{i_2}}} \cdots \underbrace{\# y_{i_n}^1 \# \cdots \# y_{i_n}^{s_n} \#}_{\overleftarrow{y_{i_n}}} \$ \end{aligned}$$

Somit ist  $(1, i_2 + 1, i_3 + 1, \dots, i_n + 1, k + 2)$  Lösung von  $f(z)$  in  $\text{PCP}_{\Sigma \cup \{\$, \#\}}$ .

( $\Leftarrow$ ) Hat  $f(z)$  die Lösung  $(i_1, i_2, \dots, i_n)$  in  $\text{PCP}_{\Sigma \cup \{\$, \#\}}$ , sodass kein Präfix eine Lösung ist, so kann dies wegen der Bauart der Wortpaare nur gelten, wenn  $i_1 = 1$  und  $i_n = k + 2$  und  $i_j \in \{2, \dots, k + 1\}$  für  $j \in \{2, \dots, n - 1\}$ . Folglich ist  $(1, i_2 - 1, i_3 - 1, \dots, i_{n-1} - 1)$  eine Lösung von  $z$  in  $\text{MPCP}_\Sigma$ . ■

**Lemma 11.19**  $H \leq_m \text{MPCP}$ .

**Beweis.** Gegeben seien eine TM  $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$ , geeignet codiert durch das Wort  $\text{code}(M) \in \{0, 1\}^*$ , und ein Eingabewort  $w \in \Sigma^*$ . Wir wollen die Berechnung von

$M(w)$  in eine Instanz des PCP codieren. Wir suchen also eine Reduktion  $f \in \mathbb{R}$ , so dass gilt:

$$\begin{aligned} M(w) \text{ hält in Endzustand} & \quad (11.1) \\ \iff f(\text{code}(M)\#w) = ((x_1, y_1), \dots, (x_k, y_k)) \text{ hat Lösung in MPCP}_\Delta, \end{aligned}$$

wobei  $\Delta = \Gamma \cup Z \cup \{\#\}$  das verwendete Alphabet mit einem neuen Symbol  $\#$  ist.

- Als erstes Paar wählen wir  $(x_1, y_1) = (\#, \#z_0w\#)$ . Dabei ist  $z_0w$  die Startkonfiguration von  $M(w)$ .
- Die weiteren Paare  $(x_i, y_i)$ ,  $i > 1$ , teilen wir ihrem Zweck entsprechend in Gruppen ein.
- **Kopierregeln:**  $(a, a)$  für alle  $a \in \Gamma \cup \{\#\}$ .
- **Überführungsregeln:**

$$\begin{aligned} (za, z'c) & \text{ falls } \delta(z, a) = (z', c, N) \\ (za, cz') & \text{ falls } \delta(z, a) = (z', c, R) \\ (bza, z'bc) & \text{ falls } \delta(z, a) = (z', c, L) \quad \text{für alle } b \in \Gamma \\ (\#za, \#z'\square c) & \text{ falls } \delta(z, a) = (z', c, L) \\ (z\#, z'c\#) & \text{ falls } \delta(z, \square) = (z', c, N) \quad \text{für } c \neq \square \\ (z\#, z'\#) & \text{ falls } \delta(z, \square) = (z', \square, N) \\ (z\#, cz'\#) & \text{ falls } \delta(z, \square) = (z', c, R) \\ (bz\#, z'bc\#) & \text{ falls } \delta(z, \square) = (z', c, L) \quad \text{für alle } b \in \Gamma. \end{aligned}$$

- **Löschregeln:**  $(az, z)$  und  $(za, z)$  für alle  $a \in \Gamma$  und  $z \in F$ .
- **Abschlussregeln:**  $(z\#\#, \#)$  für alle  $z \in F$ .

Nun zeigen wir die Äquivalenz (11.1):

$M$  hält bei Eingabe  $w$  im Endzustand

$$\begin{aligned} \iff & \text{ es gibt eine Folge } (k_0, k_1, \dots, k_t) \text{ von Konfigurationen mit } k_0 = z_0w \text{ und} \\ & k_t = uzv \text{ für } u, v \in \Gamma^*, z \in F \text{ und } k_{i-1} \vdash_M k_i \text{ für } 1 \leq i \leq t \\ \iff & f(\text{code}(M)\#w) = ((x_1, y_1), \dots, (x_k, y_k)) \text{ hat} \\ & \text{eine Lösung } (1, i_2, \dots, i_n) \text{ für MPCP}_\Sigma. \end{aligned}$$

Das Lösungswort, das sich dabei ergibt, hat die Gestalt:

$$\begin{array}{cccccccccccccccccccc}
 x_1 & & x_{i_2} & x_{i_3} & x_{i_4} & x_{i_5} & \dots & & & & & & & & & & & x_{i_n} \\
 \# & & k_0 & \# & k_1 & \# & \dots & k_{t-1} & \# & k_t & \# & \# & k'_t & \# & k''_t & \dots & z & \# \# \\
 \# k_0 \# & & k_1 & \# & k_2 & \# & \dots & k_t & \# & k_{t-1} & \# & k_t & \# & \# & k'_t & \dots & \# & \\
 y_1 & & y_{i_2} & y_{i_3} & y_{i_4} & y_{i_5} & \dots & & & & & & & & & & & y_{i_n}
 \end{array}$$

bzw.

$$\begin{aligned}
 x_1 x_{i_2} \dots x_{i_n} &= \# k_0 \# k_1 \# \dots \# k_t \# \# k'_t \# k''_t \# \dots \# z \# \# \\
 &= y_1 y_{i_2} \dots y_{i_n},
 \end{aligned}$$

wobei die  $k'_t, k''_t, \dots$  aus der Endkonfiguration  $k_t = uzv$  durch sukzessives Löschen der Nachbarsymbole des Endzustandes  $z \in F$  entstehen. Die  $x_{i_j}$  „hinken“ dabei den  $y_{i_j}$  um genau eine Konfiguration nach.

Hat umgekehrt  $f(\text{code}(M)\#w) = ((x_1, y_1), \dots, (x_k, y_k))$  eine Lösung  $(1, i_2, \dots, i_n)$  in  $\text{MPCP}_\Sigma$ , so kann aus dieser eine Rechnung von  $M(w)$  abgelesen werden, die im Endzustand hält. ■

**Beispiel 11.20** Wir veranschaulichen die Reduktion im letzten Beweis an der Turingmaschine aus Beispiel 7.2 für  $n = 3$  (s. Übungen).

**Satz 11.21** Die Probleme MPCP und PCP sind  $\leq_m$ -vollständig in RE und somit unentscheidbar, d.h.,  $\text{MPCP} \notin \text{REC}$  und  $\text{PCP} \notin \text{REC}$ .

**Beweis.** Nach den Lemmata 11.18 und 11.19 und der  $\leq_m$ -Vollständigkeit von  $H$  in RE (Satz 11.13) gilt für alle  $A \in \text{RE}$ :  $A \leq_m H \leq_m \text{MPCP} \leq_m \text{PCP}$ . Dass MPCP und PCP in RE liegen, lässt sich einfach mit dem Projektionssatz (Satz 10.26) zeigen. ■

Es gilt sogar:

**Satz 11.22**  $\text{PCP}_{\{0,1\}} \notin \text{REC}$ .

**Beweis.** Wir zeigen dass  $\text{PCP}_\Sigma \leq_m \text{PCP}_{\{0,1\}}$ . Das Alphabet von PCP sei  $\Sigma = \{a_1, \dots, a_n\}$ . Wir betrachten eine umkehrbare Codierung  $f : \Sigma^* \rightarrow \{0,1\}^*$  der Wörter über  $\Sigma$  mit der Eigenschaft, dass sich Wortketten eindeutig reproduzieren lassen. Ein Beispiel für eine solche Codierung von  $\Sigma = \{a_1, \dots, a_n\}$  ist die totale und berechenbare Funktion  $f : \Sigma^* \rightarrow \{0,1\}^*$  mit  $f(a_r) = 01^r$  und  $f(a_{i_1} \dots a_{i_k}) = f(a_{i_1}) \dots f(a_{i_k})$ . An der Position der Nullen kann man eindeutig erkennen, dass dort die Codierung eines neuen Symbols beginnt und an der Anzahl der Einsen kann man erkennen, welches Symbol dort codiert ist.

Es gilt:

$$\begin{aligned}
 &((x_1, y_1), \dots, (x_k, y_k)) \text{ hat eine Lösung} \\
 &\Leftrightarrow \\
 &((f(x_1), f(y_1)), \dots, (f(x_k), f(y_k))) \text{ hat eine Lösung.}
 \end{aligned}$$

■

Im Beweis des letzten Satzes haben wir ein beliebiges Alphabet auf ein zweielementiges Alphabet abgebildet, für einelementige Alphabete wird das PCP entscheidbar.

