

# **Automatentheorie und Formale Sprachen**

Sommersemester 2025

**Prof. Dr. David Sabel**

Theoretische Informatik  
Fachbereich DCSM  
Hochschule RheinMain  
Unter den Eichen 5  
65195 Wiesbaden

Email: [david.sabel@hs-rm.de](mailto:david.sabel@hs-rm.de)

Dieses Skript wird im Laufe der Vorlesung überarbeitet!

Stand: 2. Juli 2025

# Inhaltsverzeichnis

<b>1. Einleitung und Grundlagen</b>	<b>1</b>
1.1. Notationen, Alphabete, Wörter und Sprachen . . . . .	1
<b>2. Chomsky-Grammatiken und die Chomsky-Hierarchie</b>	<b>5</b>
2.1. Grammatiken . . . . .	6
2.2. Die Chomsky-Hierarchie . . . . .	9
2.2.1. Erzeugung des leeren Worts und $\varepsilon$ -Produktionen . . . . .	10
2.2.2. Beziehungen zwischen den Typ $i$ -Sprachen . . . . .	11
2.3. Das Wortproblem . . . . .	13
2.4. Weitere Probleme für Formale Sprachen . . . . .	14
2.5. Syntaxbäume . . . . .	15
2.6. Die Backus-Naur-Form für Grammatiken . . . . .	17
<b>3. Reguläre Sprachen</b>	<b>18</b>
3.1. Deterministische endliche Automaten . . . . .	18
3.1.1. Minimierung von DFAs . . . . .	21
3.1.2. DFAs akzeptieren reguläre Sprachen . . . . .	26
3.2. Nichtdeterministische Endliche Automaten . . . . .	26
3.3. Reguläre Sprachen können durch NFAs erkannt werden . . . . .	28
3.4. Überführung von NFAs in DFAs . . . . .	29
3.5. NFAs mit $\varepsilon$ -Übergängen . . . . .	32
3.6. Reguläre Ausdrücke . . . . .	36
3.7. Zusammenfassende Darstellung der Formalismen für reguläre Sprachen . . . . .	41
3.8. Widerlegen der Regulärität: Das Pumping-Lemma . . . . .	41
3.9. Abschlusseigenschaften der regulären Sprachen . . . . .	45
3.10. Entscheidbarkeitsresultate zu regulären Sprachen . . . . .	47
<b>4. Kontextfreie Sprachen</b>	<b>48</b>
4.1. Normalformen für kontextfreie Grammatiken . . . . .	48
4.1.1. Die Chomsky-Normalform . . . . .	48
4.1.2. Die Greibach-Normalform . . . . .	50
4.2. Widerlegen der Kontextfreiheit . . . . .	50
4.3. Abschlusseigenschaften kontextfreier Sprachen . . . . .	55
4.4. Effiziente Lösung des Wortproblems: Der CYK-Algorithmus . . . . .	56

4.5.	Kellerautomaten . . . . .	58
4.5.1.	Akzeptanz durch Endzustände . . . . .	62
4.5.2.	Äquivalenz von Kellerautomaten und kontextfreien Sprachen . . . . .	63
4.5.2.1.	Kontextfreie Sprachen werden durch Kellerautomaten erkannt . . . . .	64
4.5.2.2.	Kellerautomaten akzeptieren kontextfreie Sprachen . . . . .	65
4.6.	Deterministisch kontextfreie Sprachen . . . . .	68
4.7.	Entscheidbarkeitsresultate . . . . .	70
<b>5.</b>	<b>Kontextsensitive und Typ 0-Sprachen</b>	<b>72</b>
5.1.	Turingmaschinen . . . . .	72
5.2.	Linear beschränkte Turingmaschinen und kontextsensitive Sprachen . . . . .	75
5.3.	Turingmaschinen und Typ 0-Grammatiken . . . . .	76
<b>6.</b>	<b>Überblick über die Chomsky-Hierarchie</b>	<b>77</b>
<b>7.</b>	<b>Berechenbarkeit</b>	<b>79</b>
7.1.	Einleitung . . . . .	79
7.2.	Turingmaschinen und Turingberechenbarkeit . . . . .	80
7.2.1.	Mehrspuren- und Mehrband-Turingmaschinen . . . . .	82
7.2.2.	Mehrspuren-Turingmaschinen . . . . .	83
7.2.3.	Mehrband-Turingmaschinen . . . . .	85
7.3.	Unentscheidbarkeit . . . . .	87
7.3.1.	Gödelisierung von Turingmaschinen . . . . .	89
7.3.2.	Das Halteproblem . . . . .	90
7.3.2.1.	Unentscheidbarkeit des speziellen Halteproblems . . . . .	90
7.3.2.2.	Reduktionen . . . . .	92
7.3.2.3.	Unentscheidbarkeit des Halteproblems . . . . .	93
7.3.2.4.	Halteproblem bei leerer Eingabe . . . . .	93
7.4.	Das Postsche Korrespondenzproblem . . . . .	94
<b>8.</b>	<b>Komplexitätstheorie: Zeitkomplexität, <math>\mathcal{P}</math> und <math>\mathcal{NP}</math></b>	<b>96</b>
8.1.	Deterministische Zeitkomplexität . . . . .	96
8.2.	Nichtdeterminismus . . . . .	98
8.3.	Das P-vs-NP-Problem . . . . .	99
8.4.	NP-Vollständigkeit . . . . .	99
8.4.1.	Polynomialzeit-Reduktionen . . . . .	99
8.4.2.	Satz von Cook . . . . .	102
8.4.3.	NP-vollständige Probleme und die Praxis . . . . .	104
8.5.	Eine Auswahl NP-vollständiger Probleme . . . . .	104
8.5.1.	Das 3-CNF-SAT-Problem . . . . .	105
8.5.2.	Das CLIQUE-Problem . . . . .	107

8.5.3.	Das INDEPENDENT-SET-Problem . . . . .	109
8.5.4.	Das VERTEX-COVER-Problem . . . . .	111
8.5.5.	Das SETCOVER-Problem . . . . .	112
8.5.6.	Das SUBSETSUM-Problem . . . . .	113
8.5.7.	Das KNAPSACK-Problem . . . . .	115
8.5.8.	Das PARTITION-Problem . . . . .	116
8.5.9.	Das BINPACKING-Problem . . . . .	117
8.5.10.	Das DIRECTED-HAMILTON-CYCLE-Problem . . . . .	118
8.5.11.	Das UNDIRECTED-HAMILTON-CYCLE-Problem . . . . .	121
8.5.12.	Das TRAVELLING-SALESPERSON-Problem . . . . .	122
8.5.13.	Das GRAPH-COLORING-Problem . . . . .	123
 <b>Literatur</b>		 <b>126</b>
 <b>A. Zusatzmaterial</b>		 <b>128</b>
A.1.	Mathematische Grundlagen . . . . .	128
A.1.1.	Vollständige Induktion . . . . .	128
A.1.2.	Relationen . . . . .	129
A.1.3.	Funktionen . . . . .	131
A.1.4.	Abzählbarkeit und Überabzählbarkeit . . . . .	132
A.1.5.	Asymptotische Notation . . . . .	134
A.2.	Entfernen von $\varepsilon$ -Produktionen in kontextfreien Grammatiken . . . . .	134
A.3.	Der Satz von Myhill und Nerode . . . . .	136
A.4.	Operationen auf CFGs . . . . .	139
A.5.	Herstellen der Chomsky-Normalform . . . . .	140
A.5.1.	Entfernen von Einheitsproduktionen . . . . .	140
A.5.2.	Ersetzen von Terminalen und Zerlegen von langen rechten Seiten . . . . .	142
A.6.	Herstellen der Greibach-Normalform . . . . .	145
A.7.	Die Kuroda-Normalform für kontextsensitive Grammatiken . . . . .	148
A.8.	Beweis des Satzes von Kuroda . . . . .	149
A.9.	Der Satz von Rice . . . . .	153
A.10.	Unentscheidbarkeit des Postschen Korrespondenzproblems . . . . .	154
A.10.1.	$MPCP \leq PCP$ . . . . .	154
A.10.2.	$H \leq MPCP$ . . . . .	155
A.11.	$\mathcal{NP}$ -Schwere von SAT . . . . .	157

# 1. Einleitung und Grundlagen

Dieses Skript fasst die wesentlichen Inhalte der Vorlesung „Automatentheorie und Formale Sprachen“ zusammen. Dabei orientieren sich die Inhalte auf eine Auswahl aus Schöning’s Buch „Theoretische Informatik – kurz gefasst“ (Sch08).

Die Vorlesung behandelt drei wesentliche Teilgebiete der Theoretischen Informatik. Der größte Teil befasst sich mit den formalen Grammatiken und den Grundlagen der Automatentheorie. Auch Teile der Berechenbarkeitstheorie werden behandelt, die sich mit der Frage beschäftigt, welche Probleme *überhaupt* mit dem Rechner *gelöst* werden können. Schließlich wird eine Einführung in die Komplexitätstheorie gegeben, die sich mit der Frage beschäftigt, welche Probleme *effizient lösbar* sind und was selbiges heißt.

Im Rest dieses Einleitungskapitel erläutern wir wichtige Grundlagen. Weiterführende Grundlagen, die als bekannt vorausgesetzt werden, sowie vertiefendes Material, das nicht in der Vorlesung behandelt wird, sind im Anhang A zu finden.

## 1.1. Notationen, Alphabete, Wörter und Sprachen

Wir bezeichnen die Menge der natürlichen Zahlen (ohne 0) mit  $\mathbb{N}$ , d. h.  $\mathbb{N} = \{1, 2, 3, \dots\}$  und mit  $\mathbb{N}_0$  bezeichnen wir die Menge der natürlichen Zahlen mit Null, d. h.  $\mathbb{N}_0 = \{1, 2, 3, \dots\}$ .

In logischen Formeln schreiben wir die Implikation als  $F \implies G$  und die Bimplikation als  $F \iff G$ . Die logische Äquivalenz (als Äquivalenzrelation auf Formeln) schreiben wir als  $F \longleftrightarrow G$ .

Das Summenzeichen  $\sum$  ist auch ein griechisches Sigma  $\Sigma$ . Im folgenden wird das Symbol  $\Sigma$  jedoch mit einer anderen Bedeutung verwendet:

**Definition 1.1.1** (Alphabet, Wort, Konkatenation). *Ein Alphabet ist eine endliche nicht-leere Menge von Zeichen (oder Symbolen). Wir bezeichnen Alphabete oft mit dem Symbol  $\Sigma$ .*

*Sei ein Alphabet  $\Sigma$  gegeben. Ein Wort  $w$  über  $\Sigma$  ist eine endliche Folge von Zeichen aus  $\Sigma$ . Die leere Folge ist auch ein solches Wort und wird mit  $\varepsilon$  notiert und als leeres Wort bezeichnet. Für ein Wort  $w = a_1 \cdots a_n$  notieren wir mit  $|w| = n$  die Länge des Wortes (wobei  $|\varepsilon| = 0$ ). Sei  $a \in \Sigma$  und  $w$  ein Wort über  $\Sigma$ . Mit  $\#_a(w) \in \mathbb{N}_0$  notieren wir die Anzahl an Vorkommen des Zeichens  $a$  im Wort  $w$ . Für  $1 \leq i \leq |w|$  bezeichnen wir mit  $w[i]$  das  $i$ . Zeichen von Wort  $w$ . Mit  $\Sigma^*$  bezeichnen wir die Menge aller Wörter über  $\Sigma$ . Eine formale (rekursive) Definition für  $\Sigma^*$  ist die folgende, wobei  $\Sigma^i$  die Menge der Wörter der Länge  $i$  über dem Alphabet  $\Sigma$  bezeichnet. Die Definition der Mengen  $\Sigma^i$  ist:*

- $\Sigma^0 := \{\varepsilon\}$
- $\Sigma^i := \{aw \mid a \in \Sigma, w \in \Sigma^{i-1}\}$  für  $i > 0$ .

Schließlich definiere  $\Sigma^* := \bigcup_{i \in \mathbb{N}_0} \Sigma^i$  und  $\Sigma^+ := \bigcup_{i \in \mathbb{N}} \Sigma^i$ . Es gilt  $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ .

Seien  $u$  und  $v$  Wörter über  $\Sigma$ , dann bezeichne  $u \circ v$  (alternativ auch nur  $uv$ ) die Konkatenation von  $u$  und  $v$ , d. h. jenes Wort, das entsteht, indem  $v$  hinten an  $u$  angehängt wird.

**Beispiel 1.1.2.** Sei  $\Sigma = \{a, b\}$ . Dann ist  $\Sigma^0 = \{\varepsilon\}$ ,  $\Sigma^1 = \Sigma$ ,  $\Sigma^2 = \{aa, ab, ba, bb\}$  und  $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \dots\}$  und z. B.  $aabbb \in \Sigma^*$  aber  $abc \notin \Sigma^*$ . Für  $u = aab$  und  $v = aabbb$  gilt  $|u| = 3$ ,  $|v| = 6$  und  $u \circ v = uv = aabaabbb$ , sowie z. B.  $\#_b(u) = 1$  und  $\#_a(uv) = \#_a(u) + \#_a(v) = 2 + 3 = 5$ .

Wir schreiben  $w^m$  für das  $m$ -malige Konkatenieren von  $w$ , d. h.  $w^0 = \varepsilon$  und  $w^m = w \circ w^{m-1}$  für  $m > 0$ .

Für eine endliche Menge  $M$  bezeichne  $|M|$  die Mächtigkeit von  $M$ .

**Definition 1.1.3.** Sei  $\Sigma$  ein Alphabet und  $w$  ein Wort über  $\Sigma$ . Mit  $\bar{w}$  bezeichnen wir das rückwärtsgelesene Wort  $w$ , d. h.  $\bar{\varepsilon} = \varepsilon$ ,  $\bar{a} = a$  für  $a \in \Sigma$  und  $\overline{aw} = \bar{w}a$  für alle  $a \in \Sigma, w \in \Sigma^*$ .

Ein Wort  $w \in \Sigma^*$  ist ein Palindrom, falls gilt  $w = \bar{w}$ , d. h.  $w$  ist vorwärts wie rückwärts gelesen dasselbe Wort.

**Beispiel 1.1.4.** Sei  $w = \text{informatik}$  ein Wort über  $\Sigma = \{a, \dots, z\}$ . Dann ist  $\bar{w} = \text{kitamrofni}$  und  $w$  ist kein Palindrom. Hingegen ist  $w' = \text{reliefpfeiler}$  ein Palindrom, denn  $\overline{w'} = \text{reliefpfeiler}$ .

**Definition 1.1.5** (Sprache und Operationen auf Sprachen). Eine (formale) Sprache über dem Alphabet  $\Sigma$  ist eine Teilmenge von  $\Sigma^*$ . Oft bezeichnen wir Sprachen mit dem Buchstaben  $L$  (für „language“). Für Sprachen  $L, L_1$  und  $L_2$  über dem Alphabet  $\Sigma$  sei

- $L_1 \cup L_2 := \{w \mid w \in L_1 \text{ oder } w \in L_2\}$  (die Vereinigung der Sprachen  $L_1$  und  $L_2$ )
- $L_1 \cap L_2 := \{w \mid w \in L_1 \text{ und } w \in L_2\}$  (der Schnitt der Sprachen  $L_1$  und  $L_2$ )
- $\bar{L} := \Sigma^* \setminus L$  (das Komplement von  $L$ )
- $L_1 \circ L_2 = L_1 L_2 := \{uv \mid u \in L_1 \text{ und } v \in L_2\}$  (das Produkt von  $L_1$  und  $L_2$ )

**Bemerkung 1.1.6.** Für Mengen kennt man das Kreuzprodukt (oder auch kartesisches Produkt): Wenn  $M_1$  und  $M_2$  Mengen sind, so ist

$$M_1 \times M_2 := \{(e_1, e_2) \mid e_1 \in M_1, e_2 \in M_2\}$$

Dieses berechnet die Menge aller geordneten Paare  $(e_1, e_2)$ , wobei  $e_1$  aus  $M_1$  und  $e_2$  aus  $M_2$  stammt. Auch für Sprachen  $L_1, L_2$  können wir das Kreuzprodukt verwenden (da Sprachen auch Mengen sind):  $L_1 \times L_2$  ist die Menge aller Paare  $(w_1, w_2)$  wobei  $w_1$  ein Wort der Sprache  $L_1$  und  $w_2$  ein Wort der Sprache  $L_2$  ist. Im Unterschied dazu berechnet das Produkt  $L_1 L_2$  keine Paare, sondern konkateniert die Wörter  $w_1$  und  $w_2$  direkt zu einem neuen Wort.

**Beispiel 1.1.7.** Sei  $\Sigma = \{a, b\}$ ,  $L_1 = \{a^i \mid i \in \mathbb{N}_0\}$  und  $L_2 = \{b^j \mid j \in \mathbb{N}_0\}$ . Dann ist  $L_1 \cup L_2$  die Sprache aller Wörter, die nur aus  $a$ 's oder nur aus  $b$ 's bestehen;  $L_1 \cap L_2 = \{\varepsilon\}$ ;  $\overline{L_1}$  ist die Sprache der Wörter, die mindestens ein  $b$  enthalten;  $L_1 L_2 = \{a^i b^j \mid i, j \in \mathbb{N}_0\}$ ,  $L_2 L_1 = \{b^j a^i \mid i, j \in \mathbb{N}_0\}$  und  $L_1 L_1 = L_1$ .

Für  $L_1 = \{\spadesuit, \clubsuit, \diamond, \heartsuit\}$  und  $L_2 = \{7, 8, 9, 10, J, D, K, A\}$  stellt  $L_1 L_2$  eine Repräsentation der Spielkarten eines Skatblatts dar.

**Definition 1.1.8.** Seien  $u, v$  Wörter über einem Alphabet  $\Sigma$ . Man sagt

- $u$  ist ein Präfix von  $v$ , wenn es ein Wort  $w$  über  $\Sigma$  gibt, sodass  $uw = v$ .
- $u$  ist ein Suffix von  $v$ , wenn es ein Wort  $w$  über  $\Sigma$  gibt, sodass  $wu = v$ .
- $u$  ist ein Teilwort von  $v$ , wenn es Wörter  $w_1, w_2$  über  $\Sigma$  gibt, sodass  $w_1 u w_2 = v$ .

**Beispiel 1.1.9.** Sei  $w = ababbaba$ . Dann ist das Wort  $aba$  ein Präfix, Suffix und ein Teilwort von  $w$ , während  $ababb$  ein Präfix (und Teilwort) von  $w$  ist, aber kein Suffix von  $w$  ist. Das Wort  $bab$  ist Teilwort von  $w$ , aber weder ein Präfix noch ein Suffix von  $w$ . Das Wort  $bbb$  ist weder Teilwort, noch Präfix, noch Suffix von  $w$ .

**Definition 1.1.10** (Abgeschlossenheit). Eine Klasse  $\mathcal{L}$  von Sprachen (d. h. eine Menge von Mengen) heißt abgeschlossen bezüglich

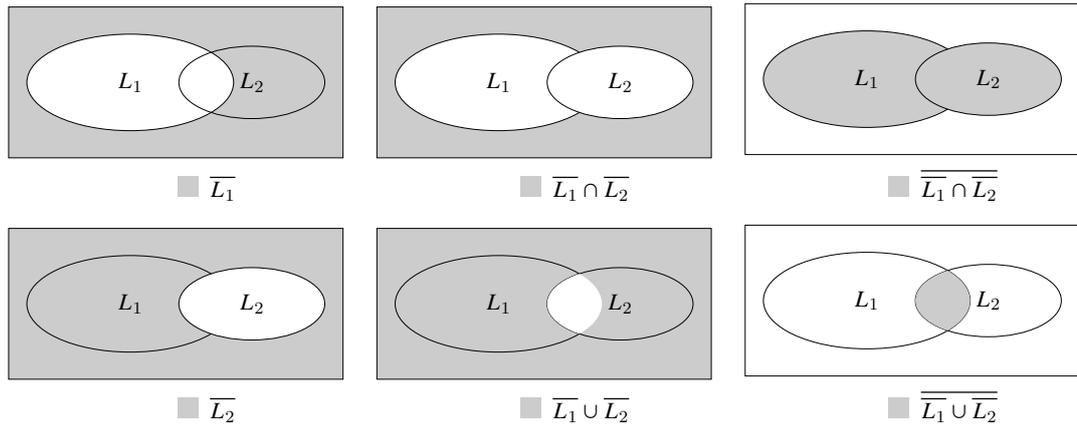
- Vereinigung g. d. w. aus  $L_1 \in \mathcal{L}$  und  $L_2 \in \mathcal{L}$  folgt stets  $(L_1 \cup L_2) \in \mathcal{L}$ ,
- Schnittbildung g. d. w. aus  $L_1 \in \mathcal{L}$  und  $L_2 \in \mathcal{L}$  folgt stets  $(L_1 \cap L_2) \in \mathcal{L}$ ,
- Komplementbildung g. d. w. aus  $L \in \mathcal{L}$  folgt stets  $\overline{L} \in \mathcal{L}$  und
- Produktbildung g. d. w. aus  $L_1 \in \mathcal{L}$  und  $L_2 \in \mathcal{L}$  folgt stets  $(L_1 L_2) \in \mathcal{L}$ .

**Satz 1.1.11.** Sei die Klasse von Sprachen  $\mathcal{L}$  abgeschlossen bezüglich Komplementbildung. Dann ist  $\mathcal{L}$  abgeschlossen bezüglich Schnittbildung genau dann, wenn  $\mathcal{L}$  abgeschlossen bezüglich Vereinigung ist.

*Beweis.* Dies folgt, da sich Vereinigung durch Schnitt und Komplement (bzw. Schnitt durch Vereinigung und Komplement) darstellen lässt:

$$L_1 \cup L_2 = \overline{\overline{L_1} \cap \overline{L_2}} \qquad L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Durch Venn-Diagramme lassen sich diese Zusammenhänge leicht illustrieren:



□

Analog zu  $\Sigma^i, \Sigma^*, \Sigma^+$  definieren wir die Operationen  $\cdot^i, \cdot^*, \cdot^+$  auch für Sprachen. Sei  $L$  eine Sprache. Dann ist:

$$\begin{aligned}
 L^0 &:= \{\varepsilon\} & L^* &:= \bigcup_{i \in \mathbb{N}_0} L^i \\
 L^i &:= L \circ L^{i-1} \text{ für } i > 0 & L^+ &:= \bigcup_{i \in \mathbb{N}} L^i
 \end{aligned}$$

Die Sprache  $L^*$  nennt man auch den *Kleeneschen Abschluss* von  $L$  (benannt nach Stephen Cole Kleene).

**Beispiel 1.1.12.** Sei  $L = \{ab, ac\}$ . Dann ist  $L^2 = \{abab, abac, acab, acac\}$  (alle Wörter, die sich aus 2 Wörtern aus  $L$  bilden lassen) und  $L^* = \{\varepsilon\} \cup \{ax_1ax_2 \cdots ax_i \mid i \in \mathbb{N}, x_j \in \{b, c\}, j = 1, \dots, i\}$ .

**Beispiel 1.1.13.** Die Sprache

$$(\{\varepsilon, 1\} \circ \{0, \dots, 9\}) \cup (\{2\} \circ \{0, 1, 2, 3\}) \circ \{:\} \circ \{0, 1, 2, 3, 4, 5\} \circ \{0, \dots, 9\}$$

stellt alle gültigen Uhrzeiten dar. Die Sprache  $\{0\} \cup (\{1, \dots, 9\} \circ \{0, \dots, 9\}^*)$  stellt alle natürlichen Zahlen dar.

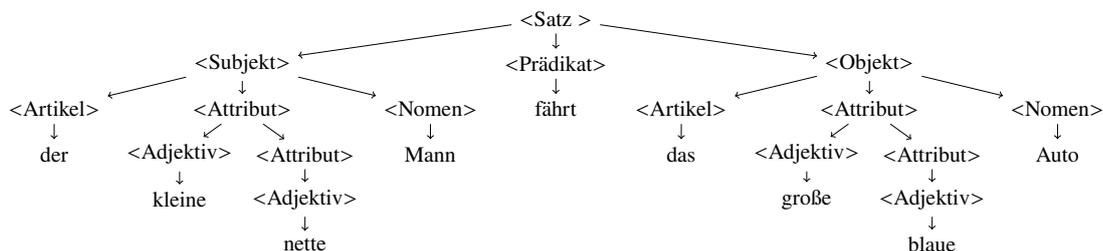
## 2. Chomsky-Grammatiken und die Chomsky-Hierarchie

Sei  $\Sigma$  ein Alphabet. Eine Sprache über  $\Sigma$  ist eine beliebige Teilmenge von  $\Sigma^*$ . Z. B. könnten wir für  $\Sigma = \{ (, ), +, -, *, /, a \}$  die Sprache  $L_{ArEx}$  definieren, welche als Wörter die korrekt geklammerten arithmetischen Ausdrücke über  $\Sigma$  enthält. Dann gilt z. B.  $((a+a)-a)*a \in L_{ArEx}$  aber  $(a-)+a \notin L_{ArEx}$ . Für die Definition von  $L_{ArEx}$  benötigen wir jedoch einen *Formalismus*, der es erlaubt, die Sprache mit einer *endlichen Beschreibung* genau festzulegen. Dies muss möglich sein, obwohl die Sprache selbst aus unendlichen vielen Objekten besteht! Die zwei wesentlichen Formalismen hierfür sind die Grammatiken und die Automaten.

Ein Beispiel für eine Grammatik, die einen Teil der deutschen Grammatik approximiert, ist:

<Satz> → <Subjekt><Prädikat><Objekt>	<Adjektiv> → kleine
<Subjekt> → <Artikel><Attribut><Nomen>	<Adjektiv> → große
<Objekt> → <Artikel><Attribut><Nomen>	<Adjektiv> → nette
<Artikel> → $\varepsilon$	<Adjektiv> → blaue
<Artikel> → der	<Nomen> → Mann
<Artikel> → das	<Nomen> → Auto
<Attribut> → <Adjektiv>	<Prädikat> → fährt
<Attribut> → <Adjektiv><Attribut>	<Prädikat> → liebt

Bei dieser Menge von Regeln, die jeweils von der Form „linke Seite“ → „rechte Seite“ sind, sind die Symbole in spitzen Klammern wie z. B. <Artikel> *Variablen*, d. h. sie sind nur Platzhalter und müssen weiter ersetzt werden. Durch die Grammatik kann z. B. der Satz „der kleine nette Mann fährt das große blaue Auto“ abgeleitet werden. D. h. dieser Satz wäre Teil der durch die Grammatik definierten Sprache. Eine Ableitung korrespondiert zum sogenannten *Syntaxbaum*: Dieser stellt dar, wie der Satz aus den Variablen entsteht. Elternknoten sind jeweils mit der linken Seite einer Regel beschriftet und Kindknoten sind die Objekte auf der rechten Seite der Regel:



Diese Grammatik kann bereits unendlich viele Wörter erzeugen, denn z. B. sind alle Sätze der Form „der kleine Mann liebt das große große große . . . Auto“ damit erzeugbar.

## 2.1. Grammatiken

Grammatiken sind endliche Mengen von Regeln der Form

„linke Seite“  $\rightarrow$  „rechte Seite“.

Wir unterscheiden wir zwischen *Variablen* und *Terminalsymbolen* (üblicherweise sind dies die Zeichen aus einem Alphabet  $\Sigma$ ). Linke und rechte Seite der Regeln sind Folgen bestehend aus Variablen und Terminalen. In obigem Beispiel waren die linken Regelseiten genau eine Variable, was einen Spezialfall darstellt (es handelt sich um eine sogenannte *kontextfreie* Grammatik).

Grammatiken werden benutzt, um aus einer ausgezeichneten Variablen (dem *Startsymbol*) ein Wort über  $\Sigma$  *abzuleiten*. Ableiten meint hierbei mehrfaches Ausführen von Ableitungsschritten, wobei ein Ableitungsschritt ein Vorkommen einer linken Seite (d. h. ein Teilwort, welches der linken Seite einer Regel entspricht) durch die rechte Seite der entsprechenden Regel ersetzt.

Die Menge aller Wörter über  $\Sigma$ , die vom Startsymbol aus abgeleitet werden können, definiert die von der Grammatik erzeugte Sprache.

**Definition 2.1.1** (Grammatik, Satzform, Ableitung, erzeugte Sprache). *Eine Grammatik ist ein 4-Tupel  $G = (V, \Sigma, P, S)$  wobei*

- $V$  ist eine endliche Menge von Variablen (auch Nichtterminale, Nichtterminalsymbole).
- $\Sigma$  ist ein Alphabet von Zeichen (auch Terminale, Terminalsymbole).
- $V \cap \Sigma = \emptyset$  (ein Symbol kann nicht zugleich Variable als auch Zeichen sein).
- $P$  ist eine endliche Menge von Produktionen (auch Regeln genannt), wobei Elemente von  $P$  von der Form  $\ell \rightarrow r$  sind, mit  $\ell \in (V \cup \Sigma)^+$  und  $r \in (V \cup \Sigma)^*$  (d. h. wir können auch schreiben:  $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ , wenn wir Regeln  $\ell \rightarrow r$  als Paare  $(\ell, r)$  auffassen).
- $S \in V$  ist das Startsymbol (manchmal auch Startvariable genannt).

*Manchmal notieren wir nur die Produktionen  $P$  anstelle des gesamten 4-Tupels, wenn klar ist, was Terminale und was Variablen sind und das Startsymbol offensichtlich ist.*

*Ein Wort  $u \in (V \cup \Sigma)^*$  nennt man auch eine Satzform. Für Satzformen  $u, v$  definieren wir  $u \Rightarrow_G v$  ( $u$  geht unter Grammatik  $G$  unmittelbar in  $v$  über), falls es Satzformen  $w_1, w_2$  gibt mit:*

- $u = w_1 \ell w_2$
- $v = w_1 r w_2$
- $\ell \rightarrow r$  ist eine Produktion aus  $P$  (d. h.  $(\ell \rightarrow r) \in P$ ).

*Falls klar ist, welche Grammatik  $G$  behandelt wird, schreiben wir  $u \Rightarrow v$  anstelle von  $u \Rightarrow_G v$ . Mit  $\Rightarrow_G^*$  bezeichnen wir die reflexiv-transitive Hülle von  $\Rightarrow_G$  (d. h.  $\Rightarrow_G^*$  entspricht dem 0- oder mehrfachen Anwenden von  $\Rightarrow_G$ ).*

*Die von der Grammatik  $G$  erzeugte Sprache  $L(G)$  ist*

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

Eine Folge  $(w_0, w_1, \dots, w_n)$  mit  $w_0 = S$ ,  $w_n \in \Sigma^*$  und  $w_{i-1} \Rightarrow w_i$  für  $i = 1, \dots, n$  heißt Ableitung von  $w_n$ . Statt  $(w_0, w_1, \dots, w_n)$  schreiben wir auch  $w_0 \Rightarrow w_1 \Rightarrow \dots \Rightarrow w_n$ .

**Beispiel 2.1.2.** Sei  $G = (\{E, M, Z\}, \{+, *, 1, 2, (, )\}, P, E)$  mit

$$P = \{E \rightarrow M, \\ E \rightarrow E + M, \\ M \rightarrow Z, \\ M \rightarrow M * Z, \\ Z \rightarrow 1, \\ Z \rightarrow 2, \\ Z \rightarrow (E)\}$$

Dann ist  $L(G)$ , die Sprache der korrekt geklammerten arithmetischen Ausdrücke mit  $+$  und  $*$  und den Zahlen 1 und 2. Z. B.  $(2 + 1) * (2 + 2) \in L(G)$ , denn:

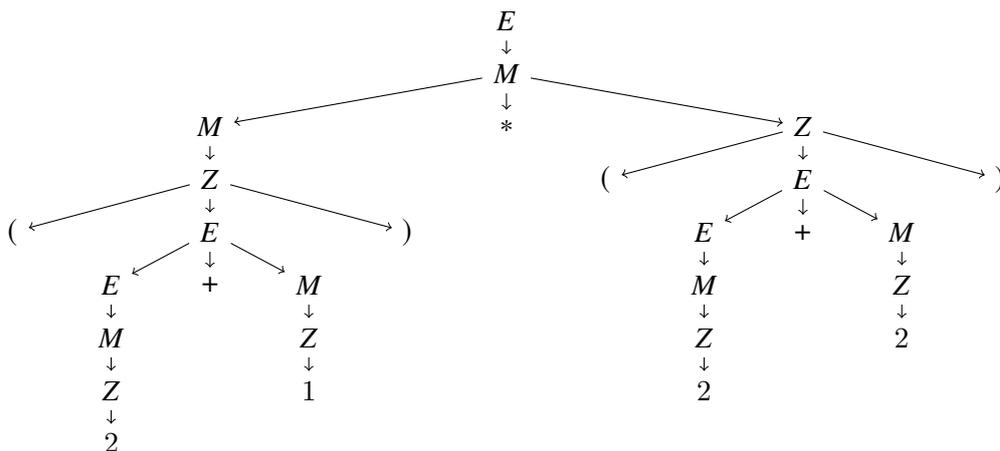
$$E \Rightarrow M \Rightarrow M * Z \Rightarrow Z * Z \Rightarrow Z * (E) \Rightarrow Z * (E + M) \Rightarrow (E) * (E + M) \Rightarrow (E) * (E + Z) \Rightarrow \\ (E + M) * (E + Z) \Rightarrow (M + M) * (E + Z) \Rightarrow (M + M) * (M + Z) \Rightarrow (M + M) * (Z + Z) \Rightarrow \\ (M + M) * (Z + 2) \Rightarrow (M + Z) * (Z + 2) \Rightarrow (M + Z) * (2 + 2) \Rightarrow (Z + Z) * (2 + 2) \Rightarrow \\ (2 + Z) * (2 + 2) \Rightarrow (2 + 1) * (2 + 2)$$

Hierbei ist die ersetzte Variable jeweils grau hinterlegt.

Beachte, dass dies nicht die einzig mögliche Ableitung für  $(2 + 1) * (2 + 2)$  ist. Eine sogenannte Linksableitung, in der immer die linkeste Variable ersetzt wird, ist:

$$E \Rightarrow M \Rightarrow M * Z \Rightarrow Z * Z \Rightarrow (E) * Z \Rightarrow (E + M) * Z \Rightarrow (M + M) * Z \Rightarrow (Z + M) * Z \Rightarrow \\ (2 + M) * Z \Rightarrow (2 + Z) * Z \Rightarrow (2 + 1) * Z \Rightarrow (2 + 1) * (E) \Rightarrow (2 + 1) * (E + M) \Rightarrow \\ (2 + 1) * (M + M) \Rightarrow (2 + 1) * (Z + M) \Rightarrow (2 + 1) * (2 + M) \Rightarrow (2 + 1) * (2 + Z) \Rightarrow \\ (2 + 1) * (2 + 2)$$

Der Syntaxbaum dazu ist:



Das letzte Beispiel zeigt bereits, dass Ableiten kein deterministisches Verfahren ist, sondern es mehrere Möglichkeiten gibt. Es kann für eine Satzform  $w$  mehrere Ableitungsschritte  $w \Rightarrow w_i$  (für verschiedene  $w_1, w_2, \dots$ ) geben: Zum einen dadurch, dass verschiedene Produktionen auf ein Teilwort von  $w$  anwendbar sind, aber auch zum anderen dadurch, dass ein und dieselbe Produktion auf verschiedene Teilworte von  $w$  anwendbar ist. Beachte, dass es jedoch in jedem Schritt nur endliche viele verschiedene Möglichkeiten gibt (da die Satzformen endlich lang sind und die Menge der Produktionen endlich ist). Beachte ferner, dass man u.U. jedoch unendliche viele Ableitungsschritte hintereinander ausführen kann, und dass es auch Satzformen geben kann, die noch keine Wörter aus Terminalsymbolen sind, aber kein Ableitungsschritt mehr anwendbar ist. In beiden Fällen entsteht daraus kein Wort der durch die Grammatik erzeugten Sprache.

**Beispiel 2.1.3.** Sei  $G = (\{S\}, \{a\}, \{S \rightarrow aS\}, S)$ . Dann gilt  $L(G) = \emptyset$ , denn es lassen sich keine Wörter aus  $\{a\}^*$  ableiten: Die einzig mögliche Folge von Ableitungsschritten ist  $S \Rightarrow aS \Rightarrow aaS \Rightarrow \dots$ , die jedoch nie endet.

Für die Grammatik  $G' = (\{S'\}, \{a, b\}, \{S' \rightarrow aS', S' \rightarrow b\}, S')$  gilt  $L(G') = \{a^n b \mid n \in \mathbb{N}_0\}$ .

**Beispiel 2.1.4.** Betrachte die Grammatik  $G = (\{S, B, C\}, \{a, b, c\}, P, S)$  mit

$$P = \{S \rightarrow aSBC, S \rightarrow aBC, CB \rightarrow BC, aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc\}$$

(die z. B. in (Sch08, S.7) zu finden ist). Es gilt  $a^4 b^4 c^4 \in L(G)$ , denn

$$\begin{aligned} S &\Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow aaaSBCBCBC \Rightarrow aaaaBCBCBCBC \Rightarrow \\ &aaaabCBCBCBC \Rightarrow aaaabBCCBCBC \Rightarrow aaaabbCCBCBC \Rightarrow aaaabbCBCCBC \Rightarrow \\ &aaaabbBCCBC \Rightarrow aaaabbBCCBCC \Rightarrow aaaabbBCBCCC \Rightarrow aaaabbBBCCCC \Rightarrow \\ &aaaabbbBCCCC \Rightarrow aaaabbbbBCCCC \Rightarrow aaaabbbbcbCCC \Rightarrow aaaabbbbcbccc \Rightarrow \\ &aaaabbbbcbccc \Rightarrow aaaabbbbcbccc. \end{aligned}$$

Ein Beispiel für eine „steckengebliebene Folge von Ableitungsschritten“ ist

$$S \Rightarrow aSBC \Rightarrow aaBCBC \Rightarrow aabCBC \Rightarrow aabcBC,$$

denn auf die letzte Satzform  $aabcBC$  ist keine Produktion mehr anwendbar (es gibt kein Teilwort von  $aabcBC$ , das linke Seite einer Produktion ist).

Genauer gilt  $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ . Der Beweis hierfür kann in zwei Teilen erfolgen:

- Teil 1: Zeige  $a^n b^n c^n \in L(G)$  für alle  $n \in \mathbb{N}$ : Wende  $n - 1$  mal die Regel  $S \rightarrow aSBC$  und dann einmal die Regel  $S \rightarrow aBC$  an. Das ergibt  $S \Rightarrow^* a^n (BC)^n$ . Nun wende die Regel  $CB \rightarrow BC$  solange an, bis es kein Teilwort  $CB$  mehr gibt. Danach müssen alle Vorkommen von  $C$  rechts der Vorkommen von  $B$  stehen, daher gilt  $a^n (BC)^n \Rightarrow^* a^n B^n C^n$ . Wende Regel  $aB \rightarrow ab$  und anschließend  $n - 1$  mal  $bB \rightarrow bb$  an. Das ergibt  $a^n B^n C^n \Rightarrow^* a^n b^n C^n$ . Schließlich wende einmal  $bC \rightarrow bc$  und anschließend  $n - 1$  mal  $cC \rightarrow cc$  an, sodass  $a^n b^n C^n \Rightarrow^* a^n b^n c^n$ . Zusammensetzen aller Ableitungsschritte zeigt  $S \Rightarrow^* a^n b^n c^n$ .
- Teil 2: Zeige, dass alle von  $G$  erzeugten Wörter von der Form  $a^n b^n c^n$  sind. Betrachte die Produktionen und verifiziere:

- für jede Satzform  $w$  mit  $S \Rightarrow^* w$  gilt:  $\#_a(w) = \#_b(w) + \#_B(w) = \#_c(w) + \#_C(w)$ .
- $a$ 's werden ganz links erzeugt, d. h. jede Satzform  $w$  mit  $S \Rightarrow^* w$  ist von der Form  $a^n w'$  wobei in  $\#_b(w') + \#_B(w') = \#_c(w') + \#_C(w') = n$ .
- Sei  $S \Rightarrow^* a^n w$  mit  $w \in \{b, c\}^*$ . Dann muss das erste Zeichen von  $w$  ein  $b$  sein (denn ein auf  $a$  folgendes Zeichen kann nur durch  $aB \rightarrow ab$  erzeugt werden, und die Regeln vertauschen keine Terminalsymbole). Ebenso kann das linkeste  $c$  nur durch die Regel  $bC \rightarrow bc$  erzeugt sein. Alle andere Terminale  $b$  und  $c$  können nur durch  $bB \rightarrow bb$  und  $bC \rightarrow bc$  und  $cC \rightarrow cc$  erzeugt werden und danach auch nicht mehr vertauscht werden (es gibt dafür keine Produktion). Dieses Erzeugen erlaubt aber keine Wechsel mehr von  $c$  zu  $b$ , daher muss  $w$  von der Form  $b^i c^j$  sein. Da jedoch wie vorher argumentiert  $\#_a(w) = \#_b(w) = \#_c(w)$  gilt muss gelten  $a^n w = a^n b^n c^n$ .

**Beispiel 2.1.5.** Sei  $G = (\{S, T, A, B, \$\}, \{a, b\}, P, S)$  mit

$$P = \{S \rightarrow \$T\$, T \rightarrow aAT, T \rightarrow bBT, T \rightarrow \varepsilon, \$a \rightarrow a\$, \$b \rightarrow b\$, \\ Aa \rightarrow aA, Ab \rightarrow bA, Ba \rightarrow aB, Bb \rightarrow bB, A\$ \rightarrow \$a, B\$ \rightarrow \$b, \$\$ \rightarrow \varepsilon\}$$

Z.B. erzeugt  $G$  das Wort  $aabaab$ :

$$S \Rightarrow \$T\$ \Rightarrow \$aAT\$ \Rightarrow \$aAaAT\$ \Rightarrow \$aAaAbBT\$ \Rightarrow \$aAaAbB\$ \Rightarrow \$aaAAbB\$ \Rightarrow \\ \$aaAbAB\$ \Rightarrow \$aabAAB\$ \Rightarrow \$aabAA\$b \Rightarrow \$aabA\$ab \Rightarrow \$aab\$aab \Rightarrow a\$ab\$aab \Rightarrow \\ aa\$b\$aab \Rightarrow aab\$\$aab \Rightarrow aabaab$$

Wir begründen, dass  $L(G) = \{ww \mid w \in \{a, b\}^*\}$  gilt. Die Regel  $S \rightarrow \$T\$$  erzeugt zunächst eine Umrahmung mit  $\$\$$ . Anschließend erzeugen die drei Regeln  $T \rightarrow aAT$ ,  $T \rightarrow bBT$  und  $T \rightarrow \varepsilon$  ein Wort, welches aus Blöcken  $aA$  und  $bB$  besteht (und mit je einem  $\$$  links und rechts umrahmt ist). Streichen von  $A$  und  $B$  und  $\$$  stellt dann schon ein Wort  $w \in \{a, b\}^*$  dar und die Kopie davon findet man durch Streichen von  $\$, a$  und  $b$  und anschließendem Ersetzen von  $A$  durch  $a$  und  $B$  durch  $b$ . Die Erzeugung muss die Kopie aus  $A$  und  $B$  nun rechts vom Wort bestehend aus  $a$  und  $b$  platzieren. Dies geschieht dadurch, dass zunächst mit den Regeln  $Aa \rightarrow aA, Ab \rightarrow bA, Ba \rightarrow aB, Bb \rightarrow bB$  die  $A$ 's und  $B$ 's bis vor das rechte  $\$$  geschoben werden. Dann werden mit  $A\$ \rightarrow \$a$  und  $B\$ \rightarrow \$b$  die  $A$ 's und  $B$ 's in  $a$ 's und  $b$ 's verwandelt, wobei sie dabei über das rechte  $\$$  hüpfen, und dadurch „getrennt aufbewahrt“ werden. Mit den Regeln  $\$a \rightarrow a\$, \$b \rightarrow b\$, \$\$ \rightarrow \varepsilon$  wird das linke  $\$$ -Symbol zum rechten  $\$$ -Symbol hingeschoben und schließlich werden beide  $\$$  eliminiert. Bei allen Schritten wird die relative Lage aller  $a$  und  $b$  sowie aller  $A$  und  $B$  nicht geändert.

## 2.2. Die Chomsky-Hierarchie

Noam Chomsky nahm die folgende Einteilung der Grammatiken in Typen 0 bis 3 vor:

**Definition 2.2.1** (Chomsky-Hierarchie). Es gilt die folgende Einteilung von Grammatiken in Typen 0 bis 3:

- *Typ 0: Jede Grammatik ist automatisch vom Typ 0.*
- *Typ 1: Eine Grammatik  $G = (V, \Sigma, P, S)$  ist vom Typ 1 und wird kontextsensitiv genannt, wenn für alle Produktionen  $(\ell \rightarrow r) \in P$  gilt:  $|\ell| \leq |r|$ .*
- *Typ 2: Eine Typ 1-Grammatik  $G = (V, \Sigma, P, S)$  ist vom Typ 2 und wird kontextfrei genannt, wenn für alle Produktionen  $(\ell \rightarrow r) \in P$  gilt:  $\ell = A \in V$ , d. h. die linken Seiten der Produktionen bestehen aus genau einer Variablen.*
- *Typ 3: Eine Typ 2-Grammatik  $G = (V, \Sigma, P, S)$  ist vom Typ 3 und wird regulär genannt, wenn für alle Produktionen  $(A \rightarrow r) \in P$  gilt:  $r$  ist von der Form  $a$  oder von der Form  $aA'$  mit  $a \in \Sigma, A' \in V$ , d. h. die rechten Seiten sind Wörter aus  $(\Sigma \cup (\Sigma V))$  und bestehen daher aus einem Terminalsymbol, welchem optional eine Variable folgt.*

Für  $i = 0, 1, 2, 3$  nennt man eine formale Sprache  $L \subseteq \Sigma^*$  vom Typ  $i$ , falls es eine Typ  $i$ -Grammatik  $G$  gibt, sodass  $L(G) = L$  gilt. Spricht man von dem Typ einer formalen Sprache, so ist stets der größtmögliche Typ gemeint.

**Bemerkung 2.2.2.** Die Definition erlaubt Aussagen der Form: Typ  $i + k$ -Sprachen sind eine Teilmenge der Typ  $i$ -Sprachen, da jede Typ  $i + k$ -Grammatik auch eine Typ  $i$ -Grammatik ist.

Die Bezeichnungen „kontextfrei“ und „kontextsensitiv“ rühren daher, dass bei kontextfreien Grammatiken ein Vorkommen einer Variablen  $A$  immer (ohne Beschränkung) durch die rechte Seite  $r$  ersetzt werden kann, wenn es eine Produktion  $A \rightarrow r$  gibt. Bei kontextsensitiven Grammatiken können die Produktionen diese Ersetzung auf einen bestimmten Kontext einschränken: Z. B. kann durch die Produktion  $uAv \rightarrow urv$  zugesichert werden, dass  $A$  nur dann durch  $r$  ersetzt wird, wenn es umrahmt von  $u$  und  $v$  vorkommt (diese Umrahmung ist der „Kontext“).

**Beispiel 2.2.3.** Die Grammatik aus Beispiel 2.1.2 ist kontextfrei (vom Typ 2), während die Grammatik aus Beispiel 2.1.4 kontextsensitiv (vom Typ 1) ist. Die Grammatiken  $G$  und  $G'$  aus Beispiel 2.1.3 sind regulär (vom Typ 3). Die Grammatik aus Beispiel 2.1.5 ist vom Typ 0.

Der Unterschied zwischen Typ 0- und Typ 1-Grammatiken besteht darin, dass Typ 1-Grammatiken keine verkürzenden Regeln erlauben, d. h. bei Ableitungen mit Typ 1-Grammatiken wächst die Länge des erzeugten Wortes monoton, während dies bei Typ 0-Grammatiken nicht der Fall ist. Grafisch kann die mögliche Ableitung eines Wortes der Länge  $n$  daher je nach Typ der Grammatik wie in Abb. 2.1 veranschaulicht werden (vergl. (Sch08, S.10)).

### 2.2.1. Erzeugung des leeren Wortes und $\varepsilon$ -Produktionen

Grammatiken des Typs 1,2,3 erlauben nach unserer bisherigen Definition nicht die Ableitung des leeren Wortes. Um dies jedoch zu ermöglichen, erlauben wir folgende Sonderregel:

**Definition 2.2.4** ( $\varepsilon$ -Regel für Typ 1,2,3-Grammatiken). Eine Grammatik  $G = (V, \Sigma, P, S)$  vom Typ 1, 2 oder 3 darf eine Produktion  $(S \rightarrow \varepsilon) \in P$  enthalten, vorausgesetzt, dass keine rechte Seite einer Produktion in  $P$ , die Variable  $S$  enthält.

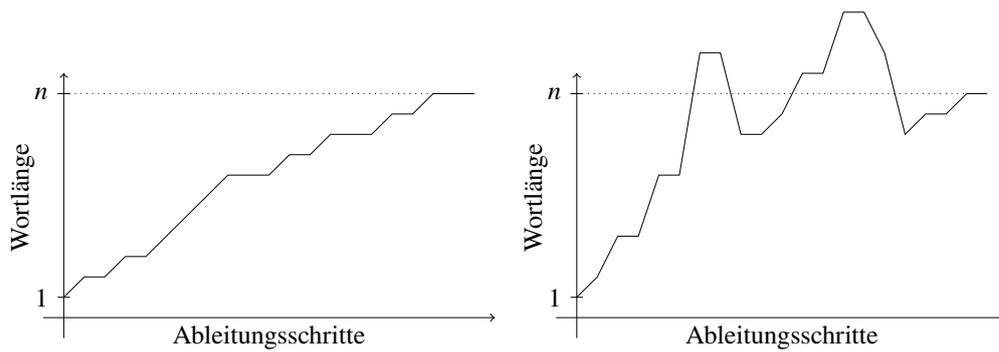


Abbildung 2.1.: Illustrationen zur Ableitung eines Wortes der Länge  $n$  mit einer Typ 1-Grammatik (links) und einer Typ 0-Grammatik (rechts)

Die Einschränkung, dass  $S$  nicht in rechten Seiten von Produktionen vorkommen darf, ist keine echte Beschränkung, wie die folgende Aussage zeigt:

**Satz 2.2.5.** Sei  $G = (V, \Sigma, P, S)$  eine Grammatik vom Typ  $i$  ( $i \in \{1, 2, 3\}$ ) mit  $\varepsilon \notin L(G)$ . Sei  $G' = (V \cup \{S'\}, \Sigma, P', S')$  wobei  $P'$  aus  $P$  entsteht, indem die Produktion  $S' \rightarrow \varepsilon$  hinzugefügt wird und alle Produktionen für  $S$  für das neue Startsymbol  $S'$  kopiert werden, d. h.

$$P' := P \cup \{S' \rightarrow r \mid (S \rightarrow r) \in P\} \cup \{S' \rightarrow \varepsilon\}.$$

Dann ist  $G'$  vom Typ  $i$  (mit der  $\varepsilon$ -Regel gemäß Definition 2.2.4) und  $L(G') = L(G) \cup \{\varepsilon\}$ .

In Typ 2- und Typ 3-Grammatiken kann man generell Produktionen der Form  $A \rightarrow \varepsilon$  zulassen, da diese dort entfernt werden können, ohne die Sprache (bis auf Enthaltensein des leeren Wortes) oder den Typ der Grammatik zu ändern. Dies behandeln wir nun:

**Definition 2.2.6** ( $\varepsilon$ -Produktionen in kontextfreien und regulären Grammatiken). In Grammatiken des Typs 2 und 3 erlauben wir Produktionen der Form  $A \rightarrow \varepsilon$  (sogenannte  $\varepsilon$ -Produktionen).

**Satz 2.2.7** (Entfernen von  $\varepsilon$ -Produktionen in kontextfreien Grammatiken). Sei  $G = (V, \Sigma, P, S)$  eine kontextfreie (bzw. reguläre) Grammatik mit  $\varepsilon \notin L(G)$ . Dann gibt es eine kontextfreie (bzw. reguläre) Grammatik  $G'$  mit  $L(G) = L(G')$  und  $G'$  enthält keine  $\varepsilon$ -Produktionen.

Der zugehörige Algorithmus und ein Beispiel ist im Anhang A.2 zu finden.

### 2.2.2. Beziehungen zwischen den Typ $i$ -Sprachen

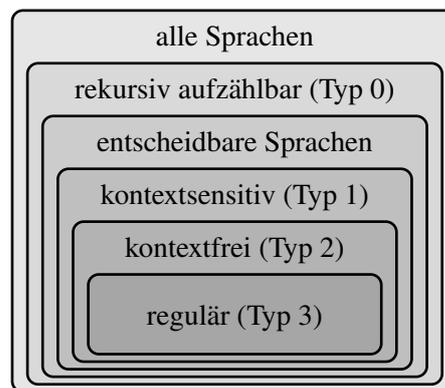
Offensichtlich gilt: Typ 3-Sprachen  $\subseteq$  Typ 2-Sprachen  $\subseteq$  Typ 1-Sprachen  $\subseteq$  Typ 0-Sprachen. Wie wir in späteren Kapiteln sehen (und beweisen) werden, sind alle diese Teilmengenbeziehungen *echt*, d. h. es gilt: Typ 3-Sprachen  $\subset$  Typ 2-Sprachen  $\subset$  Typ 1-Sprachen  $\subset$  Typ 0-Sprachen.

Trennende Beispiele sind: Die Sprache  $L = \{a^n b^n \mid n \in \mathbb{N}\}$  ist von Typ 2, aber nicht von Typ 3, die Sprache  $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$  ist von Typ 1, aber nicht von Typ 2. Schließlich ist die Sprache  $H = \{w \# x \mid \text{Turingmaschine } M_w \text{ h\u00e4lt f\u00fcr Eingabe } x\}$  (das sogenannte Halteproblem) eine Typ 0- aber keine Typ 1-Sprache.

Eine Sprache hei\u00dft *entscheidbar*, wenn es einen Algorithmus gibt, der bei Eingabe der Grammatik  $G$  und einem Wort  $w$  in endlicher Zeit feststellen kann, ob  $w \in L(G)$  gilt oder nicht. Die Typ 1-, 2- und 3-Sprachen sind entscheidbar, w\u00e4hrend es Typ 0-Sprachen gibt, die nicht entscheidbar sind (f\u00fcr obige Sprache  $H$  trifft dies zu). Typ 0-Sprachen sind jedoch rekursiv-aufz\u00e4hlbar (oder semi-entscheidbar), d. h. es gibt einen Algorithmus, der bei Eingabe der Grammatik  $G$  und einem Wort  $w \in G$  in endlicher Zeit feststellt, dass  $w \in L(G)$  gilt und bei einem Wort  $w \notin G$  entweder feststellt, dass  $w \notin L(G)$  gilt, oder nicht-terminiert.

Die Menge der Typ 0-Grammatiken ist abz\u00e4hlbar, da jede Grammatik eine endliche Beschreibung hat und die Grammatiken daher der Gr\u00f6\u00dfe nach aufgez\u00e4hlt werden k\u00f6nnen. Hingegen ist die Menge aller Sprachen \u00fcberabz\u00e4hlbar und hat dieselbe Kardinalit\u00e4t wie die reellen Zahlen.

Daher ergibt sich folgendes Bild:



F\u00fcr die praktische Verwendung in der Informatik sind insbesondere die Typ 3- und Typ 2-Sprachen im Rahmen der lexikalischen und der syntaktischen Analyse im Compilerbau von Interesse und daher sehr gut untersucht. Z. B. gibt es zwischen Typ 3- und Typ 2-Sprachen noch weitere Unterteilungen (z. B. lineare kontextfreie Sprachen, deterministisch kontextfreie Sprache, etc.). Viele (auch praktische) Fragestellungen sind jedoch eher kontextsensitiv oder sogar vom Typ 0. Wegen der schwierigeren Handhabung solcher Sprachen, versucht man oft Probleme als kontextfreie Sprachen zuz\u00fcglich einiger Nebenbedingungen zu formulieren und zu behandeln. Z. B. sind die meisten Programmiersprachen nicht kontextfrei (denn Bedingungen wie Deklaration der verwendeten Variablen, korrekte Typisierung von Ausdr\u00fccken, u. s. w. sind nicht mit einer kontextfreien Sprache darstellbar). Dennoch wird die Syntax von Programmiersprachen oft durch kontextfreie Grammatiken beschrieben und zus\u00e4tzliche Nebenbedingungen (wie korrekte Typisierung) dar\u00fcber hinaus festgelegt.

## 2.3. Das Wortproblem

Sei  $S \Rightarrow w_1 \Rightarrow \dots \Rightarrow w_m$  eine Ableitung des Wortes  $w_m$  der Länge  $n$  in einer kontextsensitiven Grammatik. Da  $|\ell| \leq |r|$  für alle Produktionen  $\ell \rightarrow r$  einer kontextsensitiven Grammatik gilt, wissen wir, dass alle Satzformen  $w_i$  höchstens die Länge  $n$  haben. Da es nur endliche viele Satzformen der Länge  $\leq n$  über  $(\Sigma \cup V)^*$  gibt, kann man erahnen, dass man durch systematisches Durchprobieren all dieser Satzformen, entscheiden kann, welche Wörter  $w_m \in \Sigma^*$  der Länge  $\leq n$  von einer Grammatik erzeugt werden und welche nicht.

**Definition 2.3.1** (Wortproblem für Typ i-Sprachen). *Das Wortproblem für Typ i-Sprachen ist die Frage, ob für eine gegebene Typ i-Grammatik  $G = (V, \Sigma, P, S)$  und ein Wort  $w \in \Sigma^*$  gilt:  $w \in L(G)$  oder  $w \notin L(G)$ .*

**Satz 2.3.2.** *Das Wortproblem für Typ 1-Sprachen ist entscheidbar, d. h. es gibt einen Algorithmus, der bei Eingabe von Typ 1-Grammatik  $G$  und Wort  $w$  nach endlicher Zeit entscheidet, ob  $w \in L(G)$  oder  $w \notin L(G)$  gilt.*

*Beweis.* Sei  $G = (V, \Sigma, P, S)$  eine Typ 1-Grammatik und  $w \in \Sigma^*$ . Für  $m, n \in \mathbb{N}_0$  sei

$$L_m^n := \{w \in (V \cup \Sigma)^* \mid |w| \leq n \text{ und } S \Rightarrow_G^k w, k \leq m\},$$

d. h.  $L_m^n$  enthält alle Satzformen der Länge höchstens  $n$ , die in höchstens  $m$  Schritten vom Startsymbol aus ableitbar sind.

Die Mengen  $L_m^n$  (für  $n > 0$ ) lassen sich rekursiv wie folgt berechnen:

$$\begin{aligned} L_0^n &:= \{S\} \\ L_m^n &:= \text{next}(L_{m-1}^n, n) \text{ für } m > 0 \\ &\text{wobei } \text{next}(L, n) := L \cup \{w' \mid w \in L, w \Rightarrow_G w', |w'| \leq n\} \end{aligned}$$

Beachte, dass diese Berechnung für eine Typ 0-Grammatik falsch wäre, da dort zwischendrin auch Wörter der Länge  $> n$  entstehen dürfen, die im Anschluss daran wieder gekürzt werden. Ferner ist klar, dass die Berechnung von  $L_m^n$  für gegebenes  $m$  und  $n$  terminiert.

Die Mächtigkeit der Mengen  $L_m^n$  sind durch die  $|\Sigma \cup V|^{n+1}$  beschränkt (mehr Satzformen der Länge  $\leq n$  gibt es nicht). Für den Übergang von  $L_{i-1}^n$  zu  $L_i^n$  gilt  $L_{i-1}^n = L_i^n$  oder  $L_{i-1}^n \subset L_i^n$ . Ferner gilt: Falls  $L_{i-1}^n = L_i^n$  dann  $L_{i-1}^n = L_{i+k}^n$  für alle  $k \in \mathbb{N}_0$ . Aus den vorherigen Aussagen folgt, dass es irgendein  $m_0$  geben muss, für das  $L_{m_0}^n = L_{m_0+k}^n$  für alle  $k \in \mathbb{N}_0$  gilt. Nach Berechnung dieser Menge  $L_{m_0}^n$  reicht es daher zu prüfen, ob  $w \in L_{m_0}^n$  gilt oder nicht.

Daher entscheidet Algorithmus 1 das Wortproblem für Typ 1-Grammatiken.  $\square$

**Korollar 2.3.3.** *Das Wortproblem für Typ 2-Sprachen und das Wortproblem für Typ 3-Sprachen sind jeweils entscheidbar.*

**Algorithmus 1** : Entscheiden des Wortproblems für Typ 1-Grammatiken**Eingabe** : Typ 1-Grammatik  $G = (V, \Sigma, P, S)$  und ein Wort  $w \in \Sigma^*$ **Ausgabe** : Ja, wenn  $w \in L(G)$  und Nein, wenn  $w \notin L(G)$ **Beginn** $n := |w|;$  $L := \{S\};$ **wiederhole** $L_{old} := L;$  $L := next(L_{old}, n);$ **bis** ( $w \in L$ ) oder ( $L_{old} = L$ );**wenn**  $w \in L$  **dann**| **return** Ja;**sonst**| **return** Nein;**Ende****Ende**

**Beispiel 2.3.4.** Als Beispiel betrachte die Grammatik  $G = (\{S, B\}, \{a, b, c\}, P, S)$  mit  $P = \{S \rightarrow aSBc, S \rightarrow abc, cB \rightarrow Bc, bB \rightarrow bb\}$ . Wir berechnen  $L_m^6$  für alle  $m$ :

$$L_0^6 = \{S\}$$

$$L_1^6 = next(L_0^6) = L_0^6 \cup \{aSBc, abc\} = \{S, aSBc, abc\}$$

$$L_2^6 = next(L_1^6) = L_1^6 \cup \{aSBc, abc, aabcBc\} = \{S, aSBc, abc, aabcBc\}$$

$$L_3^6 = next(L_2^6) = L_2^6 \cup \{aSBc, abc, aabcBc, aabBcc\} = \{S, aSBc, abc, aabcBc, aabBcc\}$$

$$L_4^6 = next(L_3^6) = L_3^6 \cup \{aSBc, abc, aabcBc, aabBcc, aabbcc\}$$

$$= \{S, aSBc, abc, aabcBc, aabBcc, aabbcc\}$$

$$L_5^6 = next(L_4^6) = L_4^6 \cup \{aSBc, abc, aabcBc, aabBcc, aabbcc\}$$

$$= \{S, aSBc, abc, aabcBc, aabBcc, aabbcc\}$$

Da  $L_4^5 = L_5^6$ , gilt  $L_m^6 = \{S, aSBc, abc, aabcBc, aabBcc, aabbcc\}$  für alle  $m \geq 4$ . Das einzige von  $G$  erzeugte Wort der Länge 6 ist daher  $aabbcc$ .

## 2.4. Weitere Probleme für Formale Sprachen

Neben dem Wort-Problem existieren andere Fragestellungen für formale Sprachen, die wir in den folgenden Definitionen spezifizieren. Die Entscheidbarkeit dieser Probleme werden wir in späteren Kapiteln betrachten.

### Definition 2.4.1.

- Das Leerheitsproblem für Sprachen vom Typ  $i$  ist die Frage, ob Typ  $i$ -Grammatik  $G$  die Gleichheit  $L(G) = \emptyset$  gilt.

- Das Endlichkeitsproblem für Sprachen vom Typ  $i$  ist die Frage, ob für Typ  $i$ -Grammatik  $G$  die Ungleichheit  $|L| < \infty$  gilt.
- Das Schnittproblem für Sprachen vom Typ  $i$  ist die Frage, ob für Typ  $i$ -Grammatiken  $G_1, G_2$  gilt:  $L(G_1) \cap L(G_2) = \emptyset$ .
- Das Äquivalenzproblem für Sprachen vom Typ  $i$  ist, die Frage, ob für Typ  $i$ -Grammatiken  $G_1, G_2$  gilt:  $L(G_1) = L(G_2)$ .

## 2.5. Syntaxbäume

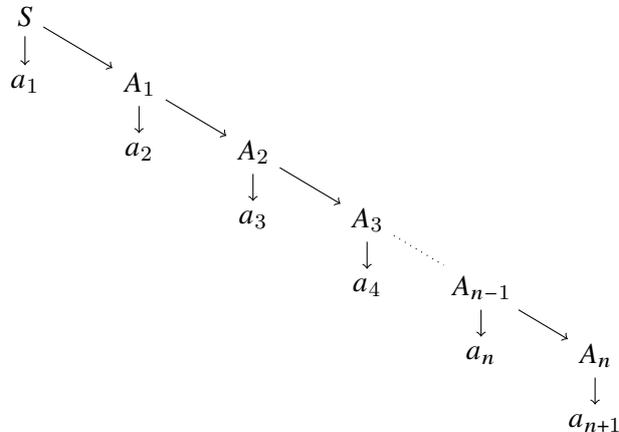
Wir haben Syntaxbäume bereits verwendet und definieren diese nun formal:

**Definition 2.5.1.** Sei  $G = (V, \Sigma, P, S)$  eine Typ 2-Grammatik und  $S \Rightarrow w_0 \Rightarrow \dots \Rightarrow w_n$  eine Ableitung von  $w_n \in \Sigma^*$ . Der Syntaxbaum zur Ableitung wird wie folgt erstellt:

- Die Wurzel des Baums ist mit  $S$  markiert.
- Wenn  $w_i \Rightarrow w_{i+1}$  und  $w_i = uAv$  und  $w_{i+1} = urv$  (die angewendete Produktion ist  $A \rightarrow r$ ), dann erzeuge im Syntaxbaum  $|r|$  viele Knoten als Kinder des mit  $A$  markierten Knotens (an der passenden Stelle im Syntaxbaum). Markiere die Kinder mit den Symbolen aus  $r$  (in der Reihenfolge von links nach rechts).

Die Blätter sind daher genau mit dem Wort  $w_n$  markiert.

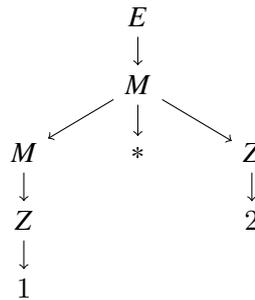
Beachte, dass für Typ-3-Grammatiken, Syntaxbäume immer eine listenartige Struktur folgender Form haben



**Beispiel 2.5.2.** Wir betrachten erneut die Grammatik  $G = (\{E, M, Z\}, \{+, *, 1, 2, (, )\}, P, E)$  mit

$$P = \{E \rightarrow M, E \rightarrow E + M, M \rightarrow Z, M \rightarrow M * Z, Z \rightarrow 1, Z \rightarrow 2, Z \rightarrow (E)\}$$

(siehe Beispiel 2.1.2). Die Ableitung  $E \Rightarrow M \Rightarrow M * Z \Rightarrow Z * Z \Rightarrow 1 * Z \Rightarrow 1 * 2$  hat den Syntaxbaum



Die Ableitung  $E \Rightarrow M \Rightarrow M * Z \Rightarrow M * 2 \Rightarrow Z * 2 \Rightarrow 1 * 2$  hat denselben Syntaxbaum.

Die erste der beiden Ableitungen im vorangegangenen Beispiel hat die Eigenschaft, dass immer die linkeste Variable in der Satzform im nächsten Ableitungsschritt ersetzt wird. Man spricht in diesem Fall von einer *Linksableitung*. Analog gibt es die *Rechtsableitung*, bei der stets die am weitesten rechts stehende Variable beim Ableiten ersetzt wird. Nicht jede Ableitung ist Links- oder Rechtsableitung. Für einen gegebenen Syntaxbaum kann man immer eine dazu passende Links- oder Rechtsableitung angeben (durch Ablesen am Baum). Daher gilt:

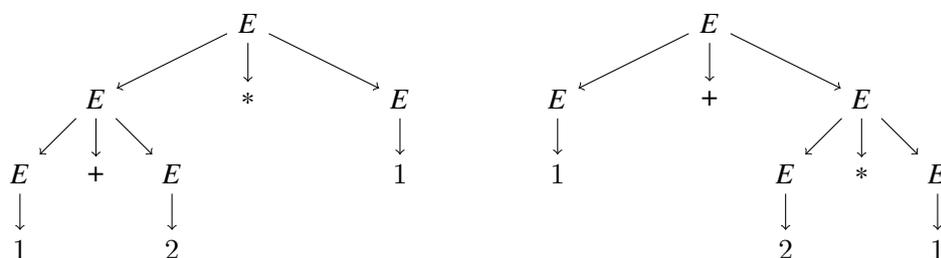
**Satz 2.5.3.** Sei  $G$  eine Typ 2-Grammatik und  $w \in L(G)$ . Dann gibt es eine Linksableitung (und eine Rechtsableitung) von  $w$ .

*Beweis.* Da  $w \in L(G)$ , gibt es irgendeine Ableitung von  $w$ . Dieser Ableitung entspricht ein Syntaxbaum. Für diesen Syntaxbaum kann man eine Linksableitung (bzw. Rechtsableitung) ablesen. □

Es gibt Grammatiken, für die dasselbe Wort mit unterschiedlichen Syntaxbäumen hergeleitet werden kann. Betrachte z. B. die Grammatik

$$(E, \{*, +, 1, 2\}, \{E \rightarrow E * E, E \rightarrow E + E, E \rightarrow 1, E \rightarrow 2\}, E).$$

Eine Ableitung des Worts  $1+2*1$  ist  $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow 1 + E * E \Rightarrow 1 + 2 * E \Rightarrow 1 + 2 * 1$  eine andere Ableitung ist  $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow 1 + E * E \Rightarrow 1 + 2 * E \Rightarrow 1 + 2 * 1$ . Die Syntaxbäume der beiden Ableitungen sind



In solchen Fällen (es gibt verschieden strukturierte Syntaxbäume für dasselbe Wort) spricht man von einer *mehrdeutigen Grammatik*. Tatsächlich gibt es Sprachen, für die es ausschließlich mehrdeutige Grammatiken gibt. Dies nennt man *inhärent mehrdeutig*. Eine kontextfreie, inhärent mehrdeutige Sprache, ist die Sprache  $\{a^m b^m c^n d^n \mid m, n \in \mathbb{N}\} \cup \{a^m b^n c^n d^m \mid m, n \in \mathbb{N}\}$  (siehe (HMU06, Abschnitt 5.4.4)).

## 2.6. Die Backus-Naur-Form für Grammatiken

Von John Backus und Peter Naur wurde im Rahmen der Entwicklung der Programmiersprache ALGOL 60 ein Formalismus eingeführt, um kontextfreie Grammatiken in kompakter Form aufzuschreiben. Wir verwenden analoge abkürzende Schreibweisen:

**Definition 2.6.1** (Erweiterte Backus-Naur-Form (EBNF)). *Für Typ 2-Grammatiken erlauben wir die folgenden abkürzenden Schreibweisen für die Menge der Produktionen  $P$ :*

1. *Statt  $A \rightarrow w_1, A \rightarrow w_2, \dots, A \rightarrow w_n$  schreiben wir auch  $A \rightarrow w_1 \mid w_2 \mid \dots \mid w_n$ .*
2. *Die Schreibweise  $A \rightarrow u[v]w$  steht für die beiden Produktionen  $A \rightarrow uvw$  und  $A \rightarrow uw$  (d. h.  $[v]$  meint, dass  $v$  optional ist).*
3. *Die Schreibweise  $A \rightarrow u\{v\}w$  steht für  $A \rightarrow uw$  oder  $A \rightarrow uBw$  mit  $B \rightarrow v \mid vB$  (d. h.  $\{v\}$  meint, dass  $v$  beliebig oft wiederholt werden kann).*

*Grammatiken, die diese Notation verwenden, nennen wir auch Grammatiken in erweiterter Backus-Naur-Form (EBNF)*

Beachte, dass Typ 2-Grammatiken in EBNF äquivalent zu Typ 2-Grammatiken in normaler Darstellung sind, und daher genau die kontextfreien Sprachen darstellen können.

## 3. Reguläre Sprachen

In diesem Kapitel beschäftigen wir uns mit der einfachsten Sprachklasse der Chomsky-Hierarchie – den Typ 3- bzw. regulären Sprachen. Neben regulären Grammatiken (d. h. Typ 3-Grammatiken) gibt es weitere Formalismen, um reguläre Sprachen zu repräsentieren, die wir in diesem Kapitel kennenlernen werden (endliche Automaten (deterministische als auch nichtdeterministische und Varianten davon) und reguläre Ausdrücke). Wir werden zeigen, dass all diese Formalismen genau die regulären Sprachen repräsentieren und wie man einen Formalismus in einen anderen übersetzen kann. Im Anschluss betrachten wir, wie man zeigt, dass eine formale Sprache nicht regulär ist, wobei wir das sogenannte Pumping-Lemma für reguläre Sprachen kennenlernen werden. Neben der Berechnung von Automaten mit minimaler Anzahl von Zuständen, zeigen wir, dass die regulären Sprachen bez. Vereinigung, Schnitt, Komplement, Produkt und Kleeneschem Abschluss abgeschlossen sind. Wir beenden das Kapitel mit Entscheidbarkeitsresultaten zum Wortproblem und verwandten Problemen für reguläre Sprachen.

### 3.1. Deterministische endliche Automaten

Endliche Automaten lesen als Eingabe ein Wort (schrittweise) und *akzeptieren* oder *verwerfen* das Wort. Die akzeptierte Sprache eines solchen Automaten besteht aus den Wörtern, die von ihm akzeptiert werden.

**Definition 3.1.1** (Deterministischer Endlicher Automat, DFA). *Ein deterministischer endlicher Automat (deterministic finite automaton, DFA) ist ein 5-Tupel  $M = (Z, \Sigma, \delta, z_0, E)$  wobei*

- $Z$  ist eine endliche Menge von Zuständen,
- $\Sigma$  ist das (endliche) Eingabealphabet mit  $(Z \cap \Sigma) = \emptyset$ ,
- $z_0 \in Z$  ist der Startzustand,
- $E \subseteq Z$  ist die Menge der Endzustände (oder auch akzeptierende Zustände) und
- $\delta : Z \times \Sigma \rightarrow Z$  ist die Zustandsüberföhrungsfunktion (oder Überföhrungsfunktion).

Deterministische endliche Automaten können durch Zustandsgraphen dargestellt werden:

**Definition 3.1.2.** *Sei  $M = (Z, \Sigma, \delta, z_0, E)$  ein DFA. Der Zustandsgraph zu  $M$  ist ein Graph, mit Knoten für jeden Zustand  $z \in Z$  und markierten Kanten zwischen diesen Knoten, wobei*

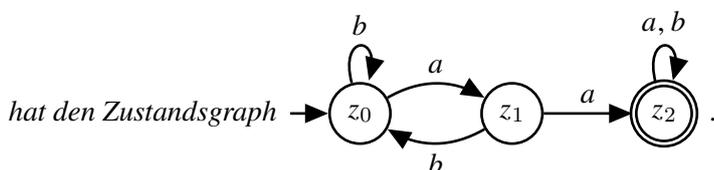
- Zustände  $z \in Z$  als  gezeichnet werden,
- der Startzustand  $z_0 \in Z$  durch einen auf ihn zeigenden Pfeil markiert wird, d. h. .

- Endzustände  $z \in E$  mit doppelten Kreisen markiert werden, d. h. als  $\textcircled{z}$ ,
- für  $\delta(z_i, a) = z_j$ , je eine mit  $a$  beschriftete Kante von  $z_i$  nach  $z_j$   $\textcircled{z_i} \xrightarrow{a} \textcircled{z_j}$  gezeichnet wird. Gleiche Kanten mit unterschiedlicher Beschriftung werden zu einer Kante zusammengefasst, die mit allen Beschriftungen (durch Kommas getrennt) markiert ist: Anstelle von  $\textcircled{z_i} \xrightarrow{a} \textcircled{z_j} \xrightarrow{b}$  zeichnen wir  $\textcircled{z_i} \xrightarrow{a,b} \textcircled{z_j}$ .

Wenn die konkreten Namen der Zustände keine Rolle spielen, lassen wir sie in Illustrationen des Zustandsgraphen manchmal weg.

**Beispiel 3.1.3.** Der DFA  $M = (\{z_0, z_1, z_2\}, \{a, b\}, \delta, z_0, \{z_2\})$  mit

$$\begin{array}{lll} \delta(z_0, a) = z_1 & \delta(z_1, a) = z_2 & \delta(z_2, a) = z_2 \\ \delta(z_0, b) = z_0 & \delta(z_1, b) = z_0 & \delta(z_2, b) = z_2 \end{array}$$



Die Abarbeitung eines Wortes mit einem Automaten kann man so veranschaulichen: Das Wort wird zeichenweise verarbeitet, wobei im Startzustand  $z_0$  begonnen wird. Für jedes Zeichen, wird in den entsprechenden Nachfolgezustand (berechnet mit der Überföhrungsfunktion  $\delta$ ) gewechselt. Ist das Wort komplett eingelesen und der aktuelle Zustand ist ein Endzustand, dann wird das Wort vom DFA *erkannt*.

**Beispiel 3.1.4.** Betrachte den DFA  $M$  aus Beispiel 3.1.3.  $M$  erkennt das Wort  $abaa$ : Beginne im Zustand  $z_0$ . Bei Einlesen des ersten Zeichens  $a$  wechselt der Automat in Zustand  $z_1$ , Einlesen des zweiten Zeichens  $b$  im Zustand  $z_1$  lässt den Automaten in Zustand  $z_0$  wechseln, Einlesen des dritten Zeichens  $a$  lässt den Automaten in  $z_1$  wechseln, und anschließendes Einlesen von  $a$  lässt den Automaten in Zustand  $z_2$  wechseln. Jetzt ist das gesamte Wort eingelesen und der Automat ist im Zustand  $z_2$ , der ein Endzustand ist.

Analog wird das Wort  $aba$  nicht vom DFA  $M$  erkannt, da  $M$  nach Abarbeitung des Wortes im Zustand  $z_1$  ist, der kein Endzustand ist.

Die Menge der erkannten Wörter ist die *akzeptierte Sprache* des Automaten:

**Definition 3.1.5** (Akzeptierte Sprache eines DFAs). Sei  $M = (Z, \Sigma, \delta, z_0, E)$  ein DFA. Wir definieren die Funktion  $\widehat{\delta} : Z \times \Sigma^* \rightarrow Z$  durch

$$\widehat{\delta}(z, \varepsilon) := z \text{ und } \widehat{\delta}(z, aw) := \widehat{\delta}(\delta(z, a), w)$$

Die von  $M$  akzeptierte Sprache ist

$$L(M) := \{w \in \Sigma^* \mid \widehat{\delta}(z_0, w) \in E\}.$$

Beachte, dass  $\widehat{\delta}$  die Überföhrungsfunktion  $\delta$  solange anwendet, bis das Wort abgearbeitet wurde, d. h.  $\widehat{\delta}(z, a_1 \cdots a_n) = \delta(\dots \delta(\delta(z, a_1), a_2), \dots, a_n)$ .

Lässt man den Automaten für ein Wort ablaufen, so spricht auch von einem *Lauf* des DFAs:

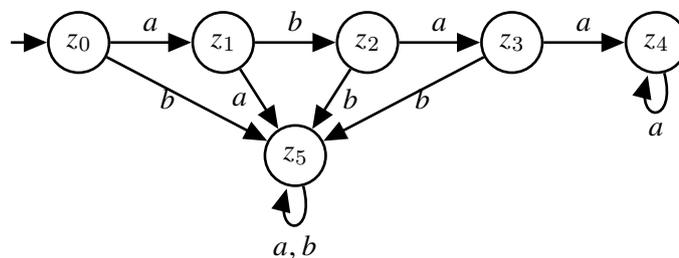
**Definition 3.1.6.** Sei  $M = (Z, \Sigma, \delta, z_0, E)$  ein DFA und  $w \in \Sigma^*$  mit  $|w| = n$ . Die Folge von Zuständen  $q_0, \dots, q_n$  mit  $q_0 = z_0$  und  $q_i = \delta(q_{i-1}, w[i])$  für  $1 \leq i \leq n$  bezeichnet man als Lauf von  $M$  für Wort  $w$ . Für einen solchen Lauf schreiben wir auch:

$$q_0 \xrightarrow{w[1]} q_1 \xrightarrow{w[2]} \cdots \xrightarrow{w[n-1]} q_{n-1} \xrightarrow{w[n]} q_n$$

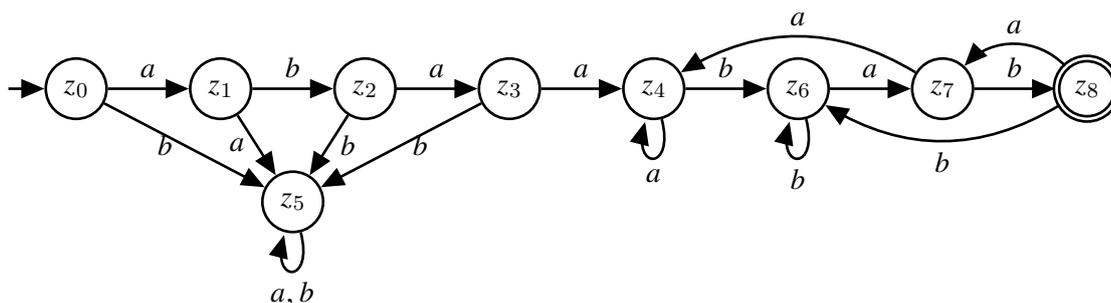
Ein Lauf der mit einem Endzustand endet, nennen wir auch akzeptierender Lauf.

**Beispiel 3.1.7.** Für den DFA  $M$  aus Beispiel 3.1.3 gilt  $L(M) = \{uaav \mid uv \in \{a, b\}^*\}$ , d. h.  $M$  akzeptiert alle Wörter, die zwei aufeinanderfolgende  $a$ 's enthalten. Dies lässt sich einsehen, da  $z_2$  von  $z_0$  aus nur über  $aa$  erreicht werden kann, und da nach Erreichen von  $z_2$  in  $z_2$  verblieben wird, und da man nach Lesen von  $b$  in  $z_1$  wieder erneut in  $z_0$  starten muss.

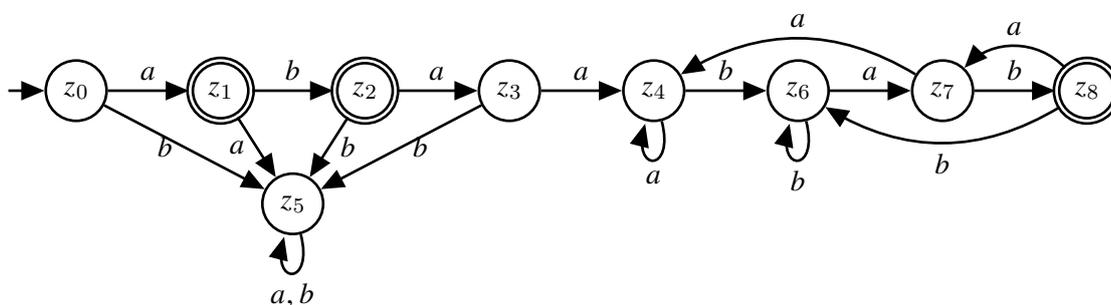
**Beispiel 3.1.8.** Wir konstruieren einen DFA über  $\Sigma = \{a, b\}$ , der die Sprache aller Wörter akzeptiert, die mit  $abaa$  beginnen und mit  $bab$  enden: Für den Präfix  $abaa$  können wir vier Zustände  $z_1, z_2, z_3, z_4$  (zusätzlich zum Startzustand  $z_0$ ) erzeugen, die von  $z_0$  aus durchlaufen werden müssen, und einen weiteren Zustand  $z_5$ , der zum Fehler führt (falls der Präfix nicht stimmt). Das ergibt bereits:



Für den Suffix  $bab$  können wir drei weitere Zustände  $z_6, z_7, z_8$  hinzufügen, sodass  $z_8$  ein Endzustand ist. Für die Zustandsübergänge ab Zustand  $z_4$  müssen wir stets die Möglichkeit beachten, dass der Suffix  $bab$  noch kommt, bzw. komplettiert wird: Wenn wir in  $z_4$  ein  $a$  lesen, verbleiben wir in  $z_4$ , da der komplette Suffix  $bab$  noch kommen muss. Wenn wir in  $z_6$  ein  $b$  lesen, muss noch  $ab$  kommen, d. h. wir verbleiben in  $z_6$ . Lesen wir in  $z_7$  ein  $a$ , dann haben wir zuletzt  $aa$  gelesen, und müssen daher noch den gesamten Suffix  $bab$  lesen und wechseln somit zurück zu  $z_4$ . Lesen wir in  $z_8$  ein  $b$ , dann haben wir  $bb$  zuletzt gelesen und müssen für den geforderten Suffix noch  $ab$  lesen, d. h. wir wechseln in  $z_6$ . Lesen wir in  $z_8$  ein  $a$ , dann haben wir  $ba$  zuletzt gelesen und müssen für den geforderten Suffix noch ein  $b$  lesen, daher wechseln wir in  $z_7$ :



**Beispiel 3.1.9.** Ein DFA, der alle Wörter über  $\{a, b\}$  akzeptiert, die mit  $abaa$  beginnen und mit  $bab$  enden, sowie die Wörter  $a$ ,  $ab$ . Kann genau wie der Automat in Beispiel 3.1.8 konstruiert werden, wobei  $z_1$  und  $z_2$  zusätzliche Endzustände sind, d. h. der folgende Automat akzeptiert die genannte Sprache:



**Übungsaufgabe 3.1.10.** Ein Kaugummi-Automat erhält als Eingabe 10- und 20-Cent Münzen und akzeptiert genau dann, wenn er 50-Cent in der Summe erhalten hat. Modellieren Sie das Akzeptanzverhalten des Kaugummi-Automaten, indem Sie einen DFA über dem Alphabet  $\{10, 20\}$  konstruieren, der genau jene Wörter akzeptiert, die in der Summe 50 Cent ergeben.

### 3.1.1. Minimierung von DFAs

Um möglichst kompakte Darstellungen (und Implementierungen) von DFAs zu erhalten, ist man an möglichst kleinen DFAs interessiert. In diesem Abschnitt erläutern wir, wie man einen gegebenen DFA minimieren kann, sodass die Zustandsmenge minimal wird. Dieser Abschnitt richtet sich im Wesentlichen nach nach (Weg99).

**Definition 3.1.11** (Äquivalenzklassenautomat). Sei  $M = (Z, \Sigma, \delta, z_0, E)$  ein DFA. Wir nennen zwei Zustände  $z, z' \in Z$  äquivalent und schreiben  $z \equiv z'$  falls gilt: für alle  $w \in \Sigma^*$  :  $\widehat{\delta}(z, w) \in E \iff \widehat{\delta}(z', w) \in E$ . Der Äquivalenzklassenautomat zu  $M$  ist der DFA  $M' = (Z', \Sigma, \delta', z'_0, E')$  mit  $Z' = \{[z]_{\equiv} \mid z \in Z\}$ ,  $z'_0 = [z_0]_{\equiv}$ ,  $E' = \{[z]_{\equiv} \mid z \in E\}$  und  $\delta'([z]_{\equiv}, a) = [\delta(z, a)]_{\equiv}$ .

Beachte, dass der Äquivalenzklassenautomat wohldefiniert ist, d. h.

- $E'$  ist wohldefiniert: Aus  $z \equiv z'$  folgt, dass  $z \in E \iff z' \in E$  gilt (dies folgt mit  $w = \varepsilon$  aus  $z \equiv z'$ , denn  $\widehat{\delta}(z, w) \in E \iff \widehat{\delta}(z', w) \in E$ )
- $\delta([z]_{\equiv}, a)$  ist eindeutig für alle  $[z]_{\equiv} \in Z', a \in \Sigma$ , d.h. aus  $z \equiv z'$  folgt  $\delta(z, a) \equiv \delta(z', a)$ : Sei  $z \equiv z'$ , dann gilt  $\widehat{\delta}(z, w) \in E \iff \widehat{\delta}(z', w) \in E$  und damit auch  $\widehat{\delta}(z, aw) \in E \iff \widehat{\delta}(z', aw) \in E$  und damit auch  $\widehat{\delta}(\delta(z, a), w) \in E \iff \widehat{\delta}(\delta(z', a), w) \in E$ . D. h. es gilt dann auch  $\delta(z, a) \equiv \delta(z', a)$ .

**Satz 3.1.12.** Sei  $M = (Z, \Sigma, \delta, z_0, E)$  ein DFA und  $M' = (Z', \Sigma, \delta', z'_0, E')$  der Äquivalenzklassenautomat zu  $M$ . Dann gilt

1.  $L(M) = L(M')$
2. Falls alle Zustände in  $Z$  vom Startzustand  $z_0$  erreichbar sind, dann ist  $M'$  minimal.

*Beweis.* Wir beweisen zunächst den ersten Teil:

1. Sei  $w \in \Sigma^*$ . Dann gilt:

- $M$  durchläuft die Zustandsfolge  $q_0, \dots, q_{|w|}$  entlang  $w$  und akzeptiert  $w$  genau dann, wenn  $q_{|w|} \in E$  gilt.
- $M'$  durchläuft die Zustandsfolge  $[q_0]_{\equiv}, \dots, [q_{|w|}]_{\equiv}$  und akzeptiert  $w$  genau dann, wenn  $[q_{|w|}]_{\equiv} \in E'$  gilt.

Da per Definition  $[q_{|w|}]_{\equiv} \in E'$  genau dann gilt, wenn  $q_{|w|} \in E$  gilt, folgt, dass  $M$  und  $M'$  dieselben Wörter akzeptieren.

Für den zweiten Teil kann man beobachten, dass es für jedes Paar von Zuständen  $[z_i]_{\equiv} \neq [z_j]_{\equiv}$  in  $Z'$  Wörter  $u, v, w$  geben muss, sodass  $\widehat{\delta}(z'_0, u) = [z_i]_{\equiv}$ ,  $\widehat{\delta}(z'_0, v) = [z_j]_{\equiv}$  und  $\widehat{\delta}([z_i]_{\equiv}, w) \in E' \iff \neg(\widehat{\delta}([z_j]_{\equiv}, w) \in E')$  (einerseits, da jeder Zustand in  $M'$  erreichbar ist, und andererseits da  $[z_i]_{\equiv} \neq [z_j]_{\equiv}$ ).

Das Tripel  $(u, v, w)$  zeigt, dass in jedem DFA  $M''$  mit Startzustand  $z''_0$  und  $L(M'') = L(M')$  die Ungleichheit  $\widehat{\delta}(z''_0, u) \neq \widehat{\delta}(z''_0, v)$  gelten muss. Da dies für alle Paare von unterschiedlichen Zuständen aus  $M'$  gilt, kann  $M''$  nicht weniger Zustände als  $M'$  haben.  $\square$

Schließlich benötigen wir noch einen Algorithmus, um alle äquivalenten Zustände eines gegebenen DFAs zu berechnen. Dann haben wir einen Minimierungsalgorithmus für DFAs gefunden, indem wir erst äquivalente Zustände berechnen und dann den Äquivalenzklassenautomat bilden.

Die Ideen bei der Berechnung der äquivalenten Zustände sind:

- Markiere Paare von Zuständen, die verschieden sein müssen (initial alle  $\{z, z'\}$  mit  $z \in E, z' \notin E$ ).
- Vervollständige das Markieren, durch Untersuchen von Übergängen: Wenn  $\{z, z'\}$  noch nicht markiert ist, dann prüfe für jedes  $a \in \Sigma$ , ob die beiden Nachfolger  $\{\delta(z, a), \delta(z', a)\}$  markiert sind. Falls ja, dann markiere  $\{z, z'\}$ . Dieser Vorgang wird solange wiederholt bis sich nichts mehr ändert.
- Alle am Ende unmarkierten Paare stellen äquivalente Zustände dar.

In Algorithmus 2 sind diese Schritte in Pseudo-Code ausformuliert.

**Algorithmus 2** : Berechnung aller äquivalenten Zustände**Eingabe** : DFA  $M = (Z, \Sigma, \delta, z_0, E)$ , der keine unerreichbaren Zustände hat**Ausgabe** : Zustandspaare  $\{z, z'\}$  mit  $z \neq z'$  für die gilt  $z \equiv z'$ **Beginn**stelle Tabelle  $T$  aller Zustandspaare  $\{z, z'\}$  mit  $z \neq z'$  und  $z, z' \in Z$  auf;markiere alle Paare  $\{z, z'\}$  in  $T$  mit  $z \in E$  und  $z' \notin E$ ;**wiederhole**  **für jedes** unmarkierte Paar  $\{z, z'\}$  in  $T$  **tue**    **für jedes**  $a \in \Sigma$  **tue**      **wenn**  $\{\delta(z, a), \delta(z', a)\}$  in  $T$  markiert ist **dann**        markiere  $\{z, z'\}$  in  $T$ ;      **Ende**    **Ende**  **Ende****bis sich**  $T$  nicht mehr verändert;**return**  $\{\{z, z'\} \mid \{z, z'\} \text{ ist nicht markiert in } T\}$ **Ende**

**Satz 3.1.13.** Sei  $M = (Z, \Sigma, \delta, z_0, E)$  ein DFA, der keine unerreichbaren Zustände hat. Die vom Algorithmus 2 für  $M$  berechneten Zustandspaare sind äquivalent und es gibt keine weitere äquivalenten Paare.

*Beweis.*

- Wenn der Algorithmus 2 ein Paar  $\{z, z'\}$  markiert, dann gilt  $z \neq z'$ : Wir zeigen, dass es für jedes markierte Paar  $\{z, z'\}$  ein Wort  $w$  gibt, sodass  $\neg(\widehat{\delta}(z, w) \in E \iff \widehat{\delta}(z', w) \in E)$ : Wir verwenden Induktion über die Anzahl an Iterationen der Wiederhole-Schleife in der  $\{z, z'\}$  markiert wird. Wenn  $\{z, z'\}$  vor der Wiederhole-Schleife markiert wird (die Anzahl der Iterationen ist 0), gilt dies für  $w = \varepsilon$ . Wenn die Anzahl an Iterationen echt größer als 0 ist, dann wird  $\{z, z'\}$  markiert, weil es  $a \in \Sigma$  und ein markiertes Paar  $\{q, q'\}$  gibt mit  $\delta(z, a) = q$  und  $\delta(z', a) = q'$ . Da  $\{q, q'\}$  schon markiert ist zu diesem Zeitpunkt, liefert die Induktionsannahme, dass es ein Wort  $w'$  gibt mit  $\neg(\widehat{\delta}(q, w') \in E \iff \widehat{\delta}(q', w') \in E)$ . Mit  $w = aw'$  folgt  $\neg(\widehat{\delta}(z, w) \in E \iff \widehat{\delta}(z', w) \in E)$ .
- Wenn  $z \neq z'$ , dann markiert der Algorithmus 2 das Paar  $\{z, z'\}$ : Wir führen einen Beweis durch Widerspruch und nehmen an, es gibt Paare  $z \neq z'$ , die vom Algorithmus nicht markiert werden. Wir wählen darunter das Paar  $\{z, z'\}$ , für welches es ein minimal langes Wort  $w$  gibt mit  $\neg(\widehat{\delta}(z, w) \in E \iff \widehat{\delta}(z', w) \in E)$ . Wenn  $w = \varepsilon$ , dann würde  $\{z, z'\}$  vor der Wiederhole-Schleife markiert, was ein Widerspruch zur Annahme ist, dass  $\{z, z'\}$  nicht markiert wird. Wenn  $w = aw'$  mit  $a \in \Sigma$ , dann gilt: Wenn  $\{\delta(z, a), \delta(z', a)\}$  vom Algorithmus markiert wird, dann auch  $\{z, z'\}$ . D. h.  $\{\delta(z, a), \delta(z', a)\}$  wird nicht markiert. Dann gilt für  $w'$ :  $\neg(\widehat{\delta}(\delta(z, a), w') \in E \iff \widehat{\delta}(\delta(z', a), w') \in E)$  und  $|w'| < |w|$ . Dies

ist ein Widerspruch dazu, dass  $\{z, z'\}$  minimal gewählt wurde.  $\square$

Der Algorithmus zur Bestimmung der äquivalenten Paare kann in Zeit  $O(|Z|^4 \cdot |\Sigma|)$  implementiert werden: Wir gehen davon aus, dass die Tabelle  $T$  durch ein zweidimensionales Array der Größe  $|Z| \times |Z|$  repräsentiert wird.

Jeder Durchlauf der Wiederhole-Schleife benötigt Zeit  $O(|Z|^2|\Sigma|)$ . Die Anzahl der Durchläufe der Wiederhole-Schleife ist durch  $|Z|^2$  begrenzt, da es nur  $|Z|^2$  Paare gibt und mindestens ein Paar pro Durchlauf markiert wird. Die restlichen Schritte benötigen konstante Laufzeit. Es gibt schnellere Implementierungen, mit Laufzeit  $O(|Z|^2|\Sigma|)$  (siehe (HMU06)).

Algorithmus 3 beschreibt den gesamten Algorithmus zur Zustandsminimierung von DFAs.

---

**Algorithmus 3 : Minimierung von DFAs**

---

**Eingabe :** DFA  $M = (Z, \Sigma, \delta, z_0, E)$

**Ausgabe :** Minimaler DFA  $M'$  mit  $L(M) = L(M')$

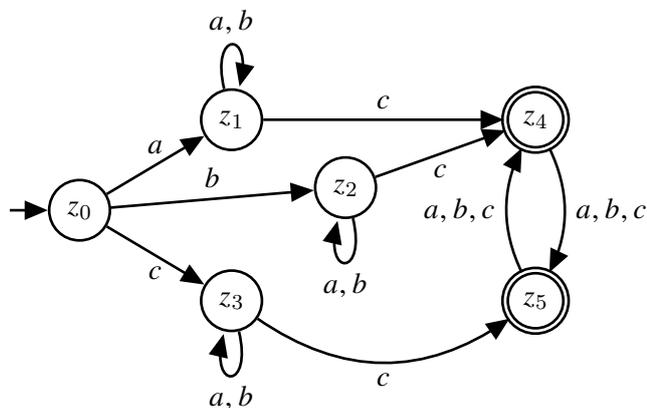
**Beginn**

entferne Zustände aus  $M$ , die vom Startzustand nicht erreichbar sind;  
 berechne äquivalente Zustände mit Algorithmus 2;  
 erzeuge den Äquivalenzklassenautomat, indem die berechneten äquivalenten Zustände vereinigt werden;

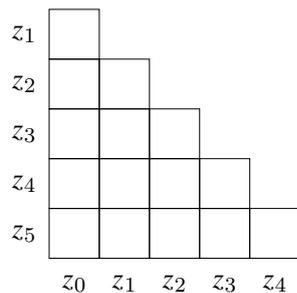
**Ende**

---

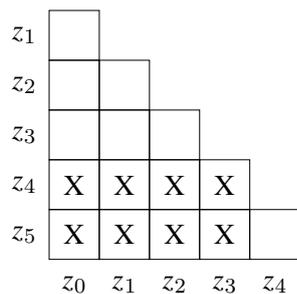
**Beispiel 3.1.14.** Sei  $\Sigma = \{a, b, c\}$  und der folgende DFA  $M$  gegeben:



Wir berechnen den Minimalautomaten zu  $M$ . Zunächst sind alle Zustände vom Startzustand  $z_0$  aus erreichbar und wir können direkt die äquivalenten Zustände berechnen: Die Tabelle  $T$  wird erstellt als



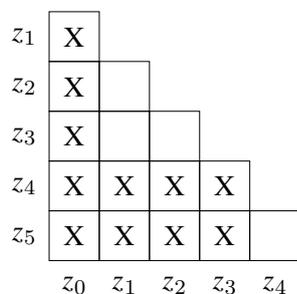
Initiales Markieren alle Paare  $\{z, z'\}$  mit  $z \in \{z_0, z_1, z_2, z_3\}$  und  $z' \in \{z_4, z_5\}$  ergibt



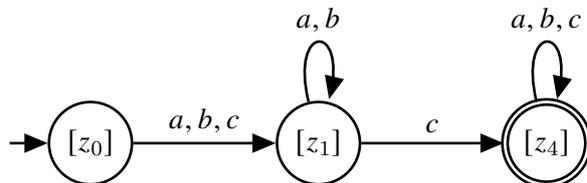
Durchlaufen der Wiederhole-Schleife markiert

- $\{z_0, z_1\}$ , da  $\{\delta(z_0, c), \delta(z_1, c)\} = \{z_3, z_4\}$  bereits markiert ist,
- $\{z_0, z_2\}$ , da  $\{\delta(z_0, c), \delta(z_2, c)\} = \{z_3, z_4\}$  bereits markiert ist, und
- $\{z_0, z_3\}$ , da  $\{\delta(z_0, c), \delta(z_3, c)\} = \{z_3, z_5\}$  bereits markiert ist.

Weitere Markierungen werden nicht durchgeführt, d. h.  $T$  hat nach Verlassen der Wiederhole-Schleife die Form:



Das ergibt  $z_1 \equiv z_3, z_1 \equiv z_2, z_2 \equiv z_3, z_4 \equiv z_5$  und daher die Äquivalenzklassen  $[z_0]_{\equiv} = \{z_0\}$ ,  $[z_1]_{\equiv} = \{z_1, z_2, z_3\}$ ,  $[z_4]_{\equiv} = \{z_4, z_5\}$  und den Minimalautomaten



### 3.1.2. DFAs akzeptieren reguläre Sprachen

Wir zeigen, dass die von DFAs akzeptierten Sprachen allesamt regulär (d. h. Typ 3-Sprachen) sind. Später werden wir auch die Umkehrung zeigen (zu jeder regulären Sprache, gibt es einen DFA, der diese Sprache akzeptiert).

**Theorem 3.1.15.** *Sei  $M = (Z, \Sigma, \delta, z_0, E)$  ein DFA. Dann ist  $L(M)$  eine reguläre Sprache.*

*Beweis.* Für einen DFA  $M = (Z, \Sigma, \delta, z_0, E)$  konstruieren wir eine reguläre Grammatik  $G = (V, \Sigma, P, S)$  mit  $L(G) = L(M)$ : Es sei  $V = Z$ ,  $S = z_0$  und  $P$  enthalte für jeden Zustand  $z_i \in Z$  und Zeichen  $a \in \Sigma$  mit  $\delta(z_i, a) = z_j$  die Produktion  $z_i \rightarrow az_j \in P$  und falls  $z_j \in E$  zusätzlich die Produktion  $z_i \rightarrow a$ . Falls  $z_0 \in E$  gilt (d. h.  $\varepsilon \in L(M)$ ) füge die Regel  $z_0 \rightarrow \varepsilon$  zu  $P$  hinzu.

Wir zeigen, dass für jedes Wort  $w \in \Sigma^*$  gilt:  $w \in L(M) \iff w \in L(G)$ . Falls  $w = \varepsilon$ , so gilt dies offensichtlich. Falls  $w = a_1 \cdots a_m$ , dann gilt  $w \in L(M)$  genau dann, wenn es Zustände  $z_1, \dots, z_m \in Z$  gibt mit  $\delta(z_{i-1}, a_i) = z_i$  und  $z_m \in E$ . Die letzte Aussage ist äquivalent dazu, dass es Ableitungsschritte  $z_0 \Rightarrow_G a_1 z_1, a_1 \cdots a_{i-1} z_{i-1} \Rightarrow_G a_1 \cdots a_i z_i$  für  $i = 1, \dots, m-1$  und  $a_1 \cdots a_{m-1} z_{m-1} \Rightarrow_G a_1 \cdots a_m$  gibt, sodass sich die Ableitung  $z_0 \Rightarrow_G^* a_1 \cdots a_m$  konstruieren lässt, was erneut genau dann gilt, wenn  $w \in L(G)$  gilt.  $\square$

**Beispiel 3.1.16.** *Die reguläre Grammatik zum DFA aus Beispiel 3.1.3 entsprechend zum Beweis von Theorem 3.1.15 ist  $G = (V, \Sigma, P, S)$  mit  $V = \{z_0, z_1, z_2\}$ ,  $\Sigma = \{a, b\}$ ,  $S = z_0$  und*

$$P = \{z_0 \rightarrow az_1, z_0 \rightarrow bz_0, z_1 \rightarrow az_2, z_1 \rightarrow a, z_1 \rightarrow bz_0, z_2 \rightarrow az_2, z_2 \rightarrow a, z_2 \rightarrow bz_2, z_2 \rightarrow b\}.$$

*Z. B. akzeptiert  $M$  das Wort babaaa, denn  $\delta(z_0, b) = z_0, \delta(z_0, a) = z_1, \delta(z_1, b) = z_0, \delta(z_1, a) = z_1, \delta(z_1, a) = z_2, \delta(z_2, a) = z_2$ . Die dazu passende Ableitung für Grammatik  $G$  ist*

$$z_0 \Rightarrow_G bz_0 \Rightarrow_G baz_1 \Rightarrow_G babz_0 \Rightarrow_G babaz_1 \Rightarrow_G babaaz_2 \Rightarrow_G babaaa.$$

Die Umkehrung der letzten Aussage (für jede reguläre Sprache  $L$  gibt es einen DFA  $M$ , der  $L$  akzeptiert) gilt auch, aber wir brauchen weitere Mittel, um sie zu beweisen. Die Kodierung im Beweis der letzten Aussage ist nicht immer rückwärts anwendbar, sobald es z. B. Produktionen  $A \rightarrow aA_1$  und  $A \rightarrow aA_2$  in der regulären Grammatik gibt, ist nicht klar, wie die Überföhrungsfunktion des DFA aussehen muss.

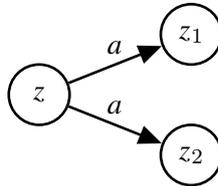
Ein Hilfsmittel sind daher die nichtdeterministischen endlichen Automaten, die wir erst später einföhren werden.

## 3.2. Nichtdeterministische Endliche Automaten

Nichtdeterministische Automaten ermöglichen es dem Automaten, nicht eindeutig, sondern durch „Raten“ in einen Zustand zu wechseln. Während die Überföhrungsfunktion  $\delta$  bei DFAs eine Funktion ist und damit eindeutig (deterministisch) angibt, welcher Nachfolgezustand in Abhängigkeit

vom aktuellen Zustand und dem gelesenen Zeichen zu besuchen ist, erlauben nichtdeterministische Automaten, den Zustandswechsel nichtdeterministisch zu machen, d. h. durch „Raten“ einen von mehreren Nachfolgezuständen aufzusuchen.

Z. B. drückt der folgende Zustandsgraph aus, dass bei Lesen des Zeichens  $a$  im Zustand  $z$  sowohl in Zustand  $z_1$  als auch in Zustand  $z_2$  gewechselt werden darf:



Formal wird dies gehandhabt, indem die Überföhrungsfunktion  $\delta$  bei nichtdeterministischen Automaten nicht mehr *einen* Nachfolgezustand, sondern eine *Menge* von Nachfolgezuständen liefert. Außerdem darf es auch mehrere Startzustände geben, daher haben nichtdeterministische endliche Automaten eine *Menge* von Startzustände:

**Definition 3.2.1.** Ein nichtdeterministischer endlicher Automat (nondeterministic finite automaton, NFA) ist ein 5-Tupel  $(Z, \Sigma, \delta, S, E)$  wobei

- $Z$  ist eine endliche Menge von Zuständen,
- $\Sigma$  ist das (endliche) Eingabealphabet mit  $Z \cap \Sigma = \emptyset$ ,
- $S \subseteq Z$  ist die Menge der Startzustände,
- $E \subseteq Z$  ist die Menge der Endzustände (oder auch akzeptierende Zustände) und
- $\delta : Z \times \Sigma \rightarrow \mathcal{P}(Z)$  ist die Zustandsüberföhrungsfunktion (oder Überföhrungsfunktion).

Beachte, dass nichtdeterministische Automaten auch erlauben, dass es *keinen* Nachfolgezustand gibt bei Lesen eines Zeichens. In diesem Fall ist  $\delta(z, a) = \emptyset$  und das Wort wird verworfen (wenn es keine andere Möglichkeit gibt, einen Endzustand zu erreichen). Passend zum Raten, erkennt ein NFA ein Wort  $w$ , wenn es *einen* Pfad von einem Startzustand zu einem Endzustand gibt. Die nächste Definition macht dies formal:

**Definition 3.2.2** (Akzeptierte Sprache eines NFA). Sei  $M = (Z, \Sigma, \delta, S, E)$  ein NFA. Wir definieren  $\widehat{\delta} : (\mathcal{P}(Z) \times \Sigma^*) \rightarrow \mathcal{P}(Z)$  induktiv durch:

$$\begin{aligned} \widehat{\delta}(X, \varepsilon) &:= X \text{ für alle } X \subseteq Z \\ \widehat{\delta}(X, aw) &:= \bigcup_{z \in X} \widehat{\delta}(\delta(z, a), w) \text{ für alle } X \subseteq Z \end{aligned}$$

Die von  $M$  akzeptierte Sprache ist  $L(M) := \{w \in \Sigma^* \mid \widehat{\delta}(S, w) \cap E \neq \emptyset\}$ .

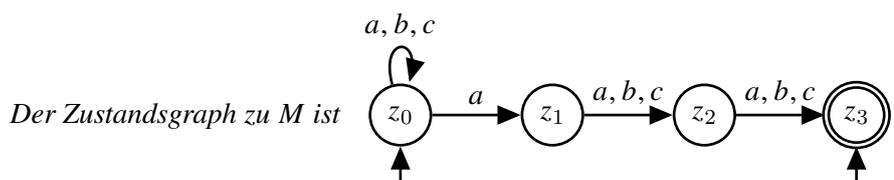
Beachte, dass die Definition  $\{w \in \Sigma^* \mid \widehat{\delta}(S, w) \cap E \neq \emptyset\}$  äquivalent dazu ist, dass ein Endzustand  $z \in E$  existiert mit  $z \in \widehat{\delta}(S, w)$  (d. h. ein Pfad von einem Startzustand zu einem Endzustand entlang  $w$  genügt, um  $w$  zu erkennen). Beachte, dass NFAs mit leerer Menge von Startzuständen möglich und wohldefiniert sind:

**Beispiel 3.2.3.** Sei  $M = (Z, \Sigma, \delta, \emptyset, E)$  ein NFA. Dann ist  $L(M) = \emptyset$ .

Für Zustandsgraphen verwenden wir für NFAs die gleiche Notation wie bei DFAs, nur dass es mehrere Startzustände geben kann, dass es mehrere ausgehende Pfeile mit gleicher Markierung geben kann, und dass es nicht für jedes Zeichen  $a \in \Sigma$  einen Pfeil geben muss.

**Beispiel 3.2.4.** Sei  $M = (\{z_0, z_1, z_2, z_3\}, \{a, b, c\}, \delta, \{z_0, z_3\}, \{z_3\})$  ein NFA mit

$$\begin{array}{llll} \delta(z_0, a) = \{z_0, z_1\} & \delta(z_1, a) = \{z_2\} & \delta(z_2, a) = \{z_3\} & \delta(z_3, a) = \emptyset \\ \delta(z_0, b) = \{z_0\} & \delta(z_1, b) = \{z_2\} & \delta(z_2, b) = \{z_3\} & \delta(z_3, b) = \emptyset \\ \delta(z_0, c) = \{z_0\} & \delta(z_1, c) = \{z_2\} & \delta(z_2, c) = \{z_3\} & \delta(z_3, c) = \emptyset \end{array}$$



Die von  $M$  akzeptierte Sprache sind alle Wörter aus  $\{a, b, c\}^*$ , die an drittletzter Stelle ein  $a$  haben, sowie das leere Wort, d. h.  $L(M) = \{\varepsilon\} \cup (\{a, b, c\}^* \circ \{a\} \circ \{a, b, c\} \circ \{a, b, c\})$  oder in anderer Schreibweise  $L(M) = \{\varepsilon\} \cup \{uaw \mid u \in \{a, b, c\}^*, w \in \{a, b, c\}^2\}$ .

Genau wie beim DFA verwenden wir die Notation eines Laufs auf einem NFA, und sprechen von einem *akzeptierenden Lauf*, wenn dieser mit einem Endzustand endet. Während beim DFA der Lauf für ein gegebenes Wort eindeutig ist, kann es beim NFA mehrere (verschiedene) Läufe für ein und dasselbe Wort geben. Der NFA akzeptiert das Wort, wenn es mindestens einen akzeptierenden Lauf für das Wort gibt.

### 3.3. Reguläre Sprachen können durch NFAs erkannt werden

Wir zeigen, dass jede reguläre Sprache  $L$  durch einen NFA  $M$  akzeptiert wird.

**Theorem 3.3.1.** Sei  $L$  eine reguläre Sprache. Dann gibt es einen NFA  $M$  mit  $L(M) = L$ .

*Beweis.* Da  $L$  regulär ist, gibt es eine reguläre Grammatik  $G = (V, \Sigma, P, S)$  mit  $L(G) = L$ . Sei  $M = (Z, \Sigma, \delta, S', E)$  ein NFA mit  $Z = V \cup \{z_E\}$  (d. h.  $z_E$  ist ein neuer Zustandsname),  $S' = \{S\}$  und sei  $\delta : Z \times \Sigma \rightarrow \mathcal{P}(Z)$  definiert durch  $\delta(A, a) := \{B \mid A \rightarrow aB \in P\} \cup \{z_E \mid \text{falls } A \rightarrow a \in P\}$  für alle  $A \in V$  und  $a \in \Sigma$  und  $\delta(z_E, a) = \emptyset$  für alle  $a \in \Sigma$ . Falls  $S \rightarrow \varepsilon \in P$  sei  $E = \{S, z_E\}$ , und ansonsten sei  $E = \{z_E\}$ .

Wir zeigen, dass  $L(M) = L(G)$  gilt. Offensichtlich gilt  $\varepsilon \in L(M)$  genau dann, wenn  $\varepsilon \in L(G)$ . Für  $w = a_1 \cdots a_n$  gilt:  $w \in L(G)$  genau dann, wenn es eine Ableitung  $S \Rightarrow_G a_1 A_1 \cdots \Rightarrow_G a_1 \cdots a_{n-1} A_{n-1} \Rightarrow_G a_1 \cdots a_n$  gibt. Mit der Konstruktion des NFAs  $M$  gilt, dass dies äquivalent

dazu ist, dass es Zustände  $A_1, \dots, A_{n-1}$  gibt, sodass  $A_1 \in \delta(S, a_1)$ ,  $A_{i+1} \in \delta(A_i, a_{i+1})$  für  $i = 1, \dots, n-2$  und  $z_E \in \delta(A_{n-1}, a_n)$ , was wiederum äquivalent zu  $w \in L(M)$  ist-  $\square$

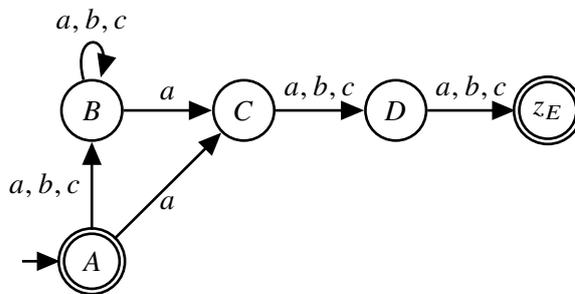
**Beispiel 3.3.2.** Betrachte die reguläre Grammatik  $G = (V, \Sigma, P, A)$  mit  $V = \{A, B, C, D\}$ ,  $\Sigma = \{a, b, c\}$  und

$$P = \{ A \rightarrow \varepsilon \mid aB \mid bB \mid cB \mid aC, \\ B \rightarrow aB \mid bB \mid cB \mid aC, \\ C \rightarrow aD \mid bD \mid cD, \\ D \rightarrow a \mid b \mid c \}$$

Der zu  $G$  passende NFA gemäß der Konstruktion im Beweis von Theorem 3.3.1 ist  $M = (Z, \Sigma, \delta, S, E)$  mit  $Z = V \cup \{z_E\} = \{A, B, C, D, z_E\}$ ,  $E = \{A, z_E\}$ ,  $S = \{A\}$  und

$$\begin{array}{llllll} \delta(A, a) = \{B, C\} & \delta(B, a) = \{B, C\} & \delta(C, a) = \{D\} & \delta(D, a) = \{z_E\} & \delta(z_E, a) = \emptyset \\ \delta(A, b) = \{B\} & \delta(B, b) = \{B\} & \delta(C, b) = \{D\} & \delta(D, b) = \{z_E\} & \delta(z_E, b) = \emptyset \\ \delta(A, c) = \{B\} & \delta(B, c) = \{B\} & \delta(C, c) = \{D\} & \delta(D, c) = \{z_E\} & \delta(z_E, c) = \emptyset \end{array}$$

Der Zustandsgraph zu  $M$  ist



Z. B. wird  $bacabc$  von  $G$  erzeugt (denn  $A \Rightarrow bB \Rightarrow baB \Rightarrow bacB \Rightarrow bacacC \Rightarrow bacabD \Rightarrow bacabc$ ) und von  $M$  akzeptiert, denn  $A$  ist Startzustand,  $B \in \delta(A, b)$ ,  $B \in \delta(B, a)$ ,  $B \in \delta(B, c)$ ,  $C \in \delta(B, a)$ ,  $D \in \delta(C, b)$ ,  $z_E \in \delta(D, c)$  und  $z_E$  ist Endzustand.

Es gilt  $L(G) = L(M) = \{\varepsilon\} \cup \{uaw \mid u \in \{a, b, c\}^*, w \in \{a, b, c\}^2\}$ , d. h. es wird dieselbe Sprache wie in Beispiel 3.2.4 vom NFA  $M$  akzeptiert bzw. von der Grammatik  $G$  erzeugt.

### 3.4. Überführung von NFAs in DFAs

In diesem Abschnitt zeigen wir, dass wir zu jedem NFA  $M$  einen DFA  $M'$  konstruieren können, sodass  $L(M) = L(M')$  gilt. Erstmals wurde diese Konstruktion von Michael O. Rabin und Dana Scott 1959 bewiesen (siehe (RS59)).

**Theorem 3.4.1.** Jede von einem NFA akzeptierte Sprache ist auch durch einen DFA akzeptierbar.

*Beweis.* Für NFA  $M = (Z, \Sigma, \delta, S, E)$  konstruieren wir den DFA  $M' = (Z', \Sigma, \delta', S', E')$  mit

- $Z' = \mathcal{P}(Z)$ , d. h. die Zustandsmenge ist die Potenzmenge von  $Z$  (für jede Teilmenge von  $Z$  gibt es einen Zustand),
- $S' = S$  (der Startzustand ist die Teilmenge  $S \subseteq Z$ , die Menge aller Startzustände von  $M$ ),
- $E' = \{X \in Z' \mid (E \cap X) \neq \emptyset\}$  (alle Teilmengen von  $Z$ , die mindestens einen Endzustand von  $M$  enthalten) und
- $\delta'(X, a) = \bigcup_{z \in X} \delta(z, a) = \widehat{\delta}(X, a)$  (d. h.  $\delta'(X, a)$  berechnet die Menge aller von  $X$  aus erreichbaren Zustände in  $M$ , was dasselbe ist, was  $\widehat{\delta}$  berechnet.).

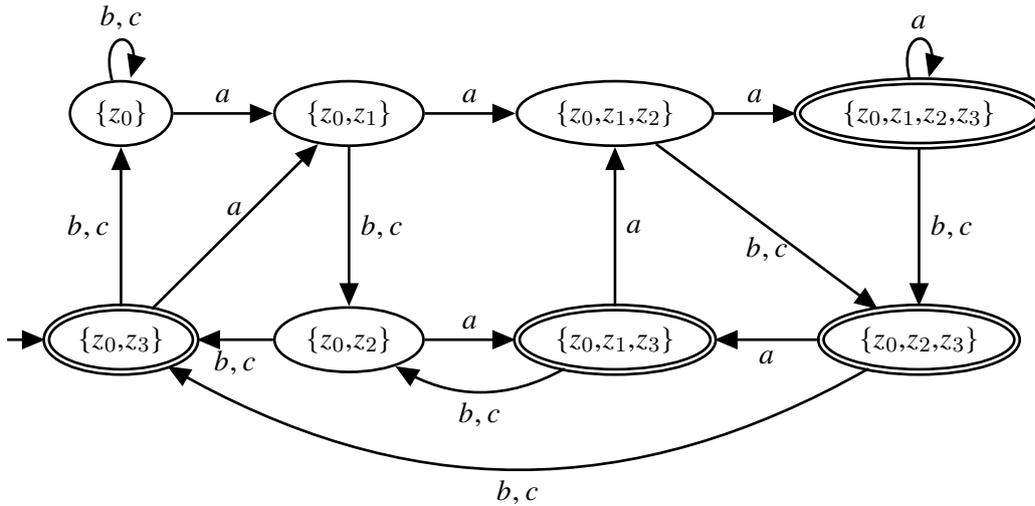
Wir beweisen, dass  $L(M) = L(M')$  gilt. Sei  $w = a_1 \cdots a_n \in \Sigma^*$ .

Es gilt  $w \in L(M)$  genau dann, wenn  $\widehat{\delta}(S, w) \cap E \neq \emptyset$ . Gemäß Definition 3.2.2 ist äquivalent dazu, dass es eine Folge  $Z_1, \dots, Z_n$  von Teilmengen von  $Z$  gibt, mit  $\delta(S, a_1) = Z_1$ ,  $\delta(Z_i, a_{i+1}) = Z_{i+1}$  für  $i = 1, \dots, n-1$  und  $Z_n \cap E \neq \emptyset$ . Gemäß obiger Konstruktion folgt, dass dies äquivalent dazu ist, dass es eine Folge  $Z_1, \dots, Z_n$  von Teilmengen von  $Z$  gibt, mit  $\delta'(S, a_1) = Z_1$ ,  $\delta'(Z_i, a_{i+1}) = Z_{i+1}$  für  $i = 1, \dots, n-1$  und  $Z_n \cap E \neq \emptyset$ . Die letzte Aussage ist äquivalent zu  $\widehat{\delta}'(S', w) \in E'$ , was wiederum äquivalent zu  $w \in L(M')$  ist.  $\square$

**Beispiel 3.4.2.** Betrachte den NFA  $M$  aus Beispiel 3.2.4. Entsprechend der Konstruktion im Beweis zu Theorem 3.4.1 wird der folgende DFA  $M'$  erstellt:  $M' = (\mathcal{P}(\{z_0, z_1, z_2, z_3\}), \{a, b, c\}, \delta', S', E')$  mit

- $S' = \{z_0, z_3\}$
- $E' = \{\{z_3\}, \{z_0, z_3\}, \{z_1, z_3\}, \{z_2, z_3\}, \{z_0, z_1, z_3\}, \{z_0, z_2, z_3\}, \{z_1, z_2, z_3\}, \{z_0, z_1, z_2, z_3\}\}$
- $\delta'(\emptyset, d) = \emptyset$  für  $d \in \{a, b, c\}$        $\delta'(\{z_1, z_2\}, d) = \{z_2, z_3\}$  für  $d \in \{a, b, c\}$   
 $\delta'(\{z_0\}, a) = \{z_0, z_1\}$        $\delta'(\{z_1, z_3\}, d) = \{z_2\}$  für  $d \in \{a, b, c\}$   
 $\delta'(\{z_0\}, d) = \{z_0\}$  für  $d \in \{b, c\}$        $\delta'(\{z_2, z_3\}, d) = \{z_3\}$  für  $d \in \{a, b, c\}$   
 $\delta'(\{z_1\}, d) = \{z_2\}$  für  $d \in \{a, b, c\}$        $\delta'(\{z_0, z_1, z_2\}, a) = \{z_0, z_1, z_2, z_3\}$   
 $\delta'(\{z_2\}, d) = \{z_3\}$  für  $d \in \{a, b, c\}$        $\delta'(\{z_0, z_1, z_2\}, d) = \{z_0, z_2, z_3\}$  für  $d \in \{b, c\}$   
 $\delta'(\{z_3\}, d) = \emptyset$  für  $d \in \{a, b, c\}$        $\delta'(\{z_0, z_1, z_3\}, a) = \{z_0, z_1, z_2\}$   
 $\delta'(\{z_0, z_1\}, a) = \{z_0, z_1, z_2\}$        $\delta'(\{z_0, z_1, z_3\}, d) = \{z_0, z_2\}$  für  $d \in \{b, c\}$   
 $\delta'(\{z_0, z_1\}, d) = \{z_0, z_2\}$  für  $d \in \{b, c\}$        $\delta'(\{z_0, z_2, z_3\}, a) = \{z_0, z_1, z_3\}$   
 $\delta'(\{z_0, z_2\}, a) = \{z_0, z_1, z_3\}$        $\delta'(\{z_0, z_2, z_3\}, d) = \{z_0, z_3\}$  für  $d \in \{b, c\}$   
 $\delta'(\{z_0, z_2\}, d) = \{z_0, z_3\}$  für  $d \in \{b, c\}$        $\delta'(\{z_1, z_2, z_3\}, d) = \{z_2, z_3\}$  für  $d \in \{a, b, c\}$   
 $\delta'(\{z_0, z_3\}, a) = \{z_0, z_1\}$        $\delta'(\{z_0, z_1, z_2, z_3\}, a) = \{z_0, z_1, z_2, z_3\}$   
 $\delta'(\{z_0, z_3\}, d) = \{z_0\}$  für  $d \in \{b, c\}$        $\delta'(\{z_0, z_1, z_2, z_3\}, d) = \{z_0, z_2, z_3\}$  für  $d \in \{b, c\}$

Gezeichnet ergibt dies, wobei wir vom Startzustand nicht erreichbare Zustände weglassen:



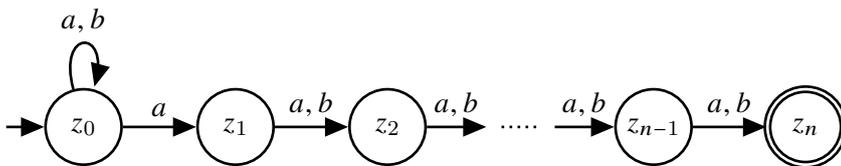
Die im Beweis zu Theorem 3.4.1 vorgestellte Potenzmengen-Konstruktion eines DFA aus einem NFA führt zu einer exponentiellen Anzahl an Zuständen im DFA bezüglich der Anzahl an Zuständen im NFA. Das Beispiel 3.4.2 zeigt, dass nicht alle Zustände tatsächlich benötigt werden (da unerreichbare entfernt werden können), aber es klärt auch nicht die Frage, ob man einen besseren Algorithmus angeben kann, der einen DFA mit weniger Zuständen generiert.

Das folgende Lemma widerlegt dies und zeigt, dass es Sprachen  $L$  gibt, für die *jeder* DFA, der  $L$  akzeptiert, exponentiell groß in der Größe eines NFA ist, der  $L$  akzeptiert.

**Lemma 3.4.3.** Sei  $L_n = \{uav \mid u \in \{a, b\}^*, v \in \{a, b\}^{n-1}\}$  für  $n \in \mathbb{N}$ , die Sprache aller Wörter aus  $\{a, b\}^*$ , die an  $n$ -letzter Stelle ein  $a$  haben. Dann gilt für alle  $n \in \mathbb{N}$ :

- Es gibt einen NFA  $M_n$  mit  $L(M_n) = L_n$  und  $M_n$  hat  $n + 1$  Zustände.
- Es gibt keinen DFA  $M'_n$  mit  $L(M'_n) = L_n$ , sodass  $M'_n$  weniger als  $2^n$  Zustände besitzt.

*Beweis.* Sei  $M_n$  der folgende NFA:



Offensichtlich akzeptiert  $M_n$  die Sprache  $L_n$ : Zum Akzeptieren müssen die Zustände  $z_0, z_1, \dots, z_n$  in dieser Reihenfolge durchlaufen werden, was genau mit dem Wort  $av$  mit  $v \in \{a, b\}^*$  und  $|v| = n - 1$  möglich ist. Zuvor kann beliebig lang im Startzustand  $z_0$  verblieben werden und dabei jedes Wort  $u \in \{a, b\}^*$  gelesen werden.

Für den zweiten Teil der Aussage führen wir einen Beweis durch Widerspruch. Nehme an, es gibt  $n \in \mathbb{N}$  und einen DFA  $M' = (Z, \{a, b\}, \delta, z_0, E)$  mit  $L(M') = L_n$  und  $M'$  hat weniger als  $2^n$  Zustände. Sei  $W = \{a, b\}^n$ , d. h.  $W$  enthält alle Wörter, die aus  $n$  Zeichen bestehen. Es gibt

genau  $2^n$  verschiedene solche Wörter (d. h.  $|W| = 2^n$ ). Da  $M'$  weniger als  $2^n$  Zustände hat, muss es zwei verschiedene Wörter  $w \neq w'$  mit  $w, w' \in W$  geben, so dass  $\widehat{\delta}(z_0, w) = \widehat{\delta}(z_0, w') = z_i$  gilt (nach Einlesen beider Wörter befindet sich  $M'$  in ein und demselben Zustand  $z_i$ ). Sei  $j$  die erste Position an der sich  $w$  und  $w'$  unterscheiden. Der Fall  $j = 1$  kann nicht gelten, da dann  $z_i \in E$  als auch  $z_i \notin E$  gelten müssten. Für den Fall  $j > 1$  erhalten wir o.B.d.A.  $w = uav$  und  $w' = ubv'$  mit  $|u| = j - 1$  und  $|v| = |v'| = n - j$ . Betrachte die Wörter  $w_0 = uavb^{j-1}$  und  $w'_0 = ubv'b^{j-1}$  (wir verlängern  $w$  und  $w'$  um  $j - 1$   $b$ 's). Dann muss gelten  $\widehat{\delta}(w_0) = \widehat{\delta}(w'_0)$ , da  $\widehat{\delta}(uav) = z_i = \widehat{\delta}(ubv')$ . Offensichtlich gilt aber  $w_0 \in L_n$  und  $w'_0 \notin L_n$ , was fordert, dass  $\widehat{\delta}(w_0) \in E$  und  $\widehat{\delta}(w'_0) \notin E$ , was ein Widerspruch ist (da  $\widehat{\delta}(w_0) = \widehat{\delta}(w'_0)$ ). Daher war unsere Annahme falsch und es gibt keinen DFA mit weniger als  $2^n$  Zuständen, der die Sprache  $L_n$  akzeptiert.  $\square$

**Theorem 3.4.4.** *DFAs und NFAs erkennen genau die regulären Sprachen.*

*Beweis.* Wir haben gezeigt:

1. DFAs erkennen reguläre Sprachen.
2. Jede reguläre Sprache wird auch durch einen NFA akzeptiert.
3. Jede von einem NFAs akzeptierte Sprache ist auch durch einen DFA akzeptierbar.

(1) wurde in Theorem 3.1.15, (2) in Theorem 3.3.1 und (3) in Theorem 3.4.1 bewiesen. Die Kombination der Aussagen (2) und (3) zeigen, dass jede reguläre Sprache durch einen DFA akzeptiert wird, was in Kombination mit Aussage (1) zeigt, dass DFAs genau die regulären Sprachen erkennen. Für NFAs wurde eine Richtung direkt bewiesen, die andere Richtung (NFAs erkennen reguläre Sprachen) folgt durch einen Beweis mit Widerspruch: Nehme an, dass NFAs auch nicht-reguläre Sprachen akzeptieren können. Dann folgt aus Aussage (3), dass auch DFAs nicht-reguläre Sprachen akzeptieren, was jedoch ein Widerspruch zu Aussage (1) ist. D. h. die Annahme war falsch und NFAs erkennen nur reguläre Sprachen.  $\square$

### 3.5. NFAs mit $\varepsilon$ -Übergängen

In diesem Abschnitt stellen wir Varianten von NFAs vor, die deren Ausdruckskraft nicht verändern, aber manchmal noch einfachere Konstruktionen oder Beweise zulassen.

Ein NFA mit  $\varepsilon$ -Übergängen, erlaubt es von einem Zustand in einen anderen Zustand zu wechseln, *ohne* ein Zeichen zu lesen (sondern durch Lesen des leeren Worts  $\varepsilon$ ). Da NFAs sowieso nichtdeterministische Zustandswechsel erlauben, führt diese Erweiterung nicht zu neuer Ausdruckskraft.

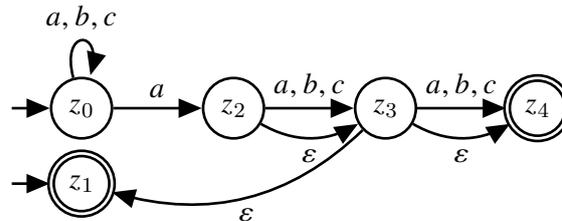
**Definition 3.5.1** (NFA mit  $\varepsilon$ -Übergängen). *Ein nichtdeterministischer endlicher Automat mit  $\varepsilon$ -Übergängen (NFA mit  $\varepsilon$ -Übergängen) ist ein Tupel  $M = (Z, \Sigma, \delta, S, E)$  wobei*

- $Z$  ist eine endliche Menge von Zuständen,
- $\Sigma$  ist das (endliche) Eingabealphabet mit  $(Z \cap \Sigma) = \emptyset$ ,
- $S \subseteq Z$  ist die Menge der Startzustände,
- $E \subseteq Z$  ist die Menge der Endzustände (oder auch akzeptierende Zustände) und

- $\delta : Z \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Z)$  ist die Zustandsüberföhrungsfunktion (oder Überföhrungsfunktion).

Beachte, dass die Definition sich von der Definition der NFAs nur dadurch unterscheidet, dass der Definitionsbereich von  $\delta$  nun  $(Z \times (\Sigma \cup \{\varepsilon\}))$  anstelle von  $(Z \times \Sigma)$  ist. D. h. der Automat kann auch bei Lesen des leeren Wortes den Zustand wechseln.

**Beispiel 3.5.2.** Ein Beispiel für einen NFA mit  $\varepsilon$ -Übergängen ist:



Wir haben die akzeptierte Sprache eines NFA mit  $\varepsilon$ -Übergängen noch nicht formal definiert, aber es ist einsichtig, dass obiger Automat die Sprache über  $\{a, b, c\}$  erkennt, die aus allen Wörtern besteht, die an drittletzter, vorletzter, oder letzter Stelle ein  $a$  haben, sowie das leere Wort: Das leere Wort wird akzeptiert, da  $z_1$  Start- und Endzustand ist. Vom Startzustand  $z_1$  aus sind andere Wörter nicht relevant. Vom Startzustand  $z_0$  kann erst beliebig oft  $a, b, c$  gelesen werden, irgendwann muss jedoch eindeutig  $a$  gelesen werden, um in Richtung Endzustand zu wandern. Nachdem Zustand  $z_2$  erreicht wurde, kann einer der Endzustände  $z_1$  oder  $z_4$  erreicht werden, indem 0, 1 oder 2 beliebige Zeichen gelesen werden.

Bevor wir die akzeptierte Sprache eines NFA mit  $\varepsilon$ -Übergängen definieren (können), erläutern wir, wie wir alle durch  $\varepsilon$ -Übergänge erreichbaren Zustände – ausgehend von einem Zustand bzw. einer Zustandsmenge – berechnen können:

**Definition 3.5.3.** Sei  $M = (Z, \Sigma, \delta, S, E)$  ein NFA mit  $\varepsilon$ -Übergängen. Die  $\varepsilon$ -Hülle  $clos_\varepsilon(z)$  eines Zustands  $z \in Z$  ist induktiv definiert als die kleinste Menge von Zuständen, welche die folgenden Eigenschaften erfüllt:

1.  $z \in clos_\varepsilon(z)$ .
2. Wenn  $z' \in clos_\varepsilon(z)$  und  $z'' \in \delta(z', \varepsilon)$ , dann ist auch  $z'' \in clos_\varepsilon(z)$ .

Für eine Zustandsmenge  $X \subseteq Z$  definieren wir  $clos_\varepsilon(X) := \bigcup_{z \in X} clos_\varepsilon(z)$ .

Die  $\varepsilon$ -Hülle für eine Zustandsmenge  $X \subseteq Z$  kann auch berechnet werden durch:

$$clos_\varepsilon(X) := \begin{cases} X, & \text{wenn } \bigcup_{z \in X} \delta(z, \varepsilon) \subseteq X \\ clos_\varepsilon(X \cup \bigcup_{z \in X} \delta(z, \varepsilon)), & \text{sonst} \end{cases}$$

**Beispiel 3.5.4.** Betrachte den NFA mit  $\varepsilon$ -Übergängen aus Beispiel 3.5.2. Dann gilt:

$$\begin{aligned} \text{clos}_\varepsilon(z_0) &= \{z_0\} \\ \text{clos}_\varepsilon(z_1) &= \{z_1\} \\ \text{clos}_\varepsilon(z_4) &= \{z_4\} \\ \text{clos}_\varepsilon(z_3) &= \{z_1, z_3, z_4\} \\ \text{clos}_\varepsilon(z_2) &= \{z_1, z_2, z_3, z_4\} \end{aligned}$$

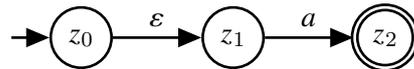
Für die Definition der akzeptierten Sprache eines NFA mit  $\varepsilon$ -Übergängen wird nun stets nach dem Anwenden der Überföhrungsfunktion  $\delta$  die erhaltene Menge noch abgeschlossen bezüglich der  $\varepsilon$ -Übergänge (durch Anwenden des  $\text{clos}_\varepsilon$ -Operators).

**Definition 3.5.5** (Akzeptierte Sprache eines NFA mit  $\varepsilon$ -Übergängen). Sei  $M = (Z, \Sigma, \delta, S, E)$  ein NFA mit  $\varepsilon$ -Übergängen. Wir definieren  $\tilde{\delta} : (\mathcal{P}(Z) \times \Sigma^*) \rightarrow \mathcal{P}(Z)$  induktiv durch:

$$\tilde{\delta}(X, \varepsilon) := X \quad \text{und} \quad \tilde{\delta}(X, aw) := \bigcup_{z \in X} \tilde{\delta}(\text{clos}_\varepsilon(\delta(z, a)), w) \text{ für alle } X \subseteq Z$$

Die von  $M$  akzeptierte Sprache ist  $L(M) := \{w \in \Sigma^* \mid \tilde{\delta}(\text{clos}_\varepsilon(S), w) \cap E \neq \emptyset\}$ .

**Bemerkung 3.5.6.** Die Anwendung des  $\text{clos}_\varepsilon$ -Operators auf die Menge  $S$  ist notwendig. Betrachte den NFA mit  $\varepsilon$ -Übergängen:



Die akzeptierte Sprache ist  $\{a\}$ , aber  $\tilde{\delta}(S, a) = \tilde{\delta}(\text{clos}_\varepsilon(\delta(z_0, a)), \varepsilon) = \text{clos}_\varepsilon(\delta(z_0, a)) = \text{clos}_\varepsilon(\emptyset) = \emptyset$  während  $\tilde{\delta}(\text{clos}_\varepsilon(S), a) = \tilde{\delta}(\{z_0, z_1\}, a) = \text{clos}_\varepsilon(\delta(z_0, a)) \cup \text{clos}_\varepsilon(\delta(z_1, a)) = \emptyset \cup \text{clos}_\varepsilon(\{z_2\}) = \{z_2\}$ .

**Satz 3.5.7.** NFAs mit  $\varepsilon$ -Übergängen akzeptieren genau die regulären Sprachen.

*Beweis.* Wir zeigen zuerst, dass jede reguläre Sprache von einem NFA mit  $\varepsilon$ -Übergängen akzeptiert wird: Sei  $L$  regulär. Dann gibt es einen NFA  $M = (Z, \Sigma, \delta, S, E)$ , der  $L$  akzeptiert. Der NFA mit  $\varepsilon$ -Übergängen  $M' = (Z, \Sigma, \delta', S, E)$  mit  $\delta'(z, a) := \delta(z, a)$  für alle  $a \in \Sigma$  und  $\delta'(z, \varepsilon) := \emptyset$  akzeptiert ebenfalls  $L$  (da  $L(M) = L(M')$ ).