**Satz 11.23** Für jedes  $\Sigma$  mit  $|\Sigma| = 1$  ist  $\text{PCP}_\Sigma$  entscheidbar.

**Beweis.** O.B.d.A. sei  $\Sigma = \{1\}$ . Fallunterscheidung:

1. Falls es ein  $i \in \{1, \dots, k\}$  gibt mit  $|x_i| = |y_i|$ , dann ist die Indexfolge  $(i)$  Lösung des PCP.

Zum Beispiel hat

$$\begin{pmatrix} x_1 = 11 & x_2 = 11 & x_3 = 11111 \\ y_1 = 111 & y_2 = 11 & y_3 = 11 \end{pmatrix}$$

offensichtlich die Indexfolge (2) eine Lösung, denn

$$\begin{aligned}
 x_2 &= 11 \\
 y_2 &= 11.
 \end{aligned}$$

2. Falls es ein  $i \in \{1, \dots, k\}$  gibt mit  $|x_i| < |y_i|$  und ein  $j \in \{1, \dots, k\}$  gibt mit  $|x_j| > |y_j|$ , dann ist jede Folge mit  $|y_i| - |x_i|$  mal Index  $j$  und  $|x_j| - |y_j|$  mal Index  $i$  eine Lösung des PCP.

Die Korrektheit folgt, da

$$\begin{aligned}
 &|x_i| \cdot (|x_j| - |y_j|) + |x_j| \cdot (|y_i| - |x_i|) \\
 &= |x_j| \cdot |y_i| - |x_i| \cdot |y_j| \\
 &= |y_i| \cdot (|x_j| - |y_j|) + |y_j| \cdot (|y_i| - |x_i|)
 \end{aligned}$$

gilt.

Als Beispiel betrachten wir

$$\begin{pmatrix} x_1 = 111 & x_2 = 1111 & x_3 = 11111 \\ y_1 = 11111 & y_2 = 1 & y_3 = 111 \end{pmatrix}$$

Es ist  $|x_1| < |y_1|$  ( $i = 1$ ) und  $|x_2| > |y_2|$  ( $j = 2$ ), somit ist  $|y_1| - |x_1| = 2$  mal Index  $j = 2$  und  $|x_2| - |y_2| = 3$  mal Index  $i = 1$  eine Lösung  $(1, 1, 1, 2, 2)$ , denn

$$\begin{array}{ccccccccc}
 x_1 & x_1 & x_1 & x_2 & x_2 & = & 111 & 111 & 111 & 1111 & 1111 \\
 y_1 & y_1 & y_1 & y_2 & y_2 & = & 11111 & 11111 & 11111 & 1 & 1
 \end{array}$$

3. Falls für alle  $i \in \{1, \dots, k\}$  gilt  $|x_i| < |y_i|$  oder für alle  $j \in \{1, \dots, k\}$  gilt  $|x_j| > |y_j|$ , dann kann es keine Lösung des PCP geben, da für jede Indexfolge  $|x_{i_1}x_{i_2} \cdots x_{i_n}| < |y_{i_1}y_{i_2} \cdots y_{i_n}|$  bzw.  $|x_{i_1}x_{i_2} \cdots x_{i_n}| > |y_{i_1}y_{i_2} \cdots y_{i_n}|$  gilt.

■

**Bemerkung 11.24** Definiert man  $k$ -PCP als die Einschränkung von PCP, in der für ein festes  $k \in \mathbb{N}$  höchstens  $k$ -Tupel  $((x_1, y_1), \dots, (x_k, y_k))$  auftreten, so ist bekannt, dass

- $k$ -PCP ist entscheidbar für  $k \leq 2$ , aber
- $k$ -PCP ist nicht entscheidbar für  $k \geq 9$ .

Offen ist die Frage nach der Entscheidbarkeit von  $k$ -PCP für  $3 \leq k \leq 8$ .

Die hier gezeigten Reduktionen implizieren die folgenden Semi-Entscheidbarkeitsresultate.

**Korollar 11.25** Das Halteproblem  $H$  und das spezielle Halteproblem  $K$  sind semi-entscheidbar.

Es gibt sogar nicht semi-entscheidbare Probleme:

**Satz 11.26** Das Komplement des speziellen Halteproblems  $\overline{K} = \{0, 1\}^* - K$  ist nicht semi-entscheidbar.

**Beweis.** Nach Korollar 11.25 ist das spezielle Halteproblem  $K$  semi-entscheidbar. Wäre nun auch  $\overline{K} = \{0, 1\}^* - K$  semi-entscheidbar, so wäre nach Satz 10.7 das spezielle Halteproblem  $K$  entscheidbar, was ein Widerspruch zu Satz 10.21 ist. ■

## 11.4 Unentscheidbarkeit in der Chomsky-Hierarchie

Mit Hilfe der Unentscheidbarkeit des PCP und des Halteproblems, lassen sich eine Reihe von Unentscheidbarkeitsresultate von Problemen in der Theorie der formalen Sprachen nachweisen.

Erinnerung: Die Klassen der Chomsky-Hierarchie sind  $\mathcal{L}_0 = \text{RE}$ ,  $\mathcal{L}_1 = \text{CS}$ ,  $\mathcal{L}_2 = \text{CF}$  und  $\mathcal{L}_3 = \text{REG}$ . Wir definieren nun eine Reihe von Problemen bzgl. der Klassen der Chomsky-Hierarchie.

**Definition 11.27** Für  $i \in \{0, 1, 2, 3\}$  definieren wir das Wortproblem, das Leerheitsproblem, das Schnittproblem, das Äquivalenzproblem und das Mehrdeutigkeitsproblem und das Äquivalenzproblem für Typ- $i$ -Grammatiken:

$$\begin{aligned} \text{Wort}_i &= \{(G, w) \mid G \text{ ist Typ-}i\text{-Grammatik und } w \in L(G)\}, \\ \text{Leer}_i &= \{G \mid G \text{ ist Typ-}i\text{-Grammatik und } L(G) \neq \emptyset\}, \\ \text{Schnitt}_i &= \{(G_1, G_2) \mid G_1, G_2 \text{ sind Typ-}i\text{-Grammatiken und } L(G_1) \cap L(G_2) \neq \emptyset\}, \\ \text{Äquiv}_i &= \{(G_1, G_2) \mid G_1, G_2 \text{ sind Typ-}i\text{-Grammatiken und } L(G_1) = L(G_2)\}, \\ \text{Ambig}_i &= \{G \mid G \text{ ist eine mehrdeutige Typ-}i\text{-Grammatik}\}. \end{aligned}$$

Statt der Grammatiken in den Problemen können auch jeweils äquivalente Automatenmodelle eingesetzt werden.

**Satz 11.28** Tabelle 11.1 zeigt die Entscheidbarkeit bzw. Nicht-Entscheidbarkeit für die oben definierten Probleme in der Chomsky-Hierarchie.

$i$	$\text{Wort}_i$	$\text{Leer}_i$	$\text{Schnitt}_i$	$\text{Äquiv}_i$	$\text{Ambig}_i$
0	$\notin \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$
1	$\in \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$
2	$\in \text{REC}$	$\in \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$
3	$\in \text{REC}$	$\in \text{REC}$	$\in \text{REC}$	$\in \text{REC}$	$\in \text{REC}$

Tabelle 11.1: Entscheidbarkeit von Problemen in der Chomsky-Hierarchie

**Beweis.** Hier einige Beweisideen:

- Wort**
- Man kann zeigen, dass  $H \leq_m \text{Wort}_0$  gilt und damit folgt, dass  $\text{Wort}_0$  nicht entscheidbar ist.
  - $\text{Wort}_1$  ist nach dem Beweis von Satz 10.9 ( $\text{CS} \subseteq \text{REC}$ ) entscheidbar. (Daraus folgt bereits:  $\text{Wort}_i \in \text{REC}$  für  $i \in \{1, 2, 3\}$ , allerdings mit exponentiellem Zeitaufwand.)
  - $\text{Wort}_2$  ist (nach dem CYK-Algorithmus sogar in kubischer Zeit) entscheidbar.
  - $\text{Wort}_3$  ist (in Linearzeit) entscheidbar; einfach über DFAs.

- Leer**
- Man kann zeigen, dass  $H_0A \leq_m \text{Leer}_1$  gilt. Somit ist  $\text{Leer}_1$  nicht entscheidbar.<sup>1</sup>

<sup>1</sup>Alternativ dazu kann man eine Reduktion  $\text{Schnitt}_2 \leq_m \text{Leer}_1$  angeben – wir zeigen später, dass  $\text{Schnitt}_2$  nicht entscheidbar ist. Diese Reduktion folgt unmittelbar aus dem Beweis, dass CS unter Schnitt abgeschlossen ist, und zwar *effektiv*. Das heißt, aus zwei gegebenen kfGs,  $G_1$  und  $G_2$ , kann man eine Typ-1-Grammatik  $G$  mit  $L(G) = L(G_1) \cap L(G_2)$  konstruieren, und es gibt ein algorithmisches Verfahren für diese Konstruktion. Dieser Algorithmus berechnet die gesuchte Reduktion  $f$ .

- Da jede Typ-1-Grammatik auch eine Typ-0-Grammatik ist, folgt dass auch  $\text{Leer}_0$  nicht entscheidbar ist.
- Zur Entscheidbarkeit von  $\text{Leer}_2$ : Es sei  $G$  eine kontextfreie Grammatik mit  $k$  Nichtterminalen. Sei  $n$  die Zahl aus dem Pumping-Lemma für  $L(G)$ , nämlich  $n = 2^{k+1}$ , siehe Satz 3.17.

Wir zeigen:

$$L(G) \neq \emptyset \iff \text{es gibt ein Wort } z \in L(G) \text{ mit } |z| < n. \quad (11.2)$$

( $\Leftarrow$ ) Klar.

( $\Rightarrow$ ) Sei  $L(G) \neq \emptyset$  und sei  $z$  ein Wort minimaler Länge in  $L(G)$ . Wäre  $|z| \geq n$ , dann folgte aus Satz 3.17:  $z = uvwxy$  mit  $|vx| \geq 1$ ,  $|vwx| \leq n$  und  $(\forall i \geq 0) [uv^iwx^iy \in L]$ . Insbesondere wäre  $uwy \in L(G)$  für  $i = 0$ , und es wäre  $|uwy| < |uvwxy| = |z|$  ein Widerspruch zur Minimalität von  $z$ . Also ist  $|z| < n$ . Die Äquivalenz (11.2) ist bewiesen.

Da die rechte Seite von (11.2) entscheidbar ist, ist auch  $\text{Leer}_2$  entscheidbar.

- Somit muss auch  $\text{Leer}_3$  entscheidbar sein, da jede Typ-3-Grammatik auch eine Typ-2-Grammatik ist.

Schnitt

- Um  $\text{Schnitt}_2 \notin \text{REC}$  zu zeigen, geben wir eine Reduktion  $\text{PCP} \leq_m \text{Schnitt}_2$  an. Gegeben sei eine Eingabe  $z = ((x_1, y_1), \dots, (x_k, y_k))$  des  $\text{PCP}_\Sigma$  für ein Alphabet  $\Sigma$ . Gesucht ist eine Reduktion  $f \in \mathbb{R}$  mit  $f(z) = (G_1, G_2)$ , wobei  $G_1$  und  $G_2$  kfGs sind, so dass gilt:

$$z \text{ hat Lösung} \iff L(G_1) \cap L(G_2) \neq \emptyset. \quad (11.3)$$

**Idee:**  $G_1$  erzeugt die Folge der  $x_i$ -Wörter und  $G_2$  erzeugt die Folge der  $y_i$ -Wörter, wobei die Indizes in potentiellen Lösungswörtern übereinstimmen müssen.

**Konstruktion:** Seien  $a_1, a_2, \dots, a_k \notin \Sigma$  neue Symbole. Definiere für  $i \in \{1, 2\}$  die Grammatik

$$\begin{aligned} G_i &= (\Sigma \cup \{a_1, a_2, \dots, a_k\}, \{S_i\}, S_i, P_i), \quad \text{wobei} \\ P_1 &= \{S_1 \rightarrow a_1x_1 \mid \dots \mid a_kx_k\} \cup \{S_1 \rightarrow a_1S_1x_1 \mid \dots \mid a_kS_1x_k\} \\ P_2 &= \{S_2 \rightarrow a_1y_1 \mid \dots \mid a_ky_k\} \cup \{S_2 \rightarrow a_1S_2y_1 \mid \dots \mid a_kS_2y_k\}. \end{aligned} \quad (11.4)$$

Dann folgt die Äquivalenz (11.3) so:

$z$  hat eine Lösung  $(i_1, i_2, \dots, i_n) \iff$

$$a_{i_n} \cdots a_{i_2} a_{i_1} x_{i_1} x_{i_2} \cdots x_{i_n} = a_{i_n} \cdots a_{i_2} a_{i_1} y_{i_1} y_{i_2} \cdots y_{i_n} \text{ ist ein Wort in } L(G_1) \cap L(G_2).$$

Da  $\text{PCP} \leq_m \text{Schnitt}_2$  gilt und  $\text{PCP}$  unentscheidbar ist, ist auch  $\text{Schnitt}_2$  nicht entscheidbar.

- Folglich sind auch  $\text{Schnitt}_1$  und  $\text{Schnitt}_0$  nicht entscheidbar, denn wäre  $\text{Schnitt}_i \in \text{REC}$  für  $i \in \{0, 1\}$ , dann könnte man für beliebige zwei Typ- $i$ -Grammatiken entscheiden, ob  $L(G_1) \cap L(G_2) \neq \emptyset$ . Da  $\text{CF} \subseteq \text{CS} \subseteq \text{RE}$ , wäre dann aber  $\text{Schnitt}_2 \in \text{REC}$ , ein Widerspruch.
- $\text{Schnitt}_3$  ist entscheidbar, da man einen so genannten Kreuzproduktautomaten angeben kann, der  $L(G_1) \cap L(G_2)$  akzeptiert.

Alternativ folgt diese Aussage daraus, dass die Klasse  $\text{REG}$  *effektiv* unter Schnitt abgeschlossen ist. Das heißt, aus zwei gegebenen regulären Grammatiken,  $G_1$  und  $G_2$ , kann man eine reguläre Grammatik  $G$  mit  $L(G) = L(G_1) \cap L(G_2)$  konstruieren. Der Algorithmus, der dies leistet, liefert die Reduktion  $\text{Schnitt}_3 \leq_m \text{Leer}_3$ . Nach Aussage 2 von Lemma 11.5 folgt  $\text{Schnitt}_3 \in \text{REC}$  aus  $\text{Leer}_3 \in \text{REC}$ .

Äquiv

- Für alle Mengen  $A, B \subseteq \Sigma^*$  gilt

$$A = B \iff (A \cap \overline{B}) \cup (\overline{A} \cap B) = \emptyset. \quad (11.5)$$

Da  $\text{REG}$  *effektiv* unter Schnitt, Vereinigung und Komplement abgeschlossen ist, liefert (11.5) eine Reduktion  $\text{Äquiv}_3 \leq_m \text{Leer}_3$ . Wegen  $\text{Leer}_3 \in \text{REC}$  ist auch  $\overline{\text{Leer}_3} \in \text{REC}$ , und da  $\text{REC} \leq_m$ -abgeschlossen ist, folgt  $\text{Äquiv}_3 \in \text{REC}$ .

Alternativer Beweis:  $\text{Äquiv}_3$  ist entscheidbar durch Testen der Minimalautomaten von  $L(G_1)$  und  $L(G_2)$  auf Isomorphie.

- Um  $\text{Äquiv}_i \notin \text{REC}$  für  $i \in \{0, 1, 2\}$  zu zeigen, genügt es,  $\text{Äquiv}_2 \notin \text{REC}$  zu beweisen. Da die in (11.4) konstruierten kfGs  $G_1$  und  $G_2$  sogar deterministisch kontextfrei sind, ist auch das Schnittproblem für deterministisch kontextfreie Grammatiken,  $\text{Schnitt}_{\text{DCF}}$ , unentscheidbar. Wir suchen eine Reduktion  $f$  für  $\text{Schnitt}_{\text{DCF}} \leq_m \text{Äquiv}_2$ :

$$(G_1, G_2) \notin \text{Schnitt}_{\text{DCF}} \iff f(G_1, G_2) \in \text{Äquiv}_2. \quad (11.6)$$

Um  $f$  zu definieren, benutzen wir, dass  $\text{DCF}$  *effektiv* unter Komplement abgeschlossen ist. Somit kann man aus einer gegebenen deterministisch kontextfreien Grammatik  $G_2$  eine deterministisch kontextfreie Grammatik  $\widehat{G}_2$  mit  $L(\widehat{G}_2) = \overline{L(G_2)}$  berechnen. Da weiterhin  $\text{CF}$  *effektiv* unter Vereinigung abgeschlossen ist, kann man aus gegebenen kfGs  $G_1$  und  $\widehat{G}_2$  eine Grammatik  $G_3$  mit  $L(G_3) = L(G_1) \cup L(\widehat{G}_2)$  berechnen. Damit ist die Konstruktion von  $f(G_1, G_2) = (G_3, \widehat{G}_2)$  vollständig beschrieben. Die Äquivalenz (11.6) folgt nun so:

$$\begin{aligned} L(G_1) \cap L(G_2) = \emptyset &\iff L(G_1) \subseteq \overline{L(G_2)} \\ &\iff L(G_1) \subseteq L(\widehat{G}_2) \\ &\iff L(G_1) \cup L(\widehat{G}_2) = L(\widehat{G}_2) \\ &\iff L(G_3) = L(\widehat{G}_2). \end{aligned}$$



- Ambig
- Man kann  $\text{PCP} \leq_m \text{Ambig}_2$  wie folgt zeigen:
  - PCP-Instanz  $((x_1, y_1), \dots, (x_k, y_k))$  umwandeln in die folgende kontextfreie Grammatik:
    - $S \rightarrow A \mid B$
    - $A \rightarrow 1Ax_1 \mid 2Ax_2 \mid \dots \mid kAx_k \mid \$$
    - $B \rightarrow 1By_1 \mid 2By_2 \mid \dots \mid kBy_k \mid \$$
  - Man kann zeigen, dass diese Grammatik genau dann mehrdeutig ist, wenn die gegebene PCP-Instanz lösbar ist.