Für die andere Richtung (akzeptierte Sprachen von NFAs mit  $\varepsilon$ -Übergängen sind regulär) sei ein NFA mit  $\varepsilon$ -Übergängen  $M = (Z, \Sigma, \delta, S, E)$  gegeben. Wir konstruieren einen NFA  $M'$  mit  $L(M) = L(M')$ , woraus dann mit Theorem 3.4.4 direkt folgt, dass  $L(M)$  regulär ist.

Sei  $M' = (Z, \Sigma, \delta', S', E)$  mit  $S' = \text{clos}_\varepsilon(S)$  und  $\delta'(z, a) = \text{clos}_\varepsilon(\delta(z, a))$ .

Wir zeigen, dass für alle  $X \subseteq Z$  und alle  $w \in \Sigma^*$  gilt:  $\tilde{\delta}(\text{clos}_\varepsilon(X), w) = \tilde{\delta}'(\text{clos}_\varepsilon(X), w)$ . Daraus folgt dann sofort, dass  $L(M) = L(M')$  gilt.

Wir verwenden Induktion über die Wortlänge  $|w|$ . Die Induktionsbasis ist, dass  $|w| = 0$  und daher  $w = \varepsilon$ . Dann gilt  $\tilde{\delta}(\text{clos}_\varepsilon(X), \varepsilon) = \text{clos}_\varepsilon(X) = \tilde{\delta}'(\text{clos}_\varepsilon(X), \varepsilon)$ .

Für den Induktionsschritt sei  $w = au$  mit  $a \in \Sigma$ . Als Induktionshypothese (I.H.) verwenden wir, dass gilt:  $\tilde{\delta}(clos_\varepsilon(W), u) = \widehat{\delta}(clos_\varepsilon(W), u)$  für alle  $W \subseteq Z$ .

Wir formen um:

$$\begin{aligned} \tilde{\delta}(clos_\varepsilon(X), au) &= \bigcup_{z \in clos_\varepsilon(X)} \tilde{\delta}(clos_\varepsilon(\delta(z, a)), u) \stackrel{I.H.}{=} \bigcup_{z \in clos_\varepsilon(X)} \widehat{\delta}(clos_\varepsilon(\delta(z, a)), u) \\ &= \bigcup_{z \in clos_\varepsilon(X)} \widehat{\delta}(\delta'(z, a), u) = \widehat{\delta}(clos_\varepsilon(X), au) = \widehat{\delta}(clos_\varepsilon(X), w) \quad \square \end{aligned}$$

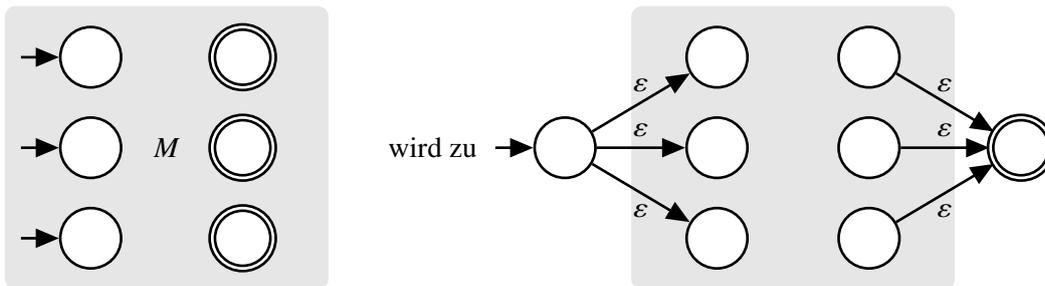
Für NFAs mit  $\varepsilon$ -Übergängen können wir leicht fordern, dass sie genau einen Startzustand und genau einen Endzustand besitzen:

**Satz 3.5.8** (NFA mit  $\varepsilon$ -Übergängen und eindeutigen Start- und Endzuständen). *Für jeden NFA  $M$  mit  $\varepsilon$ -Übergängen gibt es einen NFA  $M'$  mit  $\varepsilon$ -Übergängen, sodass  $L(M) = L(M')$  und  $M'$  genau einen Startzustand und genau einen Endzustand hat, wobei diese beiden Zustände verschieden sind.*

*Beweis.* Sei  $M = (Z, \Sigma, \delta, S, E)$ . Dann konstruiere  $M' = (Z \cup \{z_0, z_E\}, \Sigma, \delta', \{z_0\}, \{z_E\})$ , wobei  $z_0 \neq z_E$  neue Zustände sind und

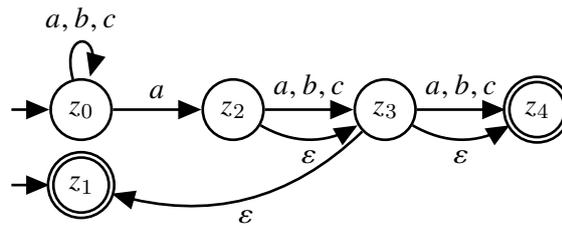
$$\begin{aligned} \delta'(z_0, \varepsilon) &= S \\ \delta'(z_0, a) &= \emptyset \quad \text{für alle } a \in \Sigma \\ \delta'(z, w) &= \delta(z, w) \quad \text{für alle } z \in Z \text{ und } w \in (\{\varepsilon\} \cup \Sigma) \\ \delta'(z, \varepsilon) &= \{z_E\} \quad \text{für alle } z \in E \\ \delta'(z_E, w) &= \emptyset \quad \text{für alle } w \in (\{\varepsilon\} \cup \Sigma) \end{aligned}$$

Bildlich dargestellt:

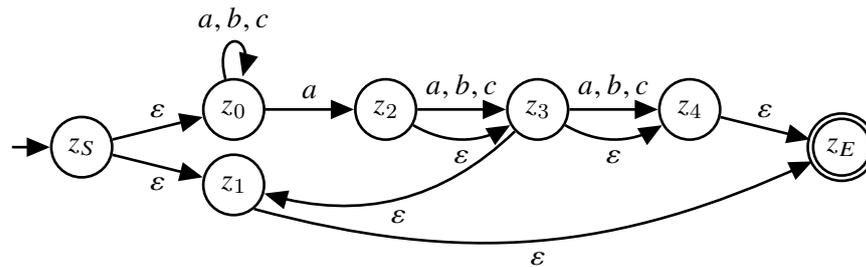


Offensichtlich gilt  $L(M) = L(M')$ . □

**Beispiel 3.5.9.** *Betrachte erneut den NFA mit  $\varepsilon$ -Übergängen aus Beispiel 3.5.2:*



Der dazu entsprechend Satz 3.5.8 konstruierte NFA mit  $\varepsilon$ -Übergängen und eindeutigen Start- und Endzuständen ist:



### 3.6. Reguläre Ausdrücke

Reguläre Ausdrücke sind (genau wie Grammatiken oder Automaten) ein Formalismus zur Darstellung von Sprachen. Hierbei wird durch einen Ausdruck, der aus Basisausdrücken und Operatoren aufgebaut ist, die Sprache definiert.

**Definition 3.6.1** (Regulärer Ausdruck). Sei  $\Sigma$  ein Alphabet. Ein regulärer Ausdruck über  $\Sigma$  ist induktiv definiert durch

- $\emptyset$  ist ein regulärer Ausdruck
- $\varepsilon$  ist ein regulärer Ausdruck
- $a$  mit  $a \in \Sigma$  ist ein regulärer Ausdruck
- Wenn  $\alpha_1$  und  $\alpha_2$  reguläre Ausdrücke sind, dann ist auch  $\alpha_1\alpha_2$  ein regulärer Ausdruck.
- Wenn  $\alpha_1$  und  $\alpha_2$  reguläre Ausdrücke sind, dann ist auch  $(\alpha_1|\alpha_2)$  ein regulärer Ausdruck.
- Wenn  $\alpha$  regulärer Ausdruck ist, dann auch  $(\alpha)^*$

Die von einem regulären Ausdruck  $\alpha$  erzeugte Sprache  $L(\alpha)$  ist induktiv über dessen Struktur definiert:

$$\begin{aligned}
 L(\emptyset) &:= \emptyset \\
 L(\varepsilon) &:= \{\varepsilon\} \\
 L(a) &:= \{a\} \text{ für } a \in \Sigma \\
 L(\alpha_1\alpha_2) &:= L(\alpha_1)L(\alpha_2) = \{uv \mid u \in L(\alpha_1), v \in L(\alpha_2)\} \\
 L(\alpha_1|\alpha_2) &:= L(\alpha_1) \cup L(\alpha_2) \\
 L((\alpha)^*) &:= L(\alpha)^*
 \end{aligned}$$

Da  $L((\alpha_1|\alpha_2)|\alpha_3) = L(\alpha_1|(\alpha_2|\alpha_3))$  für alle regulären Ausdrücke  $\alpha_1, \alpha_2, \alpha_3$ , lassen wir oft Klammern weg und schreiben  $(\alpha_1|\alpha_2|\dots|\alpha_n)$ .

**Beispiel 3.6.2.** Die vom regulären Ausdruck  $(a|b)^*aa(a|b)^*$  erzeugte Sprache (über dem Alphabet  $\{a, b\}$ ) sind alle Wörter, die zwei aufeinanderfolgende  $a$ 's enthalten.

Der reguläre Ausdruck  $(\varepsilon|((a|b|c)^*a(a|b|c)(a|b|c)(a|b|c)))$  erzeugt die Sprache aller Wörter über dem Alphabet  $\{a, b, c\}$ , die an viertletzter Stelle ein  $a$  haben, sowie das leere Wort.

Der reguläre Ausdruck

$((0|1|2|3|4|5|6|7|8|9)|1(0|1|2|3|4|5|6|7|8|9)|(2(0|1|2|3))):((0|1|2|3|4|5)(0|1|2|3|4|5|6|7|8|9))$

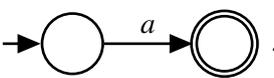
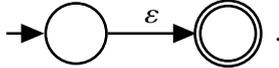
erzeugt alle gültigen Uhrzeiten im 24-Stunden-Format.

**Bemerkung 3.6.3.** Alle endlichen Sprachen sind durch reguläre Ausdrücke beschreibbar: Sei  $S = \{w_1, \dots, w_n\}$  eine endliche Sprache, dann erzeugt  $(w_1|w_2|\dots|w_n)$  die Sprache  $S$ .

**Theorem 3.6.4** (Satz von Kleene). Reguläre Ausdrücke erzeugen genau die regulären Sprachen.

*Beweis.* Wir müssen zwei Richtungen zeigen:

1. Jede von einem regulären Ausdruck erzeugte Sprache ist regulär.
  2. Für jede reguläre Sprache gibt es einen regulären Ausdruck, der sie erzeugt.
1. Sei  $\alpha$  ein regulärer Ausdruck. Wir konstruieren induktiv über die Struktur von  $\alpha$  einen NFA  $M_\alpha$  mit  $\varepsilon$ -Übergängen und eindeutigen Start- und Endzuständen, der  $L(\alpha)$  akzeptiert. Die Induktionsbasis stellen die Basisausdrücke dar:

- Für  $a \in \Sigma$  konstruiere den Automaten  .
- Für  $\varepsilon$  konstruiere den Automaten  .
- Für  $\emptyset$  konstruiere den Automaten  .

In allen drei Fällen ist offensichtlich, dass der NFA mit  $\varepsilon$ -Übergängen und eindeutigen Start- und Endzuständen die Sprache  $L(\alpha)$  akzeptiert.

Für den Induktionsschritt sei  $\alpha$  ein regulärer Ausdruck, der Operatoren enthält. Als Induktionsannahme nehmen wir an, dass für alle regulären Ausdrücke, die echte Unterausdrücke von  $\alpha$  sind, ein NFA mit  $\varepsilon$ -Übergängen und eindeutigen Start- und Endzuständen existiert, der die entsprechende Sprache akzeptiert.

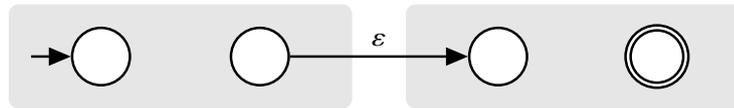
Wir betrachten alle drei möglichen Fälle für den Aufbau von  $\alpha$

- a) Falls  $\alpha = \alpha_1\alpha_2$ , dann liefert die Induktionshypothese NFAs mit  $\varepsilon$ -Übergängen und eindeutigen Start- und Endzuständen  $M_{\alpha_i}$  mit  $L(\alpha_i) = L(M_{\alpha_i})$  für  $i = 1, 2$ . Konstruiere  $M_\alpha$  wie folgt: Verbinde den Endzustand von  $M_{\alpha_1}$  mit einem  $\varepsilon$ -Übergang mit dem Startzustand von  $M_{\alpha_2}$  und mache den Startzustand von  $M_{\alpha_1}$  zum neuen

Startzustand von  $M_\alpha$  und den Endzustand von  $M_{\alpha_2}$  zum neuen Endzustand von  $M_\alpha$ . Bildlich skizziert wird

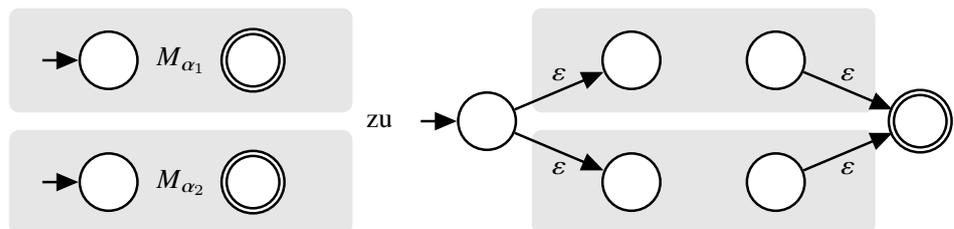


zu



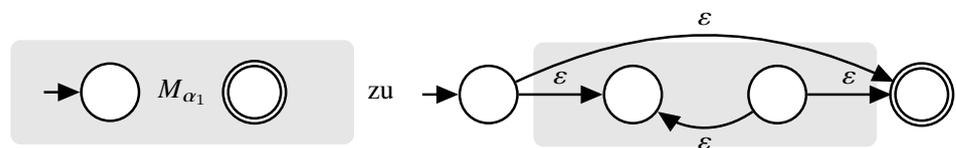
Offensichtlich gilt  $L(M_\alpha) = L(M_{\alpha_1})L(M_{\alpha_2})$ . Mit der Induktionsannahme gilt daher  $L(M_\alpha) = L(\alpha_1)L(\alpha_2)$ . Definition 3.6.1 liefert  $L(\alpha_1\alpha_2) = L(\alpha_1)L(\alpha_2)$  und daher  $L(M_\alpha) = L(\alpha)$ .

- b) Falls  $\alpha = (\alpha_1|\alpha_2)$ , so liefert die Induktionshypothese NFAs  $M_{\alpha_1}, M_{\alpha_2}$  mit  $\epsilon$ -Übergängen und eindeutigen Start- und Endzuständen, sodass  $L(\alpha_i) = L(M_{\alpha_i})$  für  $i = 1, 2$ . Konstruiere  $M_{(\alpha_1|\alpha_2)}$  wie folgt: Füge einen neuen Startzustand  $z_0$  und einen neuen Endzustand  $z_E$  hinzu. Für  $i = 1, 2$  füge einen  $\epsilon$ -Übergang von  $z_0$  zum Startzustand von  $M_{\alpha_i}$  hinzu. Für  $i = 1, 2$  füge einen  $\epsilon$ -Übergang vom Endzustand von  $M_{\alpha_i}$  zu  $z_E$  hinzu. Bildlich wird



Offensichtlich gilt  $L(M_\alpha) = L(M_{\alpha_1}) \cup L(M_{\alpha_2})$ . Mit der Induktionsannahme gilt daher  $L(M_\alpha) = L(\alpha_1) \cup L(\alpha_2)$ . Definition 3.6.1 liefert  $L(\alpha_1|\alpha_2) = L(\alpha_1) \cup L(\alpha_2)$  und daher  $L(M_\alpha) = L(\alpha)$ .

- c) Falls  $\alpha = (\alpha_1)^*$ , dann liefert die Induktionshypothese einen NFA mit  $\epsilon$ -Übergängen und einzigem Start- und Endzustand  $M_{\alpha_1}$  mit  $L(M_{\alpha_1}) = L(\alpha)$ . Konstruiere  $M_\alpha$  durch Hinzufügen eines neuen Startzustands, eines neuen Endzustands und von  $\epsilon$ -Übergängen vom alten Endzustand zum alten Startzustand, vom alten Endzustand zum neuen Endzustand, vom neuen Startzustand zum ehemaligen Startzustand, und vom neuen Startzustand zum neuen Endzustand D. h. bildlich wird



Der  $\epsilon$ -Übergang vom neuen Start- zum neuen Endzustand wird eingefügt, um das 0-fache Wiederholen von  $\alpha$  zu ermöglichen (und damit das leere Wort  $\epsilon$  zu akzeptieren).

Man kann hierfür nicht den alten Startzustand verwenden, denn dann würde man u. U. z. B. Präfixe von Wörtern akzeptieren, die vorher so nicht erzeugt wurden. Ebenso kann man nicht den alten Endzustand verwenden, da sonst Suffixe akzeptiert würden, die vorher nicht erzeugt wurden. Die neue Transition vom alten Endzustand zum alten Startzustand, kann genommen werden, jedesmal wenn man vom alten Start zum alten Endzustand gelangt ist, d.h. man hat ein Wort aus  $L(M_{\alpha_1})$  gelesen. Mit der Rückwärtskante und der Kante zum neuen Endzustand erkennt man daher  $L(M_{\alpha_1})^+$ . Insgesamt gilt  $L(M_{\alpha}) = L(M_{\alpha_1})^*$ . Mit der Induktionsannahme gilt daher  $L(M_{\alpha}) = L(\alpha_1)^*$ . Definition 3.6.1 liefert  $L(\alpha_1^*) = L(\alpha_1)^*$  und daher  $L(M_{\alpha}) = L(\alpha)$

2. O.B.d.A. sei eine reguläre Sprache durch einen DFA  $M = (Z, \Sigma, \delta, z_1, E)$ , der diese akzeptiert, gegeben. Wir konstruieren einen regulären Ausdruck  $\alpha$  sodass  $L(M) = L(\alpha)$  gilt. O.B.d.A. sei  $Z = \{z_1, \dots, z_n\}$ . Für  $w \in \Sigma^*$  und  $z_i \in Z$  mit  $\widehat{\delta}(z_i, w) = z_j$  sei  $visit_i(w) = q_1, \dots, q_m$  die Folge der durch den DFA besuchten Zustände (wobei  $q_1 = z_i$  und  $q_m = z_j$ ).

Wir definieren die Mengen  $L_{i,j}^k$  (für  $k \in \{0, \dots, n\}, i, j \in \{1, \dots, n\}$ ), sodass  $L_{i,j}^k$  genau alle Wörter  $w$  enthält, die  $M$  von Zustand  $z_i$  zu Zustand  $z_j$  führen (d. h.  $\widehat{\delta}(z_i, w) = z_j$ ), die als Zwischenzustände keine Zustände mit Index größer als  $k$  benutzen, d. h.:

$$L_{i,j}^k = \left\{ w \in \Sigma^* \mid \begin{array}{l} \widehat{\delta}(z_i, w) = z_j \text{ und } visit_i(w) = q_1, \dots, q_m, \\ \text{sodass für } l = 2, \dots, m - 1: \text{ wenn } q_l = z_p \text{ dann } p \leq k \end{array} \right\}$$

Wir zeigen per Induktion über  $k$ , dass es reguläre Ausdrücke  $\alpha_{i,j}^k$  gibt, die  $L_{i,j}^k$  erzeugen. Für  $k = 0$  betrachten wir zwei Fälle

- Wenn  $i \neq j$ , dann ist  $L_{i,j}^0 = \{a \in \Sigma \mid \delta(z_i, a) = z_j\}$ . Sei  $L_{i,j}^0 = \{a_1, \dots, a_q\}$ , dann gilt mit  $\alpha_{i,j}^0 = (a_1 | \dots | a_q)$ , dass  $L(\alpha_{i,j}^0) = L_{i,j}^0$ . Falls  $L_{i,j}^0 = \emptyset$ , dann sei  $\alpha_{i,j}^0 = \emptyset$  und es gilt ebenfalls  $L(\alpha_{i,j}^0) = L_{i,j}^0$ .
- Wenn  $i = j$ , dann ist  $L_{i,i}^0 = \{\varepsilon\} \cup \{a \in \Sigma \mid \delta(z_i, a) = z_i\}$ . Sei  $L_{i,i}^0 = \{\varepsilon, a_1, \dots, a_q\}$ , dann gilt mit  $\alpha_{i,i}^0 = (\varepsilon | a_1 | \dots | a_q)$ , dass  $L(\alpha_{i,i}^0) = L_{i,i}^0$

Für den Induktionsschritt verwenden wir als Induktionsannahme, dass es für alle  $i, j$  und festem  $k$  einen regulären Ausdruck  $\alpha_{i,j}^k$  gibt, der  $L_{i,j}^k$  erzeugt. Es gilt

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

denn ein Lauf von  $z_i$  zu  $z_j$  kann entweder den Zustand  $z_{k+1}$  nicht als Zwischenzustand verwenden (dieser Fall wird durch  $L_{i,j}^k$  abgedeckt) oder der Lauf kann aufgespalten werden in drei Teile die sequentiell aufeinander folgen:

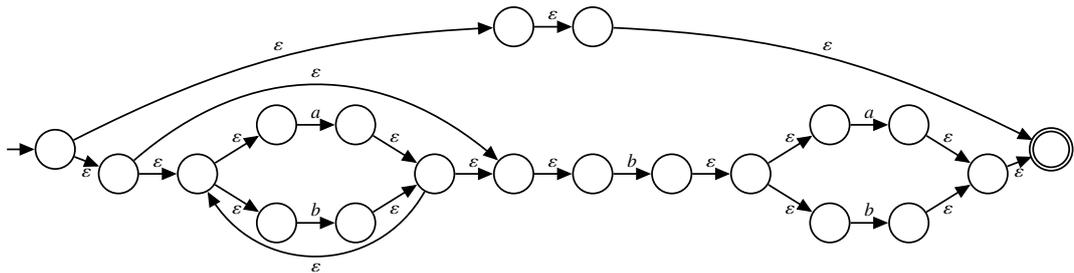
- a) Lauf von  $z_i$  bis zum ersten Besuch des Zustands  $z_{k+1}$  (abgedeckt durch  $L_{i,k+1}^k$ )
- b) Mehrmaliges, zyklisches Besuchen von  $k+1$  (beliebig oft) (abgedeckt durch  $L_{k+1,k+1}^k$ )
- c) Letztmaliges Verlassen von  $z_{k+1}$  und Lauf bis zu  $z_j$  (abgedeckt durch  $L_{k+1,j}^k$ )

Der reguläre Ausdruck, der  $L_{i,j}^{k+1}$  erzeugt, ist daher  $\alpha_{i,j}^{k+1} = (\alpha_{i,k+1}^k | \alpha_{k+1,k+1}^k)^* \alpha_{k+1,j}^k$ .

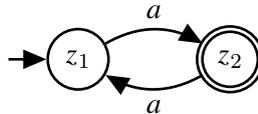
Die darin verwendeten regulären Ausdrücke sind alle von der Form  $\alpha_{i,i'}^k$  und daher durch die Induktionsannahme verfügbar.

Schließlich können wir den regulären Ausdruck angeben, der  $L(M)$  erzeugt: Sei  $E = \{z_{i_1}, \dots, z_{i_r}\}$ , dann gilt, dass  $(\alpha_{1,i_1}^n | \alpha_{1,i_2}^n | \dots | \alpha_{1,i_r}^n)$  die Sprache  $\bigcup_{z_i \in E} L_{1,i}^n = L(M)$  erzeugt.  $\square$

**Beispiel 3.6.5.** Wir konstruieren zum regulären Ausdruck  $(\varepsilon|(a|b)^*b(a|b))$  einen NFA mit  $\varepsilon$ -Übergängen und eindeutigen Start- und Endzuständen, der  $L(\varepsilon|(a|b)^*b(a|b))$  akzeptiert. Wir verwenden die Konstruktion aus dem Beweis des Satzes von Kleene (Theorem 3.6.4).



**Beispiel 3.6.6.** Betrachte den folgenden DFA  $M$



dann ist der entsprechend Theorem 3.6.4 konstruierte reguläre Ausdruck

$$\begin{aligned} \alpha_{1,2}^2 &= (\alpha_{1,2}^1 | \alpha_{1,2}^1 (\alpha_{2,2}^1)^* \alpha_{2,2}^1) \\ &= ((a|\varepsilon(\varepsilon)^*a) | (a|\varepsilon(\varepsilon)^*a) (\varepsilon|a(\varepsilon)^*a)^* (\varepsilon|a(\varepsilon)^*a)) \end{aligned}$$

denn

$$\begin{aligned} \alpha_{1,1}^0 &= \alpha_{2,2}^0 = \varepsilon \\ \alpha_{1,2}^0 &= \alpha_{2,1}^0 = a \\ \alpha_{1,2}^1 &= (\alpha_{1,2}^0 | \alpha_{1,1}^0 (\alpha_{1,1}^0)^* \alpha_{1,2}^0) = (a|\varepsilon(\varepsilon)^*a) \\ \alpha_{2,2}^1 &= (\alpha_{2,2}^0 | \alpha_{2,1}^0 (\alpha_{1,1}^0)^* \alpha_{2,2}^0) = (\varepsilon|a(\varepsilon)^*a) \end{aligned}$$

Da für alle regulären Ausdrücke  $\alpha$  die Gleichheiten  $L((\varepsilon)^*) = L(\varepsilon)$ ,  $L(\alpha\varepsilon) = L(\alpha)$ ,  $L((\varepsilon|\alpha)^*) = L((\alpha)^*)$ ,  $L((\varepsilon|\alpha)(\alpha)^*) = L((\alpha)^*(\varepsilon|\alpha)) = L((\alpha)^*)$  und  $L(\alpha|\alpha) = L(\alpha)$  gelten, kann man den regulären Ausdruck  $((a|\varepsilon(\varepsilon)^*a) | (a|\varepsilon(\varepsilon)^*a) (\varepsilon|a(\varepsilon)^*a)^* (\varepsilon|a(\varepsilon)^*a))$  vereinfachen zu  $(a|a(aa)^*)$ . Die repräsentierte Sprache besteht aus allen Wörtern  $a^i$  mit  $i \in \mathbb{N}$  und  $i$  ungerade.

### 3.7. Zusammenfassende Darstellung der Formalismen für reguläre Sprachen

Wir haben nun gezeigt, dass reguläre Grammatiken, DFAs, NFAs (mit  $\varepsilon$ -Übergängen (und eindeutigen Start- und Endzuständen)) sowie reguläre Ausdrücke allesamt Formalismen sind, um genau die Menge der regulären Sprachen zu repräsentieren.

Abbildung 3.1 zeigt nochmal die Zusammenhänge und gibt Referenzen auf die gezeigten Sätze und Theoreme. Ein Pfeil  $A \rightarrow B$  meint hierbei, dass wir eine Kodierung des einen Formalismus  $A$  in den anderen Formalismus  $B$  angeben haben.

Beachte, dass durch Benutzen von Wegen (d. h. mehrere Pfeile hintereinander), jeder Formalismus in jeden anderen Formalismus überführbar ist.

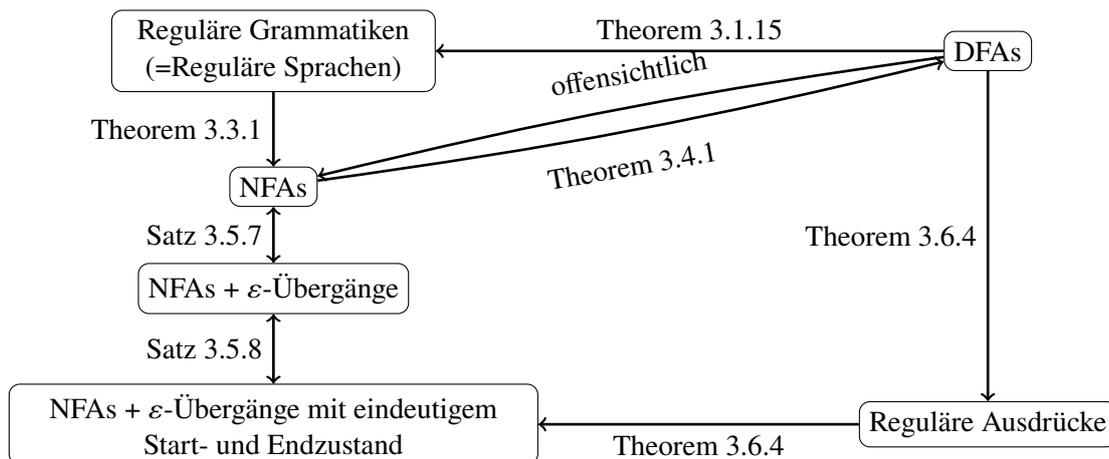


Abbildung 3.1.: Schaubild: Zusammenhänge zwischen den Formalismen für reguläre Sprachen

### 3.8. Widerlegen der Regulärität: Das Pumping-Lemma

Das Pumping-Lemma liefert eine Eigenschaft (die sogenannte Pumping-Eigenschaft, die wir noch definieren werden), die für *alle* regulären Sprachen gilt. Daher kann man es verwenden, um zu *widerlegen*, dass eine Sprache  $L$  regulär ist, indem man zeigt, dass die Sprache  $L$  die besagte Pumping-Eigenschaft *nicht* besitzt. Man kann das Pumping-Lemma *nicht verwenden*, um zu zeigen, dass eine Sprache  $L$  regulär ist, da die Pumping-Eigenschaft zwar eine notwendige aber keine hinreichende Bedingung für reguläre Sprachen ist (jede reguläre Sprache erfüllt sie, aber es gibt auch nicht-reguläre Sprachen, die sie erfüllen).

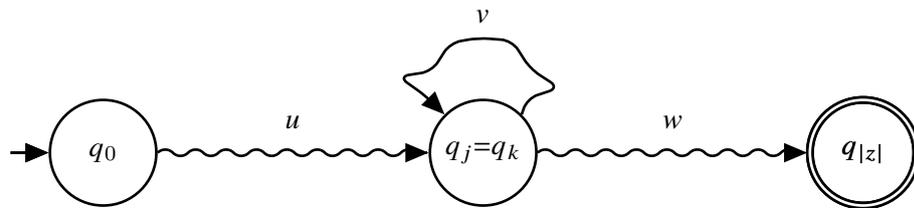
Die Idee der Pumping-Eigenschaft geht aus der Beobachtung hervor, dass ein DFA eine *endliche* Menge von Zuständen besitzt, und daher – sofern er eine unendliche Menge von Wörtern akzeptiert – Zustände mehrfach besuchen muss. Tatsächlich kann man sich klarmachen, dass mindestens ein Zustand mehrfach besucht werden muss, sobald man eine bestimmte Wortlänge überschreitet: Ein DFA mit  $n$  Zuständen, muss spätestens nach dem Einlesen von  $n$  Zeichen einen Zustand besuchen, den er vorher schon besucht hat.

**Lemma 3.8.1** (Pumping-Lemma). Sei  $L$  eine reguläre Sprache. Dann gilt die folgende, sogenannte Pumping-Eigenschaft für  $L$ : Es gibt eine Zahl  $n \in \mathbb{N}$ , sodass jedes Wort  $z \in L$ , das Mindestlänge  $n$  hat (d. h.  $|z| \geq n$ ), als  $z = uvw$  geschrieben werden kann, so dass gilt:

- $|uv| \leq n$
- $|v| \geq 1$
- für alle  $i \geq 0$ :  $uv^i w \in L$ .

Die Zahl  $n$  nennt man auch die Pumping-Konstante der Sprache  $L$ .

*Beweis.* Sei  $M = (Z, \Sigma, \delta, z_0, E)$  ein DFA der  $L$  akzeptiert. Sei  $n = |Z|$ . Dann muss jedes Wort  $z$ , das von  $L$  erkannt wird und Länge  $|z| \geq n$  hat, genau  $|z| + 1$  Zustände besuchen. O.B.d.A. sei die besuchte Folge  $q_0, q_1, \dots, q_{|z|}$  wobei  $q_0 = z_0$  und  $q_{|z|} \in E$ . Da  $|Z| = n$  gilt, wird mindestens ein Zustand zweimal besucht, spätestens nach dem Lesen der ersten  $n$  Zeichen. Sei  $q_k$  (mit  $k \leq n$ ) der erste Zustand, der bereits besucht wurde (d. h. es gibt ein  $j < k$ , sodass  $q_k = q_j$  und  $k$  ist minimal bez. dieser Eigenschaft). Sei  $u$  der Präfix von  $z$ , der von  $q_0$  zu  $q_j$  führte (d. h.  $\widehat{\delta}(q_0, u) = q_j$ ),  $v$  das Teilwort, das von  $q_j$  zu  $q_k$  führte (d. h.  $\widehat{\delta}(q_j, v) = q_k$ ) und  $w$  der verbleibende Suffix mit  $z = uvw$  für den  $\widehat{\delta}(q_k, w) = q_{|z|}$  gilt. Bildlich kann dies illustriert werden durch:



Wir zeigen nun die drei geforderten Eigenschaften der Zerlegung:

- Aus  $j < k$  folgt  $|v| \geq 1$ .
- Aus  $k \leq n$  folgt  $|uv| \leq n$
- Aus  $q_j = q_k$  folgt  $\widehat{\delta}(q_0, u) = q_j = \widehat{\delta}(q_0, uv) = q_k$  und somit  $\widehat{\delta}(q_0, uw) = \widehat{\delta}(q_0, uvw) = q_{|z|} \in E$ , d. h.  $uv^0 w \in L(M)$ . Sei  $i > 0$ , dann folgt aus  $\widehat{\delta}(q_j, v) = q_k = q_j$  auch  $\widehat{\delta}(q_j, v^i) = q_j$  und daher auch  $\widehat{\delta}(q_0, uv^i w) = \widehat{\delta}(q_k, v^i w) = \widehat{\delta}(q_j, w) = q_{|z|} \in E$ . Daher gilt  $uv^i w \in L(M)$  für alle  $i \in \mathbb{N}_0$ .  $\square$

**Bemerkung 3.8.2.** Sei  $L = \{w_1, \dots, w_m\}$  eine endliche Sprache. Dann erfüllt  $L$  die Pumping-Eigenschaft, denn mit  $k = \max\{|w_i| \mid i = \{1, \dots, m\}\}$  (die Länge des längsten Wortes in  $L$ ) und  $n = k + 1$  ist die Pumping-Eigenschaft für  $L$  erfüllt (denn es gibt keine Wörter der (Mindest-)länge  $n$  in  $L$ ).

Die wesentliche Verwendung für das Pumping-Lemma ist es, zu zeigen, dass eine Sprache *nicht regulär* ist. Hierfür wird Kontraposition verwendet: Die Aussage

$$L \text{ ist regulär} \implies L \text{ erfüllt die Eigenschaften des Pumping-Lemmas}$$

ist aquivalent zur Aussage

$L$  erfullt die Eigenschaften des Pumping-Lemmas *nicht*  $\implies L$  ist *nicht* regular

Dementsprechend konnen wir die Aussage des Pumping-Lemmas passend formulieren:

**Lemma 3.8.3** (Verwendung des Pumping-Lemmas zum Nicht-Regularitatsnachweis). *Sei  $L$  eine formale Sprache fur die gilt: Fur jede Zahl  $n \in \mathbb{N}$  gibt es ein  $z \in L$ , das Mindestlange  $n$  hat (d. h.  $|z| \geq n$ ), und fur jede Zerlegung  $z = uvw$  mit  $|uv| \leq n$  und  $|v| \geq 1$ , gibt es ein  $i \geq 0$ , sodass  $uv^i w \notin L$ . Dann ist  $L$  nicht regular.*

*Beweis.* Das Pumping-Lemma besagt „ $L$  ist regular  $\implies L$  erfullt die Pumping-Eigenschaft“. Diese Aussage (Implikation) ist logisch aquivalent ( $A \implies B$  g. d. w.  $\neg B \implies \neg A$ ) zur Aussage „ $L$  verletzt die Pumping-Eigenschaft  $\implies L$  ist nicht regular“.

Die Aussage „ $L$  erfullt die Pumping-Eigenschaft“ kann als folgende pradikatenlogische Formel geschrieben werden:

$$\exists n \in \mathbb{N} : \forall z \in L : (|z| \geq n \implies \exists u, v, w : (z = uvw \wedge |uv| \leq n \wedge |v| \geq 1 \wedge \forall i \geq 0 : (uv^i w \in L)))$$

Wir formen die negierte Formel solange um, bis sie die Form der Aussage dieses Lemmas hat:

$$\begin{aligned} & \neg(\exists n \in \mathbb{N} : \forall z \in L : (|z| \geq n \implies \exists u, v, w : (z = uvw \wedge |uv| \leq n \wedge |v| \geq 1 \wedge \forall i \geq 0 : (uv^i w \in L)))) \\ \iff & \forall n \in \mathbb{N} : \neg(\forall z \in L : (|z| \geq n \implies \exists u, v, w : (z = uvw \wedge |uv| \leq n \wedge |v| \geq 1 \wedge \forall i \geq 0 : (uv^i w \in L)))) \\ \iff & \forall n \in \mathbb{N} : (\exists z \in L : (\neg(|z| \geq n \implies \exists u, v, w : (z = uvw \wedge |uv| \leq n \wedge |v| \geq 1 \wedge \forall i \geq 0 : (uv^i w \in L)))))) \\ \iff & \forall n \in \mathbb{N} : (\exists z \in L : (\neg(\neg(|z| \geq n) \vee (\exists u, v, w : (z = uvw \wedge |uv| \leq n \wedge |v| \geq 1 \wedge \forall i \geq 0 : (uv^i w \in L))))))) \\ \iff & \forall n \in \mathbb{N} : (\exists z \in L : ((|z| \geq n) \wedge \neg(\exists u, v, w : (z = uvw \wedge |uv| \leq n \wedge |v| \geq 1 \wedge \forall i \geq 0 : (uv^i w \in L)))))) \\ \iff & \forall n \in \mathbb{N} : (\exists z \in L : ((|z| \geq n) \wedge (\forall u, v, w : (\neg(z = uvw \wedge |uv| \leq n \wedge |v| \geq 1 \wedge \forall i \geq 0 : (uv^i w \in L)))))) \\ \iff & \forall n \in \mathbb{N} : (\exists z \in L : ((|z| \geq n) \wedge (\forall u, v, w : (\neg(z = uvw \wedge |uv| \leq n \wedge |v| \geq 1) \vee \neg(\forall i \geq 0 : uv^i w \in L)))))) \\ \iff & \forall n \in \mathbb{N} : (\exists z \in L : ((|z| \geq n) \wedge (\forall u, v, w : ((z = uvw \wedge |uv| \leq n \wedge |v| \geq 1) \implies \neg(\forall i \geq 0 : uv^i w \in L)))))) \\ \iff & \forall n \in \mathbb{N} : (\exists z \in L : ((|z| \geq n) \wedge (\forall u, v, w : ((z = uvw \wedge |uv| \leq n \wedge |v| \geq 1) \implies \exists i \geq 0 : uv^i w \notin L)))) \quad \square \end{aligned}$$

**Satz 3.8.4.** *Die Sprache  $L = \{a^j b^j \mid j \in \mathbb{N}_0\}$  ist nicht regular.*

*Beweis.* Wir verwenden Lemma 3.8.3. Sei  $n \in \mathbb{N}$  beliebig. Wir wahlen fur  $z \in L$  das Wort  $a^n b^n$  (fur welches  $|z| \geq n$  erfullt ist). Sei  $z = uvw$  eine beliebige Zerlegung von  $z$ , sodass  $|uv| \leq n$  und  $|v| \geq 1$ . Dann gilt  $u = a^r$ ,  $v = a^{n-r}$  mit  $r < n$ . Daher konnen wir z. B.  $i = 2$  wahlen und erhalten  $uv^i w = uv^2 w = a^r a^{n-r} a^{n-r} b^n = a^{2n-r} b^n \notin L$ , da  $r < n$ . (Beachte: Wir hatten z. B. auch  $i = 0$  wahlen konnen!). Wir haben die Voraussetzung von Lemma 3.8.3 gezeigt, daher ist  $L$  nicht regular.  $\square$

Beachte, dass wir das Verwenden des Pumping-Lemmas zum Widerlegen der Regularität, wie im letzten Beweis auch als Gewinnstrategie für ein Spiel gegen einen Gegner (der davon überzeugen will, dass die betrachtete Sprache regulär ist) auffassen können:

Sei  $L$  die formale Sprache.

1. Der Gegner wählt die Zahl  $n \in \mathbb{N}$ .
2. Wir wählen das Wort  $z \in L$  mit  $|z| \geq n$ .
3. Der Gegner wählt die Zerlegung  $z = uvw$  mit  $|uv| \leq n$  und  $|v| \geq 1$ .
4. Wir gewinnen das Spiel, wenn wir ein  $i \geq 0$  angeben können, sodass  $uv^i w \notin L$ .

Wenn wir das Spiel für alle Wahlmöglichkeiten des Gegners gewinnen, dann haben wir die Nichtregularität von  $L$  nachgewiesen.

**Satz 3.8.5.** Die Sprache  $L = \{a^p \mid p \text{ ist Primzahl}\}$  ist nicht regulär.

*Beweis.* Wir verwenden das Pumping-Lemma in Form des eben eingeführten Spiels: Sei  $n \in \mathbb{N}$  vom Gegner gewählt. Wir wählen das Wort  $z \in L$  als  $a^p$  mit  $p$  ist die nächste Primzahl, die größer gleich  $n$  ist. Der Gegner wählt eine Zerlegung  $u = a^r$ ,  $v = a^s$ ,  $w = a^t$  mit  $uvw = a^p$ ,  $|uv| \leq n$ ,  $|v| \geq 1$ .

Wir wählen  $i = p+1$ . Dann ist  $uv^i w \notin L$ , denn  $uv^i w = a^r (a^s)^{p+1} a^t = a^{r+s \cdot (p+1)+t} = a^{r+s \cdot p+s+t} = a^{s \cdot p+p} = a^{p \cdot (s+1)}$  und für  $s \geq 1$  folgt, dass  $p \cdot (s+1)$  keine Primzahl sein kann.  $\square$

**Satz 3.8.6.** Die Sprache  $L = \{a^n \mid n \text{ ist Quadratzahl}\}$  ist nicht regulär.

*Beweis.* Sei  $n \in \mathbb{N}$  beliebig. Sei  $z \in L$  mit Mindestlänge  $n$  das Wort  $z = a^{n^2}$ . Sei  $z = uvw$ , sodass  $|uv| \leq n$  und  $|v| \geq 1$ . Dann gilt  $uv^2 w = a^k$  mit  $1 + n^2 \leq k$  (denn  $|v| \geq 1$ ) und  $k \leq n^2 + n$  (denn  $|uv| \leq n$  und daher  $|v| \leq n$ ). Dann kann  $k$  jedoch keine Quadratzahl sein, denn  $n^2 + n = (n+1) \cdot n < (n+1)^2$ . D. h.  $uv^2 w \notin L$ . Mit dem Pumping-Lemma folgt, dass  $L$  nicht regulär ist.  $\square$

**Satz 3.8.7.** Die Sprache  $L = \{a^{2^n} \mid n \in \mathbb{N}_0\}$  ist nicht regulär.

*Beweis.* Sei  $n \in \mathbb{N}$  beliebig. Sei  $z \in L$  das Wort  $z = a^{2^n}$  (welches Länge  $2^n > n$  hat). Sei  $z = uvw$  mit  $|uv| \leq n$  und  $|v| = k \geq 1$ . Dann ist  $1 \leq k \leq n$  und  $uv^2 w = a^{2^n+k}$  und  $2^n + k \neq 2^l$  da  $2^n + k < 2^{n+1} = 2^n + 2^n$ , denn  $k \leq n < 2^n$ . Mit dem Pumping-Lemma folgt, dass  $L$  nicht regulär ist.  $\square$

**Satz 3.8.8.** Die Sprache  $L = \{w \in \{a, b\}^* \mid w \text{ ist Palindrom}\}$  ist nicht regulär.

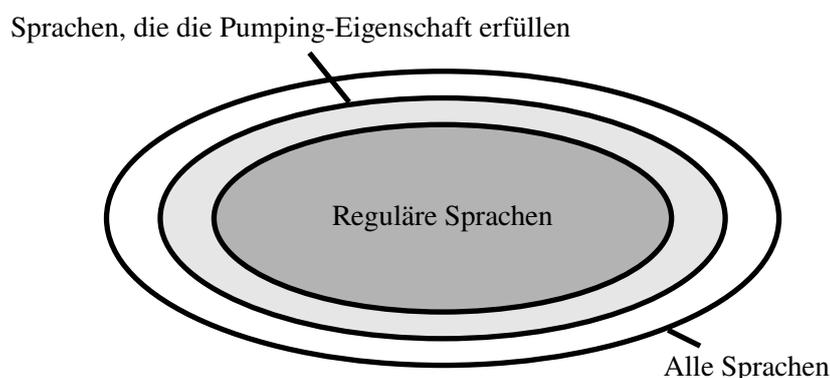
*Beweis.* Sei  $n \in \mathbb{N}$  beliebig. Wir wählen  $z = a^n b a^n$  als Wort mit Mindestlänge  $n$ . Sei  $z = uvw$  mit  $|uv| \leq n$  und  $|v| \geq 1$ . Dann ist  $uv^0 w = a^k b a^n$  mit  $k = n - |v| < n$  kein Palindrom. Mit dem Pumping-Lemma folgt, dass  $L$  nicht regulär ist.  $\square$

Das Pumping-Lemma liefert eine notwendige Bedingung für reguläre Sprachen, die jedoch nicht hinreichend ist, denn:

**Lemma 3.8.9.** *Es gibt Sprachen, welche die Pumping-Eigenschaft (siehe Lemma 3.8.1) erfüllen aber nicht regulär sind. Die Sprache  $L = \{a^k b^l c^l \mid k, l \in \mathbb{N}_0\} \cup \{b, c\}^*$  ist eine solche Sprache.*

*Beweis.* Wir zeigen hier nur, dass  $L$  die Pumping-Eigenschaft erfüllt (die Nichtregularität von  $L$  zeigen wir nicht, dies könnte man mit dem Satz von Myhill-Nerode machen, den wir in der Veranstaltung jedoch nicht behandeln). Sei  $n \geq 1$  die Zahl aus dem Pumping-Lemma und  $z \in L$  mit  $|z| \geq n$ . Wenn  $z \in \{b, c\}^*$ , zerlege z. B.  $u = \varepsilon, v$  das erste Symbol von  $z$  und  $w$  der  $n - 1$ -Zeichen lange Suffix von  $z$ . Offensichtlich gilt  $|v| \geq 1, |uv| \leq n$  und  $uv^i w \in \{b, c\}^* \subseteq L$  für alle  $i \in \mathbb{N}_0$ . Wenn  $z$  von der Form  $a^k b^l c^l$  ist und  $z \notin \{b, c\}^*$ , dann muss  $k > 0$  gelten und wir zerlegen  $z = uvw$  mit  $u = \varepsilon, v = a, w = a^{k-1} b^l c^l$ . Da  $|v| = 1, |uv| \leq n$  und  $uv^i w = a^{k+i-1} b^l c^l \in L$  für alle  $i \in \mathbb{N}_0$ , erfüllt  $L$  die Eigenschaften des Pumping-Lemmas.  $\square$

Das folgende Venn-Diagramm verdeutlicht die Beziehung zwischen allen Sprachen, den regulären Sprachen und den Sprachen, welche die Pumping-Eigenschaft erfüllen.



Nur für die, in der weiß markierten Teilmenge liegenden Sprachen, lässt sich das Pumping-Lemma verwenden, um deren Nichtregularität nachzuweisen. Eine genaue Charakterisierung der regulären Sprachen bietet der Satz von Myhill und Nerode, den wir jedoch auslassen (Interessierte können hierzu Anhang A.3 lesen).

### 3.9. Abschlusseigenschaften der regulären Sprachen

Wir untersuchen, gegenüber welchen Operationen die regulären Sprachen abgeschlossen sind.

**Satz 3.9.1.** *Die regulären Sprachen sind abgeschlossen bezüglich Vereinigung, Produkt und Kleeneschem Abschluss.*

*Beweis.* Das folgt direkt daraus, dass für reguläre Sprachen  $L_1, L_2$ , die durch reguläre Ausdrücke  $\alpha_1, \alpha_2$  erzeugt werden (d. h.  $L(\alpha_i) = L_i$  für  $i = 1, 2$ ), gilt:

- $(\alpha_1|\alpha_2)$  erzeugt  $L(\alpha_1|\alpha_2) = L(\alpha_1) \cup L(\alpha_2) = L_1 \cup L_2$ ,
- $\alpha_1\alpha_2$  erzeugt  $L(\alpha_1\alpha_2) = L(\alpha_1)L(\alpha_2) = L_1L_2$  und
- $(\alpha_1)^*$  erzeugt  $L(\alpha_1)^* = L_1^*$ .

D. h. Vereinigung, Produkt und Kleenescher Abschluss werden durch reguläre Ausdrücke erzeugt und sind daher reguläre Sprachen (Theorem 3.6.4).  $\square$

**Satz 3.9.2.** *Die regulären Sprachen sind abgeschlossen bezüglich Komplementbildung.*

*Beweis.* Sei  $L$  eine reguläre Sprache und  $M = (Z, \Sigma, \delta, z_0, E)$  ein DFA der  $L$  akzeptiert. Dann akzeptiert  $\overline{M} = (Z, \Sigma, \delta, z_0, Z \setminus E)$  die Sprache  $\overline{L}$  (d. h. das Komplement von  $L$ ): Offensichtlich gilt  $(\widehat{\delta}(z_0, w) \in E) \iff \neg(\widehat{\delta}(z_0, w) \in Z \setminus E)$   $\square$

**Satz 3.9.3.** *Die regulären Sprachen sind abgeschlossen bezüglich Schnitt.*

*Beweis.* Das folgt aus den Sätzen 3.9.1 und 3.9.2 und dem Fakt, dass  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$  gilt. Ein direkter Beweis ist: Seien  $M_1 = (Z_1, \Sigma, \delta_1, z_{01}, E_1)$  und  $M_2 = (Z_2, \Sigma, \delta_2, z_{02}, E_2)$  DFAs, die  $L_1 = L(M_1)$  und  $L_2 = L(M_2)$  akzeptieren. Der *Produktautomat* von  $M_1$  und  $M_2$  ist der DFA  $M = (Z_1 \times Z_2, \Sigma, \delta, (z_{01}, z_{02}), E_1 \times E_2)$  mit  $\delta((z, z'), a) = (\delta_1(z, a), \delta_2(z', a))$  für alle  $a \in \Sigma$  und  $(z, z') \in Z_1 \times Z_2$ .  $M$  akzeptiert  $L_1 \cap L_2$ , denn es gilt:  $\widehat{\delta}((z_{01}, z_{02}), w) \in (E_1, E_2) \iff (\widehat{\delta}_1(z_{01}, w) \in E_1 \wedge \widehat{\delta}_2(z_{02}, w) \in E_2)$ .  $\square$

Zusammenfassend halten wir fest:

**Theorem 3.9.4** (Abschlusseigenschaften der regulären Sprachen). *Die regulären Sprachen sind abgeschlossen bezüglich Vereinigung, Schnitt, Komplementbildung, Produkt und Kleeneschem Abschluss.*

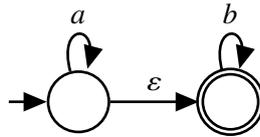
Beachte, dass die Abschlusseigenschaften der regulären Sprachen nicht nur nützlich sind, um aus regulären Sprachen weitere reguläre Sprachen zu konstruieren, sondern sie können auch verwendet werden, um die *Nichtregularität* von Sprachen zu zeigen.

**Satz 3.9.5.** *Die Sprache  $L = \{a^n \mid n \text{ ist keine Primzahl}\}$  ist nicht regulär.*

*Beweis.* Nehme an,  $L$  ist regulär. Da die regulären Sprachen abgeschlossen sind unter Komplementbildung (Theorem 3.9.4), gilt dann auch  $\overline{L} = \{a\}^* \setminus L = \{a^n \mid n \text{ ist eine Primzahl}\}$  ist regulär. Das ist jedoch ein Widerspruch zu Satz 3.8.5, der zeigt, dass  $\overline{L}$  nicht regulär ist. Daher war unsere Annahme falsch,  $L$  kann nicht regulär sein.  $\square$

**Satz 3.9.6.** *Die Sprache  $L = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$  ist nicht regulär.*

*Beweis.* Nehme an,  $L$  ist regulär. Die Sprache  $L' = \{a^n b^m \mid n, m \in \mathbb{N}_0\}$  ist regulär, da z. B. der folgende NFA mit  $\varepsilon$ -Übergängen  $L'$  akzeptiert:



Da  $L$  und  $L'$  regulär sind und die regulären Sprachen bezüglich der Schnittbildung abgeschlossen sind, folgt auch  $L \cap L'$  ist regulär. Da  $L \cap L' = \{a^j b^j \mid j \in \mathbb{N}_0\}$  ist dies ein Widerspruch zu Satz 3.8.4. Also war unsere Annahme falsch:  $L$  ist nicht regulär.  $\square$

### 3.10. Entscheidbarkeitsresultate zu regulären Sprachen

Wie bereits gezeigt, ist das Wortproblem für Typ 1,2,3-Sprachen entscheidbar (siehe Satz 2.3.2 und Korollar 2.3.3). Ist die Sprache regulär und durch einen DFA gegeben, dann kann das Wortproblem in Linearzeit in der Länge des Wortes entschieden werden, denn die Berechnung von  $\hat{\delta}(z_0, w)$  braucht für einen DFA nur  $|w|$  Schritte.

Wir betrachten nun die weiteren Probleme, wie wir sie in Abschnitt 2.4 definiert haben und zeigen, dass allesamt für die Klasse der regulären Sprachen entscheidbar sind.

**Satz 3.10.1.** *Das Leerheitsproblem für reguläre Sprachen ist entscheidbar.*

*Beweis.* Sei  $L$  eine reguläre Sprache, und  $M$  ein DFA mit  $L(M) = L$ . Dann gilt  $L = \emptyset$  genau dann, wenn es keinen Pfad vom Startzustand zu einem Endzustand in  $M$  gibt. Dies kann man leicht mit einer Tiefensuche auf dem Zustandsgraph von  $M$  überprüfen.  $\square$

**Satz 3.10.2.** *Das Endlichkeitsproblem für reguläre Sprachen ist entscheidbar.*

*Beweis.* Sei  $L$  eine reguläre Sprache und  $M$  ein DFA mit  $L(M) = L$ . Dann gilt  $|L| < \infty$  genau dann, wenn es keinen Pfad vom Startzustand zu einem Endzustand in  $M$  gibt, der eine Schleife enthält. Auch dies kann man leicht mit einer Tiefensuche auf dem Zustandsgraph von  $M$  überprüfen.  $\square$

**Satz 3.10.3.** *Das Schnittproblem für reguläre Sprachen ist entscheidbar.*

*Beweis.* Seien  $L_1$  und  $L_2$  reguläre Sprachen und  $M_1, M_2$  DFAs die  $L_1, L_2$  akzeptieren. Berechne den Produktautomaten  $M$  mit  $L(M) = L_1 \cap L_2$  (siehe Beweis von Satz 3.9.3) und prüfe anschließend das Leerheitsproblem für  $M$ .  $\square$

**Satz 3.10.4.** *Das Äquivalenzproblem für reguläre Sprachen ist entscheidbar.*

*Beweis.* Seien  $L_1$  und  $L_2$  reguläre Sprachen und  $M_1, M_2$  DFAs die  $L_1, L_2$  akzeptieren. Dann berechne die Minimalautomaten von  $M_1$  und  $M_2$  und prüfe diese auf Isomorphie.  $\square$

## 4. Kontextfreie Sprachen

In diesem Kapitel behandeln wir die Typ 2-Sprachen der Chomsky-Hierarchie, die kontextfreien Sprachen, kurz CFL (von context free language). Zur Erinnerung: Diese werden durch Typ 2-Grammatiken d. h. kontextfreie Grammatiken, kurz CFG (von context-free grammar), erzeugt. Diese fordern, dass linke Seiten der Produktionen genau aus einer Variablen bestehen.

Kontextfreie Sprachen erlauben mehr als reguläre Sprachen und sie sind insbesondere nützlich, um Klammernungen und geklammerte Sprachen zu beschreiben. Sie werden daher häufig verwendet, um die Syntax von Programmiersprachen zu beschreiben.

Z. B. beschreibt die kontextfreie Grammatik  $G = (\{E, M, Z\}, \{+, *, (\, )\} \cup \{0, \dots, 9\}, P, E)$  mit

$$\begin{aligned} P = \{ & E \rightarrow M \mid E + M, \\ & M \rightarrow Z \mid M * Z, \\ & Z \rightarrow N \mid (E), \\ & N \rightarrow 1D \mid \dots \mid 9D, \\ & D \rightarrow 0D \mid \dots \mid 9D \mid \varepsilon \} \end{aligned}$$

geklammerte arithmetische Ausdrücke.

Die Sprache  $L = \{a^j b^j \mid j \in \mathbb{N}_0\}$  ist nicht regulär (siehe Satz 3.8.4), aber kontextfrei, denn sie wird z. B. durch die Produktionen  $S \rightarrow \varepsilon \mid T$  und  $T \rightarrow aTb \mid ab$  und  $S$  als Startsymbol erzeugt.

In diesem Kapitel behandeln wir die Normalformen von kontextfreien Grammatiken (Chomsky-Normalform und Greibach-Normalform), eine Methode zum Widerlegen der Kontextfreiheit von formalen Sprachen (das Pumping-Lemma für kontextfreie Sprache). Abschlusseigenschaften von kontextfreien Sprachen, einen effizienten Algorithmus zum Lösen des Wortproblems für kontextfreie Sprachen (der CYK-Algorithmus), weitere Entscheidbarkeitsresultate und schließlich ein Automatenmodell, das genau die kontextfreien Sprachen erfasst (die Kellerautomaten).

### 4.1. Normalformen für kontextfreie Grammatiken

Wir definieren zwei Normalformen kontextfreier Grammatiken.

#### 4.1.1. Die Chomsky-Normalform

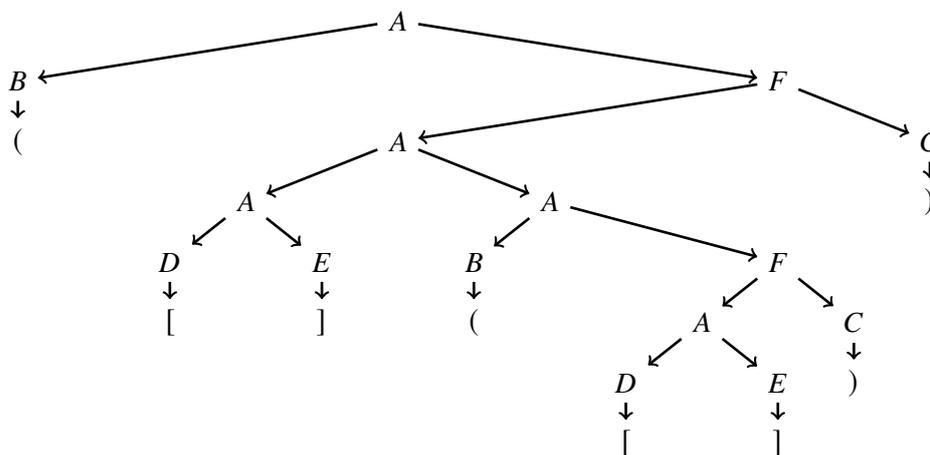
Die Chomsky-Normalform fordert eine spezielle Form aller Produktionen. Solche Normalformen sind beispielsweise nützlich, um Algorithmen auf Grammatiken zu formulieren und zu analysieren, da man sich auf die spezielle Form der Produktionen einschränken kann und daher weit weniger Fälle zu betrachten hat, als wenn man beliebige Produktionen zulässt.

**Definition 4.1.1.** Eine kontextfreie Grammatik  $G = (V, \Sigma, P, S)$  mit  $\varepsilon \notin L(G)$  ist in Chomsky-Normalform, wenn für  $A \rightarrow w \in P$  gilt:  $w = a \in \Sigma$  oder  $w = BC$  mit  $B, C \in V$ .

**Beispiel 4.1.2.** Die Grammatik  $G = (\{A\}, \{(\, , \, [, \, ]\}, \{A \rightarrow (A) \mid ( \mid [A] \mid [] \mid AA\}, A)$  ist nicht in Chomsky-Normalform (nur die Produktion  $A \rightarrow AA$  passt zum vorgeschriebenen Format). Hingegen ist  $G' = (\{A, B, C, D, E, F, G\}, \{(\, , \, [, \, ]\}, \{A \rightarrow BF \mid BC \mid DG \mid DE \mid AA, B \rightarrow (, C \rightarrow ), D \rightarrow [, E \rightarrow ], F \rightarrow AC, G \rightarrow AE\}, A)$  in Chomsky-Normalform (und erzeugt die gleiche Sprache wie  $G$ ).

Eigenschaften von Grammatiken in Chomsky-Normalform sind z. B., dass jeder Syntaxbaum zu einer Ableitung eines Wortes  $w$  ein binärer Baum ist, und dass eine Ableitung von Wort  $w$  genau  $2|w| - 1$  Ableitungsschritte benötigt.

**Beispiel 4.1.3.** Eine Linksableitung des (acht Zeichen langen) Wortes  $([]([]))$  mit der Grammatik  $G'$  ist mit den folgenden 15 Schritten möglich:  $A \Rightarrow BF \Rightarrow (F \Rightarrow (AC \Rightarrow (AAC \Rightarrow (DEAC \Rightarrow ([EAC \Rightarrow ([]AC \Rightarrow ([]BFC \Rightarrow ([](FC \Rightarrow ([](ACC \Rightarrow ([](DECC \Rightarrow ([]([ECC \Rightarrow ([]([]CC \Rightarrow ([]([])C \Rightarrow ([]([]))$ ). Der Syntaxbaum dazu ist:



Jede kontextfreie Grammatik  $G$  mit  $\varepsilon \notin L(G)$  kann in eine Grammatik  $G'$  in Chomsky-Normalform transformiert werden mit  $L(G) = L(G')$ . Das genaue Verfahren hierfür ist in Anhang A.5 angegeben. Die wesentlichen Ideen sind, dass man zunächst  $\varepsilon$ -Produktionen entfernt, indem die Variablen ausrechnet die das leere Wort erzeugen können, diese dann in anderen Produktionen wahlweise eliminiert. Anschließend werden Einheitsproduktionen der Form  $A \rightarrow B$ , wobei  $A$  und  $B$  Variablen sind, entfernt. Hierfür muss man zunächst Zyklen der Form  $A_1 \rightarrow A_2, \dots, A_{n-1} \rightarrow A_n, A_n \rightarrow A_1$  entfernen. Danach kann man sukzessive  $A \rightarrow B$  durch  $A \rightarrow w_1, \dots, A \rightarrow w_n$  ersetzen, wenn  $B \rightarrow w_1, \dots, B \rightarrow w_n$  alle Produktionen mit  $B$  als linker Seite sind. Anschließend fügt man Produktionen  $A_a \rightarrow a$  für jedes Terminalsymbol  $a$  hinzu und ersetzt dann alle Vorkommen von  $a$  in rechten Seiten (die nicht alleine aus  $a$  bestehen) durch  $A_a$ . Schließlich zerlegt man zulange rechte Seiten durch einführen neuer Variablen, z.B. wird  $A \rightarrow BCDE$  ersetzt durch  $A \rightarrow BV_1, V_1 \rightarrow CV_2, V_2 \rightarrow DE$ .

### 4.1.2. Die Greibach-Normalform

Eine weitere Normalform für kontextfreie Grammatiken ist die sogenannte Greibach-Normalform (benannt nach Sheila A. Greibach).

**Definition 4.1.4.** Eine CFG  $G = (V, \Sigma, P, S)$  mit  $\varepsilon \notin L(G)$  ist in Greibach-Normalform, falls alle Produktionen in  $P$  von der Form  $A \rightarrow aB_1B_2 \dots B_j$  sind, wobei  $j \geq 0$ ,  $A, B_1, \dots, B_j \in V$  und  $a \in \Sigma$ .

Reguläre Grammatiken sind ein Spezialfall der Greibach-Normalform: Dort ist nur  $j = 0$  oder  $j = 1$  zugelassen. Umgekehrt kann man auch sagen: Jede reguläre Grammatik (ohne  $\varepsilon$ -Produktionen) ist in Greibach-Normalform.

Jede kontextfreie Grammatik  $G$  mit  $\varepsilon \notin L(G)$  kann in Greibach-Normalform gebracht werden. Wir lassen das Verfahren aus (es kann in Anhang A.6 nachgelesen werden).

### 4.2. Widerlegen der Kontextfreiheit

Wir zeigen ein Hilfsmittel, um die Kontextfreiheit von formalen Sprachen zu widerlegen.

Als Vorüberlegung beachte, dass ein Binärbaum (ein Baum für den jeder Knoten 2 oder 0 Kinder hat), mit  $\geq 2^k$  Blättern stets einen Pfad der Länge mindestens  $k$  hat. Dies kann mit Induktion über  $k$  gezeigt werden: Für  $k = 0$  besteht der Baum nur aus einem Blatt und er hat einen Pfad der Länge 0. Wenn  $k > 0$ , dann hat der Baum  $2^k$  Blätter und einer der beiden Teilbäume unter der Wurzel hat mindestens  $2^{k-1}$  Blätter. Die Induktionshypothese liefert, dass dieser Teilbaum ein Pfad der Länge  $\geq k - 1$  hat. Verwende diesen Pfad und verlängere ihn um die Wurzel des gesamten Baums. Der Pfad hat Länge  $\geq k$ .

Wir werden diese Vorüberlegung mehrfach im Beweis des folgenden Pumping-Lemmas für kontextfreie Sprache verwenden. Ähnlich zum das Pumping-Lemma für reguläre Sprachen (siehe Lemma 3.8.1), liefert das Pumping-Lemma für CFLs eine notwendige Eigenschaft kontextfreier Sprachen, die – kurz gesprochen – besagt, dass genügend lange Wörter kontextfreier Sprachen „aufgepumpt“ werden können, ohne die Sprache zu verlassen. Im Gegensatz zu den regulären Sprachen, wird bei kontextfreien Sprachen an zwei Stellen gleichzeitig gepumpt:

**Lemma 4.2.1** (Pumping-Lemma für CFLs). Sei  $L$  eine kontextfreie Sprache. Dann gibt es eine Zahl  $n \in \mathbb{N}$ , sodass jedes Wort  $z \in L$ , das Mindestlänge  $n$  hat (d. h.  $|z| \geq n$ ), als  $z = uvwxy$  geschrieben werden kann, so dass gilt:

- $|vx| \geq 1$
- $|vwx| \leq n$
- für alle  $i \geq 0$ :  $uv^iwx^iy \in L$ .

*Beweis.* Sei  $L$  eine kontextfreie Sprache und sei  $G = (V, \Sigma, P, S)$  eine CFG für  $L \setminus \{\varepsilon\}$  in Chomsky-Normalform. Betrachte eine Ableitung und den zugehörigen Syntaxbaum eines Wortes

$z$  mit  $|z| \geq 2^{|V|} = n$ . Da  $G$  in Chomsky-Normalform ist, hat jeder Knoten im Syntaxbaum genau 2 Kinder, bis auf den jeweils letzten Schritt in jedem Pfad (der Produktionen von der Form  $A \rightarrow a$  anwendet).

Der Baum, ohne diesen letzten Schritt ist ein Binärbaum mit  $|w| \geq 2^{|V|}$  Blättern, der daher einen Pfad der Länge  $\geq |V|$  haben muss, der  $|V| + 1$  Variablen benutzt.

Das rechts gezeigte Bild illustriert dies, wobei die grau markierte Schicht, die jeweils letzten Ableitungsschritte markieren. Daher muss auf diesem Pfad mit  $|V| + 1$  Variablen mindestens eine Variable doppelt vorkommen.

Wir wählen die beiden tiefsten Vorkommen derselben Variablen, d. h. wir gehen den Pfad von unten nach oben entlang, bis zum ersten Mal eine Variable erneut vorkommt. O.B.d.A. sei dies die Variable  $A$ .

Wenn wir die beiden Teilbäume betrachten, die jeweils  $A$  als Wurzel haben, dann entsprechen diese den Ableitungen von Teilworten von  $z$ , wobei das tiefere  $A$  ein Teilwort des Wortes des vom oberen  $A$  erzeugten Wortes ist. D. h. wir können das Wort  $z$  zerlegen in  $z = uvwxy$ , wobei  $w$  das vom unteren  $A$  erzeugte Teilwort ist,  $vw$  vom oberen  $A$  erzeugt wird, und  $u$  und  $v$  die Teile von  $z$  sind, die außerhalb des vom oberen  $A$  erzeugten Teilworts liegen.

Zunächst ist klar, dass  $|w| \geq 1$  gilt, da jede Variable einer Grammatik in Chomsky-Normalform nur solche Wörter herleiten kann. Der Syntaxbaum vom oberen  $A$  hat unter der Wurzel zwei Teilbäume mit Wurzeln  $B$  und  $C$ . Nur einer der beiden Teilbäume enthält den Teilbaum mit dem unteren  $A$  als Wurzel. Der andere Teilbaum leitet daher auch Zeichen her. Daher muss  $|v| \geq 1$  oder  $|x| \geq 1$  (oder beides), vereinfacht geschrieben  $|vx| \geq 1$ , gelten.

Da wir die tiefsten Vorkommen der Variablen  $A$  gewählt haben, muss der Pfad vom oberen  $A$  bis zur der Blattebene eine Länge  $\leq |V|$  haben. Daher gilt für das abgeleitete Wort  $|vwx| \leq 2^{|V|} = n$ .