### Bemerkung 11.29

*Ob das Äquivalenzproblem für deterministisch kontextfreien Grammatiken auch unentscheidbar ist, geht aus dem obigen Beweis nicht hervor, da DCF nicht unter Vereinigung abgeschlossen ist. Tatsächlich zeigte Gérard Sénizergues 1997, dass dieses Problem entscheidbar ist. Für alle anderen Probleme bzgl. DCF gelten dieselben Eigenschaften wie für CF.*

- *Das Problem, von einer kontextfreien Grammatik festzustellen, ob sie mehrdeutig ist, ist unentscheidbar.*
- *Das Problem, von einem DFA  $M$  festzustellen, ob er eine endliche oder unendliche Sprache akzeptiert, ist dagegen entscheidbar.*

## 11.5 Zusammenfassung

In diesem Abschnitt fassen wir nochmal die Entscheidbarkeit von Wortproblemen in der Chomsky-Hierarchy in übersichtlicher Tabellenform zusammen. Wir erweitern die Tabelle 11.1 um die Zeile mit den deterministisch kontextfreien Sprachen.

	Wort	Leer	Schnitt	Äquiv
Typ 0	$\notin \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$
Typ 1	$\in \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$
Typ 2	$\in \text{REC}$	$\in \text{REC}$	$\notin \text{REC}$	$\notin \text{REC}$
det. kf.	$\in \text{REC}$	$\in \text{REC}$	$\notin \text{REC}$	$\in \text{REC}$
Typ 3	$\in \text{REC}$	$\in \text{REC}$	$\in \text{REC}$	$\in \text{REC}$

Tabelle 11.2: Entscheidbarkeit von Problemen in der Chomsky-Hierarchy

# **Teil III**

## **NP-Vollständigkeit**



# Kapitel 12

## Probleme in P und NP

In diesem Teil der Vorlesung wollen wir uns mit der NP-Vollständigkeit, einem Teilgebiet der Komplexitätstheorie, beschäftigen. Die Komplexitätstheorie untersucht den Ressourcenbedarf (insbesondere: Rechenzeit, Speicherplatz) von algorithmisch lösbaren Problemen. Dabei sind sowohl untere als auch obere Schranken für die Ressourcen von Interesse.

### 12.1 Deterministische Polynomialzeit

Um Sprachen (oder Probleme) bzgl. der benötigten Ressourcen einzuteilen, definiert man Klassen von Funktionen, so genannte Komplexitätsklassen. Dabei gehen wir von Sprachen (Probleme) aus, die als Turingprogramme formuliert sind.

Die zwei wichtigsten Zeitkomplexitätsklassen, P und NP, wollen wir nun definieren.

**Definition 12.1 (IPol)** Sei  $\mathbb{P}ol$  die Menge aller Polynome der Form

$$p(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0,$$

wobei  $n, m, a_i \in \mathbb{N}$  für  $1 \leq i \leq m$ .

**Definition 12.2 (deterministische Polynomialzeit: P)** Sei  $t : \mathbb{N} \rightarrow \mathbb{N}$  eine Funktion und  $M$  eine deterministische Turingmaschine mit  $L(M) \subseteq \Sigma^*$  für ein Alphabet  $\Sigma$ . Definiere

- die Funktion  $time_M : \Sigma^* \rightarrow \mathbb{N}$  durch

$$time_M(w) = \begin{cases} \text{Zahl der Takte von } M(w) & \text{falls } M \text{ bei Eingabe } w \text{ hält} \\ \text{nicht definiert} & \text{sonst;} \end{cases}$$

- die Funktion  $\text{Time}_M : \mathbb{N} \rightarrow \mathbb{N}$  durch

$$\text{Time}_M(n) = \begin{cases} \max_{w: |w|=n} \text{time}_M(w) & \text{falls } \text{time}_M(w) \text{ für alle } w \text{ mit} \\ & |w| = n \text{ definiert ist} \\ \text{nicht definiert} & \text{sonst;} \end{cases}$$

- die Komplexitätsklasse

$$\text{TIME}(t) = \left\{ A \mid \begin{array}{l} \text{es gibt eine deterministische TM } M \text{ mit} \\ L(M) = A \text{ und } \text{Time}_M(n) \leq t(n) \text{ für alle } n \end{array} \right\}.$$

- Die Komplexitätsklasse deterministische Polynomialzeit ist definiert durch

$$P = \bigcup_{p \in \mathbb{P}\text{ol}} \text{TIME}(p).$$

### Bemerkung 12.3

- Die Klasse  $P$  ist die Klasse der durch effiziente (polynomialzeitbeschränkte) Algorithmen lösbaren Probleme.
- Um zu zeigen, dass ein Algorithmus eine polynomielle Laufzeit hat, genügt es offensichtlich zu zeigen, dass seine Komplexität in  $\mathcal{O}(n^k)$ , für eine Konstante  $k \in \mathbb{N}$ , liegt.
- Damit ist auch jeder Algorithmus mit der Laufzeit  $\log n$  und  $n \log n$  polynomiell, da  $\log n \in \mathcal{O}(n)$  und  $n \log n \in \mathcal{O}(n^2)$ .
- Laufzeiten wie  $n^n$ ,  $n^{\log n}$ ,  $c^n$ ,  $c \geq 2$ ,  $c \in \mathbb{N}$ , sind dagegen nicht polynomiell, sondern exponentiell.

## 12.2 Das Erfüllbarkeitsproblem der Aussagenlogik

Wir betrachten nun das Erfüllbarkeitsproblem der Aussagenlogik, SAT. Abbildung 12.1 zeigt eine 3-SAT-Instanz, die durch einen Booleschen Schaltkreis repräsentiert ist. SAT ist in vielen Anwendungen wichtig, es spielt eine zentrale Rolle im Zusammenhang mit „constraint satisfaction“, in der Theorie der Schaltkreise usw. So genannte SAT-Solver werden bei vielen praktischen Aufgaben als Subroutinen verwendet.

Boolesche Formeln (oder Boolesche Ausdrücke) werden syntaktisch und semantisch wie folgt definiert.

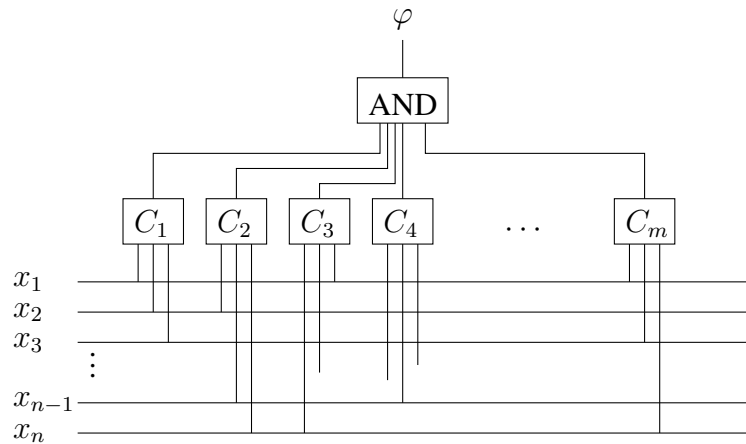


Abbildung 12.1: Eine 3-SAT-Instanz, repräsentiert durch einen Booleschen Schaltkreis

**Definition 12.4 (Boolesche Ausdrücke, KNF, DNF)**

- Die Booleschen Variablen  $x_1, x_2, \dots$  können Werte in  $\{0, 1\}$  annehmen. Die Menge aller Booleschen Variablen bezeichnen wir mit  $X$ , d.h.,  $X = \{x_1, x_2, \dots\}$ .
- **Syntax:** Die Menge der Booleschen Ausdrücke wird induktiv definiert durch:
  - Jedes  $x \in X$  ist ein (atomarer) Boolescher Ausdruck.
  - Sind  $F$  und  $G$  Boolesche Ausdrücke, so sind auch
    - \* die Negation  $\neg F$ ,
    - \* die Konjunktion  $F \wedge G$  und
    - \* die Disjunktion  $F \vee G$
 Boolesche Ausdrücke.
  - Nichts sonst ist ein Boolescher Ausdruck.

(Bemerkung: Sämtliche Booleschen Operationen lassen sich durch die Negation, die Konjunktion und die Disjunktion ausdrücken. Beispielsweise ergibt sich die Implikation  $\alpha \Rightarrow \beta$  durch  $\neg\alpha \vee \beta$ .)

- **Semantik:** Eine Belegung (mit Wahrheitswerten) ist eine Abbildung  $\beta : X \rightarrow \{0, 1\}$ . Für nicht atomare Boolesche Ausdrücke  $H$  wird der Wahrheitswert  $\beta(H)$  induktiv wie folgt definiert:
  - Ist  $H = \neg F$ , so ist  $\beta(H) = 1$ , falls  $\beta(F) = 0$ , und anderenfalls ist  $\beta(H) = 0$ .