Schließlich zeigt der Syntaxbaum, dass  $S \Rightarrow^* uAy$ ,  $A \Rightarrow^* vAx$  und  $A \Rightarrow^* w$ . Die Ableitungen für  $A$  können wir daher beliebig austauschen: Zusammengefasst zeigt dies, dass gilt  $S \Rightarrow^* uv^iwx^i y$  für alle  $i \in \mathbb{N}_0$ . Im Baum dargestellt für die Fälle  $i = 0$  und  $i = 2$ :

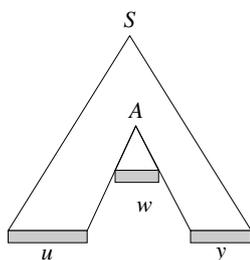


Illustration für  $uv^0wx^0y$

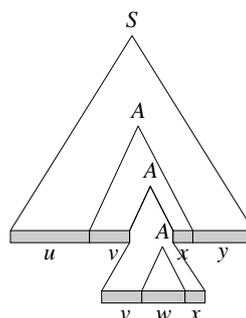


Illustration für  $uv^2wx^2y$

□

Da das Pumping-Lemma keine hinreichende Eigenschaft für kontextfreie Sprachen liefert, kann es **nicht** verwendet werden, um zu zeigen, dass eine Sprache kontextfrei ist. Es kann allerdings verwendet werden, um zu zeigen, dass eine Sprache nicht kontextfrei ist.

Entsprechend können wir die Aussage des Pumping-Lemmas für CFGs passend formulieren:

**Lemma 4.2.2** (Formulierung des Pumping-Lemmas für CFGs zum Widerlegen der Kontextfreiheit). *Sei  $L$  eine formale Sprache für die gilt: Für jede Zahl  $n \in \mathbb{N}$  gibt es ein  $z \in L$ , das Mindestlänge  $n$  hat (d. h.  $|z| \geq n$ ), und für jede Zerlegung  $z = uvwxy$  mit  $|vwx| \leq n$  und  $|vx| \geq 1$ , gibt es ein  $i \geq 0$ , sodass  $uv^iwx^iy \notin L$ . Dann ist  $L$  nicht kontextfrei.*

*Beweis.* Lemma 4.2.1 sagt „ $L$  ist kontextfrei  $\implies L$  hat die Pumping-Eigenschaft für CFLs“. Diese Implikation ist logisch äquivalent zur Aussage „ $L$  verletzt die Pumping-Eigenschaft für CFLs  $\implies L$  ist nicht kontextfrei“.

Die Aussage „ $L$  erfüllt die Pumping-Eigenschaft für CFLs“ kann als folgende prädikatenlogische Formel geschrieben werden:

$$\exists n \in \mathbb{N} : \forall z \in L : (|z| \geq n \implies \exists u, v, w, x, y : (z = uvwxy \wedge |vx| \geq 1 \wedge |vwx| \leq n \wedge \forall i \geq 0 : (uv^iwx^iy \in L)))$$

Wir formen die negierte Formel solange um, bis sie die Form der Aussage dieses Lemmas hat:

$$\begin{aligned} & \neg(\exists n \in \mathbb{N} : \forall z \in L : (|z| \geq n \implies \exists u, v, w, x, y : (z = uvwxy \wedge |vx| \geq 1 \wedge |vwx| \leq n \wedge \forall i \geq 0 : (uv^iwx^iy \in L)))) \\ \iff & \forall n \in \mathbb{N} : \exists z \in L : \neg(|z| \geq n \implies \exists u, v, w, x, y : (z = uvwxy \wedge |vx| \geq 1 \wedge |vwx| \leq n \wedge \forall i \geq 0 : (uv^iwx^iy \in L))) \\ \iff & \forall n \in \mathbb{N} : \exists z \in L : (|z| \geq n \wedge \neg(\exists u, v, w, x, y : (z = uvwxy \wedge |vx| \geq 1 \wedge |vwx| \leq n \wedge \forall i \geq 0 : (uv^iwx^iy \in L)))) \\ \iff & \forall n \in \mathbb{N} : \exists z \in L : (|z| \geq n \wedge (\forall u, v, w, x, y : \neg(z = uvwxy \wedge |vx| \geq 1 \wedge |vwx| \leq n \wedge \forall i \geq 0 : (uv^iwx^iy \in L)))) \\ \iff & \forall n \in \mathbb{N} : \exists z \in L : (|z| \geq n \wedge (\forall u, v, w, x, y : \neg(z = uvwxy \wedge |vx| \geq 1 \wedge |vwx| \leq n) \vee \neg(\forall i \geq 0 : (uv^iwx^iy \in L)))) \\ \iff & \forall n \in \mathbb{N} : \exists z \in L : (|z| \geq n \wedge (\forall u, v, w, x, y : \neg(z = uvwxy \wedge |vx| \geq 1 \wedge |vwx| \leq n) \vee (\exists i \geq 0 : (uv^iwx^iy \notin L)))) \\ \iff & \forall n \in \mathbb{N} : \exists z \in L : (|z| \geq n \wedge (\forall u, v, w, x, y : (z = uvwxy \wedge |vx| \geq 1 \wedge |vwx| \leq n) \implies (\exists i \geq 0 : (uv^iwx^iy \notin L)))) \quad \square \end{aligned}$$

Analog zur Verwendung des Pumping-Lemmas für reguläre Sprachen, lässt sich auch das Pumping-Lemma für kontextfreie Sprachen als Finden einer Gewinnstrategie eines Spiels gegen einen Gegner formulieren:

Sei  $L$  die formale Sprache.

1. Der Gegner wählt die Zahl  $n \in \mathbb{N}$ .
2. Wir wählen das Wort  $z \in L$  mit  $|z| \geq n$ .
3. Der Gegner wählt die Zerlegung  $z = uvwxy$  mit  $|vx| \geq 1$  und  $|vwx| \leq n$

4. Wir gewinnen das Spiel, wenn wir ein  $i \geq 0$  angeben können, sodass  $uv^iwx^i y \notin L$ .

Wenn wir für jede Wahl des Gegners das Spiel gewinnen können, dann haben wir gezeigt, dass  $L$  nicht kontextfrei ist.

**Satz 4.2.3.** Die Sprache  $L = \{a^l b^l c^l \mid l \in \mathbb{N}_0\}$  ist nicht kontextfrei.

*Beweis.* Wir verwenden das Pumping-Lemma für CFGs. Sei  $n \in \mathbb{N}$ . Wir wählen  $z = a^n b^n c^n$ . Für jede Zerlegung  $z = uvwxy$  mit  $|vx| \geq 1$  und  $|vwx| \leq n$  ist  $vwx$  von der Form  $a^i b^j$  oder  $b^i c^j$  mit  $i, j \geq 0, i + j \leq n$  ( $vwx$  kann nicht  $a$ 's,  $b$ 's und  $c$ 's enthalten). Da  $|vx| \geq 1$ , enthält  $|vx|$  im ersten Fall kein  $c$  aber mindestens ein  $a$  oder ein  $b$  und im zweiten Fall kein  $a$  aber mindestens ein  $b$  oder ein  $c$ . Daher hat das Wort  $uv^0wx^0y = uwy$  im ersten Fall mehr Vorkommen von  $c$  als von  $a$  oder  $b$  und im zweiten Fall mehr Vorkommen von  $a$  als von  $b$  oder  $c$  und somit gilt  $uwy \notin L$ .  $\square$

**Satz 4.2.4.** Die Sprache  $L = \{a^i b^j c^i d^j \mid i, j \in \mathbb{N}\}$  ist nicht kontextfrei.

*Beweis.* Wir verwenden das Pumping-Lemma für CFGs. Sei  $n \in \mathbb{N}$ . Wir wählen  $z = a^n b^n c^n d^n$ . Für jede Zerlegung  $z = uvwxy$  mit  $|vx| \geq 1$  und  $|vwx| \leq n$  gilt, genau eine der folgenden Möglichkeiten

1.  $vwx = a^i b^j$
2.  $vwx = b^i c^j$
3.  $vwx = c^i d^j$

mit jeweils  $i, j \in \mathbb{N}_0$  und  $i + j \leq n$ . In jedem der drei Fälle gilt  $uv^0wx^0y = uwy \notin L$ :

1. Da  $|vx| \geq 1$ , enthält  $vx$  mindestens ein  $a$  oder ein  $b$ . Dann gilt  $uwy = a^{i'} b^{j'} c^n d^n$  und  $i' < n$  und/oder  $j' < n$ .
2. Da  $|vx| \geq 1$ , enthält  $vx$  mindestens ein  $b$  oder ein  $c$ . Dann gilt  $uwy = a^n b^{i'} c^{j'} d^n$  und  $i' < n$  und/oder  $j' < n$ .
3. Da  $|vx| \geq 1$ , enthält  $vx$  mindestens ein  $c$  oder ein  $d$ . Dann gilt  $uwy = a^n b^n c^{i'} d^{j'}$  und  $i' < n$  und/oder  $j' < n$ .  $\square$

**Satz 4.2.5.** Sei  $L$  eine formale Sprache über einem unären Alphabet (d. h.  $|\Sigma| = 1$ ). Dann ist  $L$  genau dann regulär, wenn  $L$  kontextfrei ist.

*Beweis.* Wenn  $L$  regulär ist, dann ist  $L$  auch kontextfrei. D. h. wir müssen nur zeigen, dass jede kontextfreie Sprache  $L$  im Falle des unären Alphabets auch regulär ist. O.B.d.A. sei  $\Sigma = \{a\}$  und  $L$  eine kontextfreie Sprache über  $\Sigma$ . Das Pumping-Lemma für CFLs zeigt: Es gibt eine Zahl  $n \in \mathbb{N}$ , sodass:

Jedes Wort  $a^m \in L$  mit  $m \geq n$  lässt sich zerlegen als  $a^m = uvwxy$  mit  $|vx| \geq 1$ ,  $|vwx| \leq n$  und für alle  $i \in \mathbb{N}_0$ :  $uv^iwx^i y \in L$ .

Da  $u, v, w, x, y \in \{a\}^*$ , ist diese Aussage äquivalent zu folgender Aussage (in der die Reihenfolge der Wörter  $u, v, w, x, y$  getauscht ist):

Jedes Wort  $a^m \in L$  mit  $m \geq n$  lässt sich zerlegen als  $a^m = uwyvx$  mit  $|vx| \geq 1$ ,  $|vx| \leq |vwx| \leq n$  und für alle  $i \in \mathbb{N}_0$ :  $uwyv^i x^i \in L$ .

Da  $uwy$  von der Form  $a^k$  und  $vx$  von der Form  $a^l$  sein muss, lässt die Aussage noch einfacher formulieren:

Jedes Wort  $a^m \in L$  mit  $m \geq n$  lässt sich zerlegen als  $a^m = a^k a^l$  mit  $m = k + l$ ,  $1 \leq l \leq n$  und für alle  $i \in \mathbb{N}_0$ :  $a^k a^{i \cdot l} \in L$ .

Jedes Wort aus  $L$  mit Mindestlänge  $n$  lässt sich daher schreiben als  $a^k a^l$  mit  $1 \leq l \leq n$ .

Sei  $\tilde{L}_j := \{a^j a^{i \cdot n!} \mid i \in \mathbb{N}_0\}$ . Die Sprache  $\tilde{L}_j$  (für festes  $n$  und  $j$ ) ist regulär, denn der NFA mit  $\varepsilon$ -Übergängen  $M_j = (\{z_0, \dots, z_{n!+j}\}, \Sigma, \delta, z_0, \{z_{n!+j}\})$  mit  $\delta(z_i, a) = \{z_{i+1}\}$  für  $0 \leq i \leq n! + j - 1$ ,  $\delta(z_j, \varepsilon) = \{z_{n!+j}\}$  und  $\delta(z_{n!+j}, \varepsilon) = \{z_j\}$  akzeptiert  $\tilde{L}_j$ .

Wir zeigen  $\tilde{L}_j \subseteq L$  für  $j \geq n!$  und  $a^j \in L$ :

Sei  $j \geq n!$  und  $a^j \in L$ . Dann ist für alle  $i \in \mathbb{N}_0$  auch  $a^j a^{i \cdot n!} \in L$ : Da  $a^j \in L$  und  $j \geq n$ , gibt es  $l, k$  mit  $a^j = a^k a^l$  und  $1 \leq l \leq n$ . Da  $n! = l \cdot (n!/l)$  ( $l$  teilt tatsächlich  $n!$ , weil  $l \leq n$ ), gilt für beliebiges  $i \in \mathbb{N}_0$ :  $a^j a^{i \cdot n!} = a^{k+l+i \cdot l \cdot (n!/l)} = a^k a^{l \cdot (1+i \cdot (n!/l))}$  und daher ist  $a^j a^{i \cdot n!} \in L$ .

Umgekehrt gilt für jedes Wort  $a^m \in L$  mit  $m \geq n!$ :  $a^m \in \tilde{L}_j$  (mit  $j \geq n!$ ), wenn  $m = j + i \cdot n!$ . D. h. wir können ein kleinstes  $j$  wählen, für das gilt  $j \equiv m \pmod{n!}$ .

Daraus folgt einerseits, dass alle Wörter aus  $L$  mit Mindestlänge  $n!$  auch in einer der Mengen  $\tilde{L}_j$  liegen, und zudem, dass wir nur endlich viele solcher Mengen  $\tilde{L}_j$  vereinigen müssen, um alle Wörter aus  $L$  mit Mindestlänge  $n!$  darzustellen: Es gibt höchstens  $n!$  verschiedene Reste bei der Division durch  $n!$ . Daher gibt es eine Zahl  $q \in \mathbb{N}_0$ , sodass die unendliche Vereinigung  $\bigcup \{\tilde{L}_j \mid a^j \in L, j \in \mathbb{N}_0, j \geq n!\}$  als *endliche* Vereinigung

$$\bigcup \{\tilde{L}_j \mid a^j \in L, n! \leq j \leq q\} = \bigcup \{\tilde{L}_j \mid a^j \in L, j \in \mathbb{N}_0, j \geq n!\}$$

dargestellt werden kann.

Nun können wir die gesamte Sprache  $L$  darstellen durch:

$$L = \{w \in L \mid |w| < n!\} \cup \bigcup \{\tilde{L}_j \mid a^j \in L, n! \leq j \leq q\}.$$

Jede der einzelnen Teilmengen ist eine reguläre Sprache (die erste, da sie eine endliche Sprache ist, für die anderen Teilmengen existiert jeweils der NFA  $M_j$ ) und es werden nur endlich viele Sprachen vereinigt. Daher ist  $L$  regulär.  $\square$

#### Satz 4.2.6. Die Sprachen

$$\begin{aligned} L_1 &= \{a^p \mid p \text{ ist eine Primzahl}\} & L_2 &= \{a^n \mid n \text{ ist keine Primzahl}\} \\ L_3 &= \{a^n \mid n \text{ ist Quadratzahl}\} & L_4 &= \{a^{2^n} \mid n \in \mathbb{N}_0\} \end{aligned}$$

sind allesamt nicht kontextfrei.

*Beweis.* Für alle vier Sprachen haben gezeigt, dass sie nicht regulär sind (Sätze 3.8.5 bis 3.8.7 und 3.9.5), alle sind über dem unären Alphabet  $\Sigma = \{a\}$  definiert. Daher folgt mit Satz 4.2.5, dass die Sprachen auch nicht kontextfrei sind.  $\square$

### 4.3. Abschlusseigenschaften kontextfreier Sprachen

**Theorem 4.3.1.** *Die kontextfreien Sprachen sind abgeschlossen unter*

- Vereinigung
- Produkt
- Kleeneschem Abschluss

*Die kontextfreien Sprachen sind nicht abgeschlossen unter Schnitt- und Komplementbildung.*

*Beweis.* Seien  $L_1$  und  $L_2$  kontextfreie Sprachen,  $G_1 = (V_1, \Sigma_1, P_1, S_1)$  und  $G_2 = (V_2, \Sigma_2, P_2, S_2)$  CFGs mit  $L(G_i) = L_i$  für  $i = 1, 2$ . O.B.d.A. gelte  $V_1 \cap V_2 = \emptyset$  (anderenfalls benenne eine der Grammatiken um). Sei  $S$  eine neue Variable ( $S \notin (V_1 \cup V_2)$ ).

Sei  $G_\cup$  die CFG definiert durch  $G_\cup = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}, S)$ . Offensichtlich gilt  $L(G_\cup) = L(G_1) \cup L(G_2) = L_1 \cup L_2$ . Daher sind die kontextfreien Sprachen abgeschlossen bezüglich Vereinigung.

Sei  $G_\circ$  die CFG definiert durch  $G_\circ = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}, S)$ . Offensichtlich gilt  $L(G_\circ) = L(G_1)L(G_2) = L_1 L_2$ . Daher sind die kontextfreien Sprachen abgeschlossen bezüglich Produktbildung.

Sei  $G = (V, \Sigma, P, S)$ , wobei wir annehmen, dass  $S$  auf keiner rechten Seite in  $P$  vorkommt (ist dies der Fall, dann transformiere  $G$  durch Hinzufügen eines neuen Startsymbols  $S_0$  und einer Produktion  $S_0 \rightarrow S$ ). Sei  $S' \notin V$  und sei  $G_* = (V \cup \{S'\}, \Sigma, (P \setminus \{S \rightarrow \varepsilon\}) \cup \{S' \rightarrow \varepsilon, S' \rightarrow S, S \rightarrow SS'\}, S')$ . Dann gilt  $L(G_*) = L(G)^*$ . Das zeigt, dass kontextfreie Sprachen bezüglich dem Kleeneschen Stern-Operator (der reflexiv-transitiven Hülle) abgeschlossen sind.

Die Sprachen  $L_1 = \{a^n b^m c^m \mid m, n \in \mathbb{N}_0\}$  und  $L_2 = \{a^m b^m c^n \mid m, n \in \mathbb{N}_0\}$  sind beide kontextfrei, da sie von den CFGs  $G_1 = (\{A, D, S\}, \{a, b, c\}, \{S \rightarrow AD, A \rightarrow \varepsilon \mid aA, D \rightarrow bDc \mid \varepsilon\}, S)$  und  $G_2 = (\{C, D, S\}, \{a, b, c\}, \{S \rightarrow DC, C \rightarrow cC \mid \varepsilon, D \rightarrow aDb \mid \varepsilon\}, S)$  erzeugt werden. Allerdings ist  $L_1 \cap L_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$  nicht kontextfrei, was in Satz 4.2.3 gezeigt wurde. Daher sind die kontextfreien Sprachen nicht abgeschlossen bezüglich Schnittbildung.

Nehme an, die kontextfreien Sprachen seien abgeschlossen bez. Komplementbildung. Seien  $L_1, L_2$  CFLs. Dann sind  $\overline{L_1}$  und  $\overline{L_2}$  CFLs und ebenso  $\overline{L_1} \cup \overline{L_2}$  kontextfrei. Schließlich folgt auch, dass  $\overline{\overline{L_1} \cup \overline{L_2}}$  kontextfrei ist. Da  $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$  folgt, dass die kontextfreien Sprachen abgeschlossen bez. Schnittbildung sind, was ein Widerspruch ist. D. h. unsere Annahme war falsch, die kontextfreien Sprachen sind nicht abgeschlossen bez. Komplementbildung.  $\square$

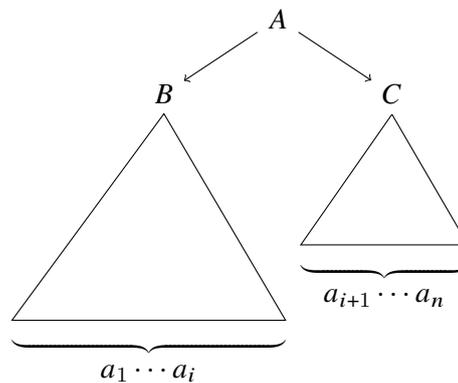
#### 4.4. Effiziente Lösung des Wortproblems: Der CYK-Algorithmus

Wir haben bereits einen Algorithmus behandelt, der das Wortproblem für Typ 1-, 2- und 3-Sprachen löst. Dieser war jedoch nicht effizient (sondern hat exponentielle Laufzeit in der Größe der Grammatik). In diesem Abschnitt lernen wir den Algorithmus von Cocke, Younger und Kasami (kurz CYK-Algorithmus) kennen (Coc69; You67; Kas65), der das Wortproblem für kontextfreie Sprachen in polynomieller Zeit löst.

Der CYK-Algorithmus arbeitet mit einer CFG in Chomsky-Normalform. Die Idee des Algorithmus kann zunächst rekursiv beschrieben werden, indem der Test beschrieben wird, ob eine Variable ein bestimmtes Wort erzeugt. Schließlich genügt es für eine CFG  $G = (V, \Sigma, P, S)$  und ein Wort  $w$  zu prüfen, ob das Startsymbol  $S$  das Wort  $w$  erzeugt. Die rekursive Beschreibung des Tests ist:

Prüfe, ob Variable  $A$  das Wort  $w = a_1 \cdots a_n$  erzeugt:

- Wenn  $w = a \in \Sigma$ , dann prüfe ob  $A \rightarrow a \in P$
- Anderenfalls ( $|w| > 1$ ) kann  $A$  nur dann das Wort  $w$  erzeugen, wenn es eine Produktion  $A \rightarrow BC \in P$  und einen Index  $1 \leq i < n$  gibt, sodass  $B$  das Wort  $a_1 \cdots a_i$  und  $C$  das Wort  $a_{i+1} \cdots a_n$  erzeugt:



Daher prüfe für alle Produktionen  $A \rightarrow BC \in P$  und alle  $i$  mit  $1 \leq i < n$  rekursiv, ob  $B$  das Wort  $w = a_1 \cdots a_i$  und  $C$  das Wort  $a_{i+1} \cdots a_n$  erzeugt.

Dieses so beschriebene rekursive Verfahren wird im CYK-Algorithmus mit dynamischer Programmierung und nicht-rekursiv umgesetzt, um einen effizienten Algorithmus zu erhalten. Daher berechnet der Algorithmus Mengen  $V(i, j) \subseteq V$ , sodass  $V(i, j)$  die Variablen enthält, die das Wort  $a_i \cdots a_{i+j-1}$  erzeugen. Die Berechnung der Mengen geschieht mit dynamischer Programmierung, indem man mit den einfachsten Mengen  $V(i, 1)$  anfängt (diese kann man bestimmen, indem alle  $A \in V$  einfügt, die das Symbol  $a_i$  erzeugen, also es Produktionen  $A \rightarrow a_i$  gibt) und anschließend die Länge  $j$  der Wörter wachsen lässt. Damit ist sichergestellt, dass die zur Berechnung von  $V(i, j)$ -benötigten Einträge alle bereits vorher berechnet wurden: Zur Berechnung von  $V(i, j)$  muss man Mengen  $V(i, k)$  und  $V(i+k, j-k)$  mit  $1 \leq k < j$  betrachten:

Gibt es Variablen  $B \in V(i, k)$  und  $C \in V(i+k, j-k)$  und eine Produktion  $A \rightarrow BC$ , dann gilt  $A \in V(i, j)$ , denn  $B$  erzeugt  $a_i \cdots a_{i+k-1}$ ,  $C$  erzeugt  $a_{i+k} \cdots a_{i+j-1}$  und

daher erzeugt  $A$  das Wort  $a_i \cdots a_{i+j-1}$ .

Nach Berechnen aller Mengen muss man schließlich prüfen, ob  $S \in (V_{1,n})$  gilt, um das Wortproblem zu entscheiden. Algorithmus 4 gibt den CYK-Algorithmus in Pseudo-Code an. Die Laufzeit wird durch die drei geschachtelten Für-Schleifen dominiert und kann mit  $O(n^3 \cdot |P|)$  abgeschätzt werden, wobei  $n = |w|$  und  $|P|$  die Anzahl der Produktionen der Grammatik ist.

**Theorem 4.4.1.** *Das Wortproblem für kontextfreie Sprachen kann in Polynomialzeit entschieden werden.*

---

**Algorithmus 4 : CYK-Algorithmus**


---

**Eingabe :** CFG  $G = (V, \Sigma, P, S)$  in Chomsky-Normalform und Wort

$$w = a_1 \cdots a_n \in \Sigma^*$$

**Ausgabe :** Ja, wenn  $w \in L(G)$  und Nein, wenn  $w \notin L(G)$

**Beginn**

**für**  $i = 1$  *bis*  $n$  **tue**

$V(i, 1) = \{A \in V \mid A \rightarrow a_i \in P\}$

**Ende**

**für**  $j = 2$  *bis*  $n - 1$  **tue**

**für**  $i = 1$  *bis*  $n + 1 - j$  **tue**

$V(i, j) = \emptyset;$

**für**  $k = 1$  *bis*  $j - 1$  **tue**

$V(i, j) = V(i, j) \cup \{A \in V \mid A \rightarrow BC \in P, B \in V(i, k), C \in$

$V(i + k, j - k)\}$

**Ende**

**Ende**

**Ende**

**wenn**  $S \in V(1, n)$  **dann**

    return Ja

**sonst**

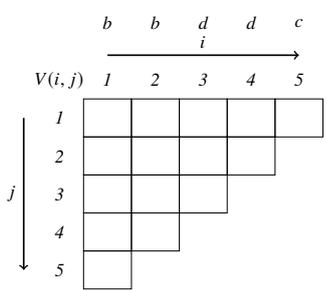
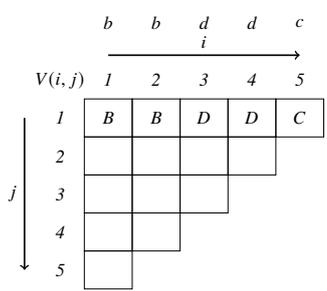
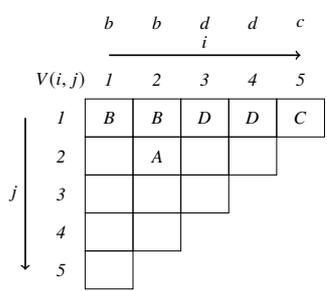
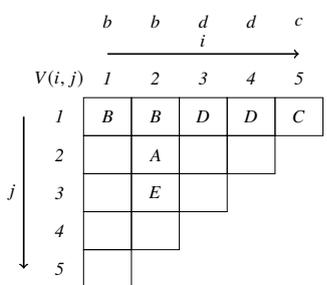
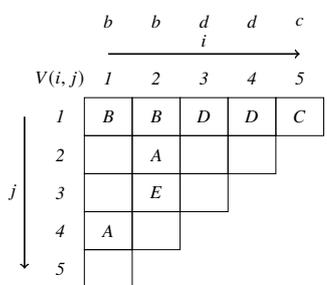
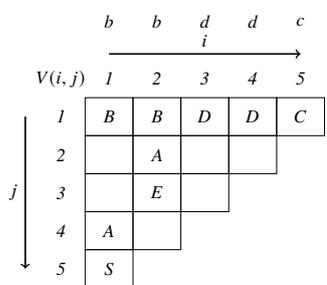
    return Nein

**Ende**

**Ende**

---

**Beispiel 4.4.2.** Sei  $G = (\{S, A, B\}, \{b, c, d\}, P, S)$  mit  $P = \{S \rightarrow AC, A \rightarrow BE, A \rightarrow BD, E \rightarrow AD, C \rightarrow c, B \rightarrow b, D \rightarrow d\}$  und  $w = bbddc$ . Wir führen den CYK-Algorithmus aus, um zu prüfen, ob  $w \in L(G)$  gilt:

<p>Am Anfang ist die Tabelle <math>V(i, j)</math> leer:</p> 	<p>Nach Abarbeiten der ersten Schleife sind alle <math>V(i, 1)</math>-Einträge gefüllt:</p> 	<p>Für <math>j = 2</math> wird <math>A \in V(2, 2)</math> eingefügt, da <math>A \rightarrow BD \in P</math>, <math>B \in V(2, 1)</math> und <math>D \in V(3, 1)</math>:</p> 
<p>Für <math>j = 3</math> wird <math>E \in V(2, 3)</math> eingefügt, da <math>E \rightarrow AD \in P</math>, <math>A \in V(2, 2)</math> und <math>D \in V(4, 1)</math>:</p> 	<p>Für <math>j = 4</math> wird <math>A \in V(1, 4)</math> eingefügt, da <math>A \rightarrow BE \in P</math>, <math>B \in V(1, 1)</math> und <math>E \in V(2, 3)</math>:</p> 	<p>Für <math>j = 5</math> wird <math>S \in V(1, 5)</math> eingefügt, da <math>S \rightarrow AC \in P</math>, <math>A \in V(1, 4)</math> und <math>C \in V(5, 1)</math>:</p> 

Da  $S \in V(1, 5)$  gibt der Algorithmus Ja aus.

Wenn man sich in der Tabelle  $V(i, j)$  nicht nur die Variablen, sondern ebenfalls die dazugehörigen Produktionen merkt, kann man im Erfolgsfall auch recht einfach eine Ableitung konstruieren, die  $w$  erzeugt. In Beispiel 4.4.2 ist die zugehörige Linksableitung  $S \Rightarrow AC \Rightarrow BEC \Rightarrow bEC \Rightarrow bADC \Rightarrow bBDDC \Rightarrow bbDDC \Rightarrow bbdDC \Rightarrow bbddC \Rightarrow bbddc$ .

### 4.5. Kellerautomaten

Endliche Automaten haben als wesentliche Beschränkung, dass sie nahezu keinen Speicher besitzen. Die einzige Speichermöglichkeit sind die Zustände selbst und dies sind nur endlich viele. Daher kann ein endlicher Automat z. B. eine Sprache der Form  $\{w\$w \mid w \in \{a, b, c\}^*\}$  nicht erkennen, da er nach dem Lesen von  $\$$  gespeichert haben müsste, welche Zeichen er vorher gelesen hat (um sie mit den kommenden Zeichen des Wortes  $w$  zu vergleichen).

Kellerautomaten erweitern das Automatenmodell um einen Speicher. Dieser ist sogar beliebig groß, aber er ist ein spezieller Speicher, nämlich ein *Keller* (Stack, LIFO-Speicher, last-in-first-out Speicher) auf den Zeichen eines Kelleralphabets von oben abgelegt und von oben gelesen

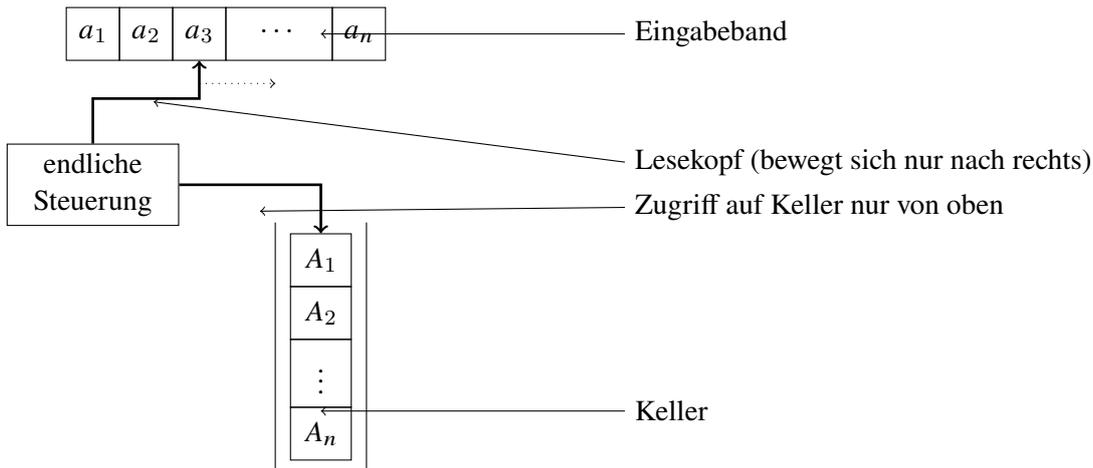


Abbildung 4.1.: Illustration des Kellerautomaten

und entnommen werden können. Abb. 4.1 illustriert den Kellerautomaten. Der Zustandsübergang (eine Aktion) hängt bei endlichen Automaten nur vom nächsten Symbol der Eingabe und dem aktuellen Zustand ab und liefert (eine Menge von) Nachfolgezustände(n). Beim Kellerautomaten hängt der Zustandsübergang zusätzlich vom Kellerinhalt (dessen oberstem Zeichen) ab und er liefert neben Nachfolgezuständen auch einen dazugehörigen abgeänderten Keller.

**Definition 4.5.1** (Kellerautomat, PDA). Ein (nichtdeterministischer) Kellerautomat (PDA, pushdown automaton) ist ein Tupel  $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ , wobei

- $Z$  ist eine endliche Menge von Zuständen,
- $\Sigma$  ist das (endliche) Eingabealphabet,
- $\Gamma$  ist das (endliche) Kelleralphabet,
- $\delta : (Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \rightarrow \mathcal{P}_e(Z \times \Gamma^*)$  ist die Zustandsüberföhrungsfunktion (oder nur Überföhrungsfunktion)
- $z_0 \in Z$  ist der Startzustand und
- $\# \in \Gamma$  ist das Startsymbol im Keller.

Wenn  $(z', B_1 \dots B_k) \in \delta(z, a, A)$  dann meint dies, dass der Kellerautomat vom Zustand  $z$  ausgehend, bei Eingabe von Zeichen  $a$  und oberstem Zeichen  $A$  auf dem Keller in den Zustand  $z'$  übergangen kann und dabei  $A$  durch die Zeichenfolge  $B_1 \dots B_k$  ersetzt, sodass  $B_1$  ganz oben auf dem Keller liegt. Abb. 4.2 illustriert diesen Übergang. Beachte, dass  $k = 0$  möglich ist und in diesem Fall das Symbol  $A$  vom Kellerspeicher gelöscht wird.

Die so definierten PDAs sind nichtdeterministisch und erlauben  $\varepsilon$ -Übergänge. Außerdem gibt es keine Menge von Endzuständen. Ein PDA akzeptiert, wenn die Eingabe gänzlich verarbeitet wurde und der Keller leer ist. Initial startet der Kellerautomat mit dem Symbol  $\#$  im Keller.

Befindet sich ein Kellerautomat in einer Berechnung, dann kann man Buchföhren über den

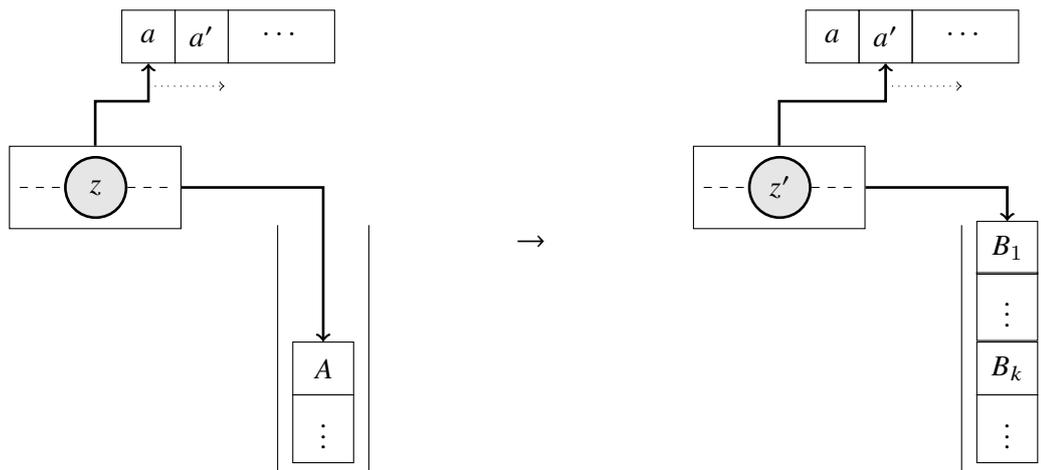


Abbildung 4.2.: Illustration der Ausführung des Schrittes  $(z', B_1 \cdots B_k) \in \delta(z, a, A)$

aktuellen Zustand, die Resteingabe und den aktuellen Kellerinhalt. Formal wird dies durch eine *Konfiguration* dargestellt:

**Definition 4.5.2** (Konfiguration eines Kellerautomaten). Sei  $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$  ein PDA. Eine Konfiguration von  $M$  ist ein Tripel  $(z, w, W)$  mit  $z \in Z$ ,  $w \in \Sigma^*$ ,  $W \in \Gamma^*$ . Die Menge aller Konfigurationen für  $M$  ist daher  $Z \times \Sigma^* \times \Gamma^*$ .

Für eine Konfiguration  $(z, w, W)$  ist  $z$  der aktuelle Zustand,  $w$  die Resteingabe und  $W$  der Inhalt des Kellers, wobei das erste Zeichen von  $W$  ganz oben im Keller steht.

Wir definieren eine binäre Relation  $\vdash$  auf Konfigurationen, die den Übergang von einer Konfiguration in eine nächste beschreibt:

**Definition 4.5.3** (Transitionsrelation für PDA-Konfigurationen). Für einen PDA  $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$  definieren wir  $\vdash_M \subseteq (Z \times \Sigma^* \times \Gamma^*) \times (Z \times \Sigma^* \times \Gamma^*)$  durch

- $(z, a_1 \cdots a_n, A_1 \cdots A_m) \vdash_M (z', a_2 \cdots a_n, WA_2 \cdots A_m)$  falls  $(z', W) \in \delta(z, a_1, A_1)$  und
- $(z, w, A_1 \cdots A_m) \vdash_M (z', w, WA_2 \cdots A_m)$  falls  $(z', W) \in \delta(z, \varepsilon, A_1)$ .

Mit  $\vdash_M^*$  bezeichnen wir (wie üblich) die reflexiv-transitive Hülle von  $\vdash_M$  und mit  $\vdash_M^i$  die  $i$ -fache Anwendung von  $\vdash_M$ . Wenn  $M$  eindeutig ist, lassen wir den Index  $M$  in  $\vdash_M$  weg und schreiben  $\vdash$ .

**Definition 4.5.4** (Akzeptierte Sprache eines PDA). Sei  $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$  ein PDA. Die durch  $M$  akzeptierte Sprache  $L(M)$  ist definiert als

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

**Beispiel 4.5.5.** Der folgende PDA  $M = (\{z_0, z_1\}, \{a, b\}, \{B, \#\}, \delta, z_0, \#)$  akzeptiert die Sprache  $\{a^i b^i \mid i \in \mathbb{N}_0\}$ , wobei wir für  $\delta$  nur die Fälle angeben, die zu nicht-leeren Mengen führen (d. h.

in allen anderen Fällen für  $i \in \{0, 1\}$ ,  $c \in \{a, b, \varepsilon\}$  und  $A \in \{B, \#\}$  ist  $\delta(z_i, c, A) = \emptyset$ :

$$\begin{array}{lll} \delta(z_0, a, \#) = \{(z_0, B\#)\} & \delta(z_0, b, B) = \{(z_1, \varepsilon)\} & \delta(z_1, \varepsilon, \#) = \{(z_1, \varepsilon)\} \\ \delta(z_0, a, B) = \{(z_0, BB)\} & \delta(z_1, b, B) = \{(z_1, \varepsilon)\} & \delta(z_0, \varepsilon, \#) = \{(z_0, \varepsilon)\} \end{array}$$

Der PDA  $M$  akzeptiert das leere Wort, denn  $(z_0, \varepsilon, \#) \vdash (z_0, \varepsilon, \varepsilon)$ . Der PDA  $M$  akzeptiert das Wort  $a^i b^i$  für  $i > 0$ , da

$$(z_0, a^i b^i, \#) \vdash (z_0, a^{i-1} b^i, B\#) \vdash^* (z_0, b^i, B^i \#) \vdash (z_1, b^{i-1}, B^{i-1} \#) \vdash^* (z_1, \varepsilon, \#) \vdash (z_1, \varepsilon, \varepsilon).$$

Andere Wörter  $w$  werden nicht von  $M$  akzeptiert, da einerseits jedes Verarbeiten eines Zeichen  $a$  ein Kellerzeichen  $B$  auf den Keller legt, das selbst wieder nur durch Verarbeiten eines  $b$ 's abgebaut werden kann (d. h.  $\#_a(w) = \#_b(w)$  muss gelten), und andererseits das Wort mit  $a$  beginnen muss und im Zustand  $z_0$  nur weitere  $a$ 's verarbeitet werden können (da ein  $b$  im Zustand  $z_0$  nicht verarbeitet werden kann) und von der Form  $a^* b^*$  sein muss, da nach einem Wechsel von  $z_0$  nach  $z_1$  nur noch  $b$ 's gelesen werden können.

**Beispiel 4.5.6.** Sei  $M = (\{z_0, z_1\}, \{a, b\}, \{A, B, \#\}, \delta, z_0, \#)$  mit

$$\begin{array}{ll} \delta(z_0, a, \#) = \{(z_0, A\#), (z_1, \#)\} & \delta(z_0, \varepsilon, A) = \{(z_1, A)\} \\ \delta(z_0, b, \#) = \{(z_0, B\#), (z_1, \#)\} & \delta(z_0, \varepsilon, B) = \{(z_1, B)\} \\ \delta(z_0, a, A) = \{(z_0, AA), (z_1, A)\} & \delta(z_0, \varepsilon, \#) = \{(z_1, \#)\} \\ \delta(z_0, b, A) = \{(z_0, BA), (z_1, A)\} & \delta(z_1, a, A) = \{(z_1, \varepsilon)\} \\ \delta(z_0, a, B) = \{(z_0, AB), (z_1, B)\} & \delta(z_1, b, B) = \{(z_1, \varepsilon)\} \\ \delta(z_0, b, B) = \{(z_0, BB), (z_1, B)\} & \delta(z_1, \varepsilon, \#) = \{(z_1, \varepsilon)\} \end{array}$$

und  $\delta(z_i, c, C) = \emptyset$  für alle anderen Fälle mit  $i \in \{0, 1\}$ ,  $c \in \{a, b, \varepsilon\}$  und  $C \in \{A, B, \#\}$ .

Der PDA akzeptiert die Sprache  $L = \{w \in \{a, b\}^* \mid w \text{ ist Palindrom}\}$ . Da  $w$  ein Palindrom sein muss, gibt es im Wesentlichen vier Möglichkeiten  $w = \varepsilon$ ,  $w = u\bar{u}$ ,  $w = ua\bar{u}$ , oder  $w = ub\bar{u}$  (mit  $u \in \{a, b\}^*$ ). Die Kellerzeichen  $A$  und  $B$  werden verwendet, um die gelesenen Zeichen von  $u$  zu speichern (im Zustand  $z_0$ ) und sie anschließend in umgekehrter Reihenfolge zum Verarbeiten von  $\bar{u}$  im Zustand  $z_1$  zu entfernen. Die Übergänge von  $z_0$  zu  $z_1$  sind sowohl ohne Lesen eines Zeichens oder mit Lesen genau eines Zeichens möglich, was genau den Unterschied zwischen den Fällen  $u\bar{u}$  und  $uc\bar{u}$  mit  $c \in \{a, b\}$  ausmacht. Welcher Fall nun eingetreten ist, wird dem Nichtdeterminismus des Automaten überlassen, der sich die richtige Position und die richtige Art und Weise vom Übergang von  $z_0$  zu  $z_1$  aussuchen muss. Tatsächlich ist dieser Nichtdeterminismus notwendig: Es gibt keinen deterministischen Kellerautomaten, der die Sprache  $L$  akzeptiert.

Z. B. gibt es für Eingabe  $abbba$  die folgende Transitionsfolge, bei der der richtige Zeitpunkt zum Übergang von  $z_0$  zu  $z_1$  verpasst wird:

$$\begin{aligned} (z_0, abbba, \#) \vdash (z_0, bbba, A\#) \vdash (z_0, bba, BA\#) \vdash (z_0, ba, BBA\#) \\ \vdash (z_0, a, BBBA\#) \vdash (z_0, \varepsilon, ABBBA\#) \vdash (z_1, \varepsilon, ABBBA\#) \end{aligned}$$

Ein erfolgreicher Lauf mit Eingabe  $abbba$  ist:

$$(z_0, abbba, \#) \vdash (z_0, bbba, A\#) \vdash (z_0, bba, BA\#) \vdash (z_1, ba, BA\#) \\ \vdash (z_1, a, A\#) \vdash (z_1, \varepsilon, \#) \vdash (z_1, \varepsilon, \varepsilon)$$

#### 4.5.1. Akzeptanz durch Endzustände

Eine alternative Definition der PDAs verwendet als zusätzliche Komponente Endzustände und definiert den Akzeptanzbegriff entsprechend anders. Wir werden in diesem Abschnitt zeigen, dass diese Definition äquivalent zur bisherigen Definition ohne Endzustände und mit Akzeptanz durch leeren Keller ist, indem wir zeigen, dass beide Formalismen ineinander überführbar sind.

**Definition 4.5.7** (PDA mit Endzuständen). *Ein (nichtdeterministischer) Kellerautomat mit Endzuständen (PDA mit Endzuständen) ist ein Tupel  $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$  wobei*

- $Z$  ist eine endliche Menge von Zuständen,
- $\Sigma$  ist das (endliche) Eingabealphabet,
- $\Gamma$  ist das (endliche) Kelleralphabet
- $\delta : Z \times ((\Sigma \cup \{\varepsilon\}) \times \Gamma) \rightarrow \mathcal{P}_e(Z \times \Gamma^*)$  ist die Zustandsüberföhrungsfunktion (oder Überföhrungsfunktion)
- $z_0 \in Z$  ist der Startzustand,
- $\# \in \Gamma$  ist das Startsymbol im Keller und
- $E \subseteq Z$  ist die Menge der Endzustände.

Ein PDA mit Endzuständen akzeptiert die Sprache

$$L(M) = \{w \in \Sigma^* \mid (z_0, w, \#) \vdash^* (z, \varepsilon, W) \text{ und } z \in E\}.$$

Beachte, dass ein PDA mit Endzuständen auch bei gefülltem Keller akzeptieren kann.

**Lemma 4.5.8.** *Für jeden Kellerautomat mit Endzuständen  $M$  kann ein Kellerautomat  $M'$  (ohne Endzustände) konstruiert werden, so dass gilt  $L(M) = L(M')$ .*

*Beweis.* Sei  $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$  ein Kellerautomat mit Endzuständen. Für die Transformation in einen äquivalenten PDA  $M'$  ohne Endzustände müssen zwei Probleme gelöst werden:

- Der PDA  $M'$  darf nicht in eine Konfiguration mit leerem Stack, leerer Eingabe und einem Zustand  $z \notin E$  laufen.
- Wenn  $M'$  in einem Zustand  $z \in E$  mit leerer Eingabe ist, muss  $M'$  in die Lage versetzt werden, den Keller zu leeren.

Beide Probleme werden dadurch gelöst, dass  $M'$  ein neues Kellersymbol erhält, welches unter alle Symbole in den Keller gelegt wird (mithilfe eines neuen Startzustands) und es einen weiteren neuen Zustand gibt, der zum Leeren des Kellers benutzt wird. Sei  $M' = (Z \cup \{z'_0, z_E\}, \Sigma, \Gamma \cup \{\$\}, \delta', z'_0, \$)$  wobei  $\delta'$  genau die Einträge enthält, die durch die folgenden Regeln definiert werden:

1.  $\delta'(z'_0, \varepsilon, \$) = \{(z_0, \#\$\}$  und  $\delta'(z'_0, a, X) = \emptyset$  für alle  $a \in \Sigma \cup \{\varepsilon\}$ ,  $X \in \Gamma \cup \{\$\}$  sonst.
2.  $\delta(z, a, A) \subseteq \delta'(z, a, A)$  für alle  $z \in Z$ ,  $a \in \{\Sigma \cup \varepsilon\}$  und  $A \in \Gamma$ .
3.  $(z_E, A) \in \delta'(z, \varepsilon, A)$  für alle  $z \in E$ ,  $A \in \Gamma \cup \{\$\}$ .
4.  $(z_E, \varepsilon) \in \delta'(z_E, \varepsilon, A)$  für alle  $A \in \Gamma \cup \{\$\}$ .

Regel (1) sorgt dafür, dass vom neuen Startzustand  $z'_0$  in den alten Startzustand  $z_0$  übergangen wird, aber das Zeichen  $\$$  ganz unten im Keller liegt. Regel (2) besagt, dass die alten Regeln von  $M$  auch in  $M'$  gelten. Regel (3) sorgt dafür, dass bei Akzeptanz von  $M$  in den neuen Zustand  $z_E$  gewechselt wird. Regel (4) sorgt dafür, dass der Keller im Zustand  $z_E$  komplett geleert wird, damit  $M'$  akzeptieren kann. Da der Keller nur im Zustand  $z_E$  leer werden kann, gibt es keine neuen Wörter für die  $M'$  akzeptieren kann.

Zusammengefasst zeigt dies die korrekte Konstruktion von  $M'$ . □

**Lemma 4.5.9.** *Für jeden Kellerautomat  $M$  kann ein Kellerautomat mit Endzuständen  $M'$  konstruiert werden, so dass  $L(M) = L(M')$  gilt.*

*Beweis.* Sei  $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$  ein Kellerautomat (ohne Endzustände). Für die Transformationen in einen äquivalenten PDA  $M'$  mit Endzuständen fügen wir einen Endzustand ein und sorgen mithilfe eines zusätzlichen Kellersymbols dafür, dass  $M'$  nach Abarbeitung des Wortes noch in den Endzustand wechseln kann. Sei  $M' = (Z \cup \{z'_0, z_E\}, \Sigma, \Gamma \cup \{\$\}, \delta', z'_0, \$, \{z_E\})$ , so dass  $\delta'$  genau die Einträge enthält, die durch die folgenden Regeln definiert werden:

1.  $\delta'(z'_0, \varepsilon, \$) = \{(z_0, \#\$\}$  und  $\delta'(z'_0, a, X) = \emptyset$  für alle  $a \in \Sigma \cup \{\varepsilon\}$ ,  $X \in \Gamma \cup \{\$\}$  sonst.
2.  $\delta(z, a, A) \subseteq \delta'(z, a, A)$  für alle  $z \in Z$ ,  $a \in \{\Sigma \cup \varepsilon\}$  und  $A \in \Gamma$
3.  $(z_E, \varepsilon) \in \delta'(z, \varepsilon, \$)$  für alle  $z \in Z$

Regel (1) sorgt dafür, dass am Anfang das Symbol  $\$$  ganz unten im Keller liegt. Es verbleibt dort, bis es durch Regel (3) entfernt wird (in diesem Fall hat  $M$  schon akzeptiert und  $M'$  wechselt in den Endzustand). Regel (2) sorgt dafür, dass alle Übergänge aus  $M$  auch in  $M'$  verwendet werden können.

Zusammengefasst zeigt dies die korrekte Konstruktion von  $M'$ . □

Lemmata 4.5.8 und 4.5.9 zeigen:

**Satz 4.5.10.** *PDA's mit Endzuständen und PDA's ohne Endzustände (mit Akzeptanz durch leeren Keller) sind äquivalente Formalismen.*

## 4.5.2. Äquivalenz von Kellerautomaten und kontextfreien Sprachen

Im Folgenden werden wir beweisen, dass Kellerautomaten genau die kontextfreien Sprachen erkennen. Wir beweisen dies in zwei Teilen. Zunächst zeigen wir, wie man einen Kellerautomaten für eine CFG (in Greibach-Normalform) konstruiert, der die von der Grammatik erzeugte Sprache akzeptiert. Die Idee ist, dass der Kellerautomat, eine Linksableitung  $S \Rightarrow^* w$  simuliert. Da

die CFG in Greibach-Normalform ist, ist die Linksableitung nach  $i$  Schritten von der Form  $S \Rightarrow^i a_1 \cdots a_i B_1 \cdots B_j$ . Für die Simulation auf dem PDA, startet dieser mit  $w$  als Eingabe und  $S$  auf dem Keller. Nach  $i$ -Schritten werden auf dem Keller nur die Variablen  $B_1 \cdots B_j$  gespeichert, während  $a_1 \cdots a_i$  (als Präfix von  $w$ ) von der Eingabe gelesen und verarbeitet wurde.

#### 4.5.2.1. Kontextfreie Sprachen werden durch Kellerautomaten erkannt

**Satz 4.5.11.** *Jede kontextfreie Sprache wird durch einen Kellerautomaten erkannt.*

*Beweis.* Sei  $L$  eine kontextfreie Sprache. Da  $L$  kontextfrei ist, gibt es eine CFG  $G = (V, \Sigma, P, S)$  in Greibach-Normalform mit  $L(G) = L \setminus \{\varepsilon\}$ . Sei  $M = (\{z_0\}, \Sigma, V, \delta, z_0, S)$  ein PDA, sodass

$$\delta(z_0, a, A) := \{(z_0, B_1 \cdots B_n) \mid (A \rightarrow aB_1 \cdots B_n) \in P\}$$

und falls  $\varepsilon \in L$  setze zusätzlich

$$\delta(z_0, \varepsilon, S) := \{(z_0, \varepsilon)\}$$

In allen anderen Fällen sei  $\delta(z_0, \varepsilon, A) = \emptyset$ .

Wir zeigen, dass  $L(M) = L$  gilt. Zunächst behandeln wir den Spezialfall, dass das leere Wort in  $L$  liegt: Es gilt  $\varepsilon \in L$  genau dann, wenn  $(z_0, \varepsilon, S) \vdash (z_0, \varepsilon, \varepsilon)$  und damit  $\varepsilon \in L(M)$ .

Für die weiteren Fälle zeigen wir, dass für alle  $i \in \mathbb{N}_0$  gilt:

$$\begin{aligned} S \Rightarrow_G^i a_1 \cdots a_i B_1 \cdots B_m \text{ mit einer Linksableitung} \\ \text{genau dann, wenn} \\ (z_0, a_1 \cdots a_i w, S) \vdash^i (z_0, w, B_1 \cdots B_m) \text{ für alle } w \in \Sigma^*. \end{aligned}$$

Daraus folgt dann offensichtlich  $u \in L \iff u \in L(M)$  für alle  $u \in \Sigma^*$ . Wir verwenden Induktion über  $i$  zum Nachweis der Aussage. Für  $i = 0$  gilt die Aussage. Für  $i > 0$  sei zunächst  $S \Rightarrow_G^i a_1 \cdots a_i B_1 \cdots B_m$  eine Linksableitung. Da  $G$  in Greibach-Normalform ist, kann diese geschrieben werden als  $S \Rightarrow_G^{i-1} a_1 \cdots a_{i-1} B_x B_{j+1} \cdots B_m \Rightarrow_G a_1 \cdots a_i B_1 \cdots B_m$ , wobei  $B_x \rightarrow a_i B_1 \cdots B_j \in P$  als letzte Produktion angewendet wurde. Die Induktionsannahme zeigt, dass  $S \Rightarrow_G^{i-1} a_1 \cdots a_{i-1} B_x B_{j+1} \cdots B_m$  genau dann, wenn  $(z_0, a_1 \cdots a_{i-1} w, S) \vdash^{i-1} (z_0, w, B_x B_{j+1} \cdots B_k)$ . Mit  $w = a_i w'$  und da  $(z_0, B_1 \cdots B_j) \in \delta(z_0, a_i, B_x)$ , gilt ebenfalls  $(z_0, a_1 \cdots a_i w', S) \vdash^i (z_0, w', B_1 \cdots B_k)$  für alle  $w'$ .

Nun betrachte die Rückrichtung. Sei  $(z_0, a_1 \cdots a_i w, S) \vdash^i (z_0, w, B_1 \cdots B_k)$ . Dann muss der letzte Schritt  $a_i$  gelesen haben, d.h. die Folge lässt sich zerlegen in  $(z_0, a_1 \cdots a_i w, S) \vdash^{i-1} (z_0, a_i w, B_x B_{j+1} \cdots B_k) \vdash (z_0, w, B_1 \cdots B_k)$ , wobei  $(z_0, B_1 \cdots B_j) \in \delta(z_0, a_i, B_x)$ . Dann muss  $B_x \rightarrow a_i B_1 \cdots B_j$  eine Produktion in  $P$  sein. Die Induktionsannahme liefert  $S \Rightarrow_G^{i-1} a_1 \cdots a_{i-1} B_x B_{j+1} \cdots B_k$  und wir können obige Produktion anwenden und erhalten  $S \Rightarrow_G^i a_1 \cdots a_i B_1 \cdots B_k$ .  $\square$

4.5.2.2. Kellerautomaten akzeptieren kontextfreie Sprachen

Bevor wir die Rückrichtung zeigen (jeder PDA akzeptiert eine kontextfreie Sprache), beweisen wir einen Hilfssatz, der zeigt, dass es ausreicht, wenn ein PDA pro Schritt maximal zwei Symbole auf den Keller legen darf.

**Lemma 4.5.12.** Für jeden PDA  $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$  gibt es einen PDA  $M' = (Z, \Sigma, \Gamma', \delta', z_0, \#)$  mit  $L(M) = L(M')$ , sodass gilt: Wenn  $(z', B_1 \cdots B_k) \in \delta'(z, a, A)$  (für  $a \in (\Sigma \cup \{\varepsilon\})$ ), dann ist  $k \leq 2$ .

*Beweis.* Transformiere  $M$  in  $M'$ , indem  $\delta'$  und  $\Gamma'$  konstruiert werden, sodass für alle  $A \in \Gamma$  und  $a \in (\Sigma \cup \{\varepsilon\})$  genau die folgenden Regeln beachtet werden:

- $(z', B_1 \cdots B_k) \in \delta'(z, a, A)$ , wenn  $(z', B_1 \cdots B_k) \in \delta(z, a, A)$  und  $k \leq 2$ .
- falls  $(z', B_1 \cdots B_k) \in \delta(z, a, A)$  mit  $k > 2$ , dann
  - sei  $(z, C_k B_k) \in \delta(z, a, A)$ , und
  - für alle  $i$  mit  $4 \leq i \leq k$ :  $\delta(z, \varepsilon, C_i) = \{(z, C_{i-1} B_{i-1})\}$ , sowie
  - $\delta(z, \varepsilon, C_3) = \{(z', B_1 B_2)\}$

wobei  $C_3, \dots, C_k \in \Gamma'$  neue Kellersymbole sind (diese werden jeweils neu erzeugt pro ersetztem Eintrag).

Es gilt  $(z, aw, AW) \vdash_{M'}^* (z', w, B_1 \cdots B_k W) \iff (z, aw, AW) \vdash_M (z', w, B_1 \cdots B_k W)$  und da die neuen Übergänge nicht zum sofortigen Akzeptieren führen können, folgt, dass die Äquivalenz  $(z, w, \#) \vdash_M^* (z, \varepsilon, \varepsilon) \iff (z, w, \#) \vdash_{M'}^* (z, \varepsilon, \varepsilon)$  gilt.  $\square$

**Bemerkung 4.5.13.** Das letzte Lemma kann auch bewiesen werden, indem statt zusätzlicher Kellersymbole zusätzliche Zustände verwendet werden.

Nun zeigen wir, dass wir für jeden Kellerautomaten, der die Bedingung aus dem vorherigen Lemma erfüllt, eine kontextfreie Grammatik angeben können, welche die vom Automaten akzeptierte Sprache erzeugt. Die wesentliche Idee des Beweises ist es, für die Variablen der Grammatik eine sogenannte Tripelkonstruktion zu verwenden: Die Variablen sind Tripel  $\langle z', A, z \rangle$ , wobei  $z, z'$  Zustände des PDA und  $A$  ein Kellersymbol ist. Die Grammatik soll für eine solche Variable genau die Wörter  $w$  erzeugen, die den Automaten vom Zustand  $z'$  mit Kellerinhalt  $A$  in den Zustand  $z$  durch Lesen von  $w$  überführen.

**Satz 4.5.14.** Kellerautomaten akzeptieren kontextfreie Sprachen.

*Beweis.* Sei  $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$  ein PDA, sodass für alle  $(z', B_1 \cdots B_k) \in \delta(z, a, A)$  (für  $a \in (\Sigma \cup \{\varepsilon\})$ )  $k \leq 2$  gilt (was mit Lemma 4.5.12 keine Einschränkung ist).

Sei  $S$  ein neues Symbol. Wir konstruieren die Grammatik  $G = (V, \Sigma, P, S)$  mit

$$\begin{aligned} V &= \{S\} \cup \{\langle z_i, A, z_j \rangle \mid z_i, z_j \in Z, A \in \Gamma\} \\ P &= \{S \rightarrow \langle z_0, \#, z \rangle \mid z \in Z\} \\ &\cup \{\langle z', A, z \rangle \rightarrow a \mid (z, \varepsilon) \in \delta(z', a, A), a \in \Sigma \cup \{\varepsilon\}, A \in \Gamma\} \\ &\cup \{\langle z', A, z \rangle \rightarrow a \langle z'', B, z \rangle \mid (z'', B) \in \delta(z', a, A), z \in Z, a \in \Sigma \cup \{\varepsilon\}, A \in \Gamma\} \\ &\cup \{\langle z', A, z \rangle \rightarrow a \langle z'', B, z_1 \rangle \langle z_1, C, z \rangle \mid (z'', BC) \in \delta(z', a, A), z, z_1 \in Z, a \in \Sigma \cup \{\varepsilon\}, A \in \Gamma\} \end{aligned}$$

Wir beweisen im Folgenden die Aussage

$$\langle z', A, z \rangle \Rightarrow_G^* w \quad \text{genau dann, wenn} \quad (z', w, A) \vdash_M^* (z, \varepsilon, \varepsilon).$$

Da  $S \rightarrow \langle z_0, A, z \rangle \in P$  folgt daraus  $w \in L(G) \iff w \in L(M)$ , d. h.  $L(G) = L(M)$ .

„ $\Rightarrow$ “: Sei  $\langle z', A, z \rangle \Rightarrow_G^i w$  eine Linksableitung. Wir verwenden Induktion über  $i$ . Für  $i = 1$  gilt  $\langle z', A, z \rangle \Rightarrow_G w$ , was nur gelten kann, wenn die verwendete Produktion  $\langle z', A, z \rangle \rightarrow a$  ist. Dann gilt aber  $(z, \varepsilon) \in \delta(z', a, A)$  und damit auch  $(z', a, A) \vdash (z, \varepsilon, \varepsilon)$ . Für den Induktionsschritt sei  $\langle z', A, z \rangle \Rightarrow_G u \Rightarrow_G^{i-1} w$  mit  $i - 1 > 0$

- Wenn  $u = a \in (\Sigma \cup \{\varepsilon\})$ , dann kann  $i - 1 > 0$  nicht gelten.
- Wenn  $u = a \langle z'', B, z \rangle$ , dann ist  $(z'', B) \in \delta(z', a, A)$  und  $u = a \langle z'', B, z \rangle \Rightarrow^{i-1} aw'$  mit  $w = aw'$ . Dann gilt  $\langle z'', B, z \rangle \Rightarrow^{i-1} w'$  und die Induktionsannahme liefert  $(z'', w', B) \vdash_M^* (z, \varepsilon, \varepsilon)$ . Mit  $(z'', B) \in \delta(z', a, A)$  zeigt dies  $(z', w, A) = (z', aw', A) \vdash_M (z'', w', B) \vdash_M^* (z, \varepsilon, \varepsilon)$ .
- Wenn  $u = a \langle z'', B, z_1 \rangle \langle z_1, C, z \rangle$ , dann ist  $(z'', BC) \in \delta(z', a, A)$  und  $u = a \langle z'', B, z_1 \rangle \langle z_1, C, z \rangle \Rightarrow^{i-1} aw'$  mit  $w = aw'$ . Dann gilt auch  $\langle z'', B, z_1 \rangle \langle z_1, C, z \rangle \Rightarrow^{i-1} w'$  und es gibt Linksableitungen  $\langle z'', B, z_1 \rangle \Rightarrow^j w'_0$  und  $\langle z_1, C, z \rangle \Rightarrow^k w'_1$  mit  $j + k \leq i - 1$  und  $w' = w'_0 w'_1$ . Für beide können wir die Induktionsannahme anwenden und erhalten  $(z'', w'_0, B) \vdash_M^* (z_1, \varepsilon, \varepsilon)$  und  $(z_1, w'_1, C) \vdash_M^* (z, \varepsilon, \varepsilon)$ . Die erste Konfigurationsfolge kann abgeändert werden: Wenn wir  $C$  unten im Keller einfügen und  $w'_1$  an das Wort  $w'_0$  anhängen, ist der Ablauf immer noch möglich und endet mit dem Symbol  $C$  im Keller und dem Restwort  $w'_1$  d. h. es existiert  $(z'', w', BC) \vdash_M^* (z_1, w'_1, C)$ . Anhängen der anderen Folge von Konfigurationen liefert  $(z'', w', BC) = (z'', w'_0 w'_1, BC) \vdash_M^* (z, \varepsilon, \varepsilon)$ . Da  $(z'', BC) \in \delta(z', a, A)$ , zeigt dies  $(z', w, A) = (z', aw', A) \vdash_M (z'', w', BC) \vdash_M^* (z, \varepsilon, \varepsilon)$ .

„ $\Leftarrow$ “: Sei  $(z', w, A) \vdash_M^i (z, \varepsilon, \varepsilon)$ . Wir zeigen  $\langle z', A, z \rangle \Rightarrow_G^* w$  mit Induktion über  $i$ . Für  $i = 1$  muss gelten  $(z, \varepsilon) \in \delta(z', w, A)$ , was nur gelten kann für  $w = a \in (\Sigma \cup \{\varepsilon\})$  und damit gibt es die Produktion  $\langle z', A, z \rangle \rightarrow a \in P$  und daher  $\langle z', A, z \rangle \Rightarrow_G a$ . Sei nun  $i > 1$ . Dann  $(z', aw', A) \vdash (z'', w', \alpha) \vdash_M^{i-1} (z, \varepsilon, \varepsilon)$  für  $i - 1 > 0$ ,  $a \in \Sigma \cup \{\varepsilon\}$  und  $\alpha = \varepsilon$ ,  $\alpha = B$  oder  $\alpha = BC$ . Wir betrachten alle drei Fälle für  $\alpha$  einzeln:

- $\alpha = \varepsilon$ : Dieser Fall ist nicht möglich, da  $i - 1 > 0$  nicht gelten kann.
- $\alpha = B$ : Da  $(z'', w', B) \vdash_M^{i-1} (z, \varepsilon, \varepsilon)$ , liefert die Induktionsannahme  $\langle z'', B, z \rangle \Rightarrow_G^* w'$  und es gilt  $\langle z', A, z \rangle \rightarrow a \langle z'', B, z \rangle \in P$  daher gilt  $\langle z', A, z \rangle \Rightarrow_G^* aw' = w$ .

- $\alpha = BC$ . Die Konfigurationsfolge  $(z'', w', BC) \vdash_M^{i-1} (z, \varepsilon, \varepsilon)$  kann zerlegt werden in die Abarbeitung von  $B$  und die anschließende Abarbeitung von  $C$ , d. h.  $w' = w'_1 w'_2$ , sodass  $(z'', w'_1 w'_2, BC) \vdash_M^j (z_1, w'_2, C) \vdash_M^k (z, \varepsilon, \varepsilon)$  mit  $j+k = i-1$ . Aus dem ersten Teil folgt  $(z'', w'_1, B) \vdash_M^j (z_1, \varepsilon, \varepsilon)$ , indem man  $C$  auf dem Keller und den Suffix  $w'_2$  weg lässt. Da  $j < i$  und  $k < i$ , kann die Induktionsannahme auf  $(z'', w'_1, B) \vdash_M^j (z_1, \varepsilon, \varepsilon)$  und  $(z_1, w'_2, C) \vdash_M^k (z, \varepsilon, \varepsilon)$  einzeln angewendet werden und liefert  $\langle z'', B, z_1 \rangle \Rightarrow_G^* w'_1$  und  $\langle z_1, C, z \rangle \Rightarrow_G^* w'_2$ . Da  $\langle z', A, z \rangle \rightarrow \langle z'', B, z_1 \rangle \langle z_1, C, z \rangle \in P$ , gilt  $\langle z', A, z \rangle \Rightarrow_G^* a \langle z'', B, z_1 \rangle \langle z_1, C, z \rangle \Rightarrow_G^* a w'_1 \langle z_1, C, z \rangle \Rightarrow_G^* a w'_1 w'_2 = w$ .  $\square$

Sätze 4.5.11 und 4.5.14 zeigen:

**Theorem 4.5.15.** *Kellerautomaten erkennen genau die kontextfreien Sprachen.*

**Bemerkung 4.5.16.** *Die zu Theorem 4.5.15 zugehörigen Beweise zeigen auch, dass PDAs mit einem Zustand auskommen, denn der in Satz 4.5.11 konstruierte PDA besitzt nur einen Zustand.*

**Beispiel 4.5.17.** *Betrachte den PDA aus Beispiel 4.5.5  $M = (\{z_0, z_1\}, \{a, b\}, \{B, \#\}, \delta, z_0, \#)$  mit*

$$\begin{aligned} \delta(z_0, a, \#) &= \{(z_0, B\#)\} & \delta(z_0, b, B) &= \{(z_1, \varepsilon)\} & \delta(z_1, \varepsilon, \#) &= \{(z_1, \varepsilon)\} \\ \delta(z_0, a, B) &= \{(z_0, BB)\} & \delta(z_1, b, B) &= \{(z_1, \varepsilon)\} & \delta(z_0, \varepsilon, \#) &= \{(z_0, \varepsilon)\} \end{aligned}$$

und  $\delta(z_i, a, A) = \emptyset$  für alle anderen Fälle. Der Beweis von Satz 4.5.14 konstruiert die Grammatik  $G = (V, \Sigma, P, S)$  mit

$$\begin{aligned} V &= \{S, \langle z_0, B, z_0 \rangle, \langle z_0, B, z_1 \rangle, \langle z_1, B, z_0 \rangle, \\ &\quad \langle z_1, B, z_1 \rangle, \langle z_0, \#, z_0 \rangle, \langle z_0, \#, z_1 \rangle, \langle z_1, \#, z_0 \rangle, \langle z_1, \#, z_1 \rangle\} \\ P &= \{S \rightarrow \langle z_0, \#, z_0 \rangle, S \rightarrow \langle z_0, \#, z_1 \rangle \cdot\} \\ &\cup \{\langle z_0, B, z_1 \rangle \rightarrow b, \langle z_1, B, z_1 \rangle \rightarrow b, \langle z_0, \#, z_0 \rangle \rightarrow \varepsilon, \langle z_1, \#, z_1 \rangle \rightarrow \varepsilon\} \\ &\cup \{\langle z_0, \#, z_0 \rangle \rightarrow a \langle z_0, B, z_0 \rangle \langle z_0, \#, z_0 \rangle, \langle z_0, \#, z_0 \rangle \rightarrow a \langle z_0, B, z_1 \rangle \langle z_1, \#, z_0 \rangle \\ &\quad \langle z_0, \#, z_1 \rangle \rightarrow a \langle z_0, B, z_0 \rangle \langle z_0, \#, z_1 \rangle, \langle z_0, \#, z_1 \rangle \rightarrow a \langle z_0, B, z_1 \rangle \langle z_1, \#, z_1 \rangle\} \\ &\cup \{\langle z_0, B, z_0 \rangle \rightarrow a \langle z_0, B, z_0 \rangle \langle z_0, B, z_0 \rangle, \langle z_0, B, z_1 \rangle \rightarrow a \langle z_0, B, z_0 \rangle \langle z_0, B, z_1 \rangle, \\ &\quad \langle z_0, B, z_0 \rangle \rightarrow a \langle z_0, B, z_1 \rangle \langle z_1, B, z_0 \rangle, \langle z_0, B, z_1 \rangle \rightarrow a \langle z_0, B, z_1 \rangle \langle z_1, B, z_1 \rangle\} \end{aligned}$$

Diese Grammatik kann man noch vereinfachen, indem man untersucht, welche Produktionen nie in einer erfolgreichen Ableitung verwendet werden können:

- $\langle z_0, B, z_0 \rangle \rightarrow a \langle z_0, B, z_1 \rangle \langle z_1, B, z_0 \rangle$ , kann gelöscht werden, da die Produktion u.a.  $\langle z_1, B, z_0 \rangle$  erzeugt, es aber keine Produktion mit  $\langle z_1, B, z_0 \rangle$  als rechter Seite gibt.
- Nun kann  $\langle z_0, B, z_0 \rangle \rightarrow a \langle z_0, B, z_0 \rangle \langle z_0, B, z_0 \rangle$ , gelöscht werden, da sie die einzige Regel mit  $\langle z_0, B, z_0 \rangle$  als rechter Seite ist, aber auf der rechten Seite wieder ein  $\langle z_0, B, z_0 \rangle$  erzeugt. Daher kann kein Wort mit ihr erzeugt werden.
- Nun können alle Regeln gelöscht werden, die  $\langle z_0, B, z_0 \rangle$  erzeugen, da es keine Produktion mehr gibt mit  $\langle z_0, B, z_0 \rangle$  auf der rechten Seite. Dies sind:

$\langle z_0, \#, z_0 \rangle \rightarrow a \langle z_0, B, z_0 \rangle \langle z_0, \#, z_0 \rangle, \langle z_0, \#, z_1 \rangle \rightarrow a \langle z_0, B, z_0 \rangle \langle z_0, \#, z_1 \rangle \langle z_0, B, z_1 \rangle \rightarrow a \langle z_0, B, z_0 \rangle \langle z_0, B, z_1 \rangle,$

- Nun kann  $\langle z_0, \#, z_0 \rangle \rightarrow a \langle z_0, B, z_1 \rangle \langle z_1, \#, z_0 \rangle$ , gelöscht werden, da es keine Produktion für  $\langle z_1, \#, z_0 \rangle$  gibt.

Es verbleiben die Produktionen

$\{S \rightarrow \langle z_0, \#, z_0 \rangle, S \rightarrow \langle z_0, \#, z_1 \rangle, \langle z_0, B, z_1 \rangle \rightarrow b, \langle z_1, B, z_1 \rangle \rightarrow b, \langle z_0, \#, z_0 \rangle \rightarrow \varepsilon, \langle z_1, \#, z_1 \rangle \rightarrow \varepsilon, \langle z_0, \#, z_1 \rangle \rightarrow a \langle z_0, B, z_1 \rangle \langle z_1, \#, z_1 \rangle, \langle z_0, B, z_1 \rangle \rightarrow a \langle z_0, B, z_1 \rangle \langle z_1, B, z_1 \rangle\}$

Mit Umbenennen, Streichen von nicht erreichbaren Variablen und Entfernen von Einheitsproduktionen erhalten wir als Grammatik

$$G = ((S, A, B), \{a, b\}, \{S \rightarrow \varepsilon \mid aB, B \rightarrow aBC \mid b, C \rightarrow b\}, S)$$

Offensichtlich erzeugt  $B$  Wörter der Form  $a^{i+1}b^i$  und damit ist  $L(G) = \{a^i b^i \mid i \in \mathbb{N}_0\}$ . Beachte: Bis auf  $S \rightarrow \varepsilon$  ist  $G$  in Greibach-Normalform.

ei  $G' = (\{S, A, B, C\}, \{a, b\}, \{S \rightarrow aB, B \rightarrow b \mid aBC, C \rightarrow b\}, S)$ , dann gilt  $L(G') = L(G) \setminus \{\varepsilon\}$  und  $G'$  ist in Greibach-Normalform. Der Beweis von Satz 4.5.11 konstruiert für  $G$  den PDA  $M = (\{z_0\}, \Sigma, \Sigma \cup V, \delta, z_0, S)$  mit

$$\begin{aligned} \delta(z_0, a, S) &= \{(z_0, B)\} & \delta(z_0, b, B) &= \{(z_0, \varepsilon)\} & \delta(z_0, a, B) &= \{(z_0, BC)\} \\ \delta(z_0, b, C) &= \{(z_0, \varepsilon)\} & \delta(z_0, \varepsilon, S) &= \{(z_0, \varepsilon)\} & \delta(z_0, d, A) &= \emptyset \text{ in allen anderen Fällen} \end{aligned}$$

Der PDA  $M$  hat (wie bereits erwähnt) nur einen Zustand und akzeptiert  $L(M) = \{a^i b^i \mid i \in \mathbb{N}_0\}$ . Eine Konfigurationsfolge für die Eingabe  $aaabbb$  ist

$$\begin{aligned} (z_0, aaabbb, S) \vdash (z_0, aabbb, B) \vdash (z_0, abbb, BC) \vdash (z_0, bbb, BCC) \\ \vdash (z_0, bb, CC) \vdash (z_0, b, C) \vdash (z_0, \varepsilon, \varepsilon) \end{aligned}$$

#### 4.6. Deterministisch kontextfreie Sprachen

Für deterministische Kellerautomaten verwenden wir Automaten mit Endzuständen. Wir erlauben jedoch weiterhin  $\varepsilon$ -Übergänge. Damit der Übergang trotzdem deterministisch bleibt, darf ein  $\varepsilon$ -Übergang nur dann möglich sein, wenn bei gleichem obersten Kellersymbol kein Übergang bei Lesen eines Zeichens möglich ist. Die folgende Definition für deterministische Kellerautomaten sichert dies zu:

**Definition 4.6.1** (Deterministischer Kellerautomat, DPDA). *Ein Kellerautomat mit Endzuständen  $M = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$  ist deterministisch (ein DPDA) wenn für alle  $(z, a, A) \in (Z, \Sigma, \Gamma)$  gilt:  $|\delta(z, a, A)| + |\delta(z, \varepsilon, A)| \leq 1$ .*

Die von DPDA's akzeptierten Sprachen heißen deterministisch kontextfrei.

**Satz 4.6.2.** *Die Sprache  $L = \{w\$ \overline{w} \mid w \in \{a, b\}^*\}$  ist deterministisch kontextfrei.*

*Beweis.* Sei  $M = (\{z_0, z_1, z_2\}, \{a, b, \$\}, \{\#, A, B\}, \delta, z_0, \#, \{z_2\})$ , wobei

$$\begin{array}{lll} \delta(z_0, a, \#) = \{(z_0, A\#)\} & \delta(z_0, a, B) = \{(z_0, AB)\} & \delta(z_0, \$, \#) = \{(z_1, \#)\} \\ \delta(z_0, b, \#) = \{(z_0, B\#)\} & \delta(z_0, b, B) = \{(z_0, BB)\} & \delta(z_1, a, A) = \{(z_1, \varepsilon)\} \\ \delta(z_0, a, A) = \{(z_0, AA)\} & \delta(z_0, \$, A) = \{(z_1, A)\} & \delta(z_1, b, B) = \{(z_1, \varepsilon)\} \\ \delta(z_0, b, A) = \{(z_0, BA)\} & \delta(z_0, \$, B) = \{(z_1, B)\} & \delta(z_1, \varepsilon, \#) = \{(z_2, \varepsilon)\} \end{array}$$

und  $\delta(z_i, c, C) = \emptyset$  für alle anderen Fälle mit  $(z_i, c, C) \in \{z_0, z_1, z_2\} \times \{a, b, \$\} \times \{\#, A, B\}$ . Der PDA mit Endzuständen  $M$  erkennt  $L$ , denn vom Startzustand  $z_0$  aus merkt er sich die gelesenen Symbole  $w$  auf dem Keller (im umgekehrter Reihenfolge) ( $A$  für  $a$  und  $B$  für  $b$ ), nach Lesen von  $\$$  wechselt er in den Zustand  $z_1$  und kann dort den Keller abbauen, indem er  $\bar{w}$  verarbeitet. Im Anschluss daran wechselt er mit dem untersten Kellersymbol in den akzeptierenden Zustand  $z_2$ . Offensichtlich ist  $M$  deterministisch (d. h. ein DPDA).  $\square$

**Satz 4.6.3.** Die Sprache  $L = \{a^i b^i \mid i \in \mathbb{N}\}$  ist deterministisch kontextfrei.

*Beweis.* Der PDA mit Endzuständen  $M = (\{z_0, z_1, z_2\}, \{a, b\}, \{\#, A\}, \delta, z_0, \#, \{z_2\})$ , wobei

$$\begin{array}{lll} \delta(z_0, a, \#) = \{(z_0, A\#)\} & \delta(z_0, b, A) = \{(z_1, \varepsilon)\} & \delta(z_1, \varepsilon, \#) = \{(z_2, \varepsilon)\} \\ \delta(z_0, a, A) = \{(z_0, AA)\} & \delta(z_1, b, A) = \{(z_1, \varepsilon)\} & \end{array}$$

und  $\delta(z_i, c, B) = \emptyset$  für alle anderen Fälle mit  $(z_i, c, B) \in \{z_0, z_1, z_2\} \times \{a, b\} \times \{\#, A\}$  erkennt  $L$  und ist deterministisch (d. h. ein DPDA).  $\square$

Für DPDAs ist jede Folge von Konfigurationen eindeutig, d. h. es gibt zu jeder Konfiguration höchstens eine Nachfolgekongfiguration. Im Folgenden erwähnen wir einige wichtige Eigenschaften der deterministisch kontextfreien Sprachen, verzichten aber an dieser Stelle auf deren Beweise:

**Theorem 4.6.4** (Eigenschaften deterministisch kontextfreier Sprachen).

1. Das Wortproblem für deterministisch kontextfreie Sprachen kann in Linearzeit entschieden werden.
2. Für deterministisch kontextfreie Sprachen gibt es eindeutige Grammatiken.
3. Deterministisch kontextfreie Sprachen sind unter Komplementbildung abgeschlossen.

Während das Theorem besagt, dass deterministische kontextfreie Sprachen bezüglich Komplementbildung abgeschlossen sind, gilt dies nicht für Schnittbildung und Vereinigung:

**Satz 4.6.5.** Deterministisch kontextfreie Sprachen sind nicht abgeschlossen bezüglich Vereinigung und Schnitt.

*Beweis.* Schnittbildung kann widerlegt werden durch die Schneiden der Sprachen  $\{a^n b^n c^m \mid n, m \in \mathbb{N}\}$  und  $\{a^n b^m c^m \mid n, m \in \mathbb{N}\}$ , denn beide Sprachen sind deterministisch kontextfrei, ihr

Schnitt hingegen ist die Sprache  $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ , die – wie bereits gezeigt, Satz 4.2.3 – nicht kontextfrei ist.

Die Nichtabgeschlossenheit bezüglich Vereinigung ergibt sich aus der gültigen Gleichung  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ : Deterministisch kontextfreie Sprachen sind abgeschlossen bezüglich Komplementbildung und bei Abgeschlossenheit bezüglich der Vereinigung würde auch folgen, dass deterministisch kontextfreie Sprachen auch abgeschlossen gegen Schnittbildung wären. Das hatten wir aber gerade eben widerlegt.  $\square$

## 4.7. Entscheidbarkeitsresultate

In diesem letzten Abschnitt des Kapitels prüfen wir die Entscheidbarkeit verschiedener Probleme für die Klasse der Typ 2-Sprachen. Das Wortproblem haben wir bereits betrachtet und mit dem CYK-Algorithmus gezeigt, dass es für kontextfreie Sprachen effizient entscheidbar ist.

Auch die Frage, ob eine kontextfreie Grammatik die leere Sprache erzeugt ist entscheidbar:

**Satz 4.7.1.** *Das Leerheitsproblem für kontextfreie Grammatiken ist entscheidbar.*

*Beweis.* Sei  $L$  eine kontextfreie Sprache gegeben durch eine kontextfreie Grammatik. Wir können entsprechend Definition 2.2.4 prüfen, ob  $\varepsilon \in L$  gilt. Ist dies der Fall, dann ist  $L$  nicht leer. Andernfalls sei  $G = (V, \Sigma, P, S)$  eine CFG in Chomsky-Normalform mit  $L(G) = L$ . Algorithmus 5 markiert alle Variablen  $A \in V$  für die gilt  $\{w \in \Sigma^* \mid A \Rightarrow_G^* w\} \neq \emptyset$ .

---

**Algorithmus 5 :** Markierung der Variablen, die nichtleere Sprachen erzeugen

---

**Eingabe :** Grammatik  $G = (V, \Sigma, P, S)$  in Chomsky-Normalform

**Ausgabe :** Menge  $W \subseteq V$  aller Variablen, die nicht die leere Sprache erzeugen

**Beginn**

$W := \{A \in V \mid A \rightarrow a \in P, a \in \Sigma\};$

**wiederhole**

$W_{alt} := W;$

$W := W_{alt} \cup \{A \mid A \rightarrow BC \in P, B \in W_{alt}, C \in W_{alt}\};$

**bis**  $W = W_{alt};$

**return**  $W$

**Ende**

---

Nach Ausführung von Algorithmus 5 müssen wir nur testen, ob das Startsymbol  $S$  markiert wurde (d. h. ob  $S \in W$  gilt).  $\square$

**Satz 4.7.2.** *Das Endlichkeitsproblem für kontextfreie Sprachen ist entscheidbar.*

*Beweis.* Sei  $G = (V, \Sigma, P, S)$  eine CFG in Chomsky-Normalform. Sei  $n$  die Zahl aus dem Pumping-Lemma für CFGs (wir können  $n = 2^{|V|}$  setzen, wie der Beweis des Pumping-Lemmas zeigt). Es gilt  $|L(G)| = \infty$  genau dann, wenn es ein Wort  $z \in L(G)$  mit  $n \leq |z| < 2n$  gibt:

- „ $\Leftarrow$ “: Wenn es ein Wort  $z \in L$  mit  $|z| \geq n$  gibt, dann zeigt, dass Pumping-Lemma für CFGs (Lemma 4.2.1), dass alle Wörter  $uv^iwx^iy \in L$  für  $i \in \mathbb{N}_0$  liegen – das sind unendliche viele Wörter.
- „ $\Rightarrow$ “: Nehme an, die Aussage sei falsch, d. h. es gibt kein Wort  $z \in L(G)$  für  $n \leq |z| < 2n$ , aber trotzdem gilt  $|L(G)| = \infty$ . Sei  $z \in L(G)$  das kürzeste Wort mit  $|z| \geq 2n$ . Das Pumping-Lemma zeigt, dass es Wörter  $u, v, w, x, y$  gibt mit  $z = uvwxy$ ,  $|vx| > 0$  und  $|vwx| \leq n$ , sodass insbesondere  $uv^0wx^0y \in L$  gilt. Da  $|uv^0wx^0y| = |uwy| < |uvwxy|$  und  $|uwy| \geq n$  gilt, war  $z$  nicht minimal gewählt, was einen Widerspruch darstellt.

D. h. wir können das Endlichkeitsproblem entscheiden, indem wir alle Wörter  $w \in \Sigma^*$  der Länge  $n \leq |w| < 2n$  aufzählen und mit dem CYK-Algorithmus testen, ob  $w \in L(G)$  gilt.  $\square$

Es gibt wesentlich effizientere Verfahren, um das Endlichkeitsproblem zu lösen (siehe z. B. (Weg99)).

Die meisten anderen Fragestellungen (z. B. das Äquivalenzproblem und das Schnittproblem) sind für kontextfreie Sprachen unentscheidbar.

## 5. Kontextsensitive und Typ 0-Sprachen

In diesem Kapitel behandeln wir Typ 1- und Typ 0-Sprachen und lernen die Turingmaschine als Maschinenmodell kennen. Typ 0-Sprachen werden genau von den Turingmaschinen erfasst, während Typ 1-Sprachen von linear platz-beschränkten Turingmaschinen erfasst werden.

Wir erinnern an die Unterschiede zwischen Typ 2-, Typ 1- und Typ 0-Sprachen. Der Unterschied zwischen kontextfreien (also Typ 2-) und kontextsensitiven (also Typ 1-) Sprachen besteht darin, dass die linken Seiten der Produktionen bei kontextsensitiven Sprachen Satzformen sein dürfen, während diese bei kontextfreien Sprachen nur einzelne Variablen sind. Der Unterschied zwischen Typ 1- und Typ 0-Sprachen ist, dass für Produktionen  $\ell \rightarrow r$  der Typ 1-Grammatiken  $|r| \geq |\ell|$  gelten muss, d. h. Anwenden der Produktionen erlaubt es nicht, Wörter zu schrumpfen. Diese Grammatiken werden auch *monotone Grammatiken* genannt. In manchen Arbeiten und Büchern werden kontextsensitive Grammatiken so definiert, dass alle Produktionen die Form  $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \alpha_3 \alpha_2$  haben müssen, wobei  $\alpha_i$  Satzformen sind und  $\alpha_3 \neq \varepsilon$  gelten muss. Wir verwenden (wie (Sch08)) die allgemeinere Definition, die jedoch die gleiche Menge an Sprachen definiert.

### 5.1. Turingmaschinen

Die kontextfreien Sprachen werden genau durch die (nichtdeterministischen) Kellerautomaten erkannt. Typ 1- und Typ 0-Sprachen umfassen mehr Sprachen. Daher muss das Automatenmodell auch mehr können. Die wesentliche Einschränkung des Kellerautomaten ist, dass Speicher nur in Form eines Kellers zur Verfügung steht auf den der Zugriff nur von oben möglich ist. Z. B. kann die Sprache  $\{a^i b^i c^i \mid i \in \mathbb{N}\}$  nicht von einem Kellerautomaten erkannt werden, da das Verifizieren, dass genau  $i$   $b$ 's auf  $i$   $a$ 's folgen, den aufgebauten Keller für die  $a$ -Symbole leer räumen muss, und daher die Anzahl  $i$  nicht mehr für das nötige Verifizieren von  $i$   $c$ 's zur Verfügung steht. Könnten wir den Keller von unten nach oben lesen, dann wäre es leicht, die Sprache  $\{a^i b^i c^i \mid i \in \mathbb{N}\}$  zu erkennen.

Daher betrachten wir als erweitertes Automatenmodell die 1936 vom britischen Informatiker Alan Turing eingeführten Turingmaschinen. Ein informelles Schaubild zur Illustration ist in Abb. 5.1. Turingmaschinen besitzen als Speicher ein (unendlich langes) Band, welches gelesen und beschrieben werden kann und der Zugriff des Schreib-Lesekopfes ist in beide Richtungen möglich.

Die formale Definition für Turingmaschinen ist:

**Definition 5.1.1** (Turingmaschine). *Eine Turingmaschine (TM) ist ein 7-Tupel  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  wobei*

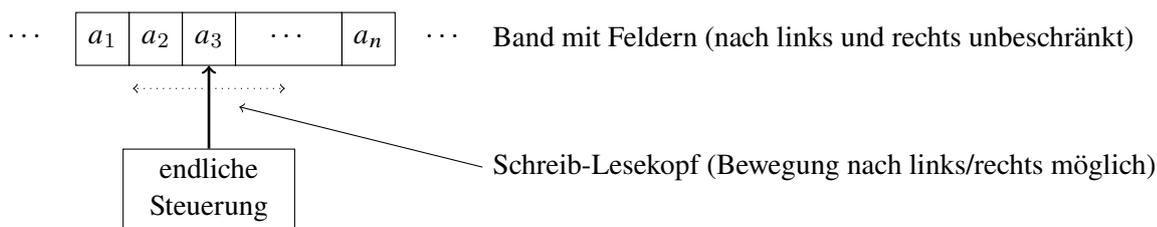


Abbildung 5.1.: Illustration der Turingmaschine

- $Z$  ist eine endliche Menge von Zuständen,
- $\Sigma$  ist das (endliche) Eingabealphabet,
- $\Gamma \supset \Sigma$  ist das (endliche) Bandalphabet,
- für eine deterministische TM ist  $\delta : (Z \times \Gamma) \rightarrow (Z \times \Gamma \times \{L, R, N\})$  die Zustandsübergangsfunktion (oder nur Übergangsfunktion) und für eine nichtdeterministische TM ist  $\delta : (Z \times \Gamma) \rightarrow \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$  die Zustandsübergangsfunktion (oder nur Übergangsfunktion)
- $z_0 \in Z$  ist der Startzustand,
- $\square \in \Gamma \setminus \Sigma$  ist das Blank-Symbol
- $E \subseteq Z$  ist die Menge der Endzustände.

Wir verwenden auch DTM bzw. NTM für die deterministische Turingmaschine bzw. nicht-deterministische Turingmaschine.

Wir beschreiben den Zustandswechsel informell für die Übergangsfunktion  $\delta$  und zunächst für die deterministische Turing-Maschine: Ein Eintrag  $\delta(z, a) = (z', b, x)$  bedeutet, dass die Turingmaschine im Zustand  $z$  bei Lesen des Zeichens  $a$  auf der aktuellen Position des Schreib-Lesekopfes auf dem Band, in den Zustand  $z'$  wechselt,  $a$  durch  $b$  auf dem Band ersetzt und den Kopf um eine Position nach links schiebt, falls  $x = L$ , um eine Position nach rechts schiebt, falls  $x = R$ , bzw. den Kopf unverändert lässt, falls  $x = N$  ( $N$  steht hier für neutral).

Für die nichtdeterministische TM gilt entsprechend, dass  $(z', b, x) \in \delta(z, a)$  bedeutet, dass die Turingmaschine vom Zustand  $z$  in den Zustand  $z'$  wechseln kann, wenn  $a$  auf der aktuellen Position des Schreib-Lesekopfes auf dem Band steht, und sie dabei  $a$  durch  $b$  auf dem Band ersetzt und den Kopf entsprechend  $x$  bewegt.

Analog zu PDAs definieren wir Konfigurationen für Turingmaschinen, um deren aktuellen Zustand einschließlich Bandinhalt und Kopfposition zu beschreiben:

**Definition 5.1.2** (Konfiguration einer Turingmaschine). Eine Konfiguration einer Turingmaschine ist ein Wort  $k \in \Gamma^* Z \Gamma^*$

Eine Konfiguration  $wzw'$  entspricht dabei einer Turingmaschine im Zustand  $z$ , sodass auf dem Band  $ww'$  steht (links und rechts davon ist der Bandinhalt jeder Zelle des Bandes das Blank-Zeichen  $\square$ ) und der Schreib-Lesekopf steht auf dem ersten Symbol von  $w'$ . Das Wort  $ww'$  ist der Bandabschnitt, der schon von der Turing-Maschine bearbeitet wurde (oder Teil der Eingabe ist).

Initial befindet sich die TM im Startzustand  $z_0$ , auf dem Band steht das Eingabewort und der Schreib-Lesekopf steht auf dem ersten Symbol des Eingabeworts<sup>1</sup>:

**Definition 5.1.3** (Startkonfiguration einer TM). *Für ein Eingabewort  $w$  ist die Startkonfiguration einer TM  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  das Wort  $z_0w$ .*

Die Transitionsrelation einer Turingmaschine  $M$  wird als binäre Relation  $\vdash_M$  auf den Konfigurationen von  $M$  definiert:

**Definition 5.1.4** (Transitionsrelation für Konfigurationen einer TM). *Sei  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  eine TM. Die binäre Relation  $\vdash_M$  ist definiert durch die folgenden Fälle (wobei  $\delta(z, a) = (z', c, x)$  im Falle einer NTM meint  $(z', c, x) \in \delta(z, a)$ ):*

- $b_1 \cdots b_m z a_1 \cdots a_n \vdash_M b_1 \cdots b_m z' c a_2 \cdots a_n$ , wenn  $\delta(z, a_1) = (z', c, N)$ ,  $m \geq 0, n \geq 1, z \notin E$
- $b_1 \cdots b_m z a_1 \cdots a_n \vdash_M b_1 \cdots z' b_m c a_2 \cdots a_n$ , wenn  $\delta(z, a_1) = (z', c, L)$ ,  $m \geq 1, n \geq 1, z \notin E$
- $b_1 \cdots b_m z a_1 \cdots a_n \vdash_M b_1 \cdots b_m c z' a_2 \cdots a_n$ , wenn  $\delta(z, a_1) = (z', c, R)$ ,  $m \geq 0, n \geq 2, z \notin E$
- $b_1 \cdots b_m z a_1 \vdash_M b_1 \cdots b_m c z' \square$ , wenn  $\delta(z, a_1) = (z', c, R)$  und  $m \geq 0, z \notin E$
- $z a_1 \cdots a_n \vdash_M z' \square c a_2 \cdots a_n$ , wenn  $\delta(z, a_1) = (z', c, L)$  und  $n \geq 1, z \notin E$

Mit  $\vdash_M^i$  bezeichnen wir die  $i$ -fache Anwendung von  $\vdash_M$ , mit  $\vdash_M^*$  die reflexiv-transitive Hülle von  $\vdash_M$ . Wir verzichten manchmal auf den Index  $M$  und schreiben nur  $\vdash$ , wenn die betrachtete Turingmaschine eindeutig ist.

In allen Fällen verbieten wir Nachfolgekongfigurationen, wenn der aktuelle Zustand ein Endzustand ist<sup>2</sup>.

Die ersten drei Fälle von Definition 5.1.4 sind die Standardfälle und unterscheiden sich darin, ob der Schreib-Lesekopf neutral, nach links oder nach rechts wechselt. Der vierte Fall beschreibt den Fall, dass der Schreib-Lesekopf nach rechts wandert, aber das Band rechts vom Schreib-Lesekopf noch nicht von der TM bearbeitet wurde: In diesem Fall wird ein Blank-Symbol rechts neben dem Schreib-Lesekopf erzeugt. Der fünfte Fall tritt ein, wenn die TM nach links in den Bereich des Bandes kommt, der noch nicht bearbeitet wurde. Auch dann wird ein Blank-Symbol eingefügt, damit der Schreib-Lesekopf nach links bewegt werden kann.

Nun haben wir alle Definitionen eingeführt, um die akzeptierte Sprache einer Turingmaschine zu definieren. Informell sind dies alle Wörter, die als Eingabe verwendet, dazu führen, dass

<sup>1</sup>Falls die Eingabe das leere Wort  $\varepsilon$  ist, steht der Schreib-Lesekopf auf einem Blank-Symbol

<sup>2</sup>Es gibt verschiedene Varianten der Definition, wie sich eine Turingmaschine im Endzustand verhält: Z. B. macht (Sch08) gar keine Annahme (sodass die Turingmaschine weiter rechnen dürfte), (HMU06) treffen die Annahme, dass die TM anhält, wenn sie in einem akzeptierenden Zustand ist (was analog zu unserem Vorgehen ist). Eine andere Möglichkeit ist es  $\delta(q, a) = (q, a, N)$  für alle  $q \in E$  und  $a \in \Gamma$  zu fordern, die Turingmaschine ist in diesem Fall im Endzustand gefangen. Schließlich kann man auch  $\delta$  als partielle Funktion sehen, die für die Fälle  $\delta(q, a)$  mit  $q \in E$  undefiniert ist. Alle Formalismen sind äquivalent und ineinander überführbar.

die Turingmaschine einen akzeptierenden Zustand erreicht (bzw. im nichtdeterministischen Fall: erreichen kann).

**Definition 5.1.5** (Akzeptierte Sprache einer TM). Sei  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  eine TM. Die von  $M$  akzeptierte Sprache  $L(M)$  ist definiert als

$$L(M) := \{w \in \Sigma^* \mid \exists u, v \in \Gamma^*, z \in E : z_0 w \vdash_M^* uzv\}$$

Beachte, dass für alle Turingmaschinen der Form  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  mit  $z_0 \in E$  gilt  $L(M) = \Sigma^*$ , denn diese Turingmaschinen akzeptieren jede Eingabe sofort.

Das folgende Beispiel ist auch in (Sch08, S.75) zu finden:

**Beispiel 5.1.6.** Die Turingmaschine  $M = (\{z_0, z_1, z_2, z_3\}, \{0, 1\}, \{0, 1, \square\}, \delta, z_0, \square, \{z_3\})$  mit

$$\begin{array}{lll} \delta(z_0, 0) = (z_0, 0, R) & \delta(z_0, 1) = (z_0, 1, R) & \delta(z_0, \square) = (z_1, \square, L) \\ \delta(z_1, 0) = (z_2, 1, L) & \delta(z_1, 1) = (z_1, 0, L) & \delta(z_1, \square) = (z_3, 1, N) \\ \delta(z_2, 0) = (z_2, 0, L) & \delta(z_2, 1) = (z_2, 1, L) & \delta(z_2, \square) = (z_3, \square, R) \\ \delta(z_3, 0) = (z_3, 0, N) & \delta(z_3, 1) = (z_3, 1, N) & \delta(z_3, \square) = (z_3, \square, N) \end{array}$$

interpretiert ein Eingabewort  $w \in \{0, 1\}^*$  als Binärzahl und addiert 1 hinzu. Im Startzustand  $z_0$  läuft die Maschine nach rechts, ohne die Eingabe zu verändern, bis sie das Blank-Symbol  $\square$  liest und in den Zustand  $z_1$  wechselt. In  $z_1$  versucht sie 1 zur aktuellen Ziffer hinzu zu addieren: Gelingt ihr das ohne Übertrag (die Ziffer ist 0), wechselt sie in  $z_2$  und läuft dann ohne weitere Veränderung zum Anfang und wenn sie das Blank-Symbol erkannt hat, wechselt sie in den Akzeptanzzustand  $z_3$ . Entsteht beim Addieren von 1 ein Übertrag, so muss auch 1 zur nächsten Ziffer links davon addiert werden. Daher verbleibt der Automat in Zustand  $z_1$  und der Lesekopf wechselt nach links. Falls der Übertrag auch nach Lesen der gesamten Zahl noch entsteht (z. B. bei 1111 als Eingabe), so wird eine neue erste Stelle durch den Fall  $\delta(z_1, \square) = (z_3, 1, N)$  hinzugefügt: Die 1 ersetzt das Blank-Symbol links von der Zahl, anschließend wird in den akzeptierenden Zustand  $z_3$  gewechselt.

Betrachte z. B. die Zahl 0011 als Eingabe. Dann läuft die Turingmaschine wie folgt:

$$z_0 0011 \vdash 0z_0 011 \vdash 00z_0 11 \vdash 001z_0 1 \vdash 0011z_0 \square \vdash 001z_1 1 \square \vdash 00z_1 10 \square \vdash 0z_1 000 \square \vdash z_2 0100 \square \vdash z_2 \square 0100 \square \vdash \square z_3 0100 \square$$

## 5.2. Linear beschränkte Turingmaschinen und kontextsensitive Sprachen

Wir betrachten als nächstes spezielle Turingmaschinen, die nur den Platz auf dem Band in Anspruch nehmen, den die Eingabe zur Verfügung stellt, d. h. diese Turingmaschinen verlassen die Eingabe nie mit dem Schreib-Lesekopf. Wir nennen diese Turingmaschinen *linear beschränkte Turingmaschinen*. Damit diese Turingmaschinen das rechte Ende der Eingabe erkennen können, markieren wir das letzte Symbol der Eingabe: Hierfür verwenden wir eine Kopie des Alphabets. Sei  $\Sigma = \{a_1, \dots, a_n\}$  ein Alphabet, dann bezeichnen wir mit  $\widehat{\Sigma}$  das Alphabet

$\widehat{\Sigma} = \{\widehat{a}_1, \dots, \widehat{a}_n\}$ . Eine Eingabe  $a_1 \cdots a_m$  auf dem Band der Turingmaschine wird dementsprechend als  $a_1 \cdots a_{m-1} \widehat{a}_m$  dargestellt und die Turingmaschine arbeitet auf dem Alphabet  $\Sigma' = \Sigma \cup \widehat{\Sigma}$ . Beachte, dass wir das linke Ende des Bandes nicht markieren, da dies die Turingmaschine selbst kann, indem sie ganz am Anfang das aktuelle Symbol auf dem Band durch ein gleichartiges aber markiertes Symbol ersetzt.

**Definition 5.2.1.** Eine NTM  $M = (Z, \Sigma \cup \widehat{\Sigma}, \Gamma, \delta, z_0, \square, E)$  heißt linear beschränkt (LBA, linear bounded automaton), wenn für alle  $a_1 \cdots a_m \in \Sigma^+$  und alle Konfigurationen  $uzv$  mit  $z_0 a_1 \cdots a_{m-1} \widehat{a}_m \vdash_M^* uzv$  gilt:  $|uv| \leq m$ .

Die akzeptierte Sprache eines LBA  $M$  ist

$$L(M) := \{a_1 \cdots a_m \in \Sigma^* \mid z_0 a_1 \cdots a_{m-1} \widehat{a}_m \vdash_M^* uzv \text{ wobei } u, v \in \Gamma^* \text{ und } z \in E\}$$

Beachte, dass LBAs als nichtdeterministische Turingmaschinen definiert sind.

**Theorem 5.2.2** (Satz von Kuroda). *Kontextsensitive Sprachen werden genau von den LBAs erkannt.*

### 5.3. Turingmaschinen und Typ 0-Grammatiken

**Satz 5.3.1.** *Die durch (allgemeine) nichtdeterministischen Turingmaschinen akzeptierten Sprachen sind genau die Typ 0-Sprachen.*

Da nichtdeterministische Turingmaschinen durch deterministische Turingmaschinen simuliert werden können, indem alle Berechnungsmöglichkeiten der NTM nacheinander von der DTM durchprobiert werden, gilt der letzte Satz auch für deterministische Turingmaschinen (der Unterschied zwischen NTMs und DTMs kommt erst zum Tragen, wenn wir das Laufzeitverhalten betrachten, wie wir es später im Kapitel zur Komplexitätstheorie machen werden).

## 6. Überblick über die Chomsky-Hierarchie

In diesem Kapitel geben wir übersichtsartig an, welche Formalismen und Resultate gelten.

Die folgende Tabelle gibt einen Überblick über die Sprachklassen (einschließlich der deterministisch kontextfreien Sprachen, die eine echte Teilmenge der Typ 2-Sprachen und eine echte Obermenge der Typ 3-Sprachen sind), den dazu passenden Grammatiken, Automatenmodellen und sonstigen Modellen, die wir kennengelernt haben.

Sprache	Grammatik	Automat	sonstiges
Typ 3	reguläre Grammatik	endlicher Automat (DFA und NFA)	regulärer Ausdruck
deterministisch kontextfrei	$LR(k)$ -Grammatik	Deterministischer Kellerautomat (DPDA)	
Typ 2	kontextfreie Grammatik	Kellerautomat (PDA) (nicht-deterministisch)	
Typ 1	kontextsensitive Grammatik	linear beschränkte Turingmaschine (LBA) (nichtdeterministisch)	
Typ 0	Typ 0-Grammatik	Turingmaschine (deterministisch und nichtdeterministisch)	

Beachte, dass wir den Formalismus der  $LR(k)$ -Grammatiken nicht behandelt haben. Diese Grammatiken spielen eine große Rolle im Compilerbau. Eine Definition ist z. B. in (Weg99) zu finden. Die  $LR(k)$ -Grammatiken erzeugen genau die deterministisch kontextfreien Sprachen.

Trennende Beispiele zwischen den verschiedenen Sprachklassen sind:

- Die Sprache  $\{a^n b^n \mid n \in \mathbb{N}_0\}$  ist Typ 2 aber nicht vom Typ 3.
- Die Sprache  $\{w \in \{a, b\}^* \mid w \text{ ist Palindrom}\}$  ist Typ 2 aber nicht deterministisch-kontextfrei.
- Die Sprache  $\{a^n b^n c^n \mid n \in \mathbb{N}_0\}$  ist Typ 1 aber nicht vom Typ 2.
- Die Sprache  $H = \{M \# w \mid \text{die durch } M \text{ beschriebene TM hält bei Eingabe } w\}$  ist Typ 0 aber nicht vom Typ 1.  
(Die Sprache  $H$  ist das Halteproblem, welches wir später noch genauer betrachten und erläutern).
- Das Komplement von  $H$  ist nicht vom Typ 0.

Bezüglich der Automatenmodelle zeigt die folgende Tabelle die deterministischen Varianten und die nichtdeterministischen Varianten und beantwortet (soweit bekannt), ob die Modelle äquivalent sind, d. h. genau dieselbe Sprachklasse akzeptieren:

Deterministischer Automat	nichtdeterministischer Automat	äquivalent?
DFA	NFA	ja
DPDA	PDA	nein
DLBA	LBA	unbekannt
DTM	NTM	ja

Es gelten die folgenden Abschlusseigenschaften für die verschiedenen Sprachklassen:

Sprachklasse	Schnitt	Vereinigung	Komplement	Produkt	Kleenescher Abschluss
Typ 3	ja	ja	ja	ja	ja
deterministisch kontextfrei	nein	nein	ja	nein	nein
Typ 2	nein	ja	nein	ja	ja
Typ 1	ja	ja	ja	ja	ja
Typ 0	ja	ja	nein	ja	ja

Die folgenden Entscheidbarkeiten gelten für die betrachteten Entscheidungsprobleme:

Sprachklasse	Wortproblem	Leerheitsproblem	Äquivalenzproblem	Schnittproblem
Typ 3	ja	ja	ja	ja
deterministisch kontextfrei	ja	ja	ja	nein
Typ 2	ja	ja	nein	nein
Typ 1	ja	nein	nein	nein
Typ 0	nein	nein	nein	nein

Die Komplexität des Wortproblems für die verschiedenen Sprachklassen ist:

Sprachklasse	
Typ 3, DFA gegeben	lineare Komplexität
deterministisch kontextfrei	lineare Komplexität
Typ 2, Chomsky-Normalform gegeben	$O(n^3)$
Typ 1	exponentiell
Typ 0	unlösbar

# 7. Berechenbarkeit

## 7.1. Einleitung

Jeder der schon einmal programmiert hat, hat ein Gefühl dafür, was mit einem Computerprogramm berechnet werden kann, und eventuell auch dafür, was mit einem Computerprogramm *nicht berechnet* werden kann. Nachzuweisen, dass ein Problem nicht mit dem Computer berechnet werden kann, erscheint jedoch ziemlich schwierig.

Diese Frage formal zu fassen, war das Anliegen einiger berühmter Informatiker und Mathematiker, insbesondere zu nennen sind Alan Turing und Alonzo Church.

Wir schränken unsere Überlegungen zunächst auf natürliche Zahlen und Funktionen über den natürlichen Zahlen ein und definieren einen Begriff der Berechenbarkeit für solche Funktionen. Ein Problem, das dabei entsteht, ist, dass wir argumentieren müssen, dass diese Definition mit dem intuitiven Begriff der Berechenbarkeit übereinstimmt (beweisen können wir das nicht, da der intuitive Begriff nicht formal definiert ist).

**Definition 7.1.1.** Eine Funktion  $f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$  nennen wir berechenbar, wenn es einen Algorithmus gibt (z.B. in Form eines Computerprogramms einer modernen Programmiersprache, wie C, Java, Haskell, ...), welches  $f$  berechnet. D.h. wenn das Programm mit der Eingabe  $(n_1, \dots, n_k) \in \mathbb{N}_0^k$  startet, dann stoppt der Algorithmus nach endlich vielen Berechnungsschritten und liefert den Wert von  $f(n_1, \dots, n_k)$  als Ergebnis. Falls  $f$  eine partielle Funktion ist (d.h. nicht für alle Eingaben definiert ist), dann soll der Algorithmus bei einer Eingabe  $(n_1, \dots, n_k)$  für die  $f(n_1, \dots, n_k)$  undefiniert ist, nicht anhalten, sondern unendlich lange laufen.

Beachte, dass der Begriff der Berechenbarkeit nicht erfordert, den berechnenden Algorithmus anzugeben, sondern dessen Existenz genügt. Daher ist z.B. die Funktion

$$f(n) = \begin{cases} 1, & \text{wenn es unendlich viele Primzahlzwillinge gibt} \\ 0, & \text{sonst} \end{cases}$$

berechenbar, obwohl die Frage, ob es unendlich viele Primzahlzwillinge gibt, ungeklärt ist: Denn entweder ist der Algorithmus, der immer 0 liefert, oder der Algorithmus, der immer 1 liefert, der „richtige“ Algorithmus. Beide Algorithmen existieren und damit auch der „richtige“.

Unter allen Funktionen  $\{f^r : \mathbb{N}_0 \rightarrow \mathbb{N}_0 \mid r \in \mathbb{R}\}$  mit

$$f^r(n) = \begin{cases} 1, & \text{falls } n \text{ ein Präfix der Ziffern der Dezimalzahldarstellung von } r \text{ ist} \\ 0, & \text{sonst} \end{cases}$$

muss es nicht berechenbare Funktionen geben, da die Menge der Algorithmen abzählbar unendlich ist, es aber überabzählbar viele Funktionen  $f^r$  gibt und ein Algorithmus der  $f^r$  berechnet nicht auch  $f^s$  mit  $r \neq s$  berechnen kann.

Insbesondere in den 1930er Jahren wurden verschiedene Begriffe für die Berechenbarkeit vorgeschlagen (u.a. die  $\lambda$ -definierbaren Funktionen (Alonzo Church und Stephen Kleene), die Turingberechenbaren Funktionen (Alan Turing),  $\mu$ -rekursive Funktionen (Kurt Gödel und Jacques Herbrand)).

Es hat sich (erstaunlicherweise!) gezeigt, dass alle diese Begriffe ineinander überführbar und äquivalent sind und daher denselben Begriff der Berechenbarkeit definieren.

Da diese Äquivalenzen gelten und man bisher keinen mächtigeren, sinnvollen Begriff der Berechenbarkeit gefunden hat, geht man allgemein davon aus, dass man durch diese Formalismen *den* Begriff der Berechenbarkeit gefunden hat. Das bedeutet insbesondere, wenn man von einer Funktion gezeigt hat, dass sie nicht Turingberechenbar ist, dann geht man davon aus, dass sie überhaupt nicht berechenbar ist. Diese Überzeugung fasst man unter der sogenannten *Churchschen These* zusammen:

**Churchsche These:** Die Klasse der Turingberechenbaren Funktionen stimmt genau mit der Klasse der intuitiv berechenbaren Funktionen überein.

Es gibt keinen *Beweis* der Churchschen These, da man den Begriff „intuitiv berechenbar“ nicht formal fassen kann.

In den folgenden Abschnitten dieses Kapitels definieren wir zunächst den Begriff der Turingberechenbarkeit, der eine exakte Definition der Berechenbarkeit auf Grundlage der Turingmaschinen bietet. Anschließend betrachten wir einfache Erweiterungen der Turingmaschine. Im Anschluß definieren wir den zur Berechenbarkeit eng verwandten Begriff der Entscheidbarkeit, zeigen die Unentscheidbarkeit des berühmten Halteproblems und erläutern eine Technik, wie man die Unentscheidbarkeit eines beliebigen (unentscheidbaren) Problems nachweisen kann. Wir beenden das Kapitel mit einem weiteren berühmten unentscheidbaren Problem, welches unabhängig von Turingmaschinen formuliert ist.

## 7.2. Turingmaschinen und Turingberechenbarkeit

Wir haben die von Alan Turing vorgeschlagenen Turingmaschinen bereits in Kapitel 5 kennengelernt. In diesem Abschnitt definieren wir zunächst darauf aufbauend den Begriff der Turingberechenbarkeit einer Funktion. Im Anschluss betrachten wir Varianten von Turingmaschinen.

Wir wiederholen die wesentlichen Begriffe und Definitionen zu Turingmaschinen: Eine Turingmaschine ist ein 7-Tupel  $(Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ , wobei  $Z$  die endliche Menge der Zustände,  $\Sigma$  das Eingabealphabet,  $\Gamma \supset \Sigma$  das Bandalphabet,  $z_0 \in Z$  der Startzustand,  $\square \in (\Gamma \setminus \Sigma)$  das Blank-Symbol,  $E \subseteq Z$  die Menge der Endzustände, und  $\delta$  die Zustandsüberföhrungsfunktion (mit  $\delta : (Z \times \Gamma) \rightarrow (Z \times \Gamma \times \{L, R, N\})$  für eine DTM und  $\delta : (Z \times \Gamma) \rightarrow \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$  für eine NTM) ist. Konfigurationen sind Wörter der Form  $a_1 \cdots a_m z a_{m+1} \cdots a_n$ , wobei  $a_1 \cdots a_n \in \Gamma^*$

der entdeckte Teil des Bandinhalts ist, die endliche Steuerung der TM im Zustand  $z \in Z$  ist und der Schreib-Lesekopf der TM unter  $a_{m+1}$  steht.

Bisher haben wir Turingmaschinen verwendet, um Sprachen zu erkennen, und daher definiert, wann eine Turingmaschine ein Eingabewort akzeptiert (siehe Definition 5.1.5). Für die Definition der Berechenbarkeit von Funktionen (auf den natürlichen Zahlen, wie im letzten Kapitel) passen wir die Definition an, indem wir neben der Akzeptanz auch die Ausgabe (auf dem Band der Turingmaschine) beobachten. Wir geben auch gleich eine Definition an, die für Funktionen auf Wörtern einer formalen Sprache passend ist.

**Definition 7.2.1.** Sei  $\text{bin}(n)$  die Binärdarstellung von  $n \in \mathbb{N}_0$ . Eine Funktion  $f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$  heißt Turingberechenbar, falls es eine (deterministische) Turingmaschine  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  gibt, so dass für alle  $n_1, \dots, n_k, m \in \mathbb{N}_0$  gilt:

$$f(n_1, \dots, n_k) = m \text{ g.d.w. } z_0 \text{bin}(n_1) \# \dots \# \text{bin}(n_k) \vdash^* \square \dots \square z_e \text{bin}(m) \square \dots \square \text{ mit } z_e \in E.$$

Eine Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  heißt Turingberechenbar, falls es eine (deterministische) Turingmaschine  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  gibt, so dass für alle  $u, v \in \Sigma^*$  gilt:  $f(u) = v$  g.d.w.

$$z_0 u \vdash^* \square \dots \square z_e v \square \dots \square \text{ mit } z_e \in E.$$

Eine Konsequenz (für beide Definitionen der Turingberechenbarkeit) ist: Falls  $f(n_1, \dots, n_k)$  bzw.  $f(u)$  undefiniert ist, dann muss die Maschine in eine Endlosschleife gehen.

**Beispiel 7.2.2.** Die Nachfolgerfunktion  $f(x) = x + 1$  für alle  $x \in \mathbb{N}_0$  ist Turingberechenbar. Wir haben eine entsprechende Turingmaschine bereits in Beispiel 5.1.6 angegeben.

**Beispiel 7.2.3.** Die Identitätsfunktion  $f(x) = x$  für alle  $x \in \mathbb{N}_0$  ist Turingberechenbar: Für die Turingmaschine  $M = (\{z_0\}, \{0, 1, \#\}, \{0, 1, \#\square\}, \delta, z_0, \square, \{z_0\})$  mit  $\delta(z_0, a) = (z_0, a, N)$  für alle  $a \in \{0, 1, \#, \square\}$  gilt:  $z_0 \text{bin}(n) \vdash^* z_0 \text{bin}(n)$  für alle  $n \in \mathbb{N}_0$ .

Die überall undefinierte Funktion  $\Omega$  ist Turingberechenbar, da die Turingmaschine  $M = (\{z_0\}, \{0, 1, \#\}, \{0, 1, \#, \square\}, \delta, z_0, \square, \emptyset)$  mit  $\delta(z_0, a) = (z_0, a, N)$  für keine Eingabe akzeptiert (sondern stets in eine Endlosschleife geht).

Eine berühmte nicht-berechenbare Funktion ist die Radó-Funktion (benannt nach Tibor Radó):

$$\Sigma(n) = \text{Anzahl an Einsen, die ein Fleißiger Biebei mit } n \text{ Zuständen auf das Band schreibt}$$

Ein fleißiger Biebei (busy beaver im Englischen) mit  $n$  Zuständen ist eine (auf leerer Eingabe) anhaltende Turingmaschine, für die keine andere Turingmaschine bei leerer Eingabe mehr Einsen auf das Band schreibt und anhält. Dabei gelten einige Änderungen / Einschränkungen an die betrachteten Turingmaschinen mit  $n$  Zuständen:

- Die Turingmaschinen sind deterministisch.
- Das Bandalphabet ist  $\{\square, 1\}$ , wobei  $\square$  das Blank-Symbol ist.

- Die Kopfbewegungen sind nur  $L$  und  $R$  ( $N$  ist nicht erlaubt)
- Die Turingmaschine hat  $n$  Nichtendzustände und genau einen Endzustand. (d.h. in  $n$  zählt der Endzustand nicht mit).

Es gilt  $\Sigma(1) = 1$ ,  $\Sigma(2) = 4$ ,  $\Sigma(3) = 6$ ,  $\Sigma(4) = 13$ ,  $\Sigma(5) = 4098$ , wobei der Wert für 5 erst im Jahr 2024 nachgewiesen werden konnte<sup>1</sup>.

Für Werte größer als 5 ist  $\Sigma(n)$  unbekannt. Für  $n = 6$  ist bekannt, dass  $\Sigma(n)$  größer ist als der Potenzturm  $10^{10^{\dots^{10}}}$  mit 15 Zehnern, was bereits eine unvorstellbar große Zahl ist.

**Satz 7.2.4.** *Die Radó-Funktion ist nicht berechenbar.*

*Beweisskizze.* Nehme an  $\Sigma$  sei berechenbar und  $M_\Sigma$  sei die DTM, die  $\Sigma$  berechnet, wobei wir o.b.d.A. annehmen, dass  $M_\Sigma$  die Beschränkungen an die TMs in der Definition der fleißigen Bieber einhält. Insbesondere sei  $M_\Sigma$  schon derart abgeändert, dass sie das Bandalphabet  $\{\square, 1\}$  verwendet, und daher ihre Eingabe unär kodiert erwartet und die Ausgabe ebenfalls unär kodiert schreibt. Man überlege sich, dass das Einhalten der Einschränkungen durch leichtes Abändern stets bewerkstelligt werden kann. Konstruiere die folgenden Turingmaschinen, die alles samt die Beschränkungen in der Definition der fleißigen Bieber einhalten sollen:

- Eine TM  $M_{2^{x+1}}$ , die bei Eingabe des Wortes  $1^x$  die Ausgabe  $1^{2^{x+1}}$  auf das Band schreibt, d.h.  $M_1$  verdoppelt alle Einsen und schreibt noch eine zusätzliche Eins.
- Eine TM  $M_N$ , die  $1^N$  auf das Band schreibt und nicht mehr als  $N$  Nichtendzustände und einen Endzustand benutzt: Das ist möglich mit  $\delta(z_i, \square) = (z_{i+1}, 1, R)$ , Zuständen  $z_0, \dots, z_N$  und  $z_N$  ist Endzustand.

Nehme an,  $M_\Sigma$  hat  $k_1$  Nichtendzustände und  $M_{2^{x+1}}$  hat  $k_2$  Nichtendzustände. Sei  $N = k_1 + k_2$ . Sei  $M$  die TM, die  $M_N$ , dann  $M_1$  und dann  $M_\Sigma$  ausführt.  $M$  kann mit  $N + N = 2N$  Zuständen einfach konstruiert werden.

Bei leerer Eingabe schreibt  $M$ ,  $\Sigma(2N+1)$  Einsen auf das Band und hält. Das ist jedoch unmöglich, denn  $M$  hat nur  $2N$  Nichtendzustände, müsste jedoch nach Definition von  $\Sigma$  mindestens  $2N + 1$  Zustände besitzen. Ein Widerspruch! Da  $M_{2^{x+1}}$  und  $M_N$  existieren, war unsere Annahme falsch:  $M_\Sigma$  existiert nicht und  $\Sigma$  ist nicht berechenbar.  $\square$

### 7.2.1. Mehrspuren- und Mehrband-Turingmaschinen

Unsere bisher eingeführten Turingmaschinen haben ein Band, das zur Eingabe, zur Verarbeitung, und zum Erzeugen der Ausgabe verwendet wird. In diesem Abschnitt betrachten wir zwei Varianten von Turingmaschinen, die etwas mehr Möglichkeiten bieten und sich dadurch oft „einfacher“ programmieren lassen. Wir zeigen aber, dass sich die Varianten wieder auf die 1-Band-Turingmaschinen zurückführen lassen. Dies zeigt, dass die Varianten den Berechenbarkeitsbegriff unverändert lassen.

<sup>1</sup>siehe <https://bbchallenge.org/>, wobei die dort für  $n = 5$  berechnete Zahl  $BB(n)$  die maximale Anzahl an Schritten einer haltenden DTM mit  $n$  Zuständen bestimmt, diese jedoch eng verwandt zu  $\Sigma(n)$  ist.

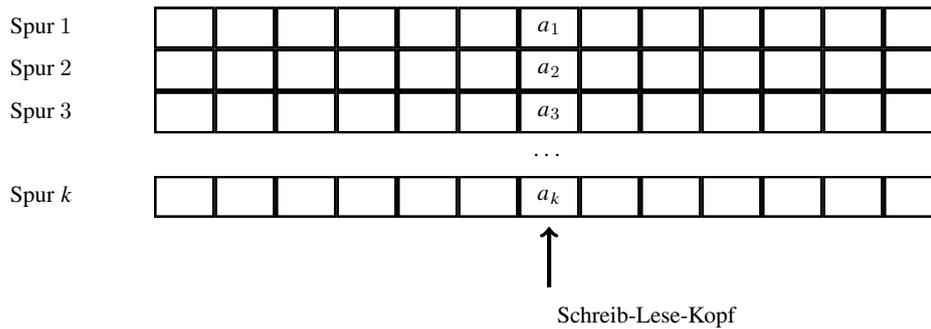


Abbildung 7.1.: Mehrspuren-Turingmaschine

### 7.2.2. Mehrspuren-Turingmaschinen

**Definition 7.2.5** (Mehrspuren-Turingmaschine). Für  $k \in \mathbb{N}$  ist eine  $k$ -Spuren-Turingmaschine ein 7-Tupel  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  wobei

- $Z$  ist eine endliche Menge von Zuständen,
- $\Sigma$  ist das (endliche) Eingabealphabet,
- $\Gamma \supset \Sigma$  ist das (endliche) Bandalphabet,
- für eine deterministische TM ist  $\delta : (Z \times \Gamma^k) \rightarrow (Z \times \Gamma^k \times \{L, R, N\})$  die Zustandsübergangsfunktion und für eine nichtdeterministische TM ist  $\delta : (Z \times \Gamma^k) \rightarrow \mathcal{P}(Z \times \Gamma^k \times \{L, R, N\})$  die Zustandsübergangsfunktion,
- $z_0 \in Z$  ist der Startzustand,
- $\square \in \Gamma \setminus \Sigma$  ist das Blank-Symbol und
- $E \subseteq Z$  ist die Menge der Endzustände.

Verglichen mit der Definition einer herkömmlichen (1-Band und 1-Spuren-TM) ist die Idee der  $k$ -Spuren-TM, dass das Band der Turingmaschine in  $k$  übereinander liegende Spuren geteilt ist. Abb. 7.1 illustriert diese Idee:

Die Turingmaschine liest ein Tupel  $(a_1, \dots, a_k)$  und kann dementsprechend agieren. Für den Begriff der Berechenbarkeit nehmen wir an, dass die Eingabe auf der ersten Spur der Mehrspurenmaschine liegt und alle anderen Spuren leer sind. Die Ausgabe wird ebenfalls von der ersten Spur (in einem Akzeptanzzustand) entnommen.

**Satz 7.2.6.** Jede Mehrspuren-Turingmaschine kann auf einer 1-Band-Turingmaschine simuliert werden.

*Beweis.* Verwende als Bandalphabet  $\Gamma \cup \Gamma^k$ , als Eingabealphabet  $\Sigma$  und als Blank-Symbol  $\square$ . Beginnend mit einer Eingabe aus  $\Sigma^*$ , kann die Mehrspurendarstellung erzeugt werden, indem jedes Symbol  $a \in \Sigma$  aus der Eingabe durch das  $k$ -Tupel  $(a, \square, \dots, \square)$  ersetzt wird. Anschließend

kann jeder Berechnungsschritt der Mehrspurenmaschine auf der 1-Band-Maschine simuliert werden und schließlich die Ausgabe erzeugt werden, indem jeder Eintrag  $(a_1, \dots, a_k)$  durch  $a_1$  ersetzt wird. Die Akzeptanz der Simulation ist dieselbe und auch die Turingberechenbarkeit bleibt dieselbe.  $\square$

**Beispiel 7.2.7.** Das folgende Beispiel ist in ähnlicher Weise in (HMU06) zu finden. Wir programmieren eine Turingmaschine, welche die Sprache  $\{w c w \mid w \in \{a, b\}^+\}$  erkennt. Dafür verwenden wir 2-Spuren und verzichten auf das Erstellen und Entfernen der 2-Spurenarstellung zu Beginn und am Ende. Stattdessen geben wir direkt eine 2-Spuren-Turingmaschine an, welche die Sprache  $\{w(c, \square)w \mid w \in \{(a, \square), (b, \square)\}^+\}$  akzeptiert. Die Ideen dabei sind:

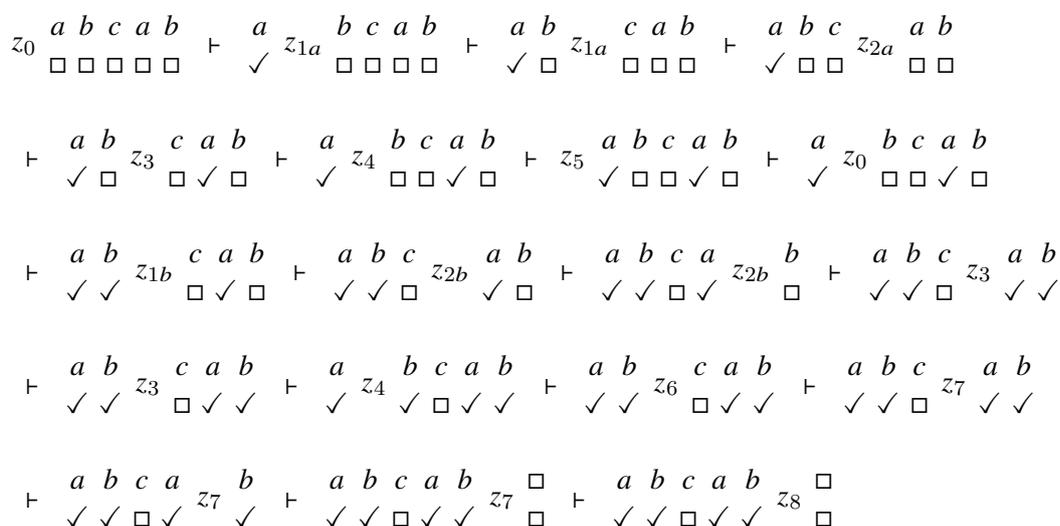
- Die Eingabe  $w c w$  auf Spur 1 wird nur gelesen, aber nicht verändert.
- Auf der zweiten Spur wird nur das Blank-Symbol und das Symbol  $\checkmark$  zum Markieren von Buchstaben auf Spur 1 verwendet.
- Das Markieren geschieht Zeichenweise von links nach rechts: Ein Zeichen wird im linken  $w$  markiert, dann im rechten  $w$  gesucht und dann dort markiert, usw.

Wir verwenden Zustände  $Z = \{z_0, z_{1a}, z_{1b}, z_{2a}, z_{2b}, z_3, \dots, z_9\}$ . Dabei „speichern“ die Zustände  $z_{1a}, z_{1b}, z_{2a}, z_{2b}$  das gelesene Zeichen ( $a$  oder  $b$ ). Der Zustand  $z_8$  ist der einzig akzeptierende, der Zustand  $z_9$  wird als Müllzustand verwendet (zum Verwerfen) und  $z_0$  ist der Startzustand. Die Übergangsfunktion  $\delta$  ist wie folgt definiert:

- $\delta(z_0, (s, \square)) = (z_{1s}, (s, \checkmark), R)$  für  $s \in \{a, b\}$  markiert das linkeste unmarkierte Zeichen  $s$  im linken  $w$ , dabei wird  $s$  im Zustand  $z_{1s}$  gespeichert. Die Suche nach  $s$  im rechten  $w$  startet.
- $\delta(z_{1s}, (s', \square)) = (z_{1s}, (s', \square), R)$  für  $s, s' \in \{a, b\}$  um nach rechts zu bis zum Zeichen  $c$  zu laufen (alle übersprungenen Zeichen sind unmarkiert).
- $\delta(z_{1s}, (c, \square)) = (z_{2s}, (c, \square), R)$  mit  $s \in \{a, b\}$  für den Fall das  $c$  erreicht wurde.
- $\delta(z_{2s}, (s', \checkmark)) = (z_{2s}, (s', \checkmark), R)$  mit  $s, s' \in \{a, b\}$ , um zum ersten unmarkierten Symbol im rechten  $w$  zu gelangen.
- $\delta(z_{2s}, (s, \square)) = (z_3, (s, \checkmark), L)$  mit  $s \in \{a, b\}$ . Das erste unmarkierte Symbol im rechten  $w$  wurde gefunden und stimmt mit dem gesuchten Zeichen überein.
- $\delta(z_3, (s, \checkmark)) = (z_3, (s, \checkmark), L)$  mit  $s \in \{a, b\}$  zum nach links laufen zum  $c$ .
- $\delta(z_3, (c, \square)) = (z_4, (c, \square), L)$  wenn  $c$  wieder erreicht wurde. Das Zeichen links vom  $c$  bestimmt nun den weiteren Verlauf.
- $\delta(z_4, (s, \square)) = (z_5, (s, \square), L)$  mit  $s \in \{a, b\}$ : Links vom  $c$  stand kein markiertes Zeichen und es wird mit  $z_5$  fortgefahren.
- $\delta(z_4, (s, \checkmark)) = (z_6, (s, \square), R)$  mit  $s \in \{a, b\}$ : Links vom  $c$  stand ein markiertes Zeichen. Dann muss geprüft werden, ob rechts vom  $c$  alle Zeichen markiert sind. Das macht Zustand  $z_6$ .
- $\delta(z_5, (s, \square)) = (z_5, (s, \square), L)$  mit  $s \in \{a, b\}$  setzt die Suche nach dem ersten markierten Zeichen im linken  $w$  fort.

- $\delta(z_5, (s, \checkmark)) = (z_0, (s, \checkmark), R)$  mit  $s \in \{a, b\}$ : Das erste markierte Zeichen im linken  $w$  wurde gefunden. Nun kann rechts davon mit dem Startzustand  $z_0$  fortgefahren werden.
- $\delta(z_6, (c, \square)) = (z_7, (c, \square), R)$ . In  $z_6$  steht der Schreib-Lesekopf direkt auf dem  $c$  und die TM sucht nun im rechten  $w$  nach unmarkierten Zeichen.
- $\delta(z_7, (s, \checkmark)) = (z_7, (s, \checkmark), R)$  für  $s \in \{a, b\}$  setzt die Suche nach unmarkierten Zeichen im rechten  $w$  fort.
- $\delta(z_7, (\square, \square)) = (z_8, (\square, \square), N)$ : Alle Zeichen im rechten  $w$  waren markiert. Die TM geht in den akzeptierenden Zustand  $z_8$ .
- $\delta(z_8, (s, t)) = (z_8, (s, t), N)$  für  $s \in \{a, b, c\}$ ,  $t \in \{\square, \checkmark\}$  verbleibe akzeptierend.
- $\delta(z_i, (s, t)) = (z_9, (s, t), N)$  für alle anderen Fälle verbleibe oder wechsele in den verwerfenden Zustand.

Ein Lauf der Turingmaschine für das Wort  $abcab$  ist:



### 7.2.3. Mehrband-Turingmaschinen

Als nächstes betrachten wir als Erweiterung der 1-Band-Turingmaschinen, die Mehrband-Turingmaschinen. Diese verwenden mehrere Bänder, wobei jedes Band *einen eignen Schreib-Lesekopf* hat. In jedem Schritt kann jeder dieser Köpfe in eine der Richtungen  $L$ ,  $R$ ,  $N$  unabhängig von den anderen Köpfen bewegt werden.

Eine Illustration der Mehrband-Turingmaschine ist in Abb. 7.2 zu finden.

**Definition 7.2.8** (Mehrband-Turingmaschine). Für  $k \in \mathbb{N}$  ist eine  $k$ -Band-Turingmaschine ein 7-Tupel  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  wobei

- $Z$  ist eine endliche Menge von Zuständen,
- $\Sigma$  ist das (endliche) Eingabealphabet,

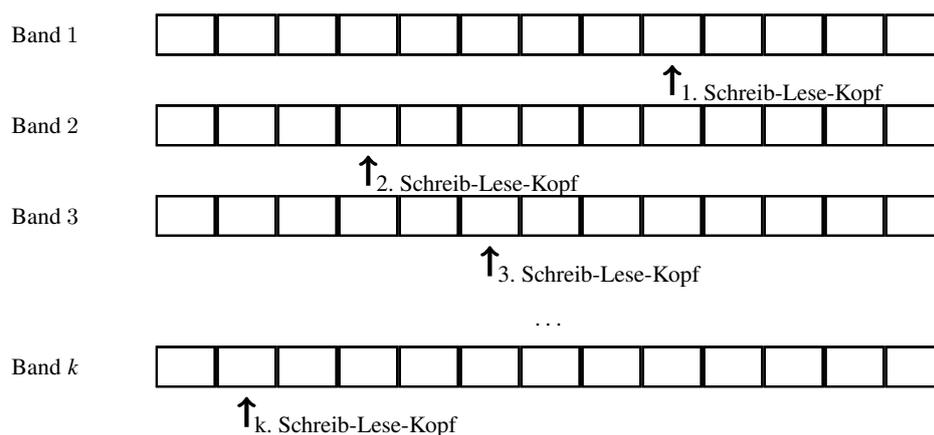


Abbildung 7.2.: Illustration der Mehrband-Turingmaschine

- $\Gamma \supset \Sigma$  ist das (endliche) Bandalphabet,
- für eine deterministische TM ist  $\delta : (Z \times \Gamma^k) \rightarrow (Z \times \Gamma^k \times \{L, R, N\}^k)$  die Zustandsüberföhrungsfunktion und für eine nichtdeterministische TM ist  $\delta : (Z \times \Gamma^k) \rightarrow \mathcal{P}(Z \times \Gamma^k \times \{L, R, N\}^k)$  die Zustandsüberföhrungsfunktion
- $z_0 \in Z$  ist der Startzustand,
- $\square \in \Gamma \setminus \Sigma$  ist das Blank-Symbol
- $E \subseteq Z$  ist die Menge der Endzustände.

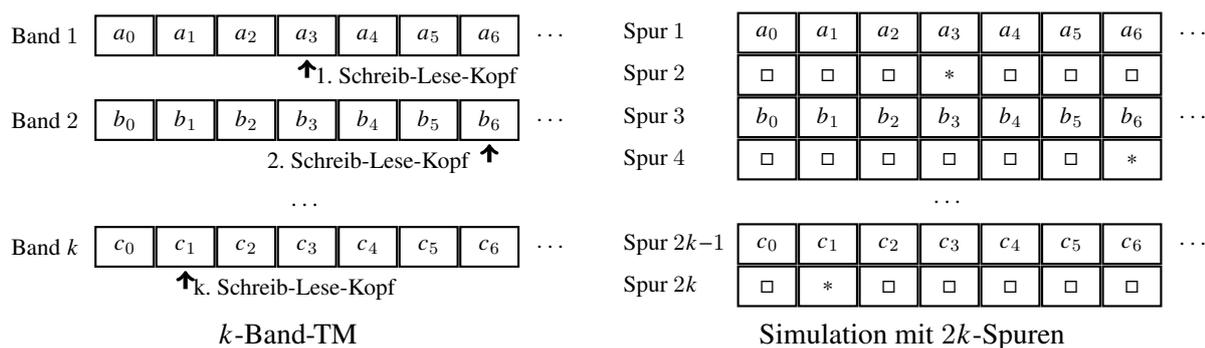
Zur Berechnung (von Funktionswerten) nehmen wir für die  $k$ -Band-TM an, dass die Eingabe auf dem ersten Band steht und alle anderen Bänder leer sind.

**Theorem 7.2.9.** Jede Mehrband-Turingmaschine kann von einer 1-Band TM simuliert werden.

*Beweis.* Sei  $M$  eine  $k$ -Band-TM. Wenn  $k = 1$ , dann ist dies eine 1-Band-TM und die Aussage ist trivial. Anderenfalls ( $k > 1$ ) verwenden wir eine  $2 \cdot k$ -Spuren-TM um  $M$  zu simulieren. Dabei repräsentieren jeweils 2 aufeinanderfolgende Spuren ein Band von  $k$ . Sei  $i$  ein Band von  $M$ , dann speichert die erste dazugehörige Spur den Inhalt von Band  $i$  und die zweite Spur speichert die Position des  $i$ -ten Schreib-Lesekopfs von  $M$ , indem sie an der entsprechenden Position ein  $*$  auf die Spur schreibt (und alle anderen Einträge mit dem Blank-Symbol belegt). Abb. 7.3 illustriert diese Darstellung.