- Ist  $H = F \wedge G$ , so ist  $\beta(H) = 1$ , falls  $\beta(F) = 1$  und  $\beta(G) = 1$ , und anderenfalls ist  $\beta(H) = 0$ .
- Ist  $H = F \vee G$ , so ist  $\beta(H) = 1$ , falls  $\beta(F) = 1$  oder  $\beta(G) = 1$ , und anderenfalls ist  $\beta(H) = 0$ .
- Ein Boolescher Ausdruck  $H$  heißt erfüllbar, falls es eine wahrmachende Belegung für ihn gibt, d.h., falls  $\beta(H) = 1$  für eine geeignete Belegung  $\beta : X \rightarrow \{0, 1\}$  gilt.
- Ein Literal ist eine Variable oder ihre Negation.
- Ein Boolescher Ausdruck  $H$  ist in konjunktiver Normalform (kurz: KNF), falls er Konjunktion von Disjunktionen von Literalen ist:

$$H = \bigwedge_{i=1}^m \left( \bigvee_{j=1}^n L_{ij} \right).$$

Die Disjunktionen  $\left( \bigvee_{j=1}^n L_{ij} \right)$  heißen Klauseln.

- Ein Boolescher Ausdruck  $H$  ist in disjunktiver Normalform (kurz: DNF), falls er Disjunktion von Konjunktionen von Literalen ist:

$$H = \bigvee_{i=1}^m \left( \bigwedge_{j=1}^n L_{ij} \right).$$

Die Konjunktionen  $\left( \bigwedge_{j=1}^n L_{ij} \right)$  heißen Monome.

- Sei  $k \geq 0$ . Ein Boolescher Ausdruck  $H$  ist in  $k$ -KNF (bzw. in  $k$ -DNF), falls  $H$  in KNF (bzw. in DNF) ist und jede Klausel von  $H$  höchstens  $k$  Literale enthält.
- Ein Boolescher Ausdruck  $H$  heißt Hornformel, falls  $H$  in KNF ist und jede Klausel von  $H$  höchstens ein positives Literal enthält.

### Beispiel 12.5 (Boolesche Ausdrücke, KNF, DNF)

- Atomare Boolesche Ausdrücke:  $x_1, x_2, x_3, x_4$
- Boolesche Ausdrücke:  $\neg x_3, x_2 \wedge x_3, (x_2 \vee x_4) \wedge \neg x_3, x_2 \vee \neg x_2, x_3 \wedge \neg x_3$
- erfüllbare Boolesche Ausdrücke:  $\neg x_3$  mit  $\beta(x_3) = 0$ ,  $x_2 \wedge x_3$  mit  $\beta(x_2) = \beta(x_3) = 1$   
nicht erfüllbar ist der Boolesche Ausdruck  $x_3 \wedge \neg x_3$
- Literale:  $x_3, x_1, \neg x_3$



- ein Boolescher Ausdruck in KNF:  $(\neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee \neg x_4)$
- ein Boolescher Ausdruck in DNF:  $(x_3 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_2 \wedge x_3 \wedge \neg x_4)$
- ein Boolescher Ausdruck in 4-KNF:  $(\neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3 \vee \neg x_4)$
- ein Boolescher Ausdruck in 3-DNF:  $(x_3 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_2 \wedge x_3 \wedge \neg x_4)$
- Hornformel:  $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$

**Definition 12.6** Varianten des Erfüllbarkeitsproblems sind:

- SAT =  $\{H \mid H \text{ ist erfüllbarer Boolescher Ausdruck}\}$ .
- KNF-SAT =  $\{H \mid H \text{ ist erfüllbarer Boolescher Ausdruck in KNF}\}$ .
- DNF-SAT =  $\{H \mid H \text{ ist erfüllbarer Boolescher Ausdruck in DNF}\}$ .
- $k$ -SAT =  $\{H \mid H \text{ ist erfüllbarer Boolescher Ausdruck in } k\text{-KNF}\}$ , wobei  $k \geq 0$ .
- Horn-SAT =  $\{H \mid H \text{ ist erfüllbare Hornformel}\}$ .

**Satz 12.7** DNF-SAT, 2-SAT und Horn-SAT sind in P.

**Beweisidee:** Insbesondere wird beim Beweis von  $2\text{-SAT} \in P$  zu einem gegebenen Booleschen Ausdruck  $H$  in 2-KNF ein gerichteter Graph  $G_H$  wie folgt konstruiert:

- Knoten von  $G_H$  sind alle Variablen von  $H$  und ihre Negationen;
- Kanten von  $G_H$ :  $(\alpha, \gamma)$  ist genau dann Kante von  $\alpha$  zu  $\gamma$ , wenn es in  $H$  eine Klausel der Form  $(\neg\alpha \vee \gamma)$  (bzw.  $(\gamma \vee \neg\alpha)$ ) gibt.

Der Graph stellt einen Implikationsgraphen dar: eine Kante von  $\alpha$  nach  $\gamma$  bedeutet, dass das Literal  $\alpha$  das Literal  $\gamma$  impliziert. Zum Beispiel aus der Klausel  $a \vee b$  folgen die zwei Kanten und Implikationen:  $\neg a \Rightarrow b$  und  $\neg b \Rightarrow a$ .

Anmerkung: für unäre Klauseln  $\alpha$  führen wir eine Dummy Variable  $D$  ein und ersetzen die Klausel durch zwei Klauseln  $\alpha \vee D$  und  $\alpha \vee \neg D$ .

**Lemma 12.8** Eine Boolesche Formel  $H$  in 2-KNF ist genau dann unerfüllbar, wenn es in  $G_H$  einen Knoten  $x$  gibt, so dass es in  $G_H$  einen Pfad von  $x$  zu  $\neg x$  und einen Pfad von  $\neg x$  zu  $x$  gibt. **ohne Beweis**

Der Polynomialzeit-Algorithmus für 2-SAT ergibt sich dann aus Lemma 12.8. ■

## 12.3 Nichtdeterministische Polynomialzeit

Nun betrachten wir *nichtdeterministische* polynomialzeitbeschränkte Turingmaschinen.

**Definition 12.9 (nichtdeterministische Polynomialzeit: NP)** Sei  $t : \mathbb{N} \rightarrow \mathbb{N}$  eine Funktion und  $M$  eine nichtdeterministische TM mit  $L(M) \subseteq \Sigma^*$  für ein Alphabet  $\Sigma$ . Definiere

- die Funktion  $ntime_M : \Sigma^* \rightarrow \mathbb{N}$  durch

$$ntime_M(w) = \begin{cases} \min[\text{Zahl der Konfigurationen in einer} & \text{falls } w \in L(M) \\ \text{akzeptierenden Rechnung von } M(w)] & \\ \text{nicht definiert} & \text{sonst;} \end{cases}$$

- die Funktion  $Ntime_M : \mathbb{N} \rightarrow \mathbb{N}$  durch

$$Ntime_M(n) = \begin{cases} \max_{w: |w|=n} ntime_M(w) & \text{falls } ntime_M(w) \text{ für alle } w \text{ mit} \\ & |w| = n \text{ definiert ist} \\ \text{nicht definiert} & \text{sonst;} \end{cases}$$

- die Komplexitätsklasse

$$NTIME(t) = \left\{ A \mid \begin{array}{l} \text{es gibt eine nichtdeterministische TM } M \text{ mit} \\ L(M) = A \text{ und } Ntime_M(n) \leq t(n) \text{ für alle } n \end{array} \right\};$$

- die Komplexitätsklasse nichtdeterministische Polynomialzeit:

$$NP = \bigcup_{p \in \mathbb{P}ol} NTIME(p).$$

Offensichtlich gilt:

$$P \subseteq NP.$$

Ob die Umkehrung auch gilt, ist ein berühmtes offenes Problem, das so genannte „P-NP-Problem“. Die allgemeine Vermutung ist, dass  $P \neq NP$  gilt, aber ein Beweis dieser Vermutung steht noch aus (und scheint sehr schwierig zu sein).

Das Verhältnis von  $P$  zu  $NP$  ist ähnlich dem Verhältnis von  $REC$  zu  $RE$ ; dies spiegelt sich in einer Vielzahl von analogen Ergebnissen wieder. Beispielsweise gilt für  $NP$  eine „polynomialzeitbeschränkte“ Version des Projektionssatzes.

**Satz 12.10**  $A \in NP \iff$  es gibt eine Menge  $B \in P$  und ein Polynom  $p \in \mathbb{P}ol$ , so dass für alle  $x \in \Sigma^*$ :

$$x \in A \iff (\exists y \in \Sigma^*) [|y| \leq p(|x|) \wedge (x, y) \in B].$$

ohne Beweis

Die Klasse NP enthält eine Vielzahl wichtiger Probleme. Eines davon ist das uneingeschränkte Erfüllbarkeitsproblem.

**Lemma 12.11** SAT ist in NP.

**Beweis.** Betrachte die folgende nichtdeterministische Turingmaschine  $M$ , die SAT wie folgt in Polynomialzeit akzeptiert:

- Eingabe: Eine Boolesche Formel  $\varphi(x_1, x_2, \dots, x_n)$ .
- Rate nichtdeterministisch eine Belegung  $t$  der Variablen  $x_1, x_2, \dots, x_n$  mit Wahrheitswerten. („Ratephase“ bzw. „Auswahlphase“)  
Es existieren  $2^n$  mögliche nichtdeterministische unabhängige Rechnungen – für jede Belegung eine.
- Verifiziere auf jedem solchen Berechnungspfad, ob die dort geratene Belegung  $t$  die Formel  $\varphi$  erfüllt. („Verifikationsphase“)
- Akzeptiere genau dann, wenn dies der Fall ist (d.h., genau dann, wenn  $t(\varphi) = 1$ ).