Für eine Eingabe  $w \in \Sigma^*$  auf dem Eingabeband der Mehrband-Maschine, erzeugt die 1-Band-Maschine für dieselbe Eingabe die Darstellung mit  $2k$ -Spuren und simuliert anschließend die Berechnungsschritte. Bei Akzeptanz der Mehrband-TM transformiert sie die Spurendarstellung in die Darstellung der Ausgabe.

Zur Simulation eines Berechnungsschrittes auf der Mehrband-TM liest die 1-Band-TM zunächst einmal komplett den verwendeten Bandbereich von links nach rechts und speichert dabei (durch

Abbildung 7.3.: Simulation der Turingmaschine mit  $k$ -Bändern

Zustände) die Kopfpositionen aller  $k$  Köpfe. Im Anschluss „weiß“ sie, welchen Schritt die Mehrband-TM machen will, und macht alle Anpassungen auf allen Schichten (Anpassen der Bandinhalte und der Kopfpositionen). Im Anschluss kann der nächste Schritt starten.  $\square$

### 7.3. Unentscheidbarkeit

In diesem Kapitel definieren wir basierend auf dem Begriff der Berechenbarkeit die Entscheidbarkeit von Problemen und betrachten im Anschluss das Halteproblem für Turingmaschinen (und Varianten davon) und zeigen, dass dieses Problem nicht entscheidbar ist. Danach erläutern wir die Technik, ein Problem auf ein anderes zu reduzieren und zeigen damit die Unentscheidbarkeit von weiteren Problemen.

**Definition 7.3.1.** Eine Sprache  $L \subseteq \Sigma^*$  heißt entscheidbar, wenn die charakteristische Funktion von  $L$ ,  $\chi_L : \Sigma^* \rightarrow \{0, 1\}$  mit

$$\chi_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{falls } w \notin L \end{cases}$$

berechenbar ist.

Eine Sprache heißt semi-entscheidbar falls  $\chi'_L : \Sigma^* \rightarrow \{0, 1\}$  mit

$$\chi'_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ \text{undefiniert}, & \text{falls } w \notin L \end{cases}$$

berechenbar ist.

Für Mengen  $L \subseteq \mathbb{N}_0$  lässt sich obige Definition sinngemäß anwenden.

Algorithmisch gesehen muss für den Fall der Entscheidbarkeit ein immer anhaltender Algorithmus formuliert werden, während im Falle der Semi-Entscheidbarkeit ein Algorithmus ausreicht, der nur im Erfolgsfall ( $w \in L$ ) anhält und ansonsten unendlich lange läuft.

**Satz 7.3.2.** Eine Sprache  $L$  ist genau dann entscheidbar, wenn  $L$  und  $\bar{L}$  jeweils semi-entscheidbar sind.

*Beweis.* Aus einer Turingmaschine, die  $\chi_L$  berechnet, können leicht TMs konstruiert werden, die  $\chi'_L$  und  $\chi''_L$  berechnen. Umgekehrt kann aus zwei TMs  $M_L$  und  $M_{\bar{L}}$ , die  $\chi'_L$  und  $\chi''_L$  berechnen, eine TM konstruiert werden, die sich wie folgt verhält und damit  $\chi_L$  berechnet:

Starte mit  $i = 1$ .  
 Simuliere  $i$ -Schritte von  $M_L$ .  
 Wenn diese akzeptiert, dann akzeptiere mit Ausgabe 1.  
 Ansonsten simuliere  $i$ -Schritte von  $M_{\bar{L}}$ .  
 Wenn diese akzeptiert, dann akzeptiere mit Ausgabe 0.  
 Ansonsten erhöhe  $i$  um 1 und starte von neuem.

□

**Korollar 7.3.3.** Wenn  $L$  entscheidbar ist, dann ist auch  $\bar{L}$  entscheidbar.

**Definition 7.3.4.** Eine Sprache  $L \subseteq \Sigma^*$  heißt rekursiv aufzählbar, falls  $L = \emptyset$  oder falls es eine totale berechenbare Funktion  $f : \mathbb{N}_0 \rightarrow \Sigma^*$  gibt, sodass  $L = \bigcup_{i \in \mathbb{N}_0} f(i)$ . Man sagt dann „ $f$  zählt  $L$  auf“.

**Lemma 7.3.5.** Die Sprache  $\Sigma^*$  ist rekursiv-aufzählbar.

*Beweis.* Sei  $|\Sigma| = b$ . Sei  $n \in \mathbb{N}_0$ . Interpretiere ein Wort  $w \in \Sigma^*$  als  $b + 1$ -äre Zahl. Konstruiere eine TM, die auf einem Band, die Eingabe  $n$  in Binärdarstellung erhält und auf dem anderen Band die  $b + 1$ -äre Zahl 0. Anschließend zählt die TM, die Binärzahl herunter und erhöht bei jedem Herunterzählen die  $b + 1$ -äre Zahl um 1. Wenn der Binärzähler auf 0 ist, dann enthält das andere Band das Wort  $f(n)$ . □

Im folgenden sei  $c(x, y)$  die Kodierung eines Paares von natürlichen Zahlen als natürliche Zahl,  $left(c(x, y))$  sei  $x$  und  $right(c(x, y))$  sei  $y$  (siehe z.B. (Sch08) für die Existenz einer solchen Kodierung).

**Satz 7.3.6.** Eine Sprache ist genau dann rekursiv aufzählbar, wenn sie semi-entscheidbar ist.

*Beweis.* Sei die Sprache  $L$  rekursiv aufzählbar und  $f$  die totale und berechenbare Funktion, die  $L$  aufzählt. Der Algorithmus

**Für**  $i = 0, 1, 2, 3, \dots$  **tue**  
**wenn**  $f(i) = w$  **dann**  
 stoppe und gebe 1 aus

berechnet  $\chi'_L(w)$ .

Sei nun  $L$  eine semi-entscheidbare Sprache und  $M$  eine TM, die  $\chi'_L$  berechnet. Wenn  $L = \emptyset$ , dann ist  $L$  rekursiv-aufzählbar.

Anderenfalls sei  $u \in L$  ein Wort. Wir müssen zeigen, dass es eine totale und berechenbare Funktion  $f$  gibt, die  $L$  aufzählt. Wir konstruieren eine TM  $M'$ . Sei  $n$  eine Eingabe. Wir interpretieren,

diese als  $c(x, y)$  und berechnen mit  $left(n)$  und  $right(n)$  die Komponenten  $x$  und  $y$ . Anschließend prüfen wir, ob TM  $M$  bei Eingabe  $g(x)$  nach  $y$  Schritten akzeptiert ( $M'$  simuliert  $y$  Schritte von  $M$ ). Hierbei ist  $g(x)$  das zu  $x$  zugehörige Wort aus  $\Sigma^*$  (dieses wird zuvor von  $M'$  berechnet, siehe Lemma 7.3.5). Ist dies der Fall, so akzeptiert die TM  $M'$  mit Ausgabe  $g(x)$ . Anderenfalls akzeptiert die TM  $M'$  mit Ausgabe  $u$ .

Die TM  $M'$  hält daher für jede Eingabe und berechnet eine totale Funktion  $f$ :

$$f(n) = \begin{cases} w \in L, & \text{falls } w = g(left(n)) \text{ und } M \text{ akzeptiert } w \text{ in } right(n) \text{ Schritten} \\ u \in L, & \text{sonst} \end{cases}$$

Für jedes  $w \in L$  gibt es eine Zahl  $x$ , sodass  $w = g(x)$ , und eine Zahl  $y$ , sodass  $M$  nach  $y$  Schritten bei Eingabe  $w$  anhält und dieses Wort akzeptiert. Dann zählt  $M'$  das Wort  $w$  bei Eingabe  $c(x, y)$  auf. Daher ist  $L$  aufzählbar.  $\square$

Insgesamt zeigen unsere bisherigen Definitionen und Resultate, dass die folgenden Aussagen äquivalent sind:

- $L$  ist vom Typ 0.
- $L$  ist semi-entscheidbar.
- $L$  ist rekursiv-aufzählbar.
- Es gibt eine Turingmaschine  $M$ , die  $L$  akzeptiert (d.h.  $L(M) = L$ ).
- $\chi'_L$  ist Turing-berechenbar.
- Es gibt berechenbare Funktionen, die  $L$  als Wertebereich (nämlich die  $L$  aufzählende Funktion) bzw. als Definitionsbereich (nämlich  $\chi'_L$ ) haben.

Beachte, dass die rekursive Aufzählbarkeit im Allgemeinen nicht dasselbe ist wie Abzählbarkeit (siehe auch Anhang A.1.4): Eine Sprache  $L$  ist abzählbar, wenn es eine totale Funktion  $f : \mathbb{N}_0 \rightarrow L$  gibt, sodass  $\bigcup_{i \in \mathbb{N}_0} f(i) = L$ . Der wesentliche Unterschied ist, dass Abzählbarkeit nicht die Berechenbarkeit von  $f$  fordert. Daher sind die Begriffe Abzählbarkeit und rekursive Aufzählbarkeit verschieden.

### 7.3.1. Gödelisierung von Turingmaschinen

Wir erläutern, wie Turingmaschinenbeschreibungen als Zahlen in Binärdarstellung, d.h. als Worte über dem Alphabet  $\{0, 1\}$  dargestellt werden können. Wir wollen dies bewerkstelligen, um im Anschluss daran, diese Beschreibungen als Eingaben anderer Turingmaschinen zu verwenden.

Sei  $(Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  eine deterministische Turingmaschine mit  $\Sigma = \{0, 1\}$ . Wir nehmen an, dass  $\Gamma$  und  $Z$  durchnummeriert sind:

- $\Gamma = \{a_0, \dots, a_k\}$
- $Z = \{z_0, \dots, z_n\}$

und dass (o.B.d.A)  $a_0 = \square$ ,  $a_1 = \#$ ,  $a_2 = 0$ ,  $a_3 = 1$ ,  $z_0$  der Startzustand ist und  $E = \{z_n\}$  gilt.

Jeder Regel  $\delta(z_p, a_i) = (z_q, a_j, D)$  ordnen wir das Wort

$$w_{p,i,q,j,D} = \#\#bin(p)\#bin(i)\#bin(q)\#bin(j)\#bin(D_m)$$

mit

$$D_m = \begin{cases} 0, & \text{falls } D = L \\ 1, & \text{falls } D = R \\ 2, & \text{falls } D = N \end{cases}$$

über dem Alphabet  $\{0, 1, \#\}$  zu.

Alle zu  $\delta$ -zugehörigen Worte, werden (in irgendeiner Reihenfolge) hintereinander geschrieben. Wir bezeichnen dieses Wort als  $w_\delta$ .

Schließlich kodieren wir das Alphabet  $\{0, 1, \#\}$  durch  $\{0 \mapsto 00, 1 \mapsto 01, \# \mapsto 11\}$ . Wir bezeichnen die so kodierte Turingmaschine  $M$  als Wort  $w_M$ .

Nicht jedes Wort über dem Alphabet  $\{0, 1\}$  entspricht der Kodierung einer Turingmaschine. Sei  $\widehat{M}$  eine beliebige aber feste Turingmaschine. Dann können wir für jedes Wort  $w \in \{0, 1\}^*$  die zugehörige Turingmaschine  $M_w$  festlegen als

$$M_w := \begin{cases} M, & \text{wenn } w = w_M \\ \widehat{M}, & \text{sonst} \end{cases}$$

**Beispiel 7.3.7.** *Das Problem  $BB := \{w \in \{0, 1\}^* \mid M_w \text{ ist ein fleißiger Bieher}\}$  ist unentscheidbar: Wenn  $BB$  entscheidbar, dann gibt es eine TM  $M$ , die  $\chi_{BB}$  berechnet. Dann könnten wir jedoch auch die Radó-Funktion  $\Sigma$  berechnen: Die TM  $M'$  zählt bei Eingabe  $n$ , alle Turingmaschinen mit  $n$  Zuständen auf und prüft durch Simulation von  $M$ , ob diese ein fleißiger Bieher ist. Ist dies der Fall, so simuliert sie den fleißigen Bieher und zählt die ausgegeben Einsen. Das  $\Sigma$  nicht berechenbar ist, ist dies ein Widerspruch.*

### 7.3.2. Das Halteproblem

#### 7.3.2.1. Unentscheidbarkeit des speziellen Halteproblems

**Definition 7.3.8** (Spezielles Halteproblem). *Das spezielle Halteproblem ist die Sprache*

$$K := \{w \in \{0, 1\}^* \mid M_w \text{ hält für Eingabe } w\}$$

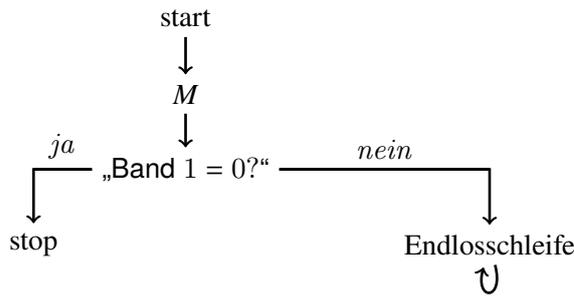
**Satz 7.3.9.** *Das spezielle Halteproblem ist nicht entscheidbar (und damit unentscheidbar).*

*Beweis.* Nimm an,  $K$  ist entscheidbar. Dann ist  $\chi_K$  berechenbar. D.h. es gibt eine TM  $M$ , die  $\chi_K$  berechnet.

Wir konstruieren eine TM  $M'$ , die zunächst  $M$  ablaufen lässt und anschließend:

- Wenn  $M$  mit 0 auf dem Band endet, dann hält  $M'$  an und akzeptiert.
- Wenn  $M$  mit 1 auf dem Band endet, dann läuft  $M'$  in eine Endlosschleife.

Illustriert durch das folgende Bild:



Betrachte nun  $M'$  auf der Eingabe  $w_{M'}$ : Es gilt

- $M'$  hält für Eingabe  $w_{M'}$
- g.d.w.  $M$  angesetzt auf  $w_{M'}$  gibt 0 aus
- g.d.w.  $\chi_K(w_{M'}) = 0$
- g.d.w.  $w_{M'} \notin K$
- g.d.w.  $M'$  hält nicht für Eingabe  $w_{M'}$

Das ergibt

$$M' \text{ hält für Eingabe } w_{M'} \iff M' \text{ hält nicht für Eingabe } w_{M'}$$

und stellt daher einen Widerspruch dar. D.h. unsere Annahme war falsch,  $K$  ist nicht entscheidbar.  $\square$

Der obige Beweis benutzt implizit ein Diagonalisierungsargument, dass wir genauer erläutern: Betrachte die folgende Tabelle:

	$w_1$	$w_2$	$w_3$	$w_4$	...
$M_{w_1}$	<i>ja</i>	<i>nein</i>	<i>ja</i>	...	...
$M_{w_2}$	<i>nein</i>	<i>nein</i>	<i>ja</i>	...	...
$M_{w_3}$	<i>ja</i>	<i>nein</i>	<i>ja</i>	...	...
$M_{w_4}$	...	...	...	...	...
...	...	...	...	...	...

Als Spaltenüberschriften enthält sie alle Worte über  $\{0,1\}$  (indem sie aufgezählt werden) und als Zeilenbeschriftungen alle Turingmaschinen  $M_{w_1}, M_{w_2}, \dots$  die sich aus der Beschreibung von  $w_i$  ergeben. Nehme an, die Tabelleneinträge in Zeile  $i$  und Spalte  $j$  sind *ja*, wenn  $M_{w_i}$  bei Eingabe  $w_j$  akzeptiert und hält und *nein* anderenfalls. Sei  $L_D$  die Sprache  $L_D = \{w_i \mid M_{w_i} \text{ hält nicht für Eingabe } w_i\}$ . Sei  $w_D$  die Beschreibung einer Turingmaschine, die  $L_D$  entscheidet und  $M_{w_D}$  die Turingmaschine dazu. Dann verhält sich die  $M_{w_D}$  genau gegensätzlich zur Diagonalen der obigen Tabelle. Irgendwann jedoch wird in der Tabelle auch das Wort  $w_D$  selbst aufgezählt, und der Eintrag in der Spalte für  $w_D$  und der Zeile für  $M_{w_D}$  wäre

das Gegenteil zum selben Eintrag:

	$w_1$	$w_2$	$w_3$	$w_4$	...	$w_D$
$M_{w_1}$	<b>ja</b>	<i>nein</i>	<i>ja</i>	...	...	...
$M_{w_2}$	<i>nein</i>	<b>nein</b>	<i>ja</i>	...	...	...
$M_{w_3}$	<i>ja</i>	<i>nein</i>	<b>ja</b>	...	...	...
$M_{w_4}$	...	...	...	...	...	...
...	...	...	...	...	...	...
$M_{w_D}$	<i>nein</i>	<i>ja</i>	<i>nein</i>	...	...	<b>?</b>
...						

Daher entsteht der Widerspruch und es ergibt sich, dass die Beschreibung  $w_D$  nicht existieren kann. Die Sprache  $L_D$  ist daher nicht entscheidbar (und auch nicht semi-entscheidbar).

### 7.3.2.2. Reduktionen

Wir führen ein „Hilfsmittel“ ein, um Unentscheidbarkeit von Sprachen  $L$  nachzuweisen. Hierbei wird eine bereits als unentscheidbar bekannte Sprache  $L_0$  verwendet und gezeigt: Wenn die neu zu betrachtende Sprache  $L$  entscheidbar wäre, dann wäre auch  $L_0$  entscheidbar. Da  $L_0$  aber bereits als unentscheidbar bekannt ist, kann auch  $L$  nicht entscheidbar sein.

**Definition 7.3.10** (Reduktion (einer Sprache auf eine andere)). Sei  $L_1 \subseteq \Sigma_1^*$  und  $L_2 \subseteq \Sigma_2^*$  Sprachen. Dann sagen wir  $L_1$  ist auf  $L_2$  *reduzierbar* (geschrieben  $L_1 \leq L_2$ ), falls es eine totale und berechenbare Funktion  $f : \Sigma_1^* \rightarrow \Sigma_2^*$  gibt, sodass für alle  $w \in \Sigma_1^*$  gilt:  $w \in L_1 \iff f(w) \in L_2$ .

**Satz 7.3.11.** Wenn  $L_1 \leq L_2$  und  $L_2$  entscheidbar (bzw. semi-entscheidbar) ist, dann ist auch  $L_1$  entscheidbar (bzw. semi-entscheidbar).

*Beweis.* Wir betrachten nur den Fall der Entscheidbarkeit. Der Beweis für die Semi-Entscheidbarkeit ist analog. Sei  $f$  die  $L_1 \leq L_2$  bezeugende (und berechenbare) Funktion. Da  $L_2$  entscheidbar ist, ist  $\chi_{L_2}$  berechenbar. Damit ist auch  $\chi_{L_1}(w) := \chi_{L_2}(f(w))$  berechenbar. Offensichtlich gilt

$$\chi_{L_1}(w) := \begin{cases} 1, & w \in L_1 \\ 0, & w \notin L_1 \end{cases} = \begin{cases} 1, & f(w) \in L_2 \\ 0, & f(w) \notin L_2 \end{cases} = \chi_{L_2}(f(w))$$

□

Mit Kontraposition folgt aus Satz 7.3.11:

**Lemma 7.3.12.** Sei  $L_1 \leq L_2$  und  $L_1$  ist unentscheidbar. Dann ist auch  $L_2$  unentscheidbar.

Wir verwenden meistens die Variante in Lemma 7.3.12, um Unentscheidbarkeit von Problemen zu zeigen. Das Vorgehen dabei ist:

- $L_1$  sei eine bekannt unentscheidbare Sprache
- Reduziere  $L_1$  auf  $L_2$  durch Angabe einer berechenbaren Funktion  $f$  mit  $w \in L_1 \iff f(w) \in L_2$ .
- Damit folgt, dass  $L_2$  unentscheidbar ist.

## 7.3.2.3. Unentscheidbarkeit des Halteproblems

**Definition 7.3.13** ((Allgemeines) Halteproblem). *Das (allgemeine) Halteproblem ist definiert als  $H := \{w\#x \mid \text{TM } M_w \text{ hält für Eingabe } x\}$*

**Satz 7.3.14.** *Das allgemeine Halteproblem ist unentscheidbar.*

*Beweis.* Wir reduzieren das spezielle Halteproblem auf das allgemeine Halteproblem, und zeigen daher  $K \leq H$ . Sei  $f(w) = w\#w$ . Dann gilt

$$\begin{aligned} & w \in K \\ \text{g.d.w. } & M_w \text{ hält für Eingabe } w \\ \text{g.d.w. } & w\#w \in H \\ \text{g.d.w. } & f(w) \in H \end{aligned}$$

Offensichtlich kann  $f$  durch eine TM berechnet werden und ist daher berechenbar.  $\square$

## 7.3.2.4. Halteproblem bei leerer Eingabe

**Definition 7.3.15.** *Das Halteproblem auf leerem Band ist die Sprache*

$$H_0 = \{w \mid M_w \text{ hält für die leere Eingabe}\}.$$

**Satz 7.3.16.** *Das Halteproblem auf leerem Band ist unentscheidbar.*

*Beweis.* Wir reduzieren das Halteproblem auf das Halteproblem auf leerem Band und zeigen daher  $H \leq H_0$ . Sei  $f(w_M\#x) = w_{M_{0,x}}$ , wobei  $M_{0,x}$  die Turingmaschine ist, die sich wie  $M$  verhält aber vorher  $x$  auf das Eingabeband schreibt. Dann gilt

$$\begin{aligned} & w_M\#x \in H \\ \text{g.d.w. } & M_w \text{ hält für Eingabe } x \\ \text{g.d.w. } & M_{0,x} \text{ hält für die leere Eingabe} \\ \text{g.d.w. } & w_{M_{0,x}} \in H_0 \\ \text{g.d.w. } & f(w_M\#x) \in H_0 \end{aligned}$$

Die Funktion  $f$  kann durch eine TM berechnet werden und ist daher berechenbar.  $\square$

### 7.4. Das Postsche Korrespondenzproblem

Das Problem ist das Postsche Korrespondenzproblem (PCP, Post Correspondence Problem) welches von Emil Post im Jahr 1946 vorgeschlagen wurde (Pos46).

**Definition 7.4.1** (Postsches Korrespondenzproblem). *Gegeben sei ein Alphabet  $\Sigma$  und eine Folge von Wortpaaren  $K = ((x_1, y_1), \dots, (x_k, y_k))$  mit  $x_i, y_i \in \Sigma^+$ . Das Postsche Korrespondenzproblem (PCP) ist die Frage, ob es für die gegebene Folge  $K$  eine Folge von Indizes  $i_1, \dots, i_m$  mit  $i_j \in \{1, \dots, k\}$  gibt, sodass  $x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$  gilt.*

Das PCP kann als Spiel aufgefasst werden, wobei die einzelnen Spielsteinarten durch die Wortpaare dargestellt sind, d.h. die Spielsteine haben die Form  $\begin{bmatrix} x_i \\ y_i \end{bmatrix}$ . Gesucht ist eine Aneinanderreihung der Spielsteine, sodass oben wie unten dasselbe Wort abgelesen werden kann. Dabei dürfen beliebig (aber endlich) viele Spielsteine verwendet werden.

**Beispiel 7.4.2.** Sei  $K = \left( \begin{bmatrix} a \\ aba \end{bmatrix}, \begin{bmatrix} baa \\ aa \end{bmatrix}, \begin{bmatrix} ab \\ bb \end{bmatrix} \right)$ . Dann ist  $I = 1, 2, 3, 2$  eine Lösung, da

$$\begin{bmatrix} a \\ aba \end{bmatrix} \begin{bmatrix} baa \\ aa \end{bmatrix} \begin{bmatrix} ab \\ bb \end{bmatrix} \begin{bmatrix} baa \\ aa \end{bmatrix} = abaaabbaa$$

$$= abaaabbaa$$

Für die Instanz  $K = \left( \begin{bmatrix} ab \\ bba \end{bmatrix}, \begin{bmatrix} ba \\ baa \end{bmatrix}, \begin{bmatrix} ba \\ aba \end{bmatrix}, \begin{bmatrix} bba \\ b \end{bmatrix} \right)$  ist das Finden einer Lösung nicht mehr so einfach. Die kürzeste Lösung benötigt bereits 66 Paare und ist 2, 1, 3, 1, 1, 2, 4, 2, 1, 3, 1, 3, 1, 1, 3, 1, 1, 2, 4, 1, 1, 2, 4, 3, 1, 4, 4, 3, 1, 1, 1, 2, 4, 2, 4, 4, 3, 1, 3, 1, 4, 2, 4, 1, 1, 2, 4, 1, 4, 4, 3, 1, 4, 4, 3, 4, 4, 3, 4, 2, 4, 1, 4, 4, 3.

**Satz 7.4.3.** *Das Postsche Korrespondenzproblem ist unentscheidbar.*

Ein ausführlicher Beweis der das Halteproblem auf PCP reduziert ist in Anhang A.10 zu finden. Wir lassen diesen hier aus und betrachten noch einige Varianten des Problems:

Während bereits für binäre Alphabet das PCP unentscheidbar ist, gilt dies nicht für unäre Alphabete:

**Lemma 7.4.4.** *Das PCP für unäre Alphabete ist entscheidbar.*

*Beweis.* Sei  $\Sigma = \{a\}$ . In diesem Fall sind alle Spielsteine von der Form  $\begin{bmatrix} a^n \\ a^m \end{bmatrix}$ . Wenn für alle Steine  $(x_i, y_i)$  gilt  $|x_i| < |y_i|$ , dann gibt es keine Lösung. Wenn für alle Steine  $(x_i, y_i)$  gilt  $|x_i| > |y_i|$ , dann gibt es keine Lösung. Wenn es zwei Steine gibt  $(x_i, y_i) = (a^n, a^{n+r})$  und  $(x_j, y_j) = (a^{m+s}, a^m)$  mit  $s, r \geq 0$ , dann ist das PCP immer lösbar: Die Lösung ist die Folge von Indizes  $\underbrace{i, \dots, i}_{s\text{-mal}}, \underbrace{j, \dots, j}_{r\text{-mal}}$ , denn diese erzeugt oben  $a^{s \cdot n + r \cdot (m+s)}$  und unten  $a^{s \cdot (n+r) + r \cdot m}$ . Daher werden oben wie unten  $(sn + rm + rs)$ -viele  $a$ 's erzeugt. □

Eine andere Variante des PCP ist es, die Anzahl der verschiedenen Spielsteinarten zu beschränken. Hier ist bekannt, dass das PCP für 1 oder 2 Spielsteinarten entscheidbar ist (siehe (EKR82)), und für 5 Spielsteinarten unentscheidbar ist (das wurde 2015 in (Nea15) gezeigt). Für 3 und 4 Spielsteinarten ist die Frage der Entscheidbarkeit ungeklärt.

## 8. Komplexitätstheorie: Zeitkomplexität, $\mathcal{P}$ und $\mathcal{NP}$

Die Komplexitätstheorie beschäftigt sich mit der Einordnung von Problemen in verschiedene Komplexitätsklassen, d.h. sie betrachtet den Ressourcenverbrauch (Rechenzeit oder Platzbedarf) von Problemen und definiert verschiedene Klassen passend zur Komplexität. Ein Untersuchungsgegenstand ist das Finden und Analysieren der Laufzeit- und Platzkomplexität von Algorithmen für konkrete Probleme. Dabei kann einerseits die Suche nach einem möglichst effizienten Verfahren das Ziel sein, um damit eine *obere Schranke* für die Komplexität des Problems anzugeben. Z.B. zeigt der CYK-Algorithmus, dass das Wortproblem für CFGs in Chomsky-Normalform  $O(|P| \cdot n^3)$  (mit  $n$  Länge des Worts,  $|P|$  Anzahl an Produktionen) als obere Schranke hat. Daher ist Problem mit dieser Zeitkomplexität lösbar, aber eventuell gibt es bessere Algorithmen. Die Frage, nach den bestmöglichen Algorithmen, fokussiert den Nachweis von *unteren Schranken*, d.h. man zeigt, dass es keinen Algorithmus mit einer besseren Zeit- bzw. Platzkomplexität gibt. Im allgemeinen ist dieses Problem schwieriger, da man über alle möglichen (unbekannten) Algorithmen, die das Problem lösen, argumentieren muss.

Ein anderer Fokus ist die Betrachtung der Frage, ob und wie sich Nichtdeterminismus gegenüber Determinismus bezüglich des Ressourcenbedarfs auswirkt. die Frage, welcher Aufwand beim Wechsel vom Nichtdeterminismus in ein deterministisches Modell entsteht und wie sich dieser im Ressourcenbedarf niederschlägt. Beispielsweise haben wir für gezeigt, dass DFAs und NFAs äquivalente Automatenmodelle sind, aber z.B. der benötigte Platz für einen DFA exponentiell größer sein kann als für einen NFA, der dieselbe Sprache erkennt. In diesem Abschnitt betrachten wir eine (sehr wichtige und bisher ungelöste) Frage, die ähnlich dazu ist. Das ist die wichtige Frage, ob  $\mathcal{P} = \mathcal{NP}$  oder  $\mathcal{P} \neq \mathcal{NP}$  gilt. Was sich dahinter genau verbirgt, wird in diesem Teil zur Komplexitätstheorie das zentrale Thema sein.

### 8.1. Deterministische Zeitkomplexität

Wir interessieren uns nur für Sprachen, die entscheidbar sind. Daher können wir für Turingmaschinen annehmen, dass diese für jede Eingabe *anhalten*: Damit meinen wir, dass wir für deterministische Turingmaschinen auch „Verwerfe“-Zustände erlauben, die zwar nicht akzeptierend ist, aber auch (genau wie Konfigurationen mit akzeptierenden Zuständen) keine Nachfolgekongfiguration besitzen. Wir verzichten darauf, die DTMs mit dieser Anpassung erneut formal zu definieren, sondern verwenden sie einfach.

Ein weiterer Punkt, der im Rahmen der Komplexitätsbetrachtungen zum Tragen kommt, ist die Frage, ob wir als Modell für unsere Untersuchungen Einband- oder Mehrbandturingmaschinen verwenden möchten. Damit wir „näher“ an normalen modernen Computern sind, ist es sinnvoller

die Mehrbandturingmaschinen zu verwenden, da diese im Vergleich zu Einband-Maschinen direkten Zugriff auf mehrere Speicherplätze ermöglichen und nicht Rechenzeit „vergeuden“ müssen, um auf dem Band hin- und herzulaufen, um Speicherabschnitte sequentiell zu suchen und zu modifizieren. Tatsächlich ist der Unterschied quadratisch: Wenn die Mehrbandmaschine  $n$  Schritte ausführt, so kann dies die Simulation der Mehrbandmaschine auf der Einbandmaschine in  $O(n^2)$  Schritten durchführen

Dieser Unterschied wird tatsächlich in der Klasse  $\mathcal{P}$  (und auch  $\mathcal{NP}$ ) keinen echten Unterschied machen, da diese gegen einen solchen quadratischen Faktor abgeschlossen ist, d.h. für *diese* Betrachtungen könnten wir sowohl mit Einband- als auch mit Mehrbandmaschinen arbeiten.

Da die Handhabung mit mehreren Bändern i.a. einfacher ist, wählen wir als Modell die Mehrbandmaschinen. Mit dieser Anpassung der Definition, definieren wir die Laufzeit einer stets anhaltenden deterministischen Mehrband-TM.

**Definition 8.1.1.** Sei  $M$  eine stets anhaltende DTM mit Startzustand  $z_0$ . Für Eingabe  $w$  definieren wir

$$time_M(w) := i, \quad \text{wenn } z_0 w \vdash_M^i uz w \text{ und } z \text{ ist Endzustand oder Verwerfe-Zustand.}$$

Aufbauend auf diesem Begriff der Rechenzeit, definieren wir nun die Klasse von Sprachen  $TIME(f(n))$ :

**Definition 8.1.2.** Für eine Funktion  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  sei die Klasse  $TIME(f(n))$  genau die Menge der Sprachen, für die es eine deterministische, stets anhaltende, Mehrband-TM  $M$  gibt, mit  $L(M) = L$  und  $time_M(w) \leq f(|w|)$  für alle  $w \in \Sigma^*$

Eine Sprache ist daher in  $TIME(f(n))$ , wenn sie von einer DTM für jede Eingabe der Länge  $n$  in nicht mehr als  $f(n)$  Schritten entschieden wird.

Die Klassen  $TIME(f(n))$  sind sehr fein, da sie eine beliebig genaue Funktion  $f$  erlauben. Die Klasse  $\mathcal{P}$  liefert nun einen wesentlich gröbereren Begriff, indem sie nur fordert, dass  $f$  ein beliebiges Polynom ist. Wir definieren zunächst, was ein Polynom ist:

**Definition 8.1.3.** Ein Polynom ist eine Funktion  $p : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  der Form:

$$p(n) = \sum_{i=0}^k a_i \cdot n^i = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 \cdot n + a_0$$

mit  $a_i \in \mathbb{N}_0$  und  $k \in \mathbb{N}_0$ . (Man spricht auch von einem Polynom des Grades  $k$ .)

**Definition 8.1.4** (Komplexitätsklasse  $\mathcal{P}$ ). Die Klasse  $\mathcal{P}$  ist definiert als

$$\mathcal{P} = \bigcup_{p \text{ Polynom}} TIME(p(n))$$

Eine Sprache  $L$  ist daher in der Klasse  $\mathcal{P}$  enthalten, wenn es eine stets anhaltende DTM  $M$  und ein Polynom  $p$  gibt, sodass  $L = L(M)$  und für alle  $w \in \Sigma^*$  :  $time_M(w) \leq p(|w|)$ .

Zum Nachweis dass ein Algorithmus polynomielle Komplexität hat, genügt es zu zeigen, dass seine Komplexität  $O(n^k)$  für eine Konstante  $k$  ist. Z.B. ist ein Algorithmus mit der Komplexität  $n \log n$  auch als polynomiell anzusehen, da  $n \log n \in O(n^2)$ . Da Funktionen wie  $n^{\log n}$ ,  $2^n$ ,  $n!$  nicht durch ein Polynom von oben beschränkt werden, sind Algorithmen mit diesen Komplexitäten nicht polynomiell.

Beachte, dass man üblicherweise die Probleme die der Klasse  $\mathcal{P}$  enthalten sind, als effizient lösbar ansieht. D.h. es gibt Algorithmen, die diese in polynomiell Zeit lösen. Algorithmen mit höheren Komplexitäten werden im allgemeinen nicht als effizient angesehen.

## 8.2. Nichtdeterminismus

Wir möchten nun analog zur Klasse  $\mathcal{P}$ , eine Komplexitätsklasse definieren, die alle Sprachen erfasst die in Polynomialzeit entschieden werden – allerdings von nichtdeterministischen Turingmaschinen.

Um eine möglichst einheitliche Definition zu haben, betrachten wir auch für den Nichtdeterministischen Fall nur Turingmaschinen, die stets anhalten – d.h. nur solche NTMs deren Berechnungen auf allen Pfaden anhält. Da für NTMs anhaltenden Nicht-Endzustände sowieso definiert werden können (durch Zustände  $z$ , für die gilt  $\delta(z, a) = \emptyset$  und  $z \notin E$ ), benötigt diese keine Anpassung an der Definition der NTMs. Beachte, dass ein solches Anhalten in einem nicht Endzustand *keine* Aussage impliziert, ob ein Wort zur erkannten Sprachen gehört oder nicht. Damit ein Wort sicher nicht zur Sprache gehört, müssen (für eine immer anhaltende NTM) *alle* Pfade in einem Nicht-Akzeptanzzustand enden.

**Definition 8.2.1.** Sei  $M$  eine Mehrband-NTM, die für jede Eingabe anhält. Dann definieren wir die Laufzeit<sup>1</sup> von  $M$  als

$$ntime_M(w) = \max\{i \mid z_0 w \vdash_M^i uz w \text{ und } uz w \neq \dots\}$$

Schließlich definieren wir basierend auf diesem Laufzeitmaß die Komplexitätsklasse  $NTIME(f(n))$  und die Klasse  $\mathcal{NP}$ :

**Definition 8.2.2.** Für eine Funktion  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  bezeichne  $NTIME(f(n))$  die Klasse aller Sprachen  $L$ , für die es eine nichtdeterministische Mehrband-TM  $M$  gibt mit  $L(M) = L$  und für alle  $w \in \Sigma^*$  gilt  $ntime_M(w) \leq f(|w|)$ .

Die Klasse  $\mathcal{NP}$  ist definiert als

$$\mathcal{NP} = \bigcup_{p \text{ Polynom}} NTIME(p(n))$$

<sup>1</sup>Es gibt hierbei verschiedene Möglichkeiten dieser Definition, welche verschiedene Turingmaschinen erlauben oder verbieten. Diese Varianten führen jedoch zum selben Begriff der Klasse  $\mathcal{NP}$

### 8.3. Das P-vs-NP-Problem

**Lemma 8.3.1.** *Es gilt  $\text{TIME}(f(n)) \subseteq \text{NTIME}(f(n))$  und damit auch  $\mathcal{P} \subseteq \mathcal{NP}$ .*

*Beweis.* Jede DTM kann leicht als NTM aufgefasst werden, wobei es nur einen möglichen Berechnungspfad gibt. Dieser bestimmt  $\text{ntime}_M(w)$  und es gilt für diese Turingmaschine  $\text{time}_M(w) = \text{ntime}_M(w)$ . Damit folgt, dass aus  $L \in \text{TIME}(f(n))$  auch  $L \in \text{NTIME}(f(n))$  folgt. Schließlich zeigt dies auch  $\mathcal{P} \subseteq \mathcal{NP}$ .  $\square$

Die bis heute ungelöste Frage ist, ob die Inklusion  $\mathcal{P} \subseteq \mathcal{NP}$  echt ist, d.h.  $\mathcal{P} \subset \mathcal{NP}$  gilt, oder  $\mathcal{P} = \mathcal{NP}$ . Es gibt viele gute Gründe, warum die Mehrheit der Forscher, vermuten dass  $\mathcal{P} \subset \mathcal{NP}$  gilt, aber ein Beweis ist bis heute niemanden gelungen.

Das  $\mathcal{P}$ -vs- $\mathcal{NP}$ -Problem ist eines der sieben sogenannten Millennium-Probleme, die vom Clay Mathematics Institute im Jahr 2000 als Liste ungelöster Probleme der Mathematik herausgegeben wurde und für dessen Lösung ein Preisgeld von einer Million US Dollar ausgelobt wurde. Würde das Problem positiv beantwortet mit  $\mathcal{P} = \mathcal{NP}$  so wüsste man, dass Probleme für die bisher nur Algorithmen bekannt sind, die in (deterministischer) Exponentialzeit laufen, effizient lösbar sind. Da es für viele Probleme (z.B. das sogenannte Travelling-Salesman-Problem) leicht ist zu zeigen, dass sie  $\mathcal{NP}$ -liegen, aber noch niemand einen Algorithmus gefunden hat, der die Probleme in deterministischer Polynomialzeit löst, ist liegt die Vermutung nahe, dass diese Probleme in  $\mathcal{NP} \setminus \mathcal{P}$  liegen und daher  $\mathcal{P} \neq \mathcal{NP}$  gilt.

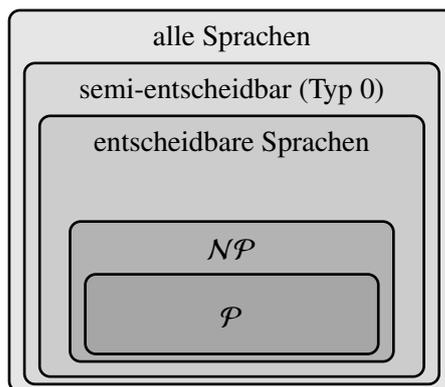
Die Arbeiten von Steven Cook und Richard Karp (Coo71; Kar72) entwickelten mit dem Begriff der  $\mathcal{NP}$ -Vollständigkeit (die genaue Definition werden in Abschnitt 8.4 sehen) eine Struktur der Klassen  $\mathcal{NP}$  und  $\mathcal{P}$ : Für viele der genannten Probleme, die in  $\mathcal{NP}$  liegen, aber für die kein Polynomialzeit-Algorithmus gefunden wurde, lässt sich zeigen, dass sie eng untereinander verknüpft sind: Wenn es einen Polynomialzeitalgorithmus gibt der eines dieser Probleme löst, dann sind alle diese Probleme in Polynomialzeit lösbar (falls  $\mathcal{P} = \mathcal{NP}$ ) oder keines (falls  $\mathcal{P} \neq \mathcal{NP}$ ).

Um den Übergang von der Berechenbarkeitstheorie zur Komplexitätstheorie und unseren Betrachtungen der Komplexitätsklassen  $\mathcal{NP}$  und  $\mathcal{P}$  zu verdeutlichen, kann das in Abb. 8.1 gezeigte Mengendiagramm helfen. Im gezeigten Diagramm sind alle Teilmengenbeziehungen als echt bekannt bis auf die unterste zwischen  $\mathcal{NP}$  und  $\mathcal{P}$ .

## 8.4. NP-Vollständigkeit

### 8.4.1. Polynomialzeit-Reduktionen

In Abschnitt 7.3.2.2 wurden Reduktionen einer Sprache auf eine andere Sprache eingeführt, um die Entscheidbarkeit (bzw. Unentscheidbarkeit) einer Sprache auf die Entscheidbarkeit (bzw. Unentscheidbarkeit) der anderen Sprache zurück zu führen. Eine solche Reduktion von  $L_1 \subseteq \Sigma_1^*$  auf  $L_2 \subseteq \Sigma_2^*$  (geschrieben  $L_1 \leq L_2$ ) verlangte als Zeugen eine berechenbare und totale Funktion

Abbildung 8.1.: Lage der Komplexitätsklassen  $\mathcal{P}$  und  $\mathcal{NP}$ 

$f$ , sodass  $w \in L_1$  g.d.w.  $f(w) \in L_2$  für alle Worte  $w \in \Sigma_1^*$ . Eine Polynomialzeit-Reduktion (oder auch Karp-Reduktion, benannt nach Richard Karp) ist analog definiert mit dem Zusatz, dass die Funktion  $f$  in deterministischer Polynomialzeit berechenbar sein muss (d.h. es gibt eine Turingmaschine, die  $f$  berechnet und für die gilt  $\text{time}_M(w) \in O(|w|^k)$  für ein festes  $k$ ).

**Definition 8.4.1** (Polynomialzeit-Reduktion (einer Sprache auf eine andere)). Sei  $L_1 \subseteq \Sigma_1^*$  und  $L_2 \subseteq \Sigma_2^*$  Sprachen. Dann sagen wir  $L_1$  ist auf  $L_2$  polynomiell reduzierbar (geschrieben  $L_1 \leq_p L_2$ ), falls es eine totale und in deterministischer Polynomialzeit berechenbare Funktion  $f: \Sigma_1^* \rightarrow \Sigma_2^*$  gibt, sodass für alle  $w \in \Sigma_1^*$  gilt:  $w \in L_1 \iff f(w) \in L_2$ .

Während Reduktionen  $L_1 \leq L_2$  in Abschnitt 7.3.2.2 verwendet wurden, um aus der Zugehörigkeit von  $L_2$  zur Klasse der entscheidbaren Sprachen, zu schließen, dass auch  $L_1$  entscheidbar ist, werden Polynomialzeit-Reduktionen  $L_1 \leq_p L_2$  verwendet, um aus der Zugehörigkeit von  $L_2$  zur Klasse  $\mathcal{P}$  (bzw.  $\mathcal{NP}$ ) zu schließen, dass auch  $L_1$  zur selben Klasse gehört:

**Lemma 8.4.2.** Falls  $L_1 \leq_p L_2$  und  $L_2 \in \mathcal{P}$ , dann gilt  $L_1 \in \mathcal{P}$ . Ebenso gilt: Falls  $L_1 \leq_p L_2$  und  $L_2 \in \mathcal{NP}$ , dann gilt  $L_1 \in \mathcal{NP}$ .

*Beweis.* Sei  $L_1 \leq_p L_2$  und  $f$  die dies bezeugende (totale und in Polynomialzeit berechenbare) Funktion. Dann gibt es eine deterministische Turingmaschine  $M_f$ , die  $f$  berechnet und dafür polynomielle Zeit benötigt.

Sei nun  $L_2 \in \mathcal{P}$ . Dann gibt es eine deterministische Turingmaschine  $M_2$  mit  $L(M_2) = L_2$ .  $M_2$  hält stets nach polynomieller Zeit an. Die Hintereinanderschaltung  $M_f; M_2$  stellt eine Turingmaschine dar, für die gilt  $L(M_f; M_2) = L_1$  und  $M_f; M_2$  hält stets in polynomieller Zeit. Daher gilt  $L_1 \in \mathcal{P}$ .

Wenn  $L_2 \in \mathcal{NP}$ , dann gibt es eine nichtdeterministische Turingmaschine  $M'_2$  mit  $L(M'_2) = L_2$ , die stets in polynomieller Zeit anhält. Die Konstruktionen  $M_f; M'_2$  führe erst  $M_f$  (deterministisch) aus und anschließend  $M'_2$  als nichtdeterministische Turingmaschine. Für sie gilt

$L(M_f; M'_2) = L_1$  und  $M_f; M'_2$  hält auf allen Berechnungspfaden nach polynomieller Zeit. Dies zeigt  $L_1 \in \mathcal{NP}$ .

In beiden Teilen des Beweises muss man sich noch verdeutlichen, dass die Berechnung des Funktionswertes  $f(w)$  für ein Wort  $w \in \Sigma_1^*$  „nur“ ein polynomiell großes Wort sein kann. Es gilt sogar: Wenn die Laufzeit von TM  $M_f$  durch  $p(|w|)$  begrenzt ist (für ein Polynom  $p$ ), Dann muss auch gelten  $|f(w)| \leq |w| + p(|w|)$ , da  $M_f$  in  $p(|w|)$ -vielen Schritten nicht mehr als  $p(|w|)$ -viele Symbole schreiben kann. Wenn die Laufzeit der TM  $M_2$  durch ein Polynom  $q$  begrenzt ist, ist die Laufzeit von  $M_f; M_2$  durch die Funktion  $r$  mit  $r(|w|) = p(|w|) + q(|w| + p(|w|))$  begrenzt, was immer noch ein Polynom ist. Analoges gilt für die nichtdeterministische Laufzeit von  $M_f; M'_2$ .  $\square$

**Lemma 8.4.3.** *Die Relation  $\leq_p$  ist transitiv, d.h. wenn  $L_1 \leq_p L_2$  und  $L_2 \leq_p L_3$ , dann gilt auch  $L_1 \leq_p L_3$*

*Beweis.* Der Beweis ist analog zum vorherigen und benutzt im Wesentlichen die Eigenschaft, dass die Komposition von zwei Polynomen immer noch ein Polynom ist.  $\square$

Wir definieren nun die  $\mathcal{NP}$ -Vollständigkeit für Probleme. Diese besteht aus zwei Teilen, einerseits, dass das Problem in der Komplexitätsklasse  $\mathcal{NP}$ -liegt und andererseits, dass *jedes* andere Problem aus  $\mathcal{NP}$  auf das  $\mathcal{NP}$ -vollständige Problem polynomiell reduzierbar ist. Damit kann man die  $\mathcal{NP}$ -vollständigen Problem als die schwierigsten Probleme in  $\mathcal{NP}$  ansehen.

**Definition 8.4.4** ( $\mathcal{NP}$ -Vollständigkeit). *Eine Sprache  $L$  heißt  $\mathcal{NP}$ -vollständig, wenn gilt*

1.  $L \in \mathcal{NP}$  und
2.  $L$  ist  $\mathcal{NP}$ -schwer<sup>2</sup>: Für alle  $L' \in \mathcal{NP}$  gilt  $L' \leq_p L$

Die Zugehörigkeit zu  $\mathcal{NP}$  für eine betrachtete Sprache  $L$  nachzuweisen ist im Allgemeinen nicht so schwer (es sei denn, dass Problem ist tatsächlich schwieriger). Für den Nachweis der  $\mathcal{NP}$ -Schwere können wir ähnlich verfahren, wie bei der Unentscheidbarkeit: Dort haben wir die Unentscheidbarkeit des speziellen Halteproblems direkt bewiesen, aber die Unentscheidbarkeit weiterer Probleme durch Reduktion des speziellen Halteproblems auf das weitere Problem nachgewiesen. Für die  $\mathcal{NP}$ -Schwere ist es ähnlich: Wenn wir eine  $\mathcal{NP}$ -vollständige Sprache  $L_0$  haben, dann ist der Nachweis der  $\mathcal{NP}$ -Schwere für weitere Sprachen  $L_1$  im Allgemeinen einfacher, da wir die Transitivität von  $\leq_p$  ausnutzen können und zeigen  $L_0 \leq_p L_1$ . Da  $L_0$  als  $\mathcal{NP}$ -vollständig bekannt ist, gilt für alle  $L' \in \mathcal{NP}$ :  $L' \leq_p L_0$  und aufgrund der Transitivität von  $\leq_p$  damit auch  $L' \leq_p L_1$  für alle  $L' \in \mathcal{NP}$ .

Bevor wir dieses eine Problem als  $\mathcal{NP}$ -vollständig sehen (als Satz von Cook im nächsten Abschnitt), zeigen wir den folgenden Satz:

**Satz 8.4.5.** *Sei  $L$  ein  $\mathcal{NP}$ -vollständiges Problem. Dann gilt  $L \in \mathcal{P} \iff \mathcal{P} = \mathcal{NP}$ .*

<sup>2</sup>manchmal auch  $\mathcal{NP}$ -hart genannt

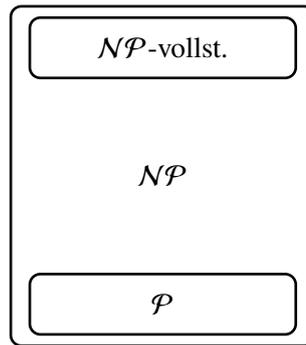


Abbildung 8.2.: Vermutete Lage der Probleme in  $\mathcal{P}$ ,  $\mathcal{NP}$  und der  $\mathcal{NP}$ -vollständigen Probleme

*Beweis.* Sei  $L \in \mathcal{P}$  und  $L$   $\mathcal{NP}$ -vollständig. Aus der  $\mathcal{NP}$ -Schwere von  $L$  folgt: jedes andere Problem  $L' \in \mathcal{NP}$  liegt ebenfalls in  $\mathcal{P}$ : Mit  $L' \leq_p L$  und Lemma 8.4.2 folgt  $L' \in \mathcal{P}$ . Da dies für alle  $L' \in \mathcal{NP}$  gilt, folgt  $\mathcal{P} = \mathcal{NP}$ .  $\square$

Dieser Satz zeigt, dass es ausreicht, für ein  $\mathcal{NP}$ -vollständiges Problem nachzuweisen, dass es in  $\mathcal{P}$  bzw. nicht in  $\mathcal{P}$  liegt, um die  $\mathcal{P}$ -vs- $\mathcal{NP}$ -Frage ein für allemal beantworten zu können.

In Abb. 8.2 ist eine Skizze zur vermuteten Lage (wenn  $\mathcal{P} \neq \mathcal{NP}$  gilt) der Klassen  $\mathcal{NP}$ ,  $\mathcal{P}$  und der  $\mathcal{NP}$ -vollständigen Probleme gezeigt. Z.B. ist bekannt (Lad75), dass es unter der Annahme  $\mathcal{P} \neq \mathcal{NP}$ , Probleme in  $\mathcal{NP}$  gibt, die nicht in  $\mathcal{P}$  liegen aber auch nicht  $\mathcal{NP}$ -vollständig sind. Ein möglicher Kandidat ist das Graph-Isomorphismus-Problem (siehe z.B. (Sch88)), für welches zum heutigen Zeitpunkt weder ein polynomieller Algorithmus noch dessen  $\mathcal{NP}$ -Vollständigkeit bekannt ist.

#### 8.4.2. Satz von Cook

In diesem Abschnitt erläutern wir das *Erfüllbarkeitsproblem der Aussagenlogik* (kurz SAT), welches  $\mathcal{NP}$ -vollständig ist. Den Beweis der  $\mathcal{NP}$ -Schwere lassen wir aus (er kann im Anhang nachgelesen werden).

Wir definieren zunächst das SAT-Problem in Form eines gegeben/gefragt Problems:

**Definition 8.4.6** (SAT-Problem). *Das Erfüllbarkeitsproblem der Aussagenlogik (kurz SAT) ist:*

*gegeben:* Eine aussagenlogische Formel  $F$

*gefragt:* Ist  $F$  erfüllbar, d.h. gibt es eine erfüllende Belegung der Variablen mit den Wahrheitswerten 0 und 1, sodass  $F$  den Wert 1 erhält.

Beachte, dass wir die zugehörige formale Sprache definieren können, indem wir genau die Sprache beschreiben, die alle erfüllenden Formeln enthält. Dabei müssen die Formeln geeignet über einem Alphabet  $\Sigma$  dargestellt werden. Bezeichne  $code(F) \in \Sigma^*$  diese Repräsentation von  $F$ . Dann ist

$$\text{SAT} = \{ \text{code}(F) \in \Sigma^* \mid F \text{ ist eine erfüllbare Formel der Aussagenlogik} \}$$

Diese Überführung der gegeben/gefragt-Notation in eine formale Sprache ist immer die gleiche: Die Sprache enthält genau die Instanzen, die mit „ja“ zu beantworten sind.

**Lemma 8.4.7.**  $\text{SAT} \in \mathcal{NP}$

*Beweis.* Sei  $\text{code}(F)$  die Eingabe einer nichtdeterministischen Turingmaschine  $M$ . Die Maschine berechnet zunächst (mit einem Durchlauf durch die Eingabe), welche Variablen in der Formel  $F$  vorkommen. Sei dies die Menge  $\{x_1, \dots, x_n\}$ . Anschließend verwendet die Maschine den Nichtdeterminismus, um eine Belegung  $I : \{x_1, \dots, x_n\} \rightarrow \{0, 1\}$  der Variablen zu „raten“. Anschließend berechnet sie den Wert von  $I(F)$ , d.h. sie setzt zunächst  $I(x_i)$  für jedes Vorkommen einer Variablen ein und rechnet anschließend den Wert von  $I(F)$  aus. Wenn  $I(F) = 1$ , dann wechselt sie in einen Akzeptanzzustand, und anderenfalls verwirft sie.

Beachte, dass im Rate-Schritt der TM  $2^n$  verschiedene mögliche Belegungen  $I$  existieren, die zu  $2^n$  verschiedenen nichtdeterministischen Berechnungen der TM führen.

Da jede Belegung überprüft wird, gilt:

$$M \text{ akzeptiert eine Formel } F \text{ g.d.w. } F \in \text{SAT}.$$

Jeder Berechnungspfad der Turingmaschine läuft in Polynomialzeit in der Größe der Eingabe (d.h. in  $|\text{code}(F)|$ ), da die Anzahl der Variablen durch die Eingabegröße beschränkt ist, das Einsetzen der Belegung und das Bewerten der variablenfreien Formel jeweils in Polynomialzeit durchführbar sind.  $\square$

Der Nachweis, dass ein Problem in  $\mathcal{NP}$  liegt, kann meistens so geführt werden, wie wir dies eben getan haben: Man rät alle möglichen Lösungen mithilfe des Nichtdeterminismus und muss anschließend nur noch zeigen, dass man in polynomieller Zeit prüfen kann, ob die mögliche Lösung eine echte Lösung ist. D.h. man zeigt, dass Lösungen in Polynomialzeit verifizierbar sind.

**Satz 8.4.8** (Satz von Cook). *Das Erfüllbarkeitsproblem der Aussagenlogik ist  $\mathcal{NP}$ -vollständig.*

*Beweis.* Dies gilt, da  $\text{SAT} \in \mathcal{NP}$  (gezeigt in Lemma 8.4.7) und  $\text{SAT}$   $\mathcal{NP}$ -schwer ist. Der Beweis der  $\mathcal{NP}$ -Schwere kann z.B. in Anhang A.11 nachgelesen werden. Die wesentliche Idee dabei ist es für jede NTM  $M$ , die nach polynomiell vielen Schritten anhält, und deren Eingabe  $w$  eine (polynomiell große) aussagenlogische Formel  $F$  zu konstruieren, so dass  $w \in L(M)$  genau dann, wenn  $F$  erfüllbar ist. Die Formel „simuliert“ dabei den Ablauf der NTM  $M$  auf  $w$ . Dann kann eine NTM, die  $\text{SAT}$  in nichtdeterministischer Polynomialzeit entscheidet mit polynomiellen zusätzlichem Aufwand auch  $L(M)$  entscheiden (Was immer noch nichtdeterministische Polynomialzeit ist.)  $\square$

### 8.4.3. NP-vollständige Probleme und die Praxis

Viele wichtige Probleme (z.B. finden optimaler Routen auf Landkarten) sind  $\mathcal{NP}$ -vollständig, aber von hoher praktischer Relevanz. Die  $\mathcal{NP}$ -Vollständigkeit eines Problems sagt uns im wesentlichen, dass wir kein Programm schreiben können, welches das Problem in annehmbarer Zeit exakt löst. Wir gehen wir damit um: Es gibt hierfür drei wesentliche Möglichkeiten:

- Ein Algorithmus mit exponentieller Laufzeit kann genutzt werden, wenn die Instanzgrößen so klein sind, dass der benötigte Zeitaufwand akzeptabel ist. Hier kann man durch Verbessern eines reinen Brute-Forces-Ansatzes (d.h. alle Möglichkeiten aufzählen und ausprobieren) noch leicht größere Instanzen lösen. Die Idee dabei ist es, die Eingabe zu vereinfachen, Teillösungen vorzuberechnen, Symmetrien auszunutzen etc. Ziel ist es, die Basis des exponentiellen Algorithmus zu verkleinern. Kurz gesprochen möchte man die Laufzeit von  $c \cdot 2^n$  auf  $c \cdot (1, x)^n$  für möglichst kleines  $x$  zu reduzieren. Nimmt man z.B. an, eine Operation kann in  $10^{-9}$  Sekunden ausgeführt werden, so benötigen  $2^{42}$  Operationen ungefähr 1,2 Stunden,  $(1, 5)^{72}$  und  $(1, 2)^{160}$  ungefähr 1,3 Stunden und  $(1, 1)^{305}$  ungefähr 1,2 Stunden. Dies verdeutlicht, welche Verbesserungen durch solche Maßnahmen möglich sind.

Eine andere Variante besteht darin, das gegebene Probleme in bekannte Probleme zu kodieren, für die es Löser gibt, die sich in der Praxis gut verhalten. Das SAT-Problem ist ein solches bekanntes Problem, für welches es verschiedene SAT-Solver gibt, die auch riesige Instanzen lösen können. Hier ist unter Umständen ein bisschen Experimentieren notwendig, ob die gewählte Kodierung sich in der Praxis zusammen mit dem Löser gut verhält.

- Es kann sein das wichtige Spezialfälle des Problems doch in polynomieller Zeit lösbar ist. Wenn nur solche Probleme in der Praxis auftreten, können diese effizienten Algorithmen verwendet werden. Ein solcher Spezialfall ist oft gegeben, wenn bestimmte Parameter der Eingabe fest und nicht variabel sind, dann werden die exponentiellen Algorithmen wieder polynomiell (z.B. wenn man bei SAT ein feste Anzahl an Variablen annimmt).
- Statt einer exakten Lösung, wird ein Algorithmus verwendet, der nur eine Näherung, d.h. eine fast optimale Lösung, in polynomieller Zeit findet. Einige Methoden hierfür sind lokale Suche, randomisierte Suche, heuristische Suche, evolutionäre Algorithmen und die Approximationsalgorithmen.

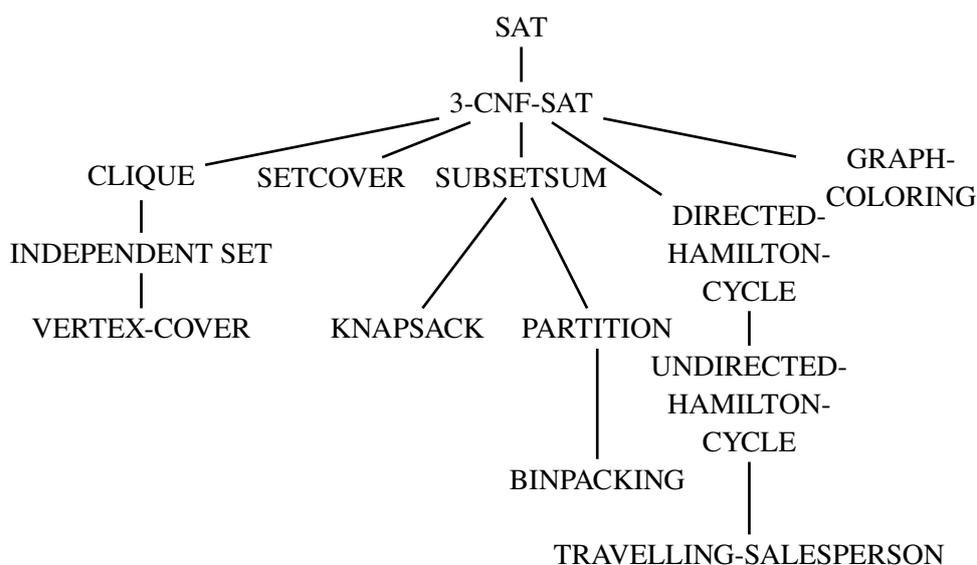
### 8.5. Eine Auswahl NP-vollständiger Probleme

In diesem Abschnitt betrachten wir verschiedene Probleme und zeigen deren  $\mathcal{NP}$ -Vollständigkeit, indem wir Reduktionen bekannter  $\mathcal{NP}$ -vollständiger Probleme auf die neuen Probleme angeben (der  $\mathcal{NP}$ -Schwere-Beweis ist dadurch erbracht) und stets kurz argumentieren, dass das gegebene Problem in  $\mathcal{NP}$  liegt. Für den zweiten Schritt ist das Vorgehen in allen Fällen, eine mögliche Lösung nichtdeterministisch zu raten und anschließend in polynomieller Zeit zu verifizieren.

Sinn und Zweck dieses Kapitels ist es, den Umgang mit polynomiellen Reduktionen und der  $\mathcal{NP}$ -Vollständigkeit zu üben, aber auch einen Katalog an  $\mathcal{NP}$ -vollständigen Problemen zu kennen, damit man diese verwenden kann, um sie auf neue Probleme zu reduzieren.

Die  $\mathcal{NP}$ -Vollständigkeitsbeweise werden hier angegeben, in der Vorlesung werden wir sie jedoch aus Zeitgründen überspringen und nur die Probleme vorstellen.

Das folgende Diagramm zeigt unser Vorgehen: Es zeigt die in diesem Abschnitt betrachteten Probleme und die Polynomialzeitreduktionen die wir präsentieren werden:



Für die in diesem Abschnitt betrachteten Graphenprobleme nehmen wir stets an, dass der Graph schlingenfrei ist (d.h. für jede Kante  $\{u, v\}$  eines Graphen gilt stets:  $u \neq v$ ).

### 8.5.1. Das 3-CNF-SAT-Problem

**Definition 8.5.1** (3-CNF-SAT). *Das 3-CNF-SAT-Problem lässt sich in der gegeben/gefragt-Notation formulieren als:*

*gegeben:* Eine aussagenlogische Formel  $F$  in konjunktiver Normalform, sodass jede Klausel höchstens 3 Literale enthält.

*gefragt:* Ist  $F$  erfüllbar? Genauer: Gibt es eine erfüllende Belegung der Variablen mit den Wahrheitswerten 0 und 1, sodass  $F$  den Wert 1 erhält?

Zur Erinnerung: Eine aussagenlogische Formel ist in konjunktiver Normalform, wenn sie von der Form  $\bigwedge_{i=1}^m (\bigvee_{j=1}^{n_i} L_{i,j})$  ist, wobei ein Literal  $L_{i,j}$  eine aussagenlogische Variable oder deren Negation ist. Die Subformeln  $(\bigvee_{j=1}^{n_i} L_{i,j})$  bezeichnet man dabei als Klausel. Oft schreibt man eine konjunktive Normalform als Menge von Mengen:  $\{\{L_{1,1}, \dots, L_{1,n_1}\}, \dots, \{L_{m,1}, \dots, L_{m,n_m}\}\}$ .

Wir nehmen stets an, dass eine CNF keine Klauseln enthält, die  $x$  und  $\neg x$  enthält: Da diese Klauseln immer wahr sind, können sie stets gelöscht werden.

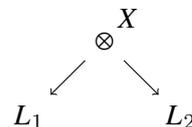
Beachte, dass eine erfüllende Belegung einer CNF, mindestens ein Literal pro Klausel wahr machen muss.

Jede aussagenlogische Formel kann in eine äquivalente konjunktive Normalform gebracht werden (der Algorithmus dazu ist in Kurzform: Implikation und Biimplikation auflösen, Negationen nach innen schieben und anschließend Ausmultiplizieren (Distributivität, Kommutativität, Assoziativität anwenden), um konjunktive Normalform herzustellen). Allerdings hat dieser Algorithmus im worst case exponentielle Laufzeit (und erzeugt im worst case exponentiell große konjunktive Normalformen). Daher kann dieser Algorithmus nicht verwendet werden, um eine Polynomialzeitreduktion von SAT auf 3-CNF-SAT anzugeben.

**Satz 8.5.2.** 3-CNF-SAT ist  $\mathcal{NP}$ -vollständig.

*Beweis.* Wir zeigen die beiden benötigten Teile:

- 3-CNF-SAT  $\in \mathcal{NP}$ : Eine NTM liest zuerst die Eingabe und die Menge der vorkommenden Variablen. Dann rät sie nicht-deterministisch eine Belegung der Variablen und schließlich verifiziert sie, ob die geratene Belegung die Formel erfüllt. Die NTM akzeptiert, wenn die Belegung eine erfüllende Belegung ist. Das Verifizieren kann in Polynomialzeit durchgeführt werden, da die Belegung angewendet werden muss und anschließend die Operatoren ausgewertet werden.
- 3-CNF-SAT ist  $\mathcal{NP}$ -schwer: Wir zeigen  $\text{SAT} \leq_p 3\text{-CNF-SAT}$ . Dafür müssen wir eine polynomiell berechenbare, totale Funktion  $f$  angeben, die jede aussagenlogische Formel  $F$  in eine 3-CNF überführt, sodass  $F$  erfüllbar g.d.w.  $f(F)$  erfüllbar. Die Transformation muss daher nicht die Äquivalenz, sondern nur die Erfüllbarkeitsäquivalenz erhalten.
  - 1. Schritt: Schiebe alle Negationen nach innen vor die Variablen. Dafür werden die Regeln  $\neg\neg F \rightarrow F$ ,  $\neg(F \wedge G) \rightarrow \neg F \vee \neg G$ ,  $\neg(F \vee G) \rightarrow \neg F \wedge \neg G$ ,  $\neg(F \iff G) \rightarrow (\neg F) \iff G$  und  $\neg(F \implies G) \rightarrow F \wedge \neg G$  angewendet.
  - 2. Schritt: Betrachte den Syntaxbaum, der die Formel repräsentiert, wobei wie Blätter mit Literalen identifizieren (d.h. Negationen tauchen nur in den Blattmarkierungen auf, aber sonst nirgends). Für jeden Nichtblatt-Knoten führe eine neue aussagenlogische Variable ein.
  - 3. Schritt: Durchlaufe den Baum von oben nach unten und erzeuge für jede Gabelung



wobei  $\otimes \in \{ \iff, \wedge, \vee, \implies \}$  und  $L_1, L_2$  entweder die aussagenlogische-Variable (für Nichtblatt-Knoten) oder das Literal am Blatt, die Formel  $(X \iff (L_1 \otimes L_2))$

Konjugiere alle diese Formeln und zusätzlich die Formel  $W$ , wobei  $W$  die Variable für die Wurzel der Formel ist. Damit erhält man eine Formel von der Form  $W \wedge \bigwedge_{i,j,k} (X_i \iff (X_j \otimes_i X_k))$ , wobei  $X_r$  Literale sind.

- 4.Schritt: Schließlich berechne für jede Subformel  $(X_i \iff (X_j \otimes_i X_k))$  die CNF mit dem üblichen Algorithmus. Die Größe dieser CNFs ist jeweils konstant und die Berechnung pro CNF ist auch konstant. Außerdem kann jede Klausel nur drei Literale enthalten, da nur drei Variablen vorkommen und – falls eine Variable mehrfach vorkommt, kann entweder die Kopie eliminiert werden, oder die gesamte Klausel gelöscht werden. Schließlich liefert die Konjunktion all dieser Klauseln eine 3-CNF.

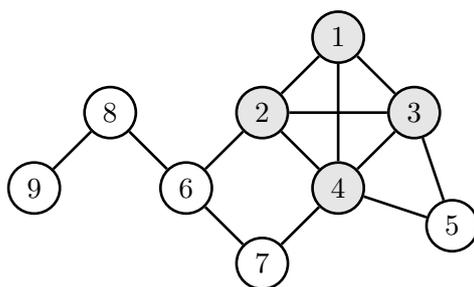
Die Größe der 3-CNF ist polynomiell in der ursprünglichen Formel und die 3-CNF kann in Polynomialzeit berechnet werden. Sei  $f$  die Funktion, die aus der aussagenlogischen Formel  $F$  diese 3-CNF berechnet. Dann gilt  $F$  erfüllbar g.d.w.  $f(F)$  erfüllbar: Wenn es eine Belegung  $I$  gibt, sodass  $I(F) = 1$  gilt, dann kann  $I$  erweitert werden zu  $I'$ , sodass  $I'(f(F)) = 1$ : Setze  $I'(X) = I(X)$  für alle Variablen, die in  $F$  vorkommen und setze  $I'(W) = 1$  für die Variable  $W$  an der Wurzel. Für alle anderen neuen Variablen  $X$ , setze  $I'(X)$  so, dass  $I'(X) = I'(L_1 \otimes L_2)$  gilt. Umgekehrt, wenn Belegung  $J$  die Formel  $f(F)$  wahr macht (d.h.  $J(f(F)) = 1$ ), dann gilt auch, dass die Restriktion von  $J$  auf die Variablen von  $F$ , die Formel  $F$  wahr macht.