Offenbar gilt  $L(M) = \text{SAT}$ . Da  $M$  in Polynomialzeit arbeitet, folgt  $\text{SAT} \in \text{NP}$ . ■

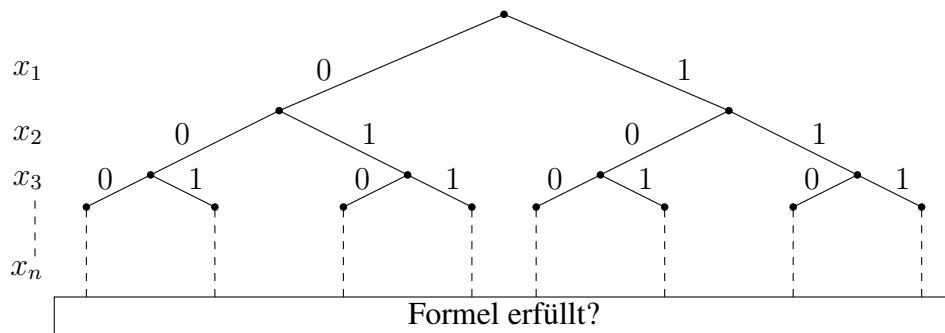


Abbildung 12.2: Berechnungsbaum eines nichtdeterministischen Algorithmus für SAT



# Kapitel 13

## NP-Vollständigkeit und der Satz von Cook

In diesem Abschnitt werden wir sehen, dass SAT eines der schwersten Probleme in NP ist: SAT ist das erste bekannte Beispiel eines NP-vollständigen Problems. Das bedeutet, dass SAT – mit bisher bekannten Techniken – nur in Exponentialzeit gelöst werden kann, also ein „störrisches“ Problem ist.

Da man in der Praxis aber sehr an „möglichst effizienten“ Algorithmen für SAT interessiert ist, versucht man:

- subexponentielle Algorithmen (d.h. Laufzeiten wie z. B.  $2^{\sqrt{n}}$  oder  $2^{\mathcal{O}(\sqrt{n \log n})}$ ) oder
- effiziente Heuristiken, die für die „praktisch relevanten“ Eingaben korrekt sind, oder
- Exponentialzeit-Algorithmen, die den trivialen  $\mathcal{O}(2^n)$ -Algorithmus für SAT verbessern,

zu finden.

Zum letzten Punkt: Abbildung 13.1 zeigt den Aufwand zur Lösung von SAT im Vergleich zweier Algorithmen, die in der Zeit  $2^n$  bzw.  $1.75^n$  laufen. Die Motivation ist die folgende:

- Läuft der triviale Algorithmus  $T$  für SAT in der Zeit  $2^n$ , dann kann ein  $c^n$ -Algorithmus  $A$  mit  $c < 2$  größere Probleminstanzen als  $T$  in derselben Zeit bearbeiten.
- Ist  $c = \sqrt{2}$ , dann kann  $A$  *doppelt so große Eingaben in derselben Zeit wie der triviale Algorithmus bearbeiten*, denn

$$\left(\sqrt{2}\right)^{2n} = 2^n.$$

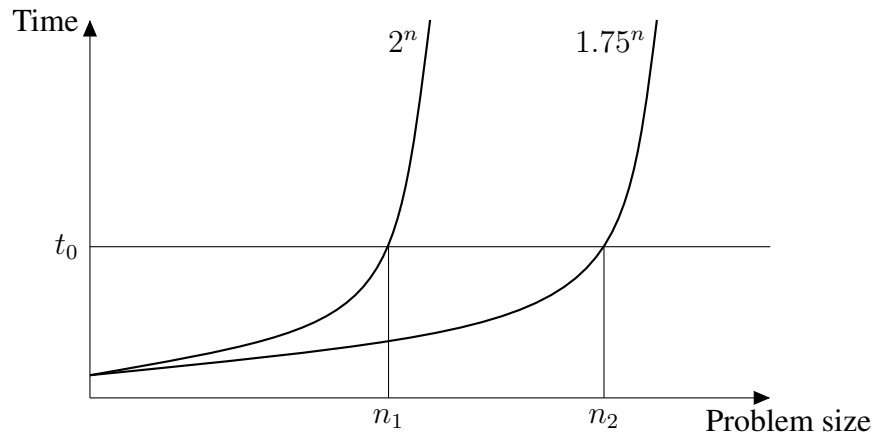


Abbildung 13.1: Verbesserter Exponentialzeit-Algorithmus für SAT

- Dieser Unterschied kann in der Praxis signifikant sein!
- Vergleiche: Ein 10-mal schnellerer Computer mit dem  $2^n$ -Algorithmus, würde  $n_1$  in Abbildung 13.1 nur geringfügig zu  $n_2$  verbessern:

$$\begin{aligned} 2^{n_2} &= 10 \cdot 2^{n_1} \\ n_2 &= n_1 + \log_2 10. \end{aligned}$$

## 13.1 NP-Vollständigkeit

**Definition 13.1 (FP, Reduzierbarkeit, NP-Vollständigkeit)** *FP bezeichne die Menge der in Polynomialzeit berechenbaren, totalen Funktionen von  $\Sigma^*$  in  $\Sigma^*$  für ein Alphabet  $\Sigma$ . Formal:*

$$\text{FP} = \left\{ f : \Sigma^* \rightarrow \Sigma^* \left| \begin{array}{l} f \text{ ist total und es gibt ein Polynom } p \in \mathbb{P} \text{ und eine} \\ \text{deterministische TM } M, \text{ die } f \text{ berechnet, und so dass} \\ \text{Time}_M(n) \leq p(n) \text{ für alle } n \end{array} \right. \right\}.$$

Seien  $A, B \subseteq \Sigma^*$  Mengen.

- $A$  ist in Polynomialzeit many-one-reduzierbar auf  $B$  (symbolisch:  $A \leq_m^p B$ ), falls gilt:

$$(\exists f \in \text{FP}) (\forall x \in \Sigma^*) [x \in A \iff f(x) \in B].$$

- Eine Menge  $B$  heißt NP-vollständig (bzgl.  $\leq_m^p$ ), falls gilt:

$$1. \ B \in \text{NP}.$$

2.  $(\forall A \in \text{NP}) [A \leq_m^P B]$ . (Sprich:  $B$  ist NP-hart (oder NP-schwer).)

Die NP-vollständigen Mengen sind also die schwersten Probleme in NP. NP-Härte wird intuitiv mit „Ineffizienz“ gleichgesetzt, denn bis heute ist für keine NP-harte Menge ein effizienter Algorithmus bekannt.

Abbildung 13.2 zeigt die Beziehungen zwischen den Klassen NP, P und der Menge aller NP-vollständigen Probleme (sofern  $P \neq \text{NP}$ ).

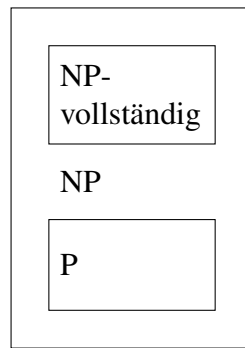


Abbildung 13.2: Beziehungen zwischen den Klassen NP, P und der Menge aller NP-vollständigen Probleme (sofern  $P \neq \text{NP}$ )

Wir fassen nun einige wichtige Eigenschaften der Relation  $\leq_m^P$  zusammen.

**Lemma 13.2** Es seien  $A, B \subseteq \Sigma^*$  Mengen.

1. Die Relation  $\leq_m^P$  ist reflexiv und transitiv, aber nicht antisymmetrisch.
2. P und NP sind  $\leq_m^P$ -abgeschlossen, d.h.,

$$(A \leq_m^P B \wedge B \in \text{P}) \Rightarrow A \in \text{P}$$

und

$$(A \leq_m^P B \wedge B \in \text{NP}) \Rightarrow A \in \text{NP}.$$

Das heißt, Mitgliedschaft in P und NP („effiziente Lösbarkeit durch (nicht-)deterministische Maschinen“) vererbt sich bzgl.  $\leq_m^P$  nach unten.

3. NP-Härte („Ineffizienz“) vererbt sich bzgl.  $\leq_m^P$  nach oben, d.h.:

$$(A \leq_m^P B \wedge A \text{ ist NP-hart}) \Rightarrow B \text{ ist NP-hart}.$$

4.  $A \leq_m^P B \Rightarrow \overline{A} \leq_m^P \overline{B}$ , aber i.a. gilt nicht:  $A \leq_m^P \overline{A}$ .

5. Ist  $B$  NP-vollständig, dann gilt:  $B \in P \iff NP = P$ .

**Beweis.**

1. Übung.

2. Es sei  $A \leq_m^P B$  mittels  $f \in FP$ . Die deterministische Turingmaschine  $M_f$  berechne  $f$  in Polynomialzeit  $p$ . Es sei  $B \in P$  mittels der deterministischen Turingmaschine  $M_B$  mit polynomieller Rechenzeit  $q$ .

Aus diesen beiden Turingmaschinen konstruieren wir eine Turingmaschine  $M_A$ , die  $A$  entscheidet: Eine Instanz  $x$  für  $M_A$  wird zuerst in  $f(x)$  umgewandelt und dann wird  $M_B(f(x)) = M_A(x)$  berechnet.

Die Laufzeit ist polynomiell:  $p(|x|) + q(|f(x)|) \leq p(|x|) + q(p(|x|))$

Falls  $B \in NP$ , so sind die Maschinen nichtdeterministisch.

3. Da  $A$  NP-hart ist, gilt  $\forall A' \in NP : A' \leq_m^P A$ . Da nach Voraussetzung  $A \leq_m^P B$  gilt, folgt aus der Transitivität von  $\leq_m^P$ , dass  $\forall A' \in NP : A' \leq_m^P B$ . Damit folgt unmittelbar aus der Definition, dass  $B$  NP-hart ist.

4. Übung.

5. ( $\Rightarrow$ ) Es sei  $B \in P$ . Da  $B$  NP-vollständig ist, gilt für alle Sprachen  $A \in NP$ :  $A \leq_m^P B$ . Da  $B \in P$ , gilt  $\forall A \in NP : A \in P$  nach Teil (2.). Da  $P \subseteq NP$ , folgt  $P = NP$ .

( $\Leftarrow$ ) Da  $B$  NP-vollständig ist, gilt  $B \in NP$ , und da nach Voraussetzung  $NP = P$  gilt, folgt  $B \in P$ .



**Bemerkung 13.3** Eigenschaft 5 von Lemma 13.2 sagt, dass ein polynomieller (effizienter) Algorithmus für (irgend)ein NP-hartes Problem impliziert, dass NP auf P kollabiert, und das P-NP-Problem wäre somit gelöst. Die weit verbreitete Annahme, dass  $P \neq NP$  gilt, bedeutet, dass es vermutlich keinen effizienten Algorithmus für ein NP-hartes Problem gibt.

## 13.2 Der Satz von Cook

**Satz 13.4 (Satz von Cook)** SAT ist NP-vollständig.

**Beweis.** Nach Definition 13.1 sind die folgenden zwei Aussagen zu zeigen:

1. SAT ist in NP: siehe Lemma 12.11.



## 2. SAT ist NP-hart.

Sei  $A$  eine beliebige Menge in NP, und sei  $M = (\Sigma, \Gamma, Z, \delta, \square, s_0, F)$  eine nicht-deterministische Turingmaschine, die  $A$  in Polynomialzeit akzeptiert. Wir nehmen an, dass  $M$  bei jeder Eingabe  $x$  der Länge  $n$  *genau*  $p(n) \geq n$  Takte rechnet, für ein  $p \in \mathbb{P}\text{ol}$ . Unser Ziel ist es, eine Reduktion  $f \in \text{FP}$  anzugeben, so dass für jedes  $x \in \Sigma^*$  gilt:

$$x \in A \iff f(x) = F_x \in \text{SAT}, \quad (13.1)$$

wobei  $F_x$  eine Boolesche Formel ist, deren Struktur und deren Variablen nun beschrieben werden.

Sei ein Eingabewort  $x = x_1x_2 \cdots x_n$  mit  $x_i \in \Sigma$  für alle  $i$  gegeben. Da  $M$  in der Zeit  $p(n)$  arbeitet, kann sich der Kopf nicht weiter als um  $p(n)$  Bandzellen nach links oder rechts bewegen. Nummeriere entsprechend die relevanten Bandzellen von  $-p(n)$  bis  $p(n)$ . Abbildung 13.3 zeigt das Band von  $M$  zu Beginn der Rechnung mit der Startkonfiguration: die Eingabesymbole  $x_1x_2 \cdots x_n$  stehen auf den Bandzellen 0 bis  $n-1$ , der Kopf liest die Zelle mit Nr. 0 und der Startzustand ist  $s_0$ .

...	$\square$	...	$\square$	$x_1$	$x_2$	...	$x_2$	$\square$	...	$\square$	...
...	$-p(n)$	...	$-1$	0	1	...	$n-1$	$n$	...	$p(n)$	...

Abbildung 13.3: Nummerierung der Bandzellen der NTM  $M$  bei Eingabe  $x$

Wir konstruieren die Formel  $F_x$ , so dass (13.1) gilt. Tabelle 13.1 gibt die Variablen von  $F_x$ , den Bereich ihrer Indizes und ihre Bedeutung an. Es gibt drei Typen von Variablen:

- $\text{state}_{t,s}$  repräsentiert den Zustand  $s$  von  $M$  in Takt  $t$ ;
- $\text{head}_{t,i}$  repräsentiert die Nummer  $i$  der Bandzelle, die der Kopf von  $M$  in Takt  $t$  liest;
- $\text{tape}_{t,i,a}$  repräsentiert das Symbol  $a \in \Gamma$  in der Bandzelle Nr.  $i$  in Takt  $t$ .

$F_x$  hat die Form:

$$F_x = S \wedge \dot{U}_1 \wedge \ddot{U}_2 \wedge E \wedge K,$$

wobei diese Teilformeln von  $F_x$  den folgenden Zweck haben:

- $S$  ist erfüllbar  $\iff$  die Rechnung von  $M(x)$  *startet* korrekt;
- $\ddot{U}_1$  ist erfüllbar  $\iff$  die *Übergänge* der Rechnung von  $M(x)$  sind in jedem Takt korrekt an den Bandfeldern mit Kopf;

Variable	Indexbereich	Bedeutung
$\text{state}_{t,s}$	$t \in \{0, 1, \dots, p(n)\}$ $s \in Z$	$\text{state}_{t,s}$ ist wahr $\iff$ $M$ ist im Takt $t$ in Zustand $s$
$\text{head}_{t,i}$	$t \in \{0, 1, \dots, p(n)\}$ $i \in \{-p(n), \dots, p(n)\}$	$\text{head}_{t,i}$ ist wahr $\iff$ $M$ 's Kopf liest im Takt $t$ die Bandzelle Nr. $i$
$\text{tape}_{t,i,a}$	$t \in \{0, 1, \dots, p(n)\}$ $i \in \{-p(n), \dots, p(n)\}$ $a \in \Gamma$	$\text{tape}_{t,i,a}$ ist wahr $\iff$ Symbol $a$ steht im Takt $t$ in Bandzelle Nr. $i$

Tabelle 13.1: Die Booleschen Variablen von  $F_x$  und ihre Bedeutung in der Cook-Reduktion

- $\dot{U}_2$  ist erfüllbar  $\iff$  die *Übergänge* der Rechnung von  $M(x)$  sind in jedem Takt korrekt an den Bandfeldern ohne Kopf;
- $E$  ist erfüllbar  $\iff$  die Rechnung von  $M(x)$  *endet* korrekt (d.h.,  $M$  akzeptiert  $x$ );
- $K$  ist erfüllbar  $\iff$  die allgemeine *Korrektheit* der Rechnung von  $M(x)$  ist gegeben, d.h.:
  - in jedem Takt der Rechnung von  $M(x)$  ist  $M$  in genau einem Zustand und der Kopf von  $M$  steht auf genau einem Feld und
  - in jedem Takt und für jede Bandposition gibt es genau ein Symbol auf diesem Feld des Bandes).

Um diese Teilformeln von  $F_x$  formal zu beschreiben, seien die Zustandsmenge  $Z$  und das Arbeitsalphabet  $\Gamma$  von  $M$  gegeben durch:

$$\begin{aligned} Z &= \{s_0, s_1, \dots, s_k\} \\ \Gamma &= \{\square, a_1, a_2, \dots, a_\ell\}. \end{aligned}$$

Dabei gilt  $\Sigma \subseteq \Gamma$ .

Definiere die Teilformel  $K$  von  $F_x$ , die die *allgemeine Korrektheit* beschreibt, durch:

$$\begin{aligned} K = \bigwedge_{0 \leq t \leq p(n)} [ & D_1(\text{state}_{t,s_0}, \text{state}_{t,s_1}, \dots, \text{state}_{t,s_k}) \wedge \\ & D_2(\text{head}_{t,-p(n)}, \text{head}_{t,-p(n)+1}, \dots, \text{head}_{t,p(n)}) \wedge \quad (13.2) \\ & \bigwedge_{-p(n) \leq i \leq p(n)} D_3(\text{tape}_{t,i,\square}, \text{tape}_{t,i,a_1}, \dots, \text{tape}_{t,i,a_\ell}) ], \end{aligned}$$

wobei die Struktur der drei Teilformeln  $D_i$  von  $K$  in (13.2) mittels Lemma 13.5 unten spezifiziert wird. Insbesondere gilt für  $i \in \{1, 2, 3\}$ :

- (a)  $D_i$  ist genau dann wahr, wenn *genau* eine der Variablen von  $D_i$  wahr ist, und
- (b) die Größe von  $D_i$  – und folglich auch die Größe von  $K$  – ist polynomiell in  $n$ .

**Lemma 13.5** *Für jedes  $m$  gibt es eine Boolesche Formel  $D$  mit  $m$  Variablen, so dass gilt:*

- (a)  $D(v_1, v_2, \dots, v_m)$  ist wahr  $\iff$  genau eine Variable  $v_i$  ist wahr;
- (b) die Größe von  $D$  (d.h., die Zahl der Vorkommen von Variablen in  $D$ ) ist in  $\mathcal{O}(m^2)$ .