Damit haben wir gezeigt  $\text{SAT} \leq_p \text{3-CNF-SAT}$ . Da  $\text{SAT}$   $\mathcal{NP}$ -schwer, ist somit auch 3-CNF-SAT  $\mathcal{NP}$ -schwer.  $\square$

### 8.5.2. Das CLIQUE-Problem

Für einen ungerichteten Graph  $G = (V, E)$  mit Knotenmengen  $V$  und Kantenmenge  $E$  ist eine *Clique* der Größe  $k$  eine Menge  $V' \subseteq V$ , sodass  $|V'| = k$  und für alle  $u, v \in V'$  mit  $u \neq v$  gilt:  $\{u, v\} \in E$ .

**Beispiel 8.5.3.** Der Graph  $G = (V, E)$  mit Knotenmenge  $V = \{1, \dots, 9\}$  und Kantenmenge  $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{2, 6\}, \{3, 4\}, \{3, 5\}, \{3, 5\}, \{4, 7\}, \{6, 7\}, \{6, 8\}, \{8, 9\}\}$  gezeichnet als



hat eine Clique der Größe 4, nämlich  $\{1, 2, 3, 4\}$  und z.B. sind die Mengen  $\{1, 2, 3\}$ ,  $\{1, 2, 4\}$ ,  $\{1, 3, 4\}$ ,  $\{2, 3, 4\}$ ,  $\{3, 4, 5\}$  Cliquen der Größe 3.

**Definition 8.5.4** (CLIQUE-Problem). Das CLIQUE-Problem lässt sich in der gegeben/gefragt-Notation formulieren durch:

gegeben: Ein ungerichteter Graph  $G = (V, E)$  und eine Zahl  $k \in \mathbb{N}_0$ .

gefragt: Besitzt  $G$  eine Clique der Größe mindestens  $k$ ?

**Satz 8.5.5.** CLIQUE ist  $\mathcal{NP}$ -vollständig.

*Beweis.* Wir zeigen die beiden benötigten Teile:

- CLIQUE  $\in \mathcal{NP}$ : Es gibt eine NTM, die zunächst alle Knoten und die Zahl  $k$  aus der Beschreibung extrahiert und anschließend nichtdeterministisch eine Menge von  $k$  Knoten rät und für jede der nichtdeterministischen Möglichkeiten verifiziert, ob die Menge eine CLIQUE ist. Die Verifikation kann in quadratischer Laufzeit durchgeführt werden. Daher ist die Laufzeit der NTM polynomiell beschränkt.
- CLIQUE ist  $\mathcal{NP}$ -schwer. Wir zeigen  $3\text{-CNF-SAT} \leq_p \text{CLIQUE}$ . Daher müssen wir eine in Polynomialzeit berechenbare, totale Funktion  $f$  angeben, die jeder 3-CNF  $F$  einen Graph  $f(F)$  zuordnet, sodass gilt:  $F$  ist erfüllbar g.d.w.  $f(F)$  eine Clique der Größe mindestens  $k$  hat (wobei wir  $k$  selbst festlegen dürfen).

Sei  $F = K_1 \wedge \dots \wedge K_m$  eine 3-CNF, wobei  $K_i = (L_{i,1} \vee L_{i,2} \vee L_{i,3})$  für  $i = 1, \dots, m$ . Hierbei nehmen wir also an, dass jede Klausel aus genau 3 Literalen besteht (ist dies nicht der Fall, so vervielfachen wir das erste Literal). Für jedes  $L_{i,j}$  erzeugen wir einen Knoten  $(i, j)$  im Graphen, d.h.  $V = \bigcup_{i=1}^m \{(i, 1), (i, 2), (i, 3)\}$ . Für die Kantenmenge ziehen wir keine Kante innerhalb der drei Knoten, die aus einer Klausel stammen, sondern nur Kanten zwischen „verschiedenen Klauseln“, d.h.  $E \subseteq \{(i, j), (i', j')\} \mid i \neq i' \wedge i, i' \in \{1, \dots, m\} \wedge j, j' \in \{1, 2, 3\}\}$ . Dabei wird die maximale Menge an Kanten genommen, sodass sich niemals zwei verbundene Literale  $L_{i,j}$  und  $L_{i',j'}$  widersprechen (d.h.  $L_{i,j} \neq \overline{L_{i',j'}}$ , wobei  $\overline{L}$  das negierte Literal zu  $L$  ist:  $\overline{\overline{x}} = x$  und  $\overline{x} = \neg x$ ).

Das ergibt  $E := \{(i, j), (i', j')\} \mid i \neq i' \wedge i, i' \in \{1, \dots, m\} \wedge j, j' \in \{1, 2, 3\}, L_{i,j} \neq \overline{L_{i',j'}}\}$ . Die Abbildung  $f(F) = ((V, E), m)$  ist in Polynomialzeit berechenbar und total.

Es verbleibt zu zeigen, dass  $F$  erfüllbar ist g.d.w.  $(V, E)$  eine Clique der Größe mindestens  $m$  hat. Wenn  $F$  erfüllbar ist, dann gibt es eine Belegung  $I$  der Variablen, die in jeder Klausel mindestens ein Literal wahr macht. D.h. es gibt  $L_{1,j_1}, \dots, L_{m,j_m}$  mit  $I(L_{1,j_1}) = 1, \dots, I(L_{m,j_m}) = 1$ . Daher können sich diese Literale nicht widersprechen und sie sind im Graphen paarweise miteinander verbunden, d.h. sie formen eine Clique der Größe  $m$ . Umgekehrt, wenn  $(V, E)$  eine Clique der Größe mindestens  $m$  hat, dann muss es eine Knotenmenge  $V' = \{(i_1, j_1), \dots, (i_m, j_m)\}$  geben, die eine Clique der Größe  $m$  formen. Da Knoten  $(i, x)$  und  $(i, y)$  nie miteinander verbunden sind, müssen alle  $i_1, \dots, i_m$  paarweise verschieden sein, also  $\{i_1, \dots, i_m\} = \{1, \dots, m\}$  gelten. Damit folgt: Die Literale  $L_{i_1,j_1}, \dots, L_{i_m,j_m}$  widersprechen sich paarweise nicht und wir können eine Belegung  $I$  konstruieren mit  $I(x) = 1$  wenn  $L_{i_k,j_k} = x$  und  $I(x) = 0$  wenn  $L_{i,k,j_k} = \neg x$  und  $I(y) = 1$

für alle anderen Variablen, die nicht dadurch bestimmt werden. Da  $I$  je ein Literal in jeder Klausel wahr macht, gilt  $I(F) = 1$ .  $\square$

**Beispiel 8.5.6.** Sei  $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$

Dann ist  $f(F) = (V, E, 3)$  mit  $V = \{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)\}$  und  $E = \{ \{(1, 1), (2, 2)\}, \{(1, 1), (2, 3)\}, \{(1, 1), (3, 1)\}, \{(1, 1), (3, 2)\}, \{(1, 1), (3, 3)\}, \{(1, 2), (2, 1)\}, \{(1, 2), (2, 2)\}, \{(1, 2), (2, 3)\}, \{(1, 2), (3, 1)\}, \{(1, 2), (3, 3)\}, \{(1, 3), (2, 1)\}, \{(1, 3), (2, 2)\}, \{(1, 3), (2, 3)\}, \{(1, 3), (3, 1)\}, \{(1, 3), (3, 2)\}, \{(2, 1), (3, 2)\}, \{(2, 1), (3, 3)\}, \{(2, 2), (3, 1)\}, \{(2, 2), (3, 3)\}, \{(2, 3), (3, 1)\}, \{(2, 3), (3, 2)\} \}$  und kann wie in Abb. 8.3 dargestellt gezeichnet werden. Z.B. ist  $\{(1, 1), (2, 2), (3, 3)\}$  eine Clique der Größe 3. Die zugehörige

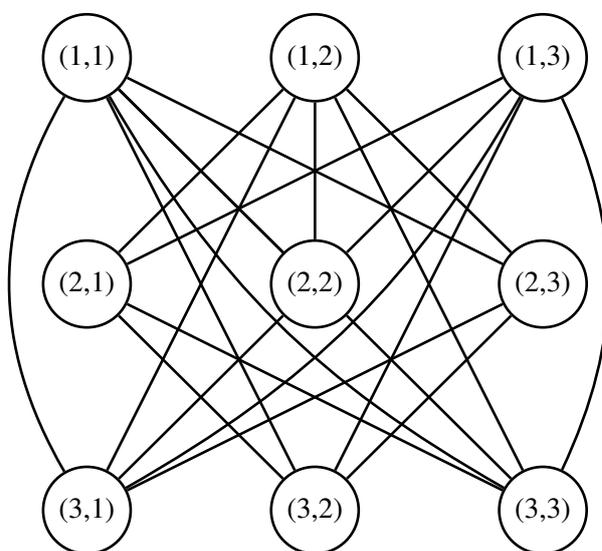


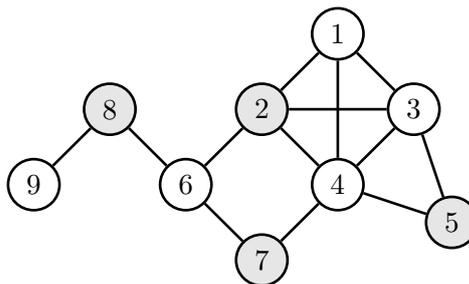
Abbildung 8.3.: Graph zu Beispiel Beispiel 8.5.6

Belegung ist  $I(x_1) = 1, I(x_2) = 1, I(x_3) = 0$ .

### 8.5.3. Das INDEPENDENT-SET-Problem

Für einen ungerichteten Graphen  $G = (V, E)$  ist  $V' \subseteq V$  eine *unabhängige Knotenmenge*, wenn keine zwei Knoten aus  $V'$  über eine Kante verbunden sind, d.h.  $u, v \in V' \implies \{u, v\} \notin E$ .

**Beispiel 8.5.7.** Der Graph



hat mehrere unabhängige Knotenmengen der Größe 4, z.B.  $\{2, 5, 7, 8\}$ , aber keine unabhängige Knotenmenge der Größe 5.

Für einen Graphen  $G = (V, E)$  ist der Komplementgraph zu  $G$ , der Graph  $\bar{G} = (V, \bar{E})$  mit  $\bar{E} = \{\{u, v\} \mid u, v \in V, u \neq v, \{u, v\} \notin E\}$ .

**Lemma 8.5.8.** Für jeden ungerichteten Graph  $G$  gilt:  $G$  hat eine unabhängige Knotenmenge der Größe  $k$  genau dann, wenn  $\bar{G}$  eine Clique der Größe  $k$  hat.

*Beweis.* Sei  $G = (V, E)$ . Dann gilt:

$V'$  ist unabhängige Knotenmenge der Größe  $k$

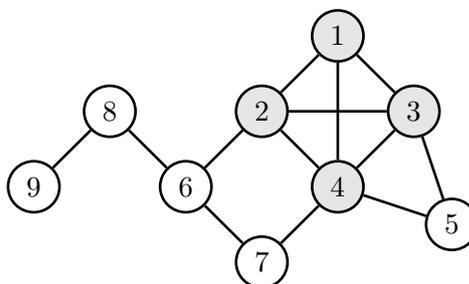
g.d.w.  $V' \subseteq V$  mit  $u, v \in V' \implies \{u, v\} \notin E$  und  $|V'| = k$

g.d.w.  $u, v \in V' \implies \{u, v\} \in \bar{E}$

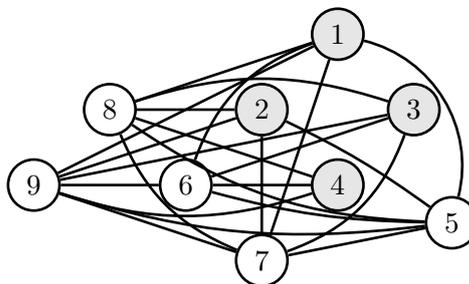
g.d.w.  $V'$  ist eine Clique der Größe  $k$  in  $\bar{G}$

□

**Beispiel 8.5.9.** Der Graph



hat eine Clique der Größe 4, nämlich  $\{1, 2, 3, 4\}$ . Der Komplementgraph dazu ist:



Der Komplementgraph hat  $\{1, 2, 3, 4\}$  als unabhängige Knotenmenge.

**Definition 8.5.10** (INDEPENDENT-SET-Problem). Das INDEPENDENT-SET-Problem lässt sich in der gegeben/gefragt-Notation formulieren durch:

gegeben: Ein ungerichteter Graph  $G = (V, E)$  und eine Zahl  $k \in \mathbb{N}_0$ .

gefragt: Besitzt  $G$  eine unabhängige Knotenmenge der Größe mindestens  $k$ ?

**Satz 8.5.11.** INDEPENDENT-SET ist NP-vollständig.

*Beweis.* Wir zeigen die beiden benötigten Teile:

- INDEPENDENT-SET  $\in \mathcal{NP}$ : Es gibt eine NTM, die zunächst alle Knoten und die Zahl  $k$  aus der Beschreibung extrahiert und anschließend nichtdeterministisch eine Menge von  $k$  Knoten rät und für jede der nichtdeterministischen Möglichkeiten verifiziert, ob für jedes Paar  $\{u, v\}$  aus der geratenen Menge gilt  $\{u, v\} \notin E$ . Die Verifikation kann in polynomieller Laufzeit durchgeführt werden. Daher ist die Laufzeit der NTM polynomiell beschränkt.
- INDEPENDENT-SET ist NP-schwer. Wir zeigen  $\text{CLIQUE} \leq_p \text{INDEPENDENT-SET}$ . Sei  $f((V, E, m)) = (V, \bar{E}, m)$  wobei  $\bar{E} = \{\{u, v\} \mid u, v \in V, \{u, v\} \notin E\}$ . Dann gilt  $(V, E)$  hat eine Clique der Größe  $m$  genau dann, wenn  $(V, \bar{E})$  eine unabhängige Knotenmenge der Größe  $m$  hat. Da die Funktion  $f$  in Polynomialzeit berechnet werden kann, gilt  $\text{CLIQUE} \leq_p \text{INDEPENDENT-SET}$  und da  $\text{CLIQUE}$  NP-schwer, folgt auch INDEPENDENT-SET ist NP-schwer.  $\square$

#### 8.5.4. Das VERTEX-COVER-Problem

Für einen ungerichteten Graph  $G = (V, E)$  ist eine überdeckende Knotenmenge eine Teilmenge  $V' \subseteq V$  aller Knoten, sodass jede Kante mindestens einen ihrer beiden Knoten in der Menge hat, d.h. es gilt für alle Knoten  $u, v \in V : \{u, v\} \in E \implies u \in V' \vee v \in V'$ .

**Lemma 8.5.12.**  $G = (V, E)$  hat eine unabhängige Knotenmenge der Größe  $k$ , g.d.w.  $G$  hat eine überdeckende Knotenmenge der Größe  $|V| - k$ .

*Beweis.*

Es gilt:

$V' \subseteq V$  ist unabhängige Knotenmenge

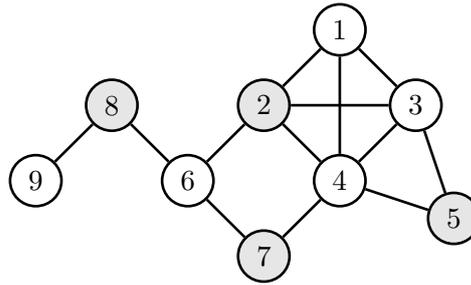
g.d.w.  $u, v \in V' \implies \{u, v\} \notin E$

g.d.w.  $\{u, v\} \in E \implies u \notin V' \vee v \notin V'$

g.d.w.  $\{u, v\} \in E \implies (u \in V \setminus V') \vee (v \in V \setminus V')$

g.d.w.  $V \setminus V'$  ist überdeckende Knotenmenge  $\square$

**Beispiel 8.5.13.** Der Graph



hat mehrere unabhängige Knotenmengen der Größe 4, z.B.  $\{2, 5, 7, 8\}$ . Nach dem vorherigen Lemma gilt  $V \setminus \{2, 5, 7, 8\} = \{1, 3, 4, 6, 9\}$  ist eine überdeckende Knotenmenge, was sich auch am Graphen verifizieren lässt: Jede Kante enthält mindestens einen weißen Knoten.

**Definition 8.5.14** (VERTEX-COVER-Problem). Das VERTEX-COVER-Problem lässt sich in der gegeben/gefragt-Notation formulieren durch:

gegeben: Ein ungerichteter Graph  $G = (V, E)$  und eine Zahl  $k \in \mathbb{N}_0$ .

gefragt: Besitzt  $G$  eine überdeckende Knotenmenge der Größe höchstens  $k$ ?

**Satz 8.5.15.** VERTEX-COVER ist  $\mathcal{NP}$ -vollständig.

*Beweis.* Wir zeigen die beiden benötigten Teile. Sei  $G = (V, E)$  ein ungerichteter Graph.

- VERTEX-COVER  $\in \mathcal{NP}$ : Es gibt eine NTM, die zunächst alle Knoten und die Zahl  $k$  aus der Beschreibung extrahiert und anschließend nichtdeterministisch eine Menge  $V' \subseteq V$  von  $k$  Knoten rät und für jede der nichtdeterministischen Möglichkeiten verifiziert, ob für jede Kante mindestens einer ihrer beteiligten Knoten in der Menge ist. Die Verifikation kann in polynomieller Laufzeit durchgeführt werden. Daher ist die Laufzeit der NTM polynomiell beschränkt.
- VERTEX-COVER ist  $\mathcal{NP}$ -schwer. Wir zeigen INDEPENDENT-SET  $\leq_p$  VERTEX-COVER. Sei  $f((V, E, m)) = (V, E, |V| - m)$ . Dann gilt (wie eben gezeigt)  $(V, E)$  hat eine unabhängige Knotenmenge der Größe mindestens  $m$  g.d.w.  $(V, E)$  hat eine überdeckende Knotenmenge der Größe höchstens  $|V| - m$ . Da  $f$  in Polynomialzeit berechnet werden kann, gilt INDEPENDENT-SET  $\leq_p$  VERTEX-COVER.  $\square$

### 8.5.5. Das SETCOVER-Problem

**Definition 8.5.16** (SETCOVER-Problem). Das SETCOVER-Problem lässt sich in der gegeben/gefragt-Notation formulieren durch:

gegeben: Ein System von Mengen  $T_1, \dots, T_k$  mit  $T_1, \dots, T_k \subseteq M$ , wobei  $M$  eine endliche Grundmenge ist und eine Zahl  $n \leq k$ .

gefragt: Gibt es eine Auswahl von  $n$  Mengen  $T_{i_1}, \dots, T_{i_n}$  ( $i_j \in \{1, \dots, k\}$ ) mit  $T_{i_1} \cup \dots \cup T_{i_n} = M$ ?

**Beispiel 8.5.17.** Für  $T_1 = \{1, 2, 3, 5\}$ ,  $T_2 = \{1, 2\}$ ,  $T_3 = \{3, 4\}$ ,  $T_4 = \{3\}$  (mit  $M = \{1, 2, 3, 4, 5\}$ ) und  $n = 2$  gibt es die Lösung  $T_1, T_3$ , da  $T_1 \cup T_3 = M$ .

**Satz 8.5.18.** SETCOVER ist NP-vollständig.

*Beweis.* Wir zeigen die beiden benötigten Teile:

- SETCOVER  $\in$  NP: Es gibt eine NTM, die zunächst alle Mengen und die Zahl  $n$  aus der Beschreibung extrahiert und anschließend nichtdeterministisch  $n$  Mengen des Mengensystems rät. Für jede der nichtdeterministischen Möglichkeiten wird anschließend (jeweils deterministisch) verifiziert, ob die Vereinigung der geratenen Menge die Grundmenge  $M$  ergibt. Die Verifikation kann in polynomieller Laufzeit durchgeführt werden. Daher ist die Laufzeit der NTM polynomiell beschränkt.
- SETCOVER ist NP-schwer. Wir zeigen  $3\text{-CNF-SAT} \leq_p \text{SETCOVER}$ . Sei  $F = K_1 \wedge \dots \wedge K_m$  eine 3-CNF. Seien  $\{x_1, \dots, x_n\}$  die aussagenlogischen Variablen, die in  $F$  vorkommen. Setze  $M = \{1, \dots, m, m+1, \dots, m+n\}$ . Für  $i = 1, \dots, n$  sei

$$\begin{aligned} T_{i,a} &= \{j \mid \text{Literal } x_i \text{ kommt in Klausel } K_j \text{ vor}\} \cup \{m+i\} \\ T_{i,b} &= \{j \mid \text{Literal } \neg x_i \text{ kommt in Klausel } K_j \text{ vor}\} \cup \{m+i\} \end{aligned}$$

Das Mengensystem sei  $T_{1,a}, \dots, T_{n,a}, T_{1,b}, \dots, T_{n,b} \subseteq M$ .

Wenn  $F$  erfüllbar, dann hat SETCOVER eine Lösung: Sei  $I$  eine Belegung mit  $I(F) = 1$ . Wenn  $I(x_i) = 1$ , dann wähle die Menge  $T_{i,a}$  und wenn  $I(x_i) = 0$ , dann wähle die Menge  $T_{i,b}$ . Die so gewählten Mengen bestehen aus  $n$  Mengen und jede Zahl aus  $M$  kommt in ihnen vor: Da jede Klausel durch  $I$  wahr gemacht wird, kommt jede Zahl  $1, \dots, m$  vor, da jede Variable mit 0 oder 1 belegt wurde, kommt jede Zahl  $m+1, \dots, m+n$  vor.

Umgekehrt gilt: Wenn es  $n$  Mengen  $U_1, \dots, U_n \subseteq T_{1,a}, \dots, T_{n,a}, T_{1,b}, \dots, T_{n,b}$  gibt, so dass  $U_1 \cup \dots \cup U_n = M$ , dann ist  $F$  erfüllbar: Damit die Zahlen  $m+i$  für  $i = 1, \dots, n$  alle enthalten sind, muss  $T_{i,a}$  oder  $T_{i,b}$  für jedes  $i = 1, \dots, n$  unter den  $U_i$  sein. Daher O.B.d.A.  $U_i \in \{T_{i,a}, T_{i,b}\}$  für  $i = 1, \dots, n$ . Wenn wir nun  $I(x_i) = 1$  setzen, wenn  $U_i = T_{i,a}$  und  $I(x_i) = 0$ , wenn  $U_i = T_{i,b}$ , dann erfüllt  $I$  die Formel  $F$ : Da die Zahlen  $1, \dots, m$  alle enthalten sind in  $\bigcup U_i$ , wird in jeder Klausel mindestens ein Literal auf wahr gesetzt.

Sei  $f$  die Funktion die aus  $F$ , das Mengensystem  $T_{1,a}, \dots, T_{n,a}, T_{1,b}, \dots, T_{n,b} \subseteq M$  und  $n$  als Zahl berechnet. Dann ist  $f$  in Polynomialzeit berechenbar und hat die Eigenschaft dass  $F$  erfüllbar ist g.d.w.  $f(F)$  lösbare SETCOVER-Instanz ist. Damit haben wir  $3\text{-CNF-SAT} \leq_p \text{SETCOVER}$  gezeigt.  $\square$

### 8.5.6. Das SUBSETSUM-Problem

**Definition 8.5.19** (SUBSETSUM-Problem). Das SUBSETSUM-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:

gegeben: Natürliche Zahlen  $a_1, \dots, a_k \in \mathbb{N}_0$  und  $s \in \mathbb{N}_0$   
 gefragt: Gibt es eine Teilmenge  $I \subseteq \{1, \dots, k\}$ , sodass  $\sum_{i \in I} a_i = s$ ?

**Satz 8.5.20.** Das SUBSETSUM-Problem ist  $\mathcal{NP}$ -vollständig.

*Beweis.* Wir zeigen die beiden benötigten Teile:

- SUBSETSUM  $\in \mathcal{NP}$ : Rate nichtdeterministisch eine Teilmenge  $I \subseteq \{1, \dots, k\}$  und anschließend verifiziere, dass  $\sum_{i \in I} a_i = s$  gilt.
- SUBSETSUM ist  $\mathcal{NP}$ -schwer: Wir geben eine Polynomialzeitreduktion von 3-CNF-SAT auf SUBSETSUM an. Sei  $F = K_1 \wedge \dots \wedge K_m$  eine 3-CNF, wobei jede Klausel genau drei Literale habe (dies kann durch Vervielfachen von Literalen stets hergestellt werden). Seien  $x_1, \dots, x_n$  die aussagenlogischen Variablen, die in  $F$  vorkommen. Wir erzeugen  $n + m$ -stellige Zahlen  $t_i, f_i$  für  $i = 1, \dots, n$ :
  - In  $t_i$  ist die  $i$ -te Stelle 1, und für  $j = 1, \dots, m$  ist jede  $n + j$ -te Stelle 1, falls Klausel  $K_j$  das Literal  $x_i$  enthält.
  - In  $f_i$  ist die  $i$ -te Stelle 1 und für  $j = 1, \dots, m$  ist jede  $n + j$ -te Stelle 1, falls Klausel  $K_j$  das Literal  $\neg x_i$  enthält.
  - Alle anderen Stellen der Zahlen  $t_i, f_i$  sind 0.

Schließlich gibt es noch Zahlen  $c_j, d_j$  für  $j = 1, \dots, m$ :

- In  $c_j$  ist genau die  $n + j$ -te Stelle 1 und alle anderen sind 0.
- In  $d_j$  ist genau die  $n + j$ -te Stelle 2 und alle anderen sind 0.

Schließlich sei  $s = \underbrace{1 \dots 1}_{n\text{-mal}} \underbrace{4 \dots 4}_{m\text{-mal}}$ .

Sei also  $f(F) = ((t_1, \dots, t_n, f_1, \dots, f_n, c_1, \dots, c_m, d_1, \dots, d_m), s)$ . Offensichtlich kann  $f$  in Polynomialzeit berechnet werden. Die folgenden Aussagen gelten:

- Die Summe jeder Teilmenge der Zahlen erzeugt keine Überträge.
- Die  $n$  Einsen in  $s$  sorgen dafür, dass in  $I$  jeweils  $t_i$  oder  $f_i$  enthalten ist aber nicht beide.
- Die Wahl der  $t_i$  und  $f_i$  zählt gleichzeitig in den Stellen  $n+1$  bis  $n+m$ , welche Klauseln durch Setzen von  $x_i$  auf 1 (bzw.  $x_i$  auf 0) wahr gemacht werden. In der Summe kann dies pro Klausel eine Zahl zwischen 0 und 3 sein (je nachdem, ob 0, 1, 2 oder 3 Literale der Klausel auf wahr gesetzt werden). Durch Hinzunahme der  $c_j$  und/oder  $d_j$  kann die Zielsumme 4 pro Stelle  $j$  stets erreicht werden, wenn mindestens 1 Literal wahr ist.

Daher gilt:

Wenn SUBSETSUM-Instanz  $((t_1, \dots, t_n, f_1, \dots, f_n, c_1, \dots, c_m, d_1, \dots, d_m), s)$  eine Lösung  $I$  hat, kann daraus eine erfüllende Belegung  $B$  für  $F$  konstruiert werden: Setze für  $i \in I$  mit  $1 \leq i \leq n$ :  $B(x_i) = 1$  und setze für  $i \in I$  mit  $n + 1 \leq i \leq n + n$ :  $B(x_i) = 0$ . Umgekehrt kann eine Belegung  $B$  für  $F$  benutzt werden, um die Indizes  $I$  der Lösung für das SUBSETSUM-Problem zu konstruieren:  $I$  enthält den Index von  $t_i$  wenn  $B(x_i) = 1$ ,  $I$  enthält den Index von  $f_i$  wenn  $B(x_i) = 0$ ,  $I$  enthält die Indizes der  $c_j, d_j$ , sodass sich die

hinteren  $m$  Stellen zu 4 aufsummieren, wobei für jede Stelle die Summe schon mindestens 1 durch die Wahl der  $t_i, f_i$  ist (da  $B$  erfüllende Belegung).

Damit haben wir gezeigt  $3\text{-CNF-SAT} \leq_p \text{SUBSETSUM}$  und aus der  $\mathcal{NP}$ -Schwere von  $3\text{-CNF-SAT}$  folgt die  $\mathcal{NP}$ -Schwere von  $\text{SUBSETSUM}$ .  $\square$

**Beispiel 8.5.21.** Die 3-CNF  $(x_1 \vee x_1 \vee x_1) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_1)$  wird durch die eben eingeführte Übersetzung  $f$  in die  $\text{SUBSETSUM}$ -Instanz

$$\begin{array}{rcl} a_1 & = & t_1 = 110 \\ a_2 & = & f_1 = 101 \\ a_3 & = & c_1 = 010 \\ a_4 & = & c_2 = 001 \\ a_5 & = & d_1 = 020 \\ a_6 & = & d_2 = 002 \\ s & & = 144 \end{array}$$

transformiert. Diese ist unlösbar (was dazu passt, dass die Formel unerfüllbar ist).

**Beispiel 8.5.22.** Betrachte die 3-CNF

$$(x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee x_4)$$

Dann erzeugt  $f$  die Zahlen:

$$\begin{array}{rcl} a_1 & = & t_1 = 100010100 \\ a_2 & = & t_2 = 010011010 \\ a_3 & = & t_3 = 001000001 \\ a_4 & = & t_4 = 000100101 \\ a_5 & = & f_1 = 100001010 \\ a_6 & = & f_2 = 010000101 \\ a_7 & = & f_3 = 001001010 \\ a_8 & = & f_4 = 000110000 \\ a_9 & = & c_1 = 000010000 \\ a_{10} & = & c_2 = 000001000 \\ a_{11} & = & c_3 = 000000100 \\ a_{12} & = & c_4 = 000000010 \\ a_{13} & = & c_5 = 000000001 \\ a_{14} & = & d_1 = 000020000 \\ a_{15} & = & d_2 = 000002000 \\ a_{16} & = & d_3 = 000000200 \\ a_{17} & = & d_4 = 000000020 \\ a_{18} & = & d_5 = 000000002 \\ s & & = 111144444 \end{array}$$

Eine Lösung ist  $I = \{2, 4, 5, 7, 9, 10, 11, 12, 13, 14, 16, 18\}$ . Die zugehörige erfüllende Belegung ist  $B(x_1) = 0, B(x_2) = 1, B(x_3) = 0, B(x_4) = 1$ .

### 8.5.7. Das KNAPSACK-Problem

**Definition 8.5.23** (KNAPSACK-Problem). Das KNAPSACK-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:

gegeben:  $k$  Gegenstände mit Gewichten  $w_1, \dots, w_k \in \mathbb{N}_0$  und Nutzenwerten  $n_1, \dots, n_k \in \mathbb{N}_0$ , sowie zwei Zahlen  $s_w, s_n \in \mathbb{N}_0$ .

gefragt: Gibt es eine Teilmenge  $I \subseteq \{1, \dots, k\}$ , sodass  $\sum_{i \in I} w_i \leq s_w$  und  $\sum_{i \in I} n_i \geq s_n$ ?

Beachte für  $w_i = n_i$  und  $s_n = s_w$  ergibt sich genau das SUBSETSUM-Problem, es ist daher ein Spezialfall. Daher ist die Reduktion  $\text{SUBSETSUM} \leq_p \text{KNAPSACK}$  sehr einfach.

**Satz 8.5.24.** *KNAPSACK ist  $\mathcal{NP}$ -vollständig.*

*Beweis.* Wir zeigen die beiden benötigten Teile:

- $\text{KNAPSACK} \in \mathcal{NP}$ : Rate eine Teilmenge  $I \subseteq \{1, \dots, k\}$  nichtdeterministisch, anschließend prüfe, ob  $\sum_{i \in I} w_i \leq s_w$  und  $\sum_{i \in I} n_i \geq s_n$  gilt. Falls ja, dann akzeptiere. Die Laufzeit der dies ausführenden NTM ist polynomiell beschränkt.
- $\text{KNAPSACK}$  ist  $\mathcal{NP}$ -schwer: Sei  $((a_1, \dots, a_k), s)$  eine SUBSETSUM-Instanz, dann sei  $f((a_1, \dots, a_k), s) = ((w_1, \dots, w_k), (n_1, \dots, n_k), s_w, s_m)$  mit  $w_i = a_i, n_i = a_i$  für  $i = 1, \dots, k$  und  $s_w = s$  und  $s_m = s$ . Es gilt  $((a_1, \dots, a_k), s)$  hat eine Lösung g.d.w.  $f((a_1, \dots, a_k), s)$  hat eine Lösung und  $f$  ist in polynomieller Zeit von einer DTM berechenbar. Daher gilt  $\text{SUBSETSUM} \leq_p \text{KNAPSACK}$ . Damit folgt aus der  $\mathcal{NP}$ -Schwere von SUBSETSUM auch die  $\mathcal{NP}$ -Schwere von KNAPSACK.  $\square$

### 8.5.8. Das PARTITION-Problem

**Definition 8.5.25** (PARTITION-Problem). *Das PARTITION-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:*

gegeben: Natürliche Zahlen  $a_1, \dots, a_k \in \mathbb{N}_0$

gefragt: Gibt es eine Teilmenge  $I \subseteq \{1, \dots, k\}$  sodass  $\sum_{i \in I} a_i = \sum_{i \in \{1, \dots, k\} \setminus I} a_i$ ?

**Satz 8.5.26.** *PARTITION ist  $\mathcal{NP}$ -vollständig.*

*Beweis.* Wir zeigen die beiden benötigten Teile:

- $\text{PARTITION} \in \mathcal{NP}$ : Rate nichtdeterministisch  $I \subseteq \{1, \dots, k\}$  und prüfe anschließend, ob  $\sum_{i \in I} a_i = \sum_{i \in \{1, \dots, k\} \setminus I} a_i$  gilt. Die dazu passende NTM läuft in (nichtdeterministischer) Polynomialzeit.
- $\text{PARTITION}$  ist  $\mathcal{NP}$ -schwer: Wir zeigen  $\text{SUBSETSUM} \leq_p \text{PARTITION}$ . Sei  $f((a_1, \dots, a_k), s) = (a_1, \dots, a_k, a_{k+1}, a_{k+2})$  mit  $a_{k+1} = A + s$  und  $a_{k+2} = 2A - s$ , wobei  $A = \sum_{i=1}^k a_i$ . Die Funktion  $f$  kann von einer DTM in polynomieller Zeit berechnet werden.

Wenn  $(a_1, \dots, a_k, a_{k+1}, a_{k+2})$  eine PARTITION-Lösung  $I \subseteq \{1, \dots, k+2\}$  hat, dann gilt  $\sum_{i \in I} a_i = 2A = \sum_{i \in \{1, \dots, k+2\} \setminus I} a_i$ . Dann können nicht beide Indizes  $k+1$  und  $k+2$  in

$I$  liegen, da sonst die Summe zu groß ist. Wenn  $k+2 \in I$ , dann sei  $I' = I \setminus \{k+2\}$  und damit  $\sum_{i \in I'} = 2A - (2A - s) = s$ , d.h.  $I'$  ist eine Lösung für die SUBSETSUM-Instanz  $((a_1, \dots, a_k), s)$ . Wenn  $k+1 \in I$ , dann sei  $I' = \{1, \dots, k\} \setminus I$  und damit  $\sum_{i \in I'} = 2A - (2A - s) = s$  und  $I'$  ist eine Lösung für die SUBSETSUM-Instanz  $((a_1, \dots, a_k), s)$ .

Wenn  $I \subseteq \{1, \dots, k\}$  eine Lösung für die SUBSETSUM-Instanz  $((a_1, \dots, a_k), s)$ , dann ist  $I \cup \{k+2\}$  eine Lösung für die PARTITION-Instanz  $(a_1, \dots, a_k, a_{k+1}, a_{k+2})$ .

Daher gilt  $((a_1, \dots, a_k), s)$  ist lösbar g.d.w.  $f((a_1, \dots, a_k), s)$  ist lösbar. Damit haben wir  $\text{SUBSETSUM} \leq_p \text{PARTITION}$  gezeigt. Da SUBSETSUM NP-schwer ist, ist auch PARTITION NP-schwer.  $\square$

**Beispiel 8.5.27.** Sei  $((1,2,3,4,5,6), 14)$  eine SUBSETSUM-Instanz. Dann ist  $(1,2,3,4,5,6,35,28)$  die von der Funktion  $f$  erzeugte PARTITION-Instanz. Eine Lösung ist  $I = \{1, 3, 4, 6, 8\}$  da  $1 + 3 + 4 + 6 + 28 = 42 = 2 + 5 + 35$ . Damit ist  $I' = \{1, 3, 4, 6\}$  eine Lösung der SUBSETSUM-Instanz (was auch stimmt, da  $1 + 3 + 4 + 6 = 14$ ).

### 8.5.9. Das BINPACKING-Problem

**Definition 8.5.28** (BINPACKING-Problem). Das BINPACKING-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:

gegeben: Natürliche Zahlen  $a_1, \dots, a_k \in \mathbb{N}_0$ , die Behältergröße  $b \in \mathbb{N}_0$ , und die Anzahl der Behälter  $m$

gefragt: Kann man alle gegebenen Zahlen, so auf die Behälter aufteilen, sodass keiner der Behälter überläuft? Formal: Gibt es eine totale Funktion  $\text{assign} : \{1, \dots, k\} \rightarrow \{1, \dots, m\}$ , sodass für alle  $j \in \{1, \dots, m\}$  gilt:  $\sum_{\text{assign}(i)=j} a_i \leq b$ ?

**Satz 8.5.29.** BINPACKING ist NP-vollständig.

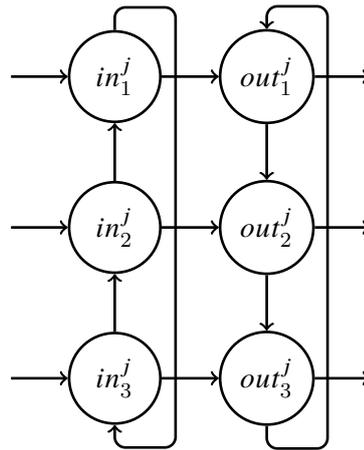
*Beweis.* Wir zeigen die beiden benötigten Teile:

- BINPACKING  $\in$  NP: Rate nichtdeterministisch für jede Zahl  $a_i$  in welchen Behälter sie gehört. (D.h. rate die Funktion  $\text{assign}$ .) Anschließend verifiziere, dass die geratene Funktion eine Lösung ist. Dies kann in Polynomialzeit auf einer NTM durchgeführt werden.
- BINPACKING ist NP-schwer. Sei  $(a_1, \dots, a_k)$  eine PARTITION-Instanz. Dann sei  $f(a_1, \dots, a_k)$  die BINPACKING-Instanz mit Zahlen  $a_1, \dots, a_k$ , Behältergröße  $b = \lfloor \frac{\sum_{i=1}^k a_i}{2} \rfloor$  und  $m = 2$  Behältern.

Wenn  $\sum_{i=1}^k a_i$  eine ungerade Zahl ist, dann ist die PARTITION-Instanz unlösbar und die BINPACKING-Instanz ebenso (da die Behälter nicht ausreichen). Anderenfalls sei  $I \subseteq \{1, \dots, k\}$  eine Lösung für die PARTITION-Instanz, dann ist

$$\text{assign}(i) = \begin{cases} 1, & \text{wenn } i \in I \\ 2, & \text{wenn } i \notin I \end{cases}$$

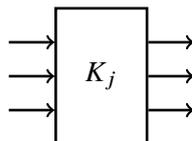




Dieser Teilgraph hat sehr besondere Eigenschaften:

- Jeder Hamilton-Kreis, der durch den Eingang  $in_i^j$  in  $K_j$  hereingeht, muss  $K_j$  durch den Knoten  $out_i^j$  verlassen: Anderenfalls gibt es „blockierte Knoten“, die durch den Hamilton-Kreis nicht mehr besucht werden können (betrachte alle endlich vielen Fälle).
- Ein Hamilton-Kreis, der in  $in_1^j$  in  $K_j$  hereingeht und  $K_j$  durch  $out_1^j$  verlässt, kann durch
  - \*  $in_1^j, out_1^j$ , oder
  - \*  $in_1^j, in_3^j, out_3^j, out_1^j$ , oder
  - \*  $in_1^j, in_3^j, in_2^j, out_2^j, out_3^j, out_1^j$
 in dieser Reihenfolge laufen.
- Ein Hamilton-Kreis, der in  $in_2^j$  in  $K_j$  hereingeht und  $K_j$  durch  $out_2^j$  verlässt, kann durch
  - \*  $in_2^j, out_2^j$ , oder
  - \*  $in_2^j, in_1^j, out_1^j, out_2^j$ , oder
  - \*  $in_2^j, in_1^j, in_3^j, out_3^j, out_1^j, out_2^j$
 in dieser Reihenfolge laufen.
- Ein Hamilton-Kreis, der in  $in_3^j$  in  $K_j$  hereingeht und  $K_j$  durch  $out_3^j$  verlässt, kann durch
  - \*  $in_3^j, out_3^j$ , oder
  - \*  $in_3^j, in_2^j, out_2^j, out_3^j$ , oder
  - \*  $in_3^j, in_2^j, in_1^j, out_1^j, out_2^j, out_3^j$
 in dieser Reihenfolge laufen.
- Je nachdem, wie daher  $K_j$  durch einen Hamilton-Kreis durchlaufen wird, kann  $K_j$  einmal, zweimal, oder dreimal durchquert werden.

Wir stellen  $K_j$  abstrakt durch



dar. Für jede Klausel  $K_j$  gibt es einen Teilgraphen  $K_j$ . Wenn  $K_j = L_1^j \vee L_2^j \vee L_3^j$ , dann wird Eingang  $in_1^j$  und Ausgang  $out_1^j$  entsprechend  $L_1^j$ , Eingang  $in_2^j$  und Ausgang  $out_2^j$  entsprechend  $L_2^j$ , Eingang  $in_3^j$  und Ausgang  $out_3^j$  entsprechend  $L_3^j$  verbunden.

Wenn  $L_k^j = x_i$ , dann liegt  $in_k^j$  nach  $out_k^j$  auf dem *oberen* Pfad zwischen Knoten  $x_i$  und  $x_{i+1}$  (bzw.  $x_n$  und  $x_1$  für  $i = n$ ).

Wenn  $L_k^j = \neg x_i$ , dann liegt  $in_k^j$  nach  $out_k^j$  auf dem *unteren* Pfad zwischen Knoten  $x_i$  und  $x_{i+1}$  (bzw.  $x_n$  und  $x_1$  für  $i = n$ ).

Beachte, dass zwischen  $x_i$  und  $x_{i+1}$  alle  $K_j$  durchlaufen werden, deren zugehörige Klauseln  $K_j$  die Variable  $x_i$  (als positives oder negative Literal) enthalten. Die Reihenfolge der  $K_j$  ist dabei irrelevant (wir nehmen die geordnete Reihenfolge an: Wenn  $K_j$  und  $K_{j'}$  zwischen  $x_i$  und  $x_{i+1}$  durch denselben Pfad (den oberen oder den unteren) durchlaufen werden, und  $j < j'$ , dann laufe erst durch  $K_j$  und dann durch  $K_{j'}$ ).

Beachte, dass diese Konstruktion in Polynomialzeit durchgeführt werden kann, indem zunächst die Knoten  $x_1, \dots, x_n$  erzeugt werden und die jeweils oberen und unteren Pfade von  $x_i$  und  $x_{i+1}$  direkt verbunden werden. Anschließend werden die Klauseln  $K_1, \dots, K_m$  in dieser Reihenfolge abgearbeitet werden, um die Verbindungen zwischen  $x_i$  und  $K_j$  und  $K_j$  und  $x_{i+1}$  zu erzeugen (durch Auftrennen der bestehenden Verbindungen).

Insgesamt werden dadurch alle Ein- und Ausgänge jedes  $K_i$  angeschlossen, d.h. der Graph ist wohl geformt.

Wenn der Graph einen Hamilton-Kreis hat, dann läuft der Kreis entweder oben durch  $x_i$  oder unten durch  $x_i$ . Dementsprechend konstruieren wir die Belegung  $I(x_i) = 1$  bzw.  $I(x_i) = 0$ . Der Hamilton-Kreis durchläuft jedes  $K_j$  mindestens einmal (bis zu dreimal). Die Anzahl der Durchläufe korrespondiert genau dazu, wie viele Literale in der Klausel durch  $I$  wahr gemacht werden. Daher ist  $I(F) = 1$ . Umgekehrt, wenn  $I$  eine Belegung mit  $I(F) = 1$  ist, dann hat der Graph einen Hamilton-Kreis: Durchlaufe die Knoten  $x_i$  oben, wenn  $I(x_i) = 1$  und unten, wenn  $I(x_i) = 0$ . Dabei wird  $K_j$  ein bis drei Mal (entsprechend der oben gezeigten Möglichkeiten) durchlaufen, je nachdem welche und wie viele Literale in der Klausel  $K_j$  durch  $I$  wahr gemacht werden.

Dies zeigt:  $3\text{-CNF-SAT} \leq_p \text{DIRECTED-HAMILTON-CYCLE}$  und damit die  $\mathcal{NP}$ -Schwere von  $\text{DIRECTED-HAMILTON-CYCLE}$ .

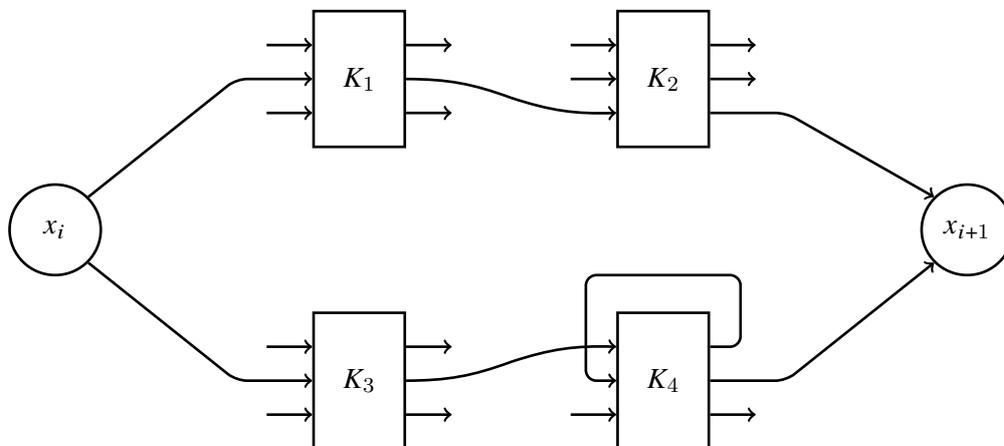
□

**Beispiel 8.5.32.** Sei  $F = K_1 \wedge K_2 \wedge K_3 \wedge K_4$  eine 3-CNF, sodass

- $x_i$  kommt an 2. Position in  $K_1$  vor
- $x_i$  kommt an 3. Position in  $K_2$  vor

- $\neg x_i$  kommt an 2. Position in  $K_3$  vor
- $\neg x_i$  kommt an 1. und 2. Position in  $K_4$  vor

Der konstruierte Ausschnitt des Graphen dazu ist



### 8.5.11. Das UNDIRECTED-HAMILTON-CYCLE-Problem

**Definition 8.5.33** (UNDIRECTED-HAMILTON-CYCLE-Problem). Das UNDIRECTED-HAMILTON-CYCLE-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:

gegeben: Ein ungerichteter Graph  $G = (V, E)$  mit  $V = \{v_1, \dots, v_n\}$

gefragt: Gibt es einen Hamilton-Kreis in  $G$ , d.h. einen Kreis, der genau alle Knoten einmal besucht? Formaler kann dies als Frage formuliert werden: Gibt es eine Permutation  $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ , sodass  $\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$  und  $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$ ?

**Satz 8.5.34.** UNDIRECTED-HAMILTON-CYCLE ist NP-vollständig.

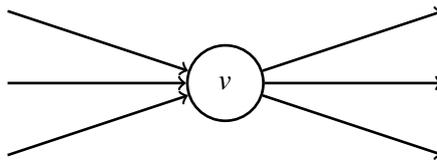
*Beweis.* Wir zeigen die beiden benötigten Teile:

- UNDIRECTED-HAMILTON-CYCLE  $\in$  NP: Rate die Permutation  $\pi$  nichtdeterministisch und verifiziere anschließend, ob  $\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$  und  $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$  gilt. Dies kann auf einer NTM in Polynomialzeit durchgeführt werden.
- UNDIRECTED-HAMILTON-CYCLE ist NP-schwer. Wir zeigen DIRECTED-HAMILTON-CYCLE  $\in$  NP  $\leq_p$  UNDIRECTED-HAMILTON-CYCLE  $\in$  NP. Sei  $f$  die Funktion, die aus einem gerichteten Graphen  $(V, E)$  einen ungerichteten Graphen macht, sodass

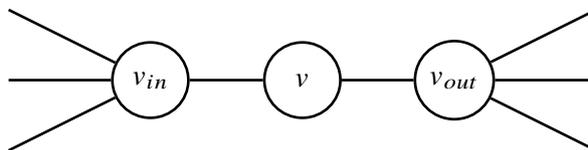
$$f(V) = \bigcup \{ \{v_{in}, v, v_{out}\} \mid v \in V \}$$

$$f(E) = \{ \{u_{out}, v_{in}\} \mid (u, v) \in E \} \cup \bigcup \{ \{ \{v_{in}, v\}, \{v, v_{out}\} \} \mid v \in V \}$$

D.h. jeder Knoten  $v$  mit Ein- und Ausgängen



wird ersetzt durch



Damit kann der Knoten  $v$  im ungerichteten Graph durch einen Hamilton-Kreis nur in einer Richtung durchlaufen werden (von  $v_{in}$  durch  $v$  durch  $v_{out}$  oder umgekehrt). Da dies für alle Kanten und Knoten gemacht wird, gibt es nur dann einen Hamilton-Kreis im ungerichteten Graphen, wenn alle Knoten in der gleichen Reihenfolge durchlaufen werden. Daher gilt: Falls

$$v_{in,\pi(1)}, v_{\pi(1)}, v_{out,\pi(1)}, \dots, v_{in,\pi(n)}, v_{\pi(n)}, v_{out,\pi(n)}, v_{in,\pi(1)}$$

ein ungerichteter Hamilton-Kreis, dann ist  $v_{\pi(1)}, \dots, v_{\pi(n)}, v_{\pi(1)}$  der gerichtete Hamilton Kreis; und bei ungerichtetem Hamilton-Kreis

$$v_{out,\pi(1)}, v_{\pi(1)}, v_{in,\pi(1)}, \dots, v_{out,\pi(n)}, v_{\pi(n)}, v_{in,\pi(n)}, v_{out,\pi(1)}$$

gibt es den gerichteten Hamilton Kreis  $v_{\pi(1)}, \dots, v_{\pi(n)}, v_{\pi(1)}$ .

Auch umgekehrt ist klar, dass es für jeden gerichteten Hamilton-Kreis einen gerichteten gibt. Da  $f$  in Polynomialzeit berechenbar ist, haben zusammenfassend gezeigt, dass  $\text{DIRECTED-HAMILTON-CYCLE} \leq_p \text{UNDIRECTED-HAMILTON-CYCLE}$  gilt.  $\square$

### 8.5.12. Das TRAVELLING-SALESPERSON-Problem

**Definition 8.5.35** (TRAVELLING-SALESPERSON-Problem). Das TRAVELLING-SALESPERSON-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:

gegeben: Ein Menge von Knoten (Städten)  $V = \{v_1, \dots, v_n\}$ , eine  $(n \times n)$ -Matrix  $(M_{i,j})$  Entfernungen  $M_{i,j} \in \mathbb{N}_0$  zwischen den Städten  $v_i$  und  $v_j$ , sowie eine Zahl  $k \in \mathbb{N}_0$ .

gefragt: Gibt es eine Rundreise, die alle Städte besucht und der Startort gleich dem Zielort ist?

Formal: Gibt es eine Permutation  $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ , sodass  $(\sum_{i=1}^{n-1} M_{\pi(i), \pi(i+1)}) + M_{\pi(n), \pi(1)} \leq k$ ?

**Satz 8.5.36.** Das TRAVELLING-SALESPERSON-Problem ist  $\mathcal{NP}$ -vollständig.

*Beweis.* Wir zeigen die beiden benötigten Teile:

- TRAVELLING-SALESPERSON  $\in \mathcal{NP}$ : Rate die Permutation  $\pi$  nichtdeterministisch und verifiziere anschließend, dass  $(\sum_{i=1}^{n-1} M_{\pi(i), \pi(i+1)}) + M_{\pi(n)+\pi(1)} \leq k$  gilt. Dies kann auf einer NTM in Polynomialzeit durchgeführt werden.
- TRAVELLING-SALESPERSON ist  $\mathcal{NP}$ -schwer: Wir reduzieren UNDIRECTED-HAMILTON-CYCLE in Polynomialzeit auf TRAVELLING-SALESPERSON: Sei  $G = (V, E)$  mit  $V = \{1, \dots, n\}$ . Dann sei  $f(G) = (V, (M_{i,j}, n))$  mit

$$M_{i,j} = \begin{cases} 1, & \text{wenn } \{i, j\} \in E \\ 2, & \text{wenn } \{i, j\} \notin E \end{cases}$$

Wenn  $G$  einen ungerichteten Hamilton-Kreis hat, dann repräsentiert dieser Kreis eine Rundreise der Länge  $n$ . Wenn  $f(G)$  eine Rundreise der Länge  $\leq n$  hat, dann muss die Länge genau  $n$  sein. Daher können nur Kanten mit Entfernung 1 verwendet werden. Daher hat  $G$  einen ungerichteten Hamilton-Kreis.

Die Funktion  $f$  kann in Polynomialzeit von einer DTM berechnet werden. Damit folgt UNDIRECTED-HAMILTON-CYCLE  $\leq_p$  TRAVELLING-SALESPERSON.

□

### 8.5.13. Das GRAPH-COLORING-Problem

**Definition 8.5.37** (GRAPH-COLORING-Problem). *Das GRAPH-COLORING-Problem lässt sich in der gegeben/gefragt-Notation wie folgt formulieren:*

*gegeben:* Ein ungerichteter Graph  $G = (V, E)$  und eine Zahl  $k \in \mathbb{N}_0$ .

*gefragt:* Gibt es Färbung der Knoten in  $V$  mit  $\leq k$  Farben, sodass keine zwei benachbarten Knoten in  $G$ , die gleiche Farbe erhalten?

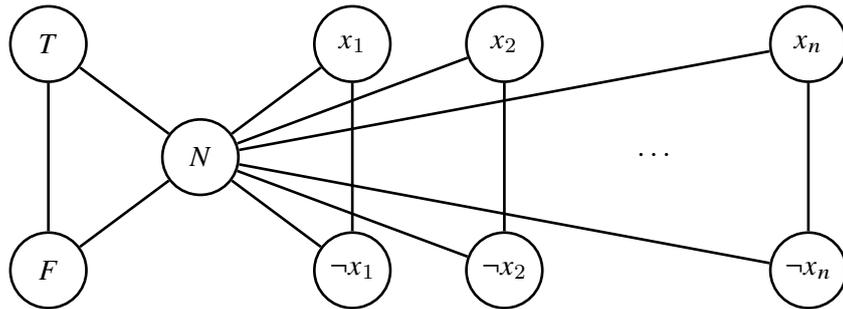
**Satz 8.5.38.** GRAPH-COLORING ist  $\mathcal{NP}$ -vollständig.

*Beweis.* Wir zeigen die beiden benötigten Teile:

- GRAPH-COLORING  $\in \mathcal{NP}$ : Rate nichtdeterministisch die Farbe aus  $\{1, \dots, k\}$  für jeden Knoten  $v \in V$ . Anschließend verifiziere, dass die Farben von  $u$  und  $v$  stets verschieden sind, für alle  $\{u, v\} \in E$ . Die Verifikation kann in Polynomialzeit durchgeführt werden. Daher kann GRAPH-COLORING auf einer NTM mit polynomieller Zeit entschieden werden.
- GRAPH-COLORING ist  $\mathcal{NP}$ -schwer. Wir zeigen 3-CNF-SAT  $\leq_p$  GRAPH-COLORING. Sei  $F = K_1 \wedge \dots \wedge K_m$  eine 3-CNF, sodass jede Klausel  $K_i$  genau 3 Literale enthält. Seien  $\{x_1, \dots, x_n\}$  die aussagenlogischen Variablen, die in  $F$  vorkommen.

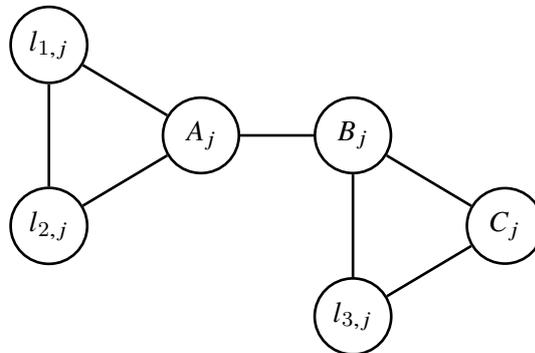
Wir erzeugen ein GRAPH-COLORING Problem mit  $k = 3$ , d.h. der Graph muss mit drei Farben färbbar sein.

Konstruiere zunächst den Teilgraphen

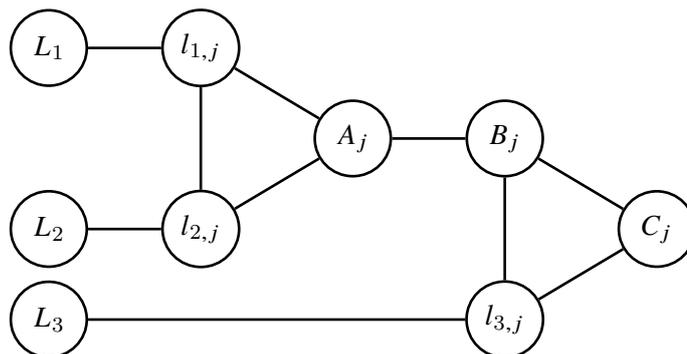


Wenn der Graph 3-färbbar ist, dann müssen  $T, N, F$  mit allen drei Farben gefärbt werden. O.B.d.A. seien dies genau die Farben  $T, N, F$  und die Knoten entsprechend gefärbt ( $T$  mit  $T$ ,  $N$  mit  $N$ ,  $F$  mit  $F$ ). dann muss für eine Färbung stets  $(x_i, \neg x_i)$  mit  $(T, F)$  oder  $(F, T)$  gefärbt werden. D.h. dieser Teilgraph sorgt dafür, dass die Färbung genau eine Belegung der aussagenlogischen Variablen erzeugt.

Für jede Klausel  $K_j$  wird der folgende Teilgraph  $G_j$  erzeugt, bestehend aus den Knoten  $l_{1,j}, l_{2,j}, l_{3,j}, A_j, B_j, C_j$ , der das logische „oder“ der drei Literale „berechnen“ soll. Sei  $K_j$ :



Verbindet man die „Eingänge“  $l_{1,j}, l_{2,j}, l_{3,j}$  mit den zugehörigen Literalen  $L_1, L_2, L_3$  (d.h. den Knoten  $x_i$ , bzw.  $\neg x_i$ ):



dann gilt:

Wenn  $L_1, L_2, L_3$  mit Farben  $T$  und  $F$  gefärbt sind, dann:

- $C_j$  kann nicht mit  $T$  gefärbt werden, wenn  $L_1, L_2, L_3$  alle mit  $F$  gefärbt sind.
- Wenn mindestens ein  $L_i$  (mit  $i = 1, 2, 3$ ) mit  $T$  gefärbt ist, dann kann der Graph so gefärbt werden, dass  $C_j$  mit  $T$  gefärbt wird.

Beide Aussagen lassen sich durch probieren aller Möglichkeiten verifizieren.

Schließlich erzeuge alle Teilgraphen  $G_j$  für  $K_1, \dots, K_m$  und verbinde  $l_{i,j}$  mit  $x_k$  wenn  $L_{i,j} = x_k$  und mit  $\neg x_k$  wenn  $L_{i,j} = \neg x_k$ . Schließlich verbinde jeweils  $C_j$  mit  $N$  und  $C_j$  mit  $F$ .

Damit gilt: Der entstandene Graph ist 3-färbbar, wenn alle  $C_j$  mit  $T$  gefärbt werden, was der Fall ist, wenn jedes  $G_j$  mit einem mit  $T$  gefärbten Literal am Eingang verbunden ist. Da dies genau den Literalen der Klausel  $K_j$  entspricht, erfüllt die Belegung  $I(x_i) = 1$ , wenn  $x_i$  mit  $T$  gefärbt wird, und  $I(x_i) = 0$  sonst, die 3-CNF.

Umgekehrt lässt sich aus einer erfüllenden Belegung eine 3-Färbung für den Graphen erzeugen.

Da  $f$  in Polynomialzeit berechnet werden kann, haben wir  $3\text{-CNF-SAT} \leq_p \text{GRAPH-COLORING}$  gezeigt.  $\square$

## Literatur

- COCKE, John: *Programming Languages and Their Compilers: Preliminary Notes*. New York, NY, USA : New York University, 1969
- COOK, Stephen A.: The Complexity of Theorem-proving Procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. New York, NY, USA : ACM, 1971 (STOC '71), S. 151–158
- EHRENFEUCHT, Andrzej ; KARHUMÄKI, Juhani ; ROZENBERG, Grzegorz: The (Generalized) Post Correspondence Problem with Lists Consisting of two Words is Decidable. In: *Theor. Comput. Sci.* 21 (1982), S. 119–144
- HOPCROFT, John E. ; MOTWANI, Rajeev ; ULLMAN, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006
- KARP, Richard M.: Reducibility Among Combinatorial Problems. In: MILLER, Raymond E. (Hrsg.) ; THATCHER, James W. (Hrsg.): *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, Plenum Press, New York, 1972 (The IBM Research Symposia Series), 85–103
- KASAMI, Tadao: An efficient recognition and syntax analysis algorithm for context-free languages / Air Force Cambridge Research Laboratory. 1965 (AFCRL-65-758). – Forschungsbericht
- LADNER, Richard E.: On the Structure of Polynomial Time Reducibility. In: *J. ACM* 22 (1975), Nr. 1, S. 155–171
- NEARY, Turlough: Undecidability in Binary Tag Systems and the Post Correspondence Problem for Five Pairs of Words. In: MAYR, Ernst W. (Hrsg.) ; OLLINGER, Nicolas (Hrsg.): *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)* Bd. 30. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015 (Leibniz International Proceedings in Informatics (LIPIcs)). – ISSN 1868–8969, 649–661
- POST, Emil L.: A variant of a recursively unsolvable problem. In: *Bull. Amer. Math. Soc.* 52 (1946), 04, Nr. 4, S. 264–268
- RICE, Henry G.: Classes of Recursively Enumerable Sets and Their Decision Problems. In: *Trans. Amer. Math. Soc.* 74 (1953), S. 358–366
- RABIN, Michael O. ; SCOTT, Dana S.: Finite Automata and Their Decision Problems. In: *IBM J. Res. Dev.* 3 (1959), April, Nr. 2, S. 114–125. – ISSN 0018–8646
- SCHÖNING, Uwe: Graph Isomorphism is in the Low Hierarchy. In: *J. Comput. Syst. Sci.* 37 (1988), Nr. 3, S. 312–323

- SCHÖNING, Uwe: *Theoretische Informatik – kurz gefasst*. 5. Auflage. Spektrum Akademischer Verlag, 2008
- WEGENER, Ingo: *Theoretische Informatik - eine algorithmenorientierte Einführung (2. Auflage)*. Teubner, 1999
- YOUNGER, Daniel H.: Recognition and Parsing of Context-Free Languages in Time  $n^3$ . In: *Information and Control* 10 (1967), Nr. 2, S. 189–208

## A. Zusatzmaterial

Dieser Anhang wiederholt einerseits einige mathematische Grundlagen, die bekannt sein sollten und enthält andererseits weiterführende Themen oder spezifische Beweise, die nicht Teil der Veranstaltung sind, aber für den oder die interessierte:n Lesende:n eventuell hilfreich sind.

### A.1. Mathematische Grundlagen

#### A.1.1. Vollständige Induktion

Um Aussagen für alle natürlichen Zahlen nachzuweisen, verwenden wir oft das Prinzip der vollständigen Induktion:

**Definition A.1.1** (Beweisprinzip der Vollständigen Induktion). *Um zu zeigen, dass eine Aussage  $A(n)$  für jede natürliche Zahl  $n \in \mathbb{N}_0$  gilt, genügt es, die folgenden beiden Aussagen zu zeigen:*

*Induktionsanfang:  $A(0)$  gilt.*

*Induktionsschritt: Für jede beliebige Zahl  $n \in \mathbb{N}_0$  gilt: Wenn  $A(n)$  gilt, dann auch  $A(n+1)$ .*

Wir demonstrieren die vollständige Induktion:

**Beispiel A.1.2.** Für alle  $n \in \mathbb{N}_0$  gilt  $\sum_{i=1}^n i = \frac{(n+1)n}{2}$ . Wir zeigen die Aussage durch Induktion über  $n$ , d.h. die Aussage  $A(n)$  ist  $\sum_{i=1}^n i = \frac{(n+1)n}{2}$ .

- *Induktionsanfang: Die Aussage  $A(0)$  ist  $\sum_{i=1}^0 i = 0$ , welche aus Definition der Summe folgt.*
- *Induktionsschritt: Sei  $n \in \mathbb{N}_0$  eine beliebige Zahl. Wir dürfen  $A(n)$  annehmen und müssen  $A(n+1)$  zeigen. Angenommen  $A(n)$  gilt, d.h.  $\sum_{i=1}^n i = \frac{(n+1)n}{2}$ . Wir müssen  $\sum_{i=1}^{n+1} i = \frac{(n+2)(n+1)}{2}$  zeigen. Es gilt  $\sum_{i=1}^{n+1} i = \left( \sum_{i=1}^n i \right) + n + 1$  nach Definition. Nach Annahme ist letzteres gleich  $\frac{(n+1)n}{2} + n + 1$ . Durch Ausrechnen erhält man  $\frac{(n+1)n}{2} + n + 1 = \frac{(n+1)n + 2n + 2}{2} = \frac{(n+2)(n+1)}{2}$ , also insgesamt die zu zeigende Eigenschaft.*

### A.1.2. Relationen

Seien  $R, R_1, R_2$  binäre Relationen auf  $\Sigma^*$  (d. h.  $R, R_i \subseteq (\Sigma^* \times \Sigma^*)$ ). Für Wörter  $u$  und  $v$  schreiben wir anstelle von  $(u, v) \in R$  auch  $uRv$ .

Wir definieren die *Komposition der Relationen*  $R_1$  und  $R_2$  als:

$$R_1R_2 = \{(u, w) \mid \text{es gibt } v \in \Sigma^* \text{ mit } (uR_1v) \text{ und } (vR_2w)\}$$

Außerdem definieren wir

$$\begin{aligned} R^0 &:= \{(w, w) \mid w \in \Sigma^*\} \text{ (die identische Abbildung)} \\ R^i &:= RR^{i-1} \text{ für } i > 0 \\ R^* &:= \bigcup_{n \in \mathbb{N}_0} R^n \\ R^+ &:= \bigcup_{n \in \mathbb{N}} R^n \end{aligned}$$

Es gilt  $uR^*w$  g. d. w.  $u = w$  oder es gibt  $v_1, \dots, v_n \in \Sigma^*$  und  $n > 0$  mit  $uRv_1, v_1Rv_2, \dots, v_nRw$ .

**Definition A.1.3.** Eine binäre Relation  $R \subseteq (\Sigma^* \times \Sigma^*)$  heißt

- reflexiv, falls für alle  $w \in \Sigma^*$  gilt:  $wRw$
- transitiv, falls für alle  $u, v, w \in \Sigma^*$  gilt: Wenn  $uRv$  und  $vRw$ , dann auch  $uRw$ .
- symmetrisch falls für alle  $u, v \in \Sigma^*$  gilt: Wenn  $uRv$ , dann auch  $vRu$ .

Wenn  $R$  reflexiv, transitiv und symmetrisch ist, dann ist  $R$  eine Äquivalenzrelation.

Sei  $R$  eine Äquivalenzrelation. Für ein Wort  $w \in \Sigma^*$  bezeichnen wir die Äquivalenzklasse von  $w$  mit  $[w]_R$ , welche alle zu  $w$  äquivalenten Wörter enthält:  $[w]_R := \{u \in \Sigma^* \mid uRw\}$ . Umgekehrt heißt  $w$  Repräsentant der Äquivalenzklasse  $[w]_R$ . Beachte, dass die Äquivalenzklassen die Grundmenge  $\Sigma^*$  in (endlich oder unendlich viele) Äquivalenzklassen disjunkt zerlegt, d. h.  $\Sigma^* = [w_1]_R \cup [w_2]_R \cup [w_3]_R \cup \dots$ , wobei  $\cup$  die disjunkte Vereinigung bezeichne<sup>1</sup>. Der Index einer Äquivalenzrelation  $R$  (geschrieben als  $\text{Index}(R)$ ) ist die Anzahl der verschiedenen Äquivalenzklassen, die  $R$  hat. Hierbei ist  $\text{Index}(R) \in \mathbb{N}_0 \cup \{\infty\}$ . Wenn  $\text{Index}(R) \neq \infty$ , dann sagt man  $R$  hat einen endlichen Index.

**Beispiel A.1.4.** Sei  $\Sigma = \{a, b\}$  und  $\text{echterPraefix}$  eine binäre Relation auf  $\Sigma^*$ , definiert durch: Für alle  $u, v \in \Sigma^*$ :  $u \text{ echterPraefix } v$  g. d. w.  $u$  ist ein Präfix von  $v$  und  $|u| < |v|$ .