**Beweis von Lemma 13.5.** Für festes  $m \geq 1$  sei die Formel  $D$  definiert durch

$$D(v_1, v_2, \dots, v_m) = \underbrace{\left( \bigvee_{i=1}^m v_i \right)}_{D_{\geq}} \wedge \underbrace{\left( \bigwedge_{j=1}^{m-1} \bigwedge_{k=j+1}^m \neg(v_j \wedge v_k) \right)}_{D_{\leq}}.$$

Die beiden Teilformeln  $D_{\geq}$  und  $D_{\leq}$  von  $D$  haben die folgenden Eigenschaften:

$$D_{\geq}(v_1, v_2, \dots, v_m) \text{ ist wahr} \iff \text{mindestens eine Variable } v_i \text{ ist wahr; (13.3)}$$

$$D_{\leq}(v_1, v_2, \dots, v_m) \text{ ist wahr} \iff \text{höchstens eine Variable } v_i \text{ ist wahr. (13.4)}$$

Die Äquivalenz (13.3) ist offensichtlich. Dass auch die Äquivalenz (13.4) gilt, ergibt sich aus der folgenden Struktur der Teilformel  $D_{\leq}$ :

$$\begin{aligned} D_{\leq}(v_1, v_2, \dots, v_m) = & (\neg v_1 \vee \neg v_2) \wedge (\neg v_1 \vee \neg v_3) \wedge \dots \wedge (\neg v_1 \vee \neg v_m) \\ & \wedge (\neg v_2 \vee \neg v_3) \wedge \dots \wedge (\neg v_2 \vee \neg v_m) \\ & \vdots \\ & \wedge (\neg v_{m-1} \vee \neg v_m). \end{aligned}$$

Die Äquivalenzen (13.3) und (13.4) zusammen implizieren, dass  $D(v_1, v_2, \dots, v_m)$  genau dann wahr ist, wenn genau eine Variable  $v_i$  wahr ist. Offenbar ist die Größe von  $D$  in  $\mathcal{O}(m^2)$ . ■ Lemma 13.5

Um den Beweis des Satzes fortzusetzen, definiere die Teilformel  $S$  von  $F_x$ , die für Takt  $t = 0$  den korrekten Beginn der Rechnung  $M(x)$  beschreibt (siehe Abbildung 13.3), durch

$$S = \text{state}_{0,s_0} \wedge \text{head}_{0,0} \wedge \bigwedge_{i=-p(n)}^{-1} \text{tape}_{0,i,\square} \wedge \bigwedge_{i=0}^{n-1} \text{tape}_{0,i,x_{i+1}} \wedge \bigwedge_{i=n}^{p(n)} \text{tape}_{0,i,\square}.$$

Definiere die Teilformel  $\ddot{U}_1$  von  $F_x$ , die den korrekten Übergang von Takt  $t$  zu Takt  $t + 1$  für die aktuell von  $M$ 's Kopf gelesenen Bandzellen beschreibt, durch

$$\ddot{U}_1 = \bigwedge_{t < p(n), s, i, a} \left( (\text{state}_{t,s} \wedge \text{head}_{t,i} \wedge \text{tape}_{t,i,a}) \implies \bigvee_{\substack{\hat{s} \in Z, \hat{a} \in \Gamma, y \in \{-1, 0, 1\} \\ \text{mit } (\hat{s}, \hat{a}, y) \in \delta(s, a)}} (\text{state}_{t+1, \hat{s}} \wedge \text{head}_{t+1, i+y} \wedge \text{tape}_{t+1, i, \hat{a}}) \right),$$

wobei  $\delta$  die Überföhrungsfunktion von  $M$  ist und  $y \in \{-1, 0, 1\}$  die Kopfbewegung repräsentiert.

Definiere die Teilformel  $\ddot{U}_2$  von  $F_x$ , die den korrekten Übergang von Takt  $t$  zu Takt  $t + 1$  für die aktuell von  $M$ 's Kopf *nicht* gelesenen Bandzellen beschreibt, durch

$$\ddot{U}_2 = \bigwedge_{t < p(n), i, a} \left( (\neg \text{head}_{t,i} \wedge \text{tape}_{t,i,a}) \implies \text{tape}_{t+1, i, a} \right).$$

Definiere die Teilformel  $E$  von  $F_x$ , die das korrekte Ende der Berechnung  $M(x)$  beschreibt und überprüft, ob  $M$  die Eingabe  $x$  akzeptiert:

$$E = \bigvee_{s \in F} \text{state}_{p(n), s},$$

wobei  $F$  die Menge der Endzustände von  $M$  ist.<sup>1</sup>

Damit ist die Reduktion  $f$  beschrieben. Analysiert man die Struktur der Formel  $f(x) = F_x$  und benutzt Lemma 13.5, so folgt  $f \in \text{FP}$ . Es bleibt zu zeigen, dass (13.1) gilt:

$$x \in A \iff f(x) = F_x \text{ ist erfüllbar.}$$

( $\Rightarrow$ ) Sei  $x \in A$ . Dann gibt es einen akzeptierenden Berechnungspfad  $\alpha$  von  $M(x)$ . Weist man nun jeder Variablen von  $F_x$  gemäß ihrer beabsichtigten Bedeutung der Variablen aus Tabelle 13.1 einen  $\alpha$  entsprechenden Wahrheitswert zu, dann erfüllt diese Belegung jede Teilformel von  $F_x$ , also auch  $F_x$  selbst. Somit ist  $F_x \in \text{SAT}$ .

( $\Leftarrow$ ) Sei nun  $F_x \in \text{SAT}$ . Dann gibt es eine Belegung  $\tau$ , die  $F_x$  erfüllt. Die Variablen  $\text{state}_{t,s}$ ,  $\text{head}_{t,i}$  und  $\text{tape}_{t,i,a}$  von  $F_x$  können gemäß  $\tau$  als eine Folge  $K_0, K_1, \dots, K_{p(n)}$  von Konfigurationen von  $M(x)$  entlang eines Berechnungspfad es interpretiert werden. Da  $\tau(F_x) = 1$ , muss  $\tau$  jede Teilformel von  $F_x$  erfüllen. Somit gilt:

<sup>1</sup>Anmerkung: ein Endzustand kann auch vor dem letzten Schritt  $p(n)$  erreicht werden. Deshalb nehmen wir an, dass Endzustände Selbstschleifen haben, oder wir passen  $\delta$  für  $\ddot{U}_1$  so an, dass  $\delta(s, a) = \{(s, a, 0)\}$  für  $s \in F$ .

- $\tau(S) = 1$  impliziert, dass  $K_0$  die Startkonfiguration von  $M(x)$  ist,
- $\tau(\ddot{U}_1) = \tau(\ddot{U}_2) = \tau(K) = 1$  impliziert, dass  $K_{t-1} \vdash_M K_t$  für jedes  $t$  mit  $1 \leq t \leq p(n)$  gilt, und
- $\tau(E) = 1$  impliziert, dass  $K_{p(n)}$  eine akzeptierende Endkonfiguration von  $M(x)$  ist.

Folglich ist  $x \in A$ .

Die Äquivalenz (13.1) ist gezeigt.

Es bleibt zu zeigen, dass  $F_x$  in polynomieller Zeit berechenbar ist.

Der Aufwand zu Erzeugung von  $F_x$  ist offenbar linear in der Länge von  $F_x$ .

$$\begin{array}{rcl}
 |S| & \in & \mathcal{O}(p(n)) \\
 |\ddot{U}_1| & \in & \mathcal{O}((p(n))^2) \\
 |\ddot{U}_2| & \in & \mathcal{O}((p(n))^2) \\
 |E| & \in & \mathcal{O}(1) \\
 |K| & \in & \mathcal{O}((p(n))^3) \\
 \hline
 |F_x| & \in & \mathcal{O}((p(n))^3)
 \end{array}$$

Damit ist der Satz bewiesen. ■

Da die im Beweis von Satz 13.4 konstruierte Boolesche Formel  $F_x$  sogar in KNF ist bzw. ohne zu große Aufblähung der Formel in KNF gebracht werden kann, erhalten wir die folgende Folgerung.

**Korollar 13.6** KNF-SAT ist NP-vollständig.

**Bemerkung 13.7** Die bekannten deterministischen Algorithmen zur Berechnung von SAT (z.B. systematisches Durchprobieren aller Belegungen) haben alle eine Zeitkomplexität von  $2^{\mathcal{O}(n)}$ . Da nach Satz 13.4 jede Sprache  $L \in \text{NP}$  auf SAT reduziert werden kann ( $\forall L \in \text{NP} : L \leq_m^p \text{SAT}$ ), kann die deterministische Zeitkomplexität von  $L$  nach oben durch  $2^{p(n)}$  für ein Polynom  $p$  abgeschätzt werden (s. Beweis von Lemma 13.2).

Das heißt,

$$\text{NP} \subseteq \bigcup_{p \in \text{Pol}} \text{TIME}(2^{p(n)}).$$