Die Relation  $\text{echterPraefix}$  ist transitiv: Seien  $u, v, w \in \Sigma^*$  mit  $u \text{ echterPraefix } v$  und  $v \text{ echterPraefix } w$ . Dann gilt  $u$  ist Präfix von  $v$  und  $v$  ist Präfix von  $w$  und  $|u| < |v|$  und  $|v| < |w|$ . Daraus folgt auch, dass  $u$  ein Präfix von  $w$  ist, und dass  $|u| < |w|$  gilt. Somit haben wir  $u \text{ echterPraefix } w$  gezeigt.

Die Relation  $\text{echterPraefix}$  ist nicht reflexiv, da z. B.  $\neg(\varepsilon \text{ echterPraefix } \varepsilon)$ .

Die Relation  $\text{echterPraefix}$  ist nicht symmetrisch, da z. B.  $aa \text{ echterPraefix } aab$ , aber  $\neg(aab \text{ echterPraefix } aa)$ .

<sup>1</sup>d. h.  $A \cup B$  g. d. w.  $A \cup B = A \cup B$  und  $A \cap B = \emptyset$ .

Die Relation  $\text{echterPraefix}^*$  ist reflexiv, da  $w \text{echterPraefix}^0 w$  für alle  $w \in \Sigma^*$ . Die Relation  $\text{echterPraefix}^*$  ist transitiv: Sei  $u \text{echterPraefix}^i v$  und  $v \text{echterPraefix}^j w$ , dann gilt  $u \text{echterPraefix}^{i+j} w$  und daher  $u \text{echterPraefix}^* w$ . Die Relation  $\text{echterPraefix}^*$  ist nicht symmetrisch, da  $\neg(a \text{echterPraefix}^i ab)$  für alle  $i \in \mathbb{N}_0$ . Es gilt  $u \text{echterPraefix}^* v$  genau dann, wenn  $u$  ein Präfix von  $v$  ist.

**Beispiel A.1.5.** Sei  $\Sigma = \{a, b\}$  und  $u$   $\text{gleicherAnfang}$   $v$  g. d.  $w$ .  $u$  und  $v$  beginnen mit dem gleichen Buchstaben oder sind beide das leere Wort.

$\text{gleicherAnfang}$  ist eine Äquivalenzrelation:

- $\text{gleicherAnfang}$  ist reflexiv: Wir prüfen  $u \text{gleicherAnfang} u$  für alle  $u \in \Sigma^*$ : Wenn  $u$  mit  $a$  anfängt (d. h.  $u = av$  für ein  $v \in \Sigma^*$ ) gilt  $u \text{gleicherAnfang} u$ . Wenn  $u$  mit  $b$  anfängt (d. h.  $u = bv$  für ein  $v \in \Sigma^*$ ) gilt  $u \text{gleicherAnfang} u$ . Auch für  $u = \varepsilon$  gilt  $u \text{gleicherAnfang} u$ .
- $\text{gleicherAnfang}$  ist symmetrisch: Offensichtlich folgt aus  $u \text{gleicherAnfang} v$  auch  $v \text{gleicherAnfang} u$  für alle  $u, v \in \Sigma^*$ :
- $\text{gleicherAnfang}$  ist transitiv: Sei  $u \text{gleicherAnfang} v$  und  $v \text{gleicherAnfang} w$  für  $u, v, w \in \Sigma^*$ . Dann fangen entweder  $u, v, w$  alle mit dem selben Buchstaben an (und  $u \text{gleicherAnfang} w$  gilt) oder  $u = v = w = \varepsilon$  und auch dann gilt  $u \text{gleicherAnfang} w$ .

Der Index von  $\text{gleicherAnfang}$  ist  $\text{Index}(\text{gleicherAnfang}) = 3$ , denn es gibt drei disjunkte Äquivalenzklassen:

- $[a]_{\text{gleicherAnfang}} = \{w \in \Sigma^* \mid a \text{gleicherAnfang} w\} = \{aw \mid a \in \Sigma^*\}$
- $[b]_{\text{gleicherAnfang}} = \{w \in \Sigma^* \mid b \text{gleicherAnfang} w\} = \{bw \mid b \in \Sigma^*\}$
- $[\varepsilon]_{\text{gleicherAnfang}} = \{w \in \Sigma^* \mid \varepsilon \text{gleicherAnfang} w\} = \{\varepsilon\}$

**Lemma A.1.6.**  $R^*$  ist die kleinste reflexive und transitive Relation, die  $R$  enthält (d. h.  $R^*$  ist die reflexiv-transitive Hülle von  $R$ ).

*Beweis.* Wir zeigen zunächst, dass  $R^*$  reflexiv und transitiv ist:  $R^*$  ist offensichtlich reflexiv, da  $R^0 \subseteq R$ .  $R^*$  ist transitiv, denn für  $uR^*v$  und  $vR^*w$  folgt: Es gibt  $i, j \in \mathbb{N}_0$  mit  $uR^i v$  und  $vR^j w$ . Daher gilt  $uR^{i+j} w$ , was direkt  $uR^*v$  impliziert, da  $R^{i+j} \subseteq R^*$ .

Wir zeigen nun, dass jede reflexiv-transitive Relation  $R'$ , die  $R$  enthält, auch  $R^*$  enthält. Genauer zeigen wir mit Induktion über  $i$ , dass  $R^i \subseteq R'$  für alle  $i \in \mathbb{N}_0$ . Die Induktionsbasis ist  $R^0 \subseteq R'$ , was gilt, da  $R'$  reflexiv ist. Ebenso gilt auch  $R^1 \subseteq R'$ , da  $R'$  die Relation  $R$  enthält. Als Induktionsannahme nehmen wir an, dass  $R^i \subseteq R'$  für  $i \in \mathbb{N}_0$  gilt. Sei  $uR^{i+1}v$ . Dann gibt es ein  $w$  mit  $uRw$  und  $wR^i v$ . Aus der Induktionsannahme und dem Fakt  $R \subseteq R'$  folgt auch  $uR'w$  und  $wR'v$ . Da  $R'$  transitiv ist, muss auch gelten  $uR'v$ . Da dies für jedes Paar  $uR^{i+1}v$  gilt, folgt  $R^{i+1} \subseteq R'$ .  $\square$

**Definition A.1.7.** Seien  $R$  und  $S$  Äquivalenzrelationen auf  $\Sigma^*$ . Dann ist  $R$  eine Verfeinerung von  $S$ , wenn für alle  $u, v \in \Sigma^*$  gilt:  $uRv \implies uSv$  (d. h.  $R \subseteq S$ ).

**Satz A.1.8.** Sei  $R$  eine Verfeinerung von  $S$ . Dann gilt  $\text{Index}(R) \geq \text{Index}(S)$ .

*Beweis.* Sei  $\Sigma^* = [u_1]_R \cup [u_2]_R \cup \dots$  die disjunkte Zerlegung von  $R$  in ihre Äquivalenzklassen. Seien  $v, w \in u_i$ . Dann gilt  $vRw$  und (da  $R$  eine Verfeinerung von  $S$  ist) auch  $vSw$ . D.h.  $\Sigma^* = [u_1]_S \cup [u_2]_S \cup \dots$  und damit gilt insbesondere, dass  $S$  nicht mehr disjunkte Äquivalenzklassen hat als  $R$ .  $\square$

**Beispiel A.1.9.** Sei  $\Sigma = \{a, b\}$ . Die Relationen `gleicherAnfang` und `gleicherAnfangUndEnde` seien definiert durch

- `gleicherAnfang`  $v$   $g$ . d. w.  $u$  und  $v$  beginnen mit dem gleichen Buchstaben oder sind beide das leere Wort.
- `gleicherAnfangUndEnde`  $v$   $g$ . d. w.  $u$ , `gleicherAnfang`  $v$ , und  $u$  und  $v$  enden mit dem gleichen Buchstaben oder sind beide das leere Wort.

Beide Relationen sind Äquivalenzrelationen (für `gleicherAnfang` haben wir dies in Beispiel A.1.5 gezeigt, der Nachweis für `gleicherAnfangUndEnde` funktioniert analog). Die Relation `gleicherAnfangUndEnde` ist eine Verfeinerung von `gleicherAnfang`, denn aus `gleicherAnfangUndEnde`  $v$  folgt stets `gleicherAnfang`  $v$ . In Beispiel A.1.5 haben wir gezeigt, dass  $\text{Index}(\text{gleicherAnfang}) = 3$  gilt. Für `gleicherAnfangUndEnde` gilt  $\text{Index}(\text{gleicherAnfangUndEnde}) = 5$ , denn `gleicherAnfangUndEnde` hat folgende disjunkten Äquivalenzklassen:

- $[\varepsilon]_{\text{gleicherAnfangUndEnde}} = \{\varepsilon\}$
- $[a]_{\text{gleicherAnfangUndEnde}} = \{u \in \Sigma^* \mid u \text{ beginnt mit } a \text{ und endet mit } a\}$
- $[b]_{\text{gleicherAnfangUndEnde}} = \{u \in \Sigma^* \mid u \text{ beginnt mit } b \text{ und endet mit } b\}$
- $[ab]_{\text{gleicherAnfangUndEnde}} = \{u \in \Sigma^* \mid u \text{ beginnt mit } a \text{ und endet mit } b\}$
- $[ba]_{\text{gleicherAnfangUndEnde}} = \{u \in \Sigma^* \mid u \text{ beginnt mit } b \text{ und endet mit } a\}$

### A.1.3. Funktionen

Seien  $D$  und  $Z$  Mengen. Eine Funktion  $f : D \rightarrow Z$  ist eine links-totale und rechtseindeutige binäre Relation, d. h.  $f \subseteq (D \times Z)$  und sie ordnet jedem Element aus  $D$  genau ein Element aus  $Z$  zu (kurzum: Für jedes  $d \in D$  gibt es genau einen Eintrag in  $(d, f(d)) \in f$ ).

Für ein Element  $z \in Z$ , sei  $f^{-1}(z) := \{d \in D \mid f(d) = z\}$  die Menge der Urbilder von  $z$  und  $f$ . Eine Funktion  $f$  heißt

- *injektiv*, falls für alle  $z \in Z$  gilt:  $|f^{-1}(z)| \leq 1$  (jedes  $z$  hat höchstens ein Urbild)
- *surjektiv*, falls für alle  $z \in Z$  gilt:  $|f^{-1}(z)| \geq 1$  (jedes  $z$  hat mindestens ein Urbild)
- *bijektiv*, falls für alle  $z \in Z$  gilt:  $|f^{-1}(z)| = 1$  (jedes  $z$  hat genau ein Urbild)

**Beispiel A.1.10.** Sei  $D$  eine Menge von Mänteln und  $Z$  eine Menge von Kleiderhaken. Sei  $f$  eine Zuordnung der Mäntel auf die Kleiderhaken, d. h. eine Funktion  $f : D \rightarrow Z$ . Wenn  $f$  injektiv ist, hängt auf jedem Kleiderhaken höchstens ein Mantel. Wenn  $f$  surjektiv ist, hängt auf jedem Kleiderhaken mindestens ein Mantel. Wenn  $f$  bijektiv ist, hängt auf jedem Kleiderhaken genau ein Mantel.

### A.1.4. Abzählbarkeit und Überabzählbarkeit

Seien  $M_1, M_2$  Mengen. Wir nennen  $M_1$  und  $M_2$  *gleichmächtig*, wenn es eine bijektive Abbildung  $f : M_1 \rightarrow M_2$  gibt. Falls eine Menge  $M$  gleichmächtig wie  $\mathbb{N}_0$  ist, dann heißt  $M$  *abzählbar unendlich*. Eine Menge  $M$  heißt *abzählbar*, wenn sie endlich oder abzählbar unendlich ist.

**Lemma A.1.11.** *Eine Menge  $M$  ist genau dann abzählbar, wenn  $M = \emptyset$  gilt oder es eine surjektive Abbildung  $f : \mathbb{N}_0 \rightarrow M$  gibt.*

*Beweis.* Wenn  $M = \emptyset$ , dann gilt die Aussage. Sei  $M \neq \emptyset$  für den Rest des Beweises.

Sei  $M$  abzählbar. Wenn  $M$  abzählbar unendlich ist, dann gibt es eine Bijektion  $f : \mathbb{N}_0 \rightarrow M$  und die Aussage gilt. Wenn  $M$  endlich ist, dann sei  $M = \{a_0, \dots, a_n\}$ . Definiere  $f : \mathbb{N}_0 \rightarrow M$  als  $f(x) = a_x$  für  $0 \leq x \leq n$  und  $f(x) = a_0$  sonst. Dann ist  $|f^{-1}(x)| \geq 1$  für alle  $x \in \mathbb{N}_0$  und  $f$  damit surjektiv.

Für die umgekehrte Richtung sei  $f : \mathbb{N}_0 \rightarrow M$  eine surjektive Abbildung. Wenn  $M$  endlich ist, dann ist  $M$  abzählbar. Wenn  $M$  nicht endlich ist, dann müssen wir zeigen, dass  $M$  gleichmächtig zu  $\mathbb{N}_0$  ist. Dazu sei  $g : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  induktiv definiert durch  $g(0) = 0$  und  $g(n+1) = k$ , wobei  $k \in \mathbb{N}_0$  minimal gewählt ist, sodass  $f(k) = f(g(n+1)) \notin \{f(g(0)), \dots, f(g(n))\}$ . Eine solche Zahl  $k$  gibt es, da  $M$  sonst endlich wäre. Die Konstruktion von  $g$  sichert zu, dass für alle  $i \in \mathbb{N}_0$  gilt:  $g(i+1) > g(i)$ .

Wir zeigen, dass  $h : \mathbb{N}_0 \rightarrow M$  mit  $h(x) = f(g(x))$  bijektiv ist. Die Konstruktion von  $g$  sichert zu, dass  $h$  injektiv ist. Für den Beweis der Surjektivität von  $h$  sei  $m \in M$ . Dann ist  $|f^{-1}(m)| > 0$ , da  $f$  surjektiv ist. Sei  $n \in \mathbb{N}_0$  daher minimal gewählt, sodass  $f(n) = m$ . Da  $g(i+1) > g(i)$  für alle  $i \in \mathbb{N}_0$ , gibt es nur endlich viele Zahlen  $i \in \{0, 1, \dots, j\}$  mit  $g(j) < n$ . Daher gilt  $g(j+1) = k \geq n$ , wobei  $k$  minimal gewählt ist, so dass  $f(k) \notin \{f(g(0)), \dots, f(g(j))\}$ . Da  $f(n) = m$  kann  $k$  nicht größer als  $n$  sein. D.h.  $g(j+1) = n$  und daher  $|g^{-1}(n)| > 0$ , was auch zeigt  $h^{-1}(m) > 0$ . □

Das letzte Lemma zeigt, dass für  $M \neq \emptyset$  und  $M$  abzählbar die Elemente von  $M$  stets durchnummeriert werden können (wobei manche Elemente auch mehrere Nummern erhalten können).

**Lemma A.1.12.** *Seien  $M_1, M_2$  abzählbar, dann ist auch  $M_1 \times M_2$  abzählbar.*

*Beweis.* Wir zeigen zunächst, dass  $(\mathbb{N}_0 \times \mathbb{N}_0)$  abzählbar ist. Die folgende Tabelle deutet Cantor's Diagonalverfahren an und zeigt, wie die Paare abgezählt werden:

	0	1	2	3	...
0	0	1	3	6	...
1	2	4	7	...	...
2	5	8	...	...	...
3	9	...	...	...	...
...	...	...	...	...	...

Insbesondere folgt daraus, dass es eine surjektive Funktion  $h : \mathbb{N}_0 \rightarrow (\mathbb{N}_0 \times \mathbb{N}_0)$  gibt<sup>2</sup>. Nun zeigen wir das Lemma: Da  $M_1$  und  $M_2$  abzählbar sind, gibt es surjektive Funktionen  $f_i : \mathbb{N}_0 \rightarrow M_i$  für  $i = 1, 2$ . Sei  $g : (\mathbb{N}_0 \times \mathbb{N}_0) \rightarrow (M_1 \times M_2)$  definiert durch  $g(i, j) = (f_1(i), f_2(j))$ . Schließlich definiere  $g' : \mathbb{N}_0 \rightarrow (M_1 \times M_2)$  als  $g'(i) = g(h(i))$ . Da  $g, h$  surjektiv sind, ist auch  $g'$  surjektiv.  $\square$

**Satz A.1.13.** *Zu einem Alphabet  $\Sigma$  ist  $\Sigma^*$  immer eine abzählbar unendliche Menge.*

*Beweis.* Nummeriere die Wörter  $w \in \Sigma^*$  wie folgt

- Nummeriere die Wörter der Länge 0
- Nummeriere die Wörter der Länge 1
- Nummeriere die Wörter der Länge 2
- Nummeriere die Wörter der Länge 3
- ...

Innerhalb einer Länge, nummeriere die Wörter anhand ihrer lexikographischen Ordnung. Beachte: Diese Nummerierung funktioniert, da es pro fester Länge  $k$  nur endliche viele Wörter gibt ( $s^k$  viele, wenn  $|\Sigma| = s$ )  $\square$

**Definition A.1.14** (Potenzmenge  $\mathcal{P}(M)$ ). *Sei  $M$  eine Menge. Mit  $\mathcal{P}(M)$  bezeichnen wir die Potenzmenge von  $M$ , d. h. die Menge aller Teilmengen von  $M$ :  $\mathcal{P}(M) := \{N \mid N \subseteq M\}$ .*

*Mit  $\mathcal{P}_e(M)$  bezeichnen wir die Menge aller endlichen Teilmengen von  $M$ .*

**Beispiel A.1.15.** *Sei  $\Sigma$  ein Alphabet. Dann ist die Menge aller Sprachen über  $\Sigma$  gerade  $\mathcal{P}(\Sigma^*)$ .*

**Satz A.1.16.** *Sei  $M$  abzählbar unendlich. Dann ist  $\mathcal{P}(M)$  nicht abzählbar (sondern überabzählbar).*

*Beweis.* Wir führen einen Beweis durch Widerspruch. Wir nehmen an, dass  $M$  abzählbar unendlich und  $\mathcal{P}(M)$  abzählbar ist. Da  $\mathcal{P}(M)$  nicht endlich sein kann, ist  $\mathcal{P}(M)$  ebenfalls abzählbar unendlich. Dann gibt es Bijektionen  $f : \mathbb{N}_0 \rightarrow M$  und  $g : \mathbb{N}_0 \rightarrow \mathcal{P}(M)$ . Definiere die Diagonalmenge  $D := \{f(i) \mid f(i) \notin g(i), i \in \mathbb{N}_0\}$ . Da  $D$  Teilmenge von  $M$  ist (d. h.  $D \subseteq M$ ), ist  $D$  ein Element der Potenzmenge, d. h.  $D \in \mathcal{P}(M)$  und es gibt  $n \in \mathbb{N}_0$  mit  $g(n) = D$ . Betrachte nun die Frage, ob  $f(n) \in D$  liegt: Einsetzen der Mengendefinition für  $D$  ergibt „ $f(n) \in D$  g. d. w.  $f(n) \notin g(n)$ “ und Einsetzen von  $g(n) = D$  ergibt „ $f(n) \in D$  g. d. w.  $f(n) \notin D$ “ was einen Widerspruch darstellt. D.h. unsere Annahme, dass  $\mathcal{P}(M)$  abzählbar ist, war falsch.  $\square$

<sup>2</sup>Eine andere surjektive Funktion ist  $h : \mathbb{N}_0 \rightarrow (\mathbb{N}_0 \times \mathbb{N}_0)$  mit  $h(0) = (0, 0)$  und  $h(n) = (i, j)$  wenn  $n = 2^i \cdot (2 \cdot j + 1)$  und  $n > 0$ , da sich jede positive natürliche Zahl eindeutig als  $2^i \cdot (2 \cdot j + 1)$  (mit  $i, j \in \mathbb{N}_0$ ) zerlegen lässt.

### A.1.5. Asymptotische Notation

Funktionen  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ , welche die Komplexität von Algorithmen beschreiben, werden meist mithilfe der asymptotischen Notation angegeben. Wir betrachten hier nur die  $O$ -Notation (gesprochen „Groß-Oh-Notation“). Diese vernachlässigt konstante Faktoren und gibt eine asymptotische Abschätzung nach oben an. Genauer definiert  $O(g)$  eine Klasse von Funktionen:

**Definition A.1.17.** Für eine Funktion  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  gilt  $f \in O(g)$  g. d. w. es gibt Konstanten  $c, n_0 > 0$ , sodass für alle  $n \geq n_0$  gilt:  $f(n) \leq c \cdot g(n)$

Oft schreibt man anstelle von  $f \in O(g)$  auch  $f(n) = O(g(n))$ . Hierbei ist das Gleichheitssymbol von links nach rechts zu lesen.

**Beispiel A.1.18.** Sei  $f(n) = 3 \cdot n^2 + 6n + 2$ . Dann gilt  $f(n) = O(n^2)$ , denn für  $c = 11$  und  $n_0 = 1$  gilt  $3 \cdot n^2 + 6n + 2 \leq c \cdot n^2$  für alle  $n \geq n_0$  (verteile die  $11n^2$  auf die einzelnen Summanden:  $3n^2 \leq 3n^2, 6n \leq 6n^2, 2 \leq 2n^2$  für alle  $n \geq 1$ ).

## A.2. Entfernen von $\varepsilon$ -Produktionen in kontextfreien Grammatiken

Sei  $G = (V, \Sigma, P, S)$  eine kontextfreie (bzw. reguläre) Grammatik mit  $\varepsilon \notin L(G)$ . Dann konstruiert Algorithmus 6 eine kontextfreie (bzw. reguläre) Grammatik  $G'$  mit  $L(G) = L(G')$  und  $G'$  enthält keine  $\varepsilon$ -Produktionen.

Die letzte Wiederhole-Schleife ist der interessante Teil des Algorithmus. Hier wird die Regelanwendung  $A \rightarrow \varepsilon$  sozusagen vorweggenommen und direkt in die Grammatik eingebaut. Zusammen mit den Schritten zuvor verändert dies die erzeugte Sprache nicht. Schließlich ist noch zu beobachten, dass im Fall einer regulären Grammatik, die hinzugefügten Produktionen immer von der Form  $A' \rightarrow a$  sein müssen und daher dem Format der regulären Grammatiken entsprechen. D. h.  $G'$  ist in diesem Fall ebenfalls regulär.

**Beispiel A.2.1** (Entfernen von  $\varepsilon$ -Produktionen). Wir entfernen die  $\varepsilon$ -Produktionen der Grammatik  $G = (\{A, B, C, D, S\}, \{0, 1\}, P, S)$  mit

$$P = \{S \rightarrow 1A, A \rightarrow AB, A \rightarrow DA, A \rightarrow \varepsilon, B \rightarrow 0, B \rightarrow 1, C \rightarrow AAA, D \rightarrow 1AC\}.$$

Die Menge der Variablen, die  $\varepsilon$  herleiten, ist  $W = \{A, C\}$  (zunächst wird  $A$  eingefügt, da es die Produktion  $A \rightarrow \varepsilon$  gibt, danach wird  $C$  eingefügt, da  $C \rightarrow AAA$ ). Löschen der Produktion  $A \rightarrow \varepsilon$  ergibt  $P' = \{S \rightarrow 1A, A \rightarrow AB, A \rightarrow DA, B \rightarrow 0, B \rightarrow 1, C \rightarrow AAA, D \rightarrow 1AC\}$ . Hinzufügen der Regeln (Löschen von Vorkommen von  $A, C$ ) ergibt

$$P' = \{S \rightarrow 1A, S \rightarrow 1, A \rightarrow AB, A \rightarrow B, A \rightarrow DA, A \rightarrow D, B \rightarrow 0, B \rightarrow 1, C \rightarrow AAA, C \rightarrow AA, C \rightarrow A, D \rightarrow 1AC, D \rightarrow 1A, D \rightarrow 1C, D \rightarrow 1\}.$$

Daher ist  $G' = (\{A, B, C, D, S\}, \{0, 1\}, P', S)$ .

---

**Algorithmus 6** : Entfernen von  $\varepsilon$ -Produktionen

---

**Eingabe** : Typ  $i$ -Grammatik  $G = (V, \Sigma, P, S)$  mit  $\varepsilon \notin L(G)$ ,  $i \in \{2, 3\}$

**Ausgabe** : Typ  $i$ -Grammatik  $G'$  ohne  $\varepsilon$ -Produktionen, sodass  $L(G) = L(G')$

**Beginn**

finde die Menge  $W \subseteq V$  aller Variablen  $A$  für die gilt  $A \Rightarrow^* \varepsilon$ :

**Beginn**

$W := \{A \mid (A \rightarrow \varepsilon) \in P\};$

**wiederhole**

füge alle solchen  $A$  zu  $W$  hinzu, für die es eine Produktion  $A \rightarrow A_1 \dots A_n$  gibt, sodass für alle  $i = 1, \dots, n$ :  $A_i \in W$ ;

**bis sich  $W$  nicht mehr ändert;**

**Ende**

$P' := P \setminus \{A \rightarrow \varepsilon \mid (A \rightarrow \varepsilon) \in P\};$

*/\* lösche Regeln  $A \rightarrow \varepsilon$  \*/*

**wiederhole**

**für alle** Produktionen der Form  $A' \rightarrow uAv$  in  $P'$  mit  $|uv| > 0$  und  $A \in W$  **tue**

füge die Produktion  $A' \rightarrow uv$  zu  $P'$  hinzu;

*/\* für eine Produktion  $A' \rightarrow u'Av'Aw'$  gibt es (mindestens) zwei Hinzufügungen: Sowohl für das Vorkommen von  $A$  nach  $u'$  als auch für das Vorkommen direkt vor  $w'$  \*/*

**Ende**

**bis sich  $P'$  nicht mehr ändert;**

Gebe  $G' = (V, \Sigma, P', S)$  als Ergebnisgrammatik aus;

**Ende**

---

### A.3. Der Satz von Myhill und Nerode

Wir definieren die sogenannte Nerode-Relation (benannt nach Anil Nerode):

**Definition A.3.1** (Nerode-Relation  $\sim_L$ ). Sei  $L$  eine formale Sprache über  $\Sigma$ . Die Nerode-Relation  $\sim_L \subseteq \Sigma^* \times \Sigma^*$  zu  $L$  ist definiert für alle Worte  $u, v \in \Sigma^*$  durch:

$$u \sim_L v \iff \forall w \in \Sigma^* : uw \in L \iff vw \in L$$

Informell sind  $u$  und  $v$  äquivalent bez.  $\sim_L$ , wenn sich ihr Enthaltensein in  $L$  gleich verhält bezüglich beliebiger Erweiterung um denselben Suffix.

**Satz A.3.2.** Die Nerode-Relation ist eine Äquivalenzrelation.

*Beweis.* Reflexivität von  $\sim_L$  gilt genau wie Symmetrie offensichtlich. Transitivität gilt auch: Wir nehmen an, dass  $u_1 \sim_L u_2$  und  $u_2 \sim_L u_3$  gilt. Sei  $w \in \Sigma^*$ . Wenn  $u_1w \in L$ , dann folgt aus  $u_1 \sim_L u_2$  auch  $u_2w \in L$ , woraus mit  $u_2 \sim_L u_3$  wiederum  $u_3w \in L$  folgt. Analog kann gezeigt werden, dass  $u_3w \in L$  auch  $u_1w \in L$  impliziert. Daher gilt  $u_1w \in L \iff u_3w \in L$  und damit  $u_1 \sim_L u_3$ .  $\square$

Wir erinnern, dass eine Äquivalenzrelation die Grundmenge  $\Sigma^*$  in disjunkte Äquivalenzklassen  $[u_1]_{\sim_L}, [u_2]_{\sim_L}, \dots$  zerlegt und dass der Index einer Äquivalenzrelation die Anzahl der disjunkten Äquivalenzklassen ist (u. U.  $\infty$  bei unendlich vielen Äquivalenzklassen). John Myhill und Anil Nerode bewiesen in den 1950er Jahren den folgenden Satz:

**Theorem A.3.3** (Satz von Myhill und Nerode). Eine formale Sprache  $L$  ist genau dann regulär, wenn der Index von  $\sim_L$  endlich ist.

*Beweis.* Wir haben zwei Richtungen zu zeigen:

1. Wenn  $L$  regulär ist, dann ist der Index von  $\sim_L$  endlich.
2. Wenn der Index von  $\sim_L$  endlich ist, dann ist  $L$  regulär.

Wir beweisen beide Teile unabhängig voneinander.

1. Sei  $L$  regulär und sei  $M = (Z, \Sigma, \delta, z_0, E)$  ein DFA der  $L$  akzeptiert (d. h.  $L(M) = L$ ). O.b.d.A. seien alle Zustände von  $M$  vom Startzustand aus erreichbar. Sei  $\approx_M \subseteq (\Sigma^* \times \Sigma^*)$  definiert durch  $u \approx_M v \iff \widehat{\delta}(z_0, u) = \widehat{\delta}(z_0, v)$ , d. h. Wörter  $u, v$  sind äquivalent bez.  $\approx_M$ , wenn sie ausgeführt auf  $M$  zum gleichen Zustand führen. Es lässt sich leicht prüfen, dass  $\approx_M$  eine Äquivalenzrelation ist. Der Index von  $\approx_M$  ist offensichtlich  $|Z|$  und daher endlich. Wir zeigen, dass  $u \approx_M v \implies u \sim_L v$  gilt und damit, dass  $\approx_M$  die Relation  $\sim_L$  verfeinert, d. h. *mehr oder gleich viele* disjunkte Äquivalenzklassen als  $\sim_L$  hat. Dann folgt sofort, dass der Index von  $\sim_L$  höchstens  $|Z|$  und daher endlich ist.

Sei  $u \approx_M v$  und  $w \in \Sigma^*$ . Dann gilt  $\widehat{\delta}(z_0, uw) = \widehat{\delta}(\widehat{\delta}(z_0, u), w) = \widehat{\delta}(\widehat{\delta}(z_0, v), w) = \widehat{\delta}(z_0, vw)$  und damit  $uw \in L \iff vw \in L$ . Da  $w$  beliebig gewählt war, zeigt dies  $u \sim_L v$ .

2. Sei der Index von  $\sim_L$  endlich. Dann gibt es  $n \in \mathbb{N}$  und Repräsentanten  $u_1, \dots, u_n$ , sodass  $\Sigma^* = [u_1]_{\sim_L} \cup \dots \cup [u_n]_{\sim_L}$ . Wir definieren den sogenannten *Nerode-Automaten*:

Sei  $M = (Z, \Sigma, \delta, [\varepsilon]_{\sim_L}, E)$  ein DFA mit  $Z = \{[u_1]_{\sim_L}, \dots, [u_n]_{\sim_L}\}$ ,  $\delta([u_i]_{\sim_L}, a) = [u_i a]_{\sim_L}$  für alle  $a \in \Sigma$  und  $E = \{[u_i]_{\sim_L} \mid i \in \{1, \dots, n\}, u_i \in L\}$ . Wir zeigen  $w \in L(M) \iff w \in L$ :  $w \in L(M)$  ist äquivalent zu  $\widehat{\delta}([\varepsilon]_{\sim_L}, w) \in E$ , was (anhand der Definition von  $\delta$ ) äquivalent zu  $[w]_{\sim_L} \subseteq L$  ist, und wiederum äquivalent zu  $w \in L$  ist. D. h. der DFA  $M$  akzeptiert die Sprache  $L$ .  $\square$

Der Satz von Myhill und Nerode gibt eine genaue Charakterisierung der regulären Sprachen. Er wird meist dazu verwendet, die Nichtregularität einer formalen Sprache  $L$  nachzuweisen, indem man zeigt, dass der Index der Nerode-Relation  $\sim_L$  unendlich ist. Dafür reicht es aus, unendlich viele disjunkte Äquivalenzklassen zu finden. Oft findet man diese, indem man versucht für alle  $i \in \mathbb{N}_0$ , Wörter  $u_i$  und  $w_i$  zu finden, sodass alle  $u_i$  paarweise verschieden sind, und alle  $w_i$  paarweise verschieden sind und  $u_i w_i \in L$ , aber  $u_j w_i \notin L$  für  $i \neq j$ .

**Beispiel A.3.4.** Betrachte die Sprache  $L = \{a^n b^n \mid n \in \mathbb{N}\}$  und deren Nerode-Relation  $\sim_L$ . Die Äquivalenzklassen  $[a^i b]_{\sim_L}$  für  $i \in \mathbb{N}$  enthalten jeweils alle Wörter, denen noch  $i - 1$   $b$ 's fehlen, um in  $L$  enthalten zu sein. Z. B. ist  $[ab]_{\sim_L} = L$ ,  $[a^2 b]_{\sim_L} = \{a^2 b, a^3 b^2, a^4 b^3, \dots\}$ ,  $[a^3 b]_{\sim_L} = \{a^3 b, a^4 b^2, a^5 b^3, a^6 b^4, \dots\}$ . All diese Äquivalenzklassen sind paarweise disjunkt bez.  $\sim_L$ , denn für  $w_i = b^{i-1}$  gilt  $a^i b w_i \in L$  aber  $a^j b w_i \notin L$ . Daher ist der Index von  $\sim_L$  nicht endlich und der Satz von Myhill und Nerode (Theorem A.3.3) zeigt, dass  $L$  nicht regulär sein (was wir bereits wussten, durch Beweis mit dem Pumping-Lemma).

Bisher konnten wir nicht zeigen, dass die Sprache  $L = \{a^k b^l c^l \mid k, l \in \mathbb{N}_0\} \cup \{b, c\}^*$  nicht regulär ist, da das Pumping-Lemma nicht anwendbar ist (siehe Lemma 3.8.9). Mithilfe des Satzes von Myhill und Nerode holen wir das jetzt nach:

**Satz A.3.5.** Die Sprache  $L = \{a^k b^l c^l \mid k, l \in \mathbb{N}_0\} \cup \{b, c\}^*$  ist nicht regulär.

*Beweis.* Betrachte die Äquivalenzklassen  $[ab^i c]_{\sim_L}$  für  $i \in \mathbb{N}$ . Diese sind alle paarweise verschieden, da für  $i, j \in \mathbb{N}$  mit  $w_i = c^{i-1}$  gilt:  $ab^i c w_i \in L$ , aber  $ab^j c w_i \notin L$  für  $i \neq j$ . Der Index von  $\sim_L$  ist daher  $\infty$  und Theorem A.3.3 zeigt daher, dass  $L$  nicht regulär ist.  $\square$

**Beispiel A.3.6.** Wir betrachten erneut die Sprache  $L = \{uaav \mid uv \in \{a, b\}^*\}$  über  $\Sigma = \{a, b\}$  (alle Wörter, die zwei aufeinanderfolgende  $a$ 's enthalten). Dann hat  $\sim_L$  drei disjunkte Äquivalenzklassen:

$$\begin{aligned} [\varepsilon]_{\sim_L} &= \{w \in \Sigma^* \mid w \text{ enthält keine zwei aufeinander folgende } a \text{ und endet nicht mit } a\} \\ [a]_{\sim_L} &= \{w \Sigma^* \mid w \text{ enthält keine zwei aufeinander } a \text{ und endet mit } a\} \\ [aa]_{\sim_L} &= \{w \in \Sigma^* \mid w \text{ enthält zwei aufeinanderfolgende } a\} = L \end{aligned}$$

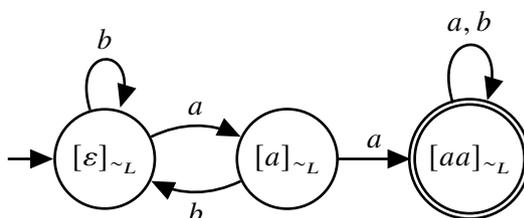
Zunächst kann man verifizieren, dass alle Fälle erfasst sind, d. h.  $[\varepsilon]_{\sim_L} \cup [a]_{\sim_L} \cup [aa]_{\sim_L} = \Sigma^*$ . Dafür muss man sich davon überzeugen, dass Worte in den Mengen äquivalent sind und die

Beschreibungen der Mengen stimmen. ( $[\varepsilon]_{\sim_L}$  sind alle Worte  $u$  denen noch ein Wort  $w$  aus  $L$  angehängt werden muss, damit  $uw \in L$  gilt,  $[a]_{\sim_L}$  sind alle Worte  $u$  denen noch ein Wort  $w$  aus  $L$  oder  $w = aw'$  mit  $w' \in \Sigma^*$  angehängt werden muss, damit  $uw \in L$  gilt,  $[aa]_{\sim_L}$  sind alle Worte  $u$ , für die für alle  $w \in \Sigma^*$  gilt  $uw \in L$ ). Die drei Äquivalenzklassen sind paarweise disjunkt: Für  $\varepsilon$  und  $a$  gilt mit  $w = a$ :  $\varepsilon w \notin L$ , aber  $aw \in L$ , d. h.  $\varepsilon \not\sim_L a$ . Für  $a$  und  $aa$  gilt mit  $w = \varepsilon$ :  $aw \notin L$  aber  $aaw \in L$ , d. h.  $a \not\sim_L aa$  und schließlich gilt für  $\varepsilon$  und  $aa$  und  $w = \varepsilon$ :  $\varepsilon w \notin L$  aber  $aaw \in L$ , d. h.  $\varepsilon \not\sim_L aa$ .

Wir erzeugen den DFA (den sogenannten Nerode-Automat), wie er im Beweis von Theorem A.3.3 konstruiert wird. Sei  $M = (Z, \Sigma, \delta, z_0, E)$  mit  $Z = \{[\varepsilon]_{\sim_L}, [a]_{\sim_L}, [aa]_{\sim_L}\}$ ,  $z_0 = [\varepsilon]_{\sim_L}$ ,  $E = \{[aa]_{\sim_L}\}$  und

$$\begin{aligned} \delta([\varepsilon]_{\sim_L}, a) &= [a]_{\sim_L} & \delta([a]_{\sim_L}, a) &= [aa]_{\sim_L} & \delta([aa]_{\sim_L}, a) &= [aa]_{\sim_L} \\ \delta([\varepsilon]_{\sim_L}, b) &= [\varepsilon]_{\sim_L} & \delta([a]_{\sim_L}, b) &= [\varepsilon]_{\sim_L} & \delta([aa]_{\sim_L}, b) &= [aa]_{\sim_L} \end{aligned}$$

Der Zustandsgraph von  $M$  ist

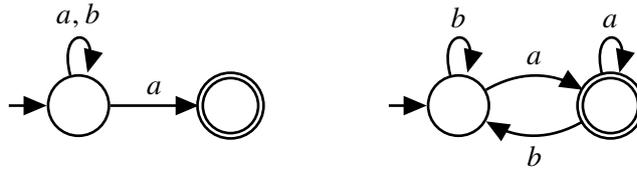


**Satz A.3.7.** Der Nerode-Automat einer regulären Sprache  $L$  ist der sogenannte Minimalautomat, d. h. jeder DFA  $M'$ , mit  $L(M') = L$  hat mindestens so viele Zustände wie der Nerode-Automat. Zudem sind alle minimalen DFAs (bis auf Umbenennung von Zuständen) identisch.

*Beweis.* Sei  $M = (Z, \Sigma, \delta, z_0, E)$  der Nerode-Automat der Sprache  $L$  und  $M' = (Z', \Sigma, \delta', z'_0, E')$  ein beliebiger DFA mit  $L(M') = L$ . Der Beweis von Theorem A.3.3 zeigt, dass  $\approx_{M'}$  eine Verfeinerung von  $\sim_L$  ist ( $\approx_{M'} \subseteq \sim_L$ ) und dass  $\approx_M = \sim_L$  gilt. Daher ist  $Index(\approx_{M'}) \geq Index(\approx_M)$  und daher auch  $|Z'| = Index(\approx_{M'}) \geq |Z| = Index(\approx_M)$ .

Sei nun  $|Z'| = |Z|$ , d. h.  $M'$  habe eine minimale Zustandsanzahl. Dann muss aufgrund von  $\approx_{M'} \subseteq \sim_L$  gelten,  $\approx_{M'} = \sim_L$ . Da auch  $\sim_L = \approx_M$  gilt, folgt, dass  $f : Z \rightarrow Z'$  mit  $f([u]_{\approx_M}) = f([u]_{\sim_L}) := [u]_{\approx_{M'}}$  ein Isomorphismus auf den Zuständen ist. Offensichtlich gilt auch  $f(\delta([u]_{\sim_L}, a)) = f([ua]_{\sim_L}) = [ua]_{\approx_{M'}} = \delta'([u]_{\approx_{M'}}, a) = \delta'(f([u]_{\sim_L}), a)$ . D. h. die Automaten  $M$  und  $M'$  verhalten sich (bis auf die Umbenennung  $f$ ) identisch.  $\square$

Die letzte Aussage zeigt, dass der Nerode-Automat eindeutig und minimal ist. Beachte, dass dies für NFAs im Allgemeinen nicht richtig ist. Es gibt strukturell unterschiedliche NFAs mit der selben minimalen Zustandsanzahl. Ein Beispiel (angelehnt an ein Beispiel aus (Sch08)) sind die beiden im folgenden gezeigten NFAs, die beide die Sprache  $\{ua \mid u \in \Sigma^*\}$  über dem Alphabet  $\Sigma = \{a, b\}$  erkennen und die minimale Anzahl an Zuständen besitzen, aber strukturell verschieden sind:



### A.4. Operationen auf CFGs

In diesem Abschnitt beweisen wir einige Hilfssätze, die Operationen auf CFGs durchführen, ohne deren erzeugte Sprache zu ändern.

**Lemma A.4.1** (Inlining von Produktionen). *Sei  $G = (V, \Sigma, P, S)$  eine CFG, sei  $A \rightarrow uBv \in P$ , seien  $B \rightarrow w_1 \mid \dots \mid w_n$  alle Regeln mit  $B$  als linker Seite und sei*

$$G' = (V, \Sigma, P \setminus \{A \rightarrow uBv\} \cup \{A \rightarrow uw_1v \mid uw_2v \mid \dots \mid uw_nv\}, S).$$

*Dann erzeugen  $G$  und  $G'$  dieselbe Sprache, d. h.  $L(G) = L(G')$ .*

*Beweis.* Sei  $S \Rightarrow_G^* w \in \Sigma^*$  eine Ableitung von  $w$  mit  $G$ . Der Syntaxbaum dieser Ableitung kann leicht modifiziert werden, sodass er einer Ableitung mit  $G'$  entspricht, indem die Ableitungsschritte, die  $A \rightarrow uBv$  und anschließendes  $B \rightarrow w_i$  im Syntaxbaum anwenden, durch einen Schritt  $A \rightarrow uw_iv$  ersetzt werden. Umgekehrt kann jede Ableitung  $S \Rightarrow_{G'}^* w$  in eine Ableitung  $S \Rightarrow_G^* w$  umkonstruiert werden, indem Schritte  $u_0Av_0 \Rightarrow_{G'} u_0uw_ivv_0$  durch die beiden Schritte  $u_0Av_0 \Rightarrow_G u_0Bv_0 \Rightarrow_G u_0uw_ivv_0$  ersetzt werden.  $\square$

**Lemma A.4.2** (Sharing von Satzformen mit neuen Produktionen). *Sei  $G$  eine CFG mit  $G = (V, \Sigma, P \cup \{A \rightarrow w_1 \dots w_n\}, S)$ . Seien  $B_1, \dots, B_n$  neue Variablen (d. h.  $V \cap \{B_1, \dots, B_n\} = \emptyset$ ) und sei  $G' = (V \cup \{B_1, \dots, B_n\}, \Sigma, P \cup \{A \rightarrow B_1 \dots B_n, B_1 \rightarrow w_1, \dots, B_n \rightarrow w_n\}, S)$ .*

*Dann gilt  $L(G) = L(G')$ .*

*Beweis.* Jeder Ableitungsschritt  $uAv \Rightarrow_G uw_1 \dots w_nv$  kann „übersetzt“ werden in  $uAv \Rightarrow_{G'} uB_1 \dots B_nv \Rightarrow_{G'}^n uw_1 \dots w_nv$ . Umgekehrt können in jeder Ableitung  $S \Rightarrow_{G'}^* w$ , die Anwendungen der Regel  $A \rightarrow B_1 \dots B_n, B_i \rightarrow w_i$  identifiziert werden (durch den Syntaxbaum) und die Ableitung entsprechend umgeordnet werden, sodass diese Schritte stets sequentiell aufeinander folgen. Für jede so normalisierte Ableitung, kann eine Ableitung mit  $G$  konstruiert werden.  $\square$

Eine Produktion nennt man *links-rekursiv*, wenn sie von der Form  $A \rightarrow Au$  ist, und *rechts-rekursiv*, wenn sie von der Form  $A \rightarrow uA$  ist, wobei in beiden Fällen  $u$  eine Satzform ist. Das folgende Lemma zeigt, wie man links-rekursive Produktionen in rechts-rekursive Produktionen umformen kann:

**Lemma A.4.3** (Elimination von Links-Rekursion). Sei  $G = (V, \Sigma, P, S)$  eine CFG mit  $P = P' \cup \{A \rightarrow Au_1 \mid \dots \mid Au_n \mid w_1 \mid \dots \mid w_m\}$ , wobei  $A \rightarrow Au_1 \mid \dots \mid Au_n \mid w_1 \mid \dots \mid w_m$  alle Produktionen in  $P$  mit Variable  $A$  als linker Seite sind, und die Satzformen  $w_1, \dots, w_m$  alle nicht mit  $A$  beginnen. Sei  $B$  eine neue Variable. Für die CFG  $G' = (V \cup \{B\}, \Sigma, P'', S)$  mit

$$P'' = P' \cup \{A \rightarrow w_1 B \mid \dots \mid w_m B \mid w_1 \mid \dots \mid w_m, B \rightarrow u_1 \mid \dots \mid u_n \mid u_1 B \mid \dots \mid u_n B\}$$

gilt  $L(G) = L(G')$ .

*Beweis.*  $L(G) \subseteq L(G')$ : Betrachte zunächst nur Linksableitungen von  $A$  ausgehend, bis zum ersten Wort, dass kein  $A$  am Anfang hat. Dann gilt  $A \Rightarrow_G^* w_r u_{f(1)} \dots u_{f(k)}$  für eine Funktion  $f : \{1, \dots, k\} \rightarrow \{1, \dots, n\}$  (mit  $k \geq 0$ ) und ein  $r \in \{1, \dots, m\}$ . Für  $G'$  können wir ebenfalls herleiten  $A \Rightarrow_{G'}^* w_r u_{f(1)} \dots u_{f(k)}$ , wobei kein  $B$  mehr in  $w_r u_{f(1)} \dots u_{f(k)}$  vorkommt. Sei nun  $w \in L(G)$  und  $S \Rightarrow_G^* w$  eine Linksableitung von  $w$  mit Grammatik  $G$ . Ersetze alle Ableitungssequenzen von  $x_1 A x_2 \Rightarrow_G^* x_1 w_r u_{f(1)} \dots u_{f(k)} x_2$  durch  $x_1 A x_2 \Rightarrow_{G'}^* x_1 w_r u_{f(1)} \dots u_{f(k)} x_2$ . Anschließend sind alle  $\Rightarrow_G$ -Ableitungsschritte auch  $\Rightarrow_{G'}$ -Ableitungsschritte und daher gibt es eine Ableitung  $S \Rightarrow_{G'}^* w$ .

$L(G') \subseteq L(G)$ : Betrachte zunächst nur Rechtsableitungen von  $A$  ausgehend, bis zum ersten Wort, dass kein  $B$  mehr hat. Dann gilt  $A \Rightarrow_{G'}^* w_r u_{f(1)} \dots u_{f(k)}$  für eine Funktion  $f : \{1, \dots, k\} \rightarrow \{1, \dots, n\}$  (mit  $k \geq 0$ ) und ein  $r \in \{1, \dots, m\}$ . Dann gibt es (eine Linksableitung)  $A \Rightarrow_G^* w_r u_{f(1)} \dots u_{f(k)}$ . Sei nun  $w \in L(G')$  und  $S \Rightarrow_{G'}^* w$  eine Rechtsableitung von  $w$  mit Grammatik  $G'$ . Ersetze alle Ableitungssequenzen von  $x_1 A x_2 \Rightarrow_{G'}^* x_1 w_r u_{f(1)} \dots u_{f(k)} x_2$  durch  $x_1 A x_2 \Rightarrow_G^* x_1 w_r u_{f(1)} \dots u_{f(k)} x_2$ . Anschließend sind alle  $\Rightarrow_{G'}$ -Ableitungsschritte auch  $\Rightarrow_G$ -Ableitungsschritte und daher gibt es eine Ableitung  $S \Rightarrow_G^* w$ .  $\square$

## A.5. Herstellen der Chomsky-Normalform

Wir nehmen im folgenden an, dass  $\varepsilon \notin L(G)$  und werden zeigen, dass jede solche kontextfreie Grammatik in Chomsky-Normalform gebracht werden kann. Diese Umformung geschieht in mehreren Schritten. Den ersten Schritt haben wir bereits in Algorithmus 6 gemacht, als wir  $\varepsilon$ -Produktionen entfernten.

### A.5.1. Entfernen von Einheitsproduktionen

Im nächsten Schritt möchten wir Produktionen der Form  $A \rightarrow B$  entfernen, wobei  $A$  und  $B$  Variablen sind. Diese sogenannten Einheitsproduktionen „verlängern“ nur die Ableitungen und es ist intuitiv klar, dass wir auf sie verzichten können, indem wir anstatt  $A \Rightarrow B \Rightarrow w$  abzuleiten direkt eine Produktion  $A \rightarrow w$  zur Grammatik hinzufügen. Allerdings muss man dieses Ersetzen in der richtigen Reihenfolge tun (wenn  $w$  selbst eine Variable  $C$  ist, ist das Hinzufügen von  $A \rightarrow C$  erst einmal nicht zielführend) und Sonderfälle (z. B.  $A \rightarrow B$  und  $B \rightarrow A$ ) anders behandeln. Der Algorithmus 7 leistet dies.

**Algorithmus 7** : Entfernen von Einheitsproduktionen in einer CFG

**Eingabe** : Eine CFG  $G = (V, \Sigma, P, S)$

**Ausgabe** : Eine CFG  $G'$  ohne Einheitsproduktionen mit  $L(G) = L(G')$

**Beginn**

Erzeuge gerichteten Graph  $D = (V, E)$ , sodass die Knoten den Variablen  $V$  entsprechen und es je eine Kante  $(A, B) \in E$  für jede Einheitsproduktion  $A \rightarrow B \in P$  gibt;

**solange** es einen Zyklus  $(A_1, A_2), \dots, (A_{n-1}, A_n), (A_n, A_1) \in E$  gibt **tue**

$P := P \setminus \{A_1 \rightarrow A_2, \dots, A_{n-1} \rightarrow A_n, A_n \rightarrow A_1\}$ ; /\* entferne die zyklischen Regeln \*/

$P := P[A_1/A_2, \dots, A_1/A_n]$ ; /\* ersetze alle Vorkommen von  $A_i$  durch  $A_1$  für  $i = 2, \dots, n$  \*/

$V := V \setminus \{A_2, A_3, \dots, A_n\}$ ; /\* lösche  $A_2, \dots, A_n$  \*/

$S := S[A_1/A_2, \dots, A_1/A_n]$ ; /\* ersetze Startsymbol durch  $A_1$ , falls es  $A_i, 2 \leq i \leq n$  war \*/

$E := E \setminus \{(A_1, A_2), \dots, (A_{n-1}, A_n), (A_n, A_1)\}$ ; /\* entferne Zyklus aus Graph \*/

$E := E[A_1/A_2, \dots, A_1/A_n]$ ; /\* ersetze alle  $A_i$  durch  $A_1$  für  $i = 2, \dots, n$  in den Kanten \*/

**Ende**

Sortiere  $D$  topologisch und nummeriere die Variablen in  $V$  durch (und benenne entsprechend in  $E, P, S$  um), so dass gilt:  $A_i \rightarrow A_j$  impliziert  $i < j$ ;

Sei  $V = \{A_1, \dots, A_k\}$ ;

**für**  $i=k$  bis 1 **tue**

**wenn**  $A_i \rightarrow A_j \in P$  **dann**

seien  $A_j \rightarrow w_1, \dots, A_j \rightarrow w_m$  alle Produktionen mit  $A_j$  als linker Seite;

$P := P \cup \{A_i \rightarrow w_1, \dots, A_i \rightarrow w_m\}$ ;

$P := P \setminus \{A_i \rightarrow A_j\}$ ;

**Ende**

**Ende**

Gib die so entstandene Grammatik als  $G'$  aus;

**Ende**

Beachte, dass die Zyklen durch eine Tiefensuche auf  $D$  gefunden werden können.

**Satz A.5.1.** Algorithmus 7 berechnet bei Eingabe einer CFG  $G$  mit  $\varepsilon \notin L(G)$  eine CFG  $G'$ , die keine Einheitsproduktionen hat, sodass gilt  $L(G) = L(G')$ . Wenn  $G$  keine  $\varepsilon$ -Produktionen hat, dann hat auch  $G'$  keine  $\varepsilon$ -Produktionen.

*Beweis.* Wir zeigen:

1. Das Entfernen eines Zyklus verändert die erzeugte Sprache nicht.
  2. Das Entfernen einer Einheitsproduktion  $A_i \rightarrow A_j$  in der rückwärts-laufenden für-Schleife ändert die erzeugte Sprache nicht.
1. Sei  $G = (V, \Sigma, P, S)$  die Grammatik mit Zyklus  $A_1 \rightarrow A_2, \dots, A_{n-1} \rightarrow A_n, A_n \rightarrow A_1$  und  $G' = (V', \Sigma, P', S')$  die Grammatik nachdem der Zyklus entfernt wurde. Sei  $\sigma$  die

Substitution  $\{A_i \mapsto A_1 \mid i \in \{2, \dots, n\}\}$ .

- „ $L(G) \subseteq L(G')$ “: Wir zeigen: Für alle Satzformen  $w_1, w_2 \in (\Sigma \cup V)^*$  gilt: Wenn  $w_1 \Rightarrow_G^* w_2$ , dann  $\sigma(w_1) \Rightarrow_{G'}^* \sigma(w_2)$ . Sei  $w_1 \Rightarrow_G^n w_2$ . Wenn  $n = 0$ , dann ist  $w_1 = w_2$  und es gilt  $\sigma(w_1) \Rightarrow_{G'}^* \sigma(w_1)$ . Wenn  $n > 0$ , dann  $w_1 \Rightarrow_G w'_1 \Rightarrow_{G'}^{n-1} w_2$ . Die Induktionsannahme liefert  $\sigma(w'_1) \Rightarrow_{G'}^* \sigma(w_2)$ . Wenn  $w_1 \Rightarrow_G w'_1$  mit einer Regel  $A_i \rightarrow A_j$  aus dem Zyklus, dann gilt  $\sigma(w_1) = \sigma(w'_1)$  (und damit auch  $\sigma(w_1) \Rightarrow_{G'}^* \sigma(w'_1)$ ) und mit der Induktionsannahme  $\sigma(w_1) \Rightarrow_{G'}^* \sigma(w_2)$ . Wenn  $w_1 \Rightarrow_G w'_1$  mit einer Regel  $B \rightarrow w$ , die nicht vom Zyklus stammt, dann gibt es die Regel  $\sigma(B) \rightarrow \sigma(w)$  in  $G'$  und es gilt  $w_1 = uBv$  und  $w'_1 = uwv$  und  $\sigma(w_1) = \sigma(u)\sigma(B)\sigma(v) \Rightarrow_{G'} \sigma(u)\sigma(w)\sigma(v) = \sigma(w'_1)$  und mit der Induktionsannahme folgt  $\sigma(w_1) \Rightarrow_{G'}^* \sigma(w_2)$ .

- „ $L(G') \subseteq L(G)$ “:

Sei  $w \in (\Sigma \cup V')^*$  eine Satzform mit  $w \Rightarrow_{G'}^n w_0$  und  $w_0 \in \Sigma^*$ . Mit Induktion über  $n$  zeigen wir  $w \Rightarrow_G^* w_0$ . Für  $n = 0$  gilt  $w = w_0$  und damit auch  $w \Rightarrow_G^* w_0$ . Falls  $n > 0$ , dann  $w \Rightarrow_{G'} w' \Rightarrow_{G'}^{n-1} w_0$ . Die Induktionsannahme liefert  $w' \Rightarrow_G^* w_0$ . Wenn  $w \Rightarrow_G w'$ , dann gilt die Behauptung. Anderenfalls muss gelten  $w = uA_1v$  und  $w' = u\sigma(w_j)v$  mit  $A_j \rightarrow w_j \in P$ . Dann gilt  $uA_1v \Rightarrow_G \dots \Rightarrow_G uA_jv \Rightarrow_G uw_jv \Rightarrow_G^* u\sigma(w_j)v \Rightarrow^* w_0$ .

Beachte, dass wir hierbei benutzen, dass gilt:  $uwv \Rightarrow_G^* u\sigma(w)v$  für alle  $u, v, w \in (\Sigma \cap V)^*$ , denn wir können durch Anwenden der Regeln jedes Vorkommen von  $A_i$  in  $w$  in  $A_1$  umbenennen.

2. Die Korrektheit des Entferns der Einheitsproduktionen folgt aus Lemma A.4.1.

Schließlich müssen wir noch beobachten, dass das Verfahren terminiert und keine neuen Einheitsproduktionen einführt. Die Solange-Schleife terminiert, da mit jeder Elimination von Zyklen ein Zyklus entfernt wird und keine neuen Zyklen eingeführt werden. Die Für-Schleife terminiert offensichtlich. Da die Produktionen entsprechend der topologischen Sortierung behandelt werden, werden keine Einheitsproduktionen eingeführt: Wenn  $A_i \rightarrow A_j$  entfernt wird, wurden bereits alle Einheitsproduktionen  $A_j \rightarrow A_k$  entfernt, d. h. alle rechten Seiten  $w_1, \dots, w_m$  der Produktionen für  $A_j$  können nicht nur aus einer Variablen bestehen.

Der zweite Teil des Satzes gilt, da sämtliche Schritte keine  $\varepsilon$ -Produktionen einführen, sofern die Eingabegrammatik keine  $\varepsilon$ -Produktionen enthält.  $\square$

### A.5.2. Ersetzen von Terminalen und Zerlegen von langen rechten Seiten

Nach Entfernen der  $\varepsilon$ -Produktionen und der Einheitsproduktionen, können wir die Chomsky-Normalform herstellen, indem wir zuerst alle Terminalsymbole  $a$ , die in rechten Seiten vorkommen (aber deren rechten Seiten nicht nur aus  $a$  bestehen) durch  $A_a$  ersetzen und Produktionen  $A_a \rightarrow a$  hinzufügen. Anschließend sind alle Regeln von der Form  $A \rightarrow a$  oder Regeln der Form  $A \rightarrow B_1 \dots B_m$  mit  $m > 1$ . Falls im zweiten Fall  $m > 2$  gilt, dann müssen wir die Regel noch

zerlegen in mehrere Regeln (durch Einführen von neuen Variablen). Algorithmus 8 formalisiert diese Schritte.

---

**Algorithmus 8 : Herstellung der Chomsky-Normalform**

---

**Eingabe :** CFG  $G$  mit  $\varepsilon \notin L(G)$

**Ausgabe :** CFG  $G'$  in Chomsky-Normalform mit  $L(G) = L(G')$

**Beginn**

Entferne die  $\varepsilon$ -Produktionen in  $G$  mit Algorithmus 6 und entferne anschließend die Einheitsproduktionen mit Algorithmus 7;

Sei  $G' = (V', \Sigma, P', S')$  die entstandene Grammatik;

**für alle**  $a \in \Sigma$  **tue**

    /\* Führe neue Variable  $A_a$  für  $a$  ein, und ersetze Vorkommen von  $a$  durch das Nichtterminal \*/

$G' := (V' \cup \{A_a\}, \Sigma, \{A \rightarrow w[A_a/a] \mid A \rightarrow w \in P' \text{ und } |w| > 1\}$   
            $\cup \{A \rightarrow w \mid A \rightarrow w \in P' \text{ und } |w| = 1\} \cup \{A_a \rightarrow a\}, S)$

**Ende**

/\* Nun sind alle Regeln von der Form  $A \rightarrow a$  oder  $A \rightarrow B_1 \cdots B_m$  mit  $m \geq 2$  \*/

**für alle**  $A \rightarrow B_1 \cdots B_m \in P'$  **mit**  $m > 2$  **tue**

    Seien  $C_1, \dots, C_{m-2}$  neue Variablen;

$V' := V' \cup \{C_1, \dots, C_{m-2}\}$ ;

    /\* Ersetze in  $P'$  die Produktion  $A \rightarrow B_1 \cdots B_m$  durch neue Regeln \*/

$P' := (P' \setminus \{A \rightarrow B_1 \cdots B_m\})$   
            $\cup \{A \rightarrow B_1 C_1\} \cup \{C_i \rightarrow B_{i+1} C_{i+1} \mid \text{für } i = 1, \dots, m-3\} \cup \{C_{m-2} \rightarrow B_{m-1} B_m\}$ ;

**Ende**

**Ende**

---

**Theorem A.5.2.** Für CFGs  $G$  mit  $\varepsilon \notin L(G)$  berechnet Algorithmus 8 eine äquivalente CFG in Chomsky-Normalform.

*Beweis.* Die Äquivalenz der erzeugten Sprache ergibt sich aus Satz 2.2.5 für das Entfernen der  $\varepsilon$ -Produktionen, aus Satz A.5.1 und aus Lemma A.4.2 für die weiteren Schritte des Algorithmus (Einführen von Produktionen für  $A \rightarrow a$  und  $A \rightarrow B_1 \cdots B_m$ ). Schließlich verifiziert man leicht, dass die Ausgabe-Grammatik in Chomsky-Normalform ist.  $\square$

**Beispiel A.5.3.** Wir verwenden Algorithmus 8 zum Berechnen der Chomsky-Normalform für die Grammatik  $G_0 = (\{A, B, C, D, S\}, \{0, 1\}, P_0, S)$  mit

$P_0 = \{S \rightarrow 1A, A \rightarrow AB, A \rightarrow DA, A \rightarrow \varepsilon, B \rightarrow 0, B \rightarrow 1, C \rightarrow AAA, D \rightarrow 1AC\}$ .

Entfernen der  $\varepsilon$ -Produktionen (siehe Beispiel A.2.1) liefert die Grammatik  $G_1 = (V, \Sigma, P_1, S)$  mit

$P_1 = \{S \rightarrow 1A, S \rightarrow 1, A \rightarrow AB, A \rightarrow B, A \rightarrow DA, A \rightarrow D, B \rightarrow 0, B \rightarrow 1,$   
 $C \rightarrow AAA, C \rightarrow AA, C \rightarrow A, D \rightarrow 1AC, D \rightarrow 1A, D \rightarrow 1C, D \rightarrow 1\}$ .

Wir wenden Algorithmus 7 an, um Einheitsproduktionen zu entfernen. Der gerichtete Graph ist  $D = (\{S, A, B, C, D\}, \{(A, A), (A, B), (A, D), (C, A)\})$  und hat keine Zyklen. Wir erhalten daher:

$$P_2 = \{S \rightarrow 1A, S \rightarrow 1, A \rightarrow AB, A \rightarrow B, A \rightarrow DA, A \rightarrow D, B \rightarrow 0, B \rightarrow 1, C \rightarrow AAA, C \rightarrow AA, C \rightarrow A, D \rightarrow 1AC, D \rightarrow 1A, D \rightarrow 1C, D \rightarrow 1\}.$$

Topologisches Sortieren und Umbenennen der Variablen, sodass „ $A_i \rightarrow A_j$  impliziert  $i < j$ “ gilt, erfordert eine Umbenennung, welche die Beziehungen „ $A < B$ “, „ $A < D$ “, „ $C < A$ “ erzeugt. Wir wählen die Umbenennung  $\rho$  mit  $\rho(C) = A_1, \rho(A) = A_2, \rho(B) = A_3, \rho(D) = A_4, \rho(S) = A_5$ . Das liefert uns  $G_3 = (\{A_1, A_2, A_3, A_4, A_5\}, \Sigma, P_3, A_5)$  mit

$$P_3 = \{A_5 \rightarrow 1A_2, A_5 \rightarrow 1, A_2 \rightarrow A_2A_3, A_2 \rightarrow A_3, A_2 \rightarrow A_4A_2, A_2 \rightarrow A_4, A_3 \rightarrow 0, A_3 \rightarrow 1, A_1 \rightarrow A_2A_2A_2, A_1 \rightarrow A_2A_2, A_1 \rightarrow A_2, A_4 \rightarrow 1A_2A_1, A_4 \rightarrow 1A_2, A_4 \rightarrow 1A_1, A_4 \rightarrow 1\}.$$

Nun läuft die Für-Schleife für  $i$  von 5 bis 1. Für  $i = 5, i = 4, i = 3$  gibt es jeweils keine Produktion der Form  $A_i \rightarrow A_j$ . Für  $i = 2$ , wird  $A_2 \rightarrow A_3$  ersetzt durch  $A_2 \rightarrow 0, A_2 \rightarrow 1$ , und  $A_2 \rightarrow A_4$  wird ersetzt durch  $A_2 \rightarrow 1A_2A_1, A_2 \rightarrow 1A_2, A_2 \rightarrow 1A_1$  und  $A_2 \rightarrow 1$ . Danach ist

$$P_4 = \{A_5 \rightarrow 1A_2, A_5 \rightarrow 1, A_2 \rightarrow A_2A_3, A_2 \rightarrow 0, A_2 \rightarrow 1, A_2 \rightarrow A_4A_2, A_2 \rightarrow 1A_2A_1, A_2 \rightarrow 1A_2, A_2 \rightarrow 1A_1, A_3 \rightarrow 0, A_3 \rightarrow 1, A_1 \rightarrow A_2A_2A_2, A_1 \rightarrow A_2A_2, A_1 \rightarrow A_2, A_4 \rightarrow 1A_2A_1, A_4 \rightarrow 1A_2, A_4 \rightarrow 1A_1, A_4 \rightarrow 1\}.$$

Für  $i = 1$  wird  $A_1 \rightarrow A_2$  ersetzt durch  $A_1 \rightarrow A_2A_3, A_1 \rightarrow 0, A_1 \rightarrow 1, A_1 \rightarrow A_4A_2, A_1 \rightarrow 1A_2A_1, A_1 \rightarrow 1A_2$  und  $A_1 \rightarrow 1A_1$ . Daher ist die Grammatik nach Entfernen der Einheitsproduktionen:  $G_5 = (V_5, \Sigma, P_5, A_5)$  mit  $V_5 = \{A_1, A_2, A_3, A_4, A_5\}$  und

$$P_5 = \{A_5 \rightarrow 1A_2, A_5 \rightarrow 1, A_2 \rightarrow A_2A_3, A_2 \rightarrow 0, A_2 \rightarrow 1, A_2 \rightarrow A_4A_2, A_2 \rightarrow 1A_2A_1, A_2 \rightarrow 1A_2, A_2 \rightarrow 1A_1, A_3 \rightarrow 0, A_3 \rightarrow 1, A_1 \rightarrow A_2A_2A_2, A_1 \rightarrow A_2A_2, A_1 \rightarrow A_2A_3, A_1 \rightarrow 0, A_1 \rightarrow 1, A_1 \rightarrow A_4A_2, A_1 \rightarrow 1A_2A_1, A_1 \rightarrow 1A_2, A_1 \rightarrow 1A_1, A_4 \rightarrow 1A_2A_1, A_4 \rightarrow 1A_2, A_4 \rightarrow 1A_1, A_4 \rightarrow 1\}.$$

Nun wird Algorithmus 8 fortgesetzt und zunächst werden alle Terminalsymbole durch neue Produktionen dargestellt: Dies seien  $B_0 \rightarrow 0$  und  $B_1 \rightarrow 1$ . Ersetzen aller Vorkommen von 0 durch  $B_0$  und 1 durch  $B_1$  in rechten Seiten mit Wortlänge  $> 1$  ergibt daher:

$$P_6 = \{B_0 \rightarrow 0, B_1 \rightarrow 1, A_5 \rightarrow B_1A_2, A_5 \rightarrow 1, A_2 \rightarrow A_2A_3, A_2 \rightarrow 0, A_2 \rightarrow 1, A_2 \rightarrow A_4A_2, A_2 \rightarrow B_1A_2A_1, A_2 \rightarrow B_1A_2, A_2 \rightarrow B_1A_1, A_3 \rightarrow 0, A_3 \rightarrow 1, A_1 \rightarrow A_2A_2A_2, A_1 \rightarrow A_2A_2, A_1 \rightarrow A_2A_3, A_1 \rightarrow 0, A_1 \rightarrow 1, A_1 \rightarrow A_4A_2, A_1 \rightarrow B_1A_2A_1, A_1 \rightarrow B_1A_2, A_1 \rightarrow B_1A_1, A_4 \rightarrow B_1A_2A_1, A_4 \rightarrow B_1A_2, A_4 \rightarrow B_1A_1, A_4 \rightarrow 1\}.$$

Schließlich werden alle rechten Seiten mit Länge  $\geq 3$  durch Einführen neuer Variablen zerlegt,

sodass wir als Ausgabegrammatik  $G_7 = (V_7, \Sigma, P_7, A_5)$  in Chomsky-Normalform erhalten, wobei

$$V_7 = \{A_1, A_2, A_3, A_4, A_5, B_0, B_1, C_1, C_2, C_3, C_4\}$$

$$P_7 = \{B_0 \rightarrow 0, B_1 \rightarrow 1, A_5 \rightarrow B_1A_2, A_5 \rightarrow 1, A_2 \rightarrow A_2A_3, A_2 \rightarrow 0, A_2 \rightarrow 1, A_2 \rightarrow A_4A_2, \\ A_2 \rightarrow B_1C_1, C_1 \rightarrow A_2A_1, A_2 \rightarrow B_1A_2, A_2 \rightarrow B_1A_1, A_3 \rightarrow 0, A_3 \rightarrow 1, \\ A_1 \rightarrow A_2C_2, C_2 \rightarrow A_2A_2, A_1 \rightarrow A_2A_2, A_1 \rightarrow A_2A_3, A_1 \rightarrow 0, A_1 \rightarrow 1, A_1 \rightarrow A_4A_2, \\ A_1 \rightarrow B_1C_3, C_4 \rightarrow A_2A_1, A_1 \rightarrow B_1A_2, A_1 \rightarrow B_1A_1, A_4 \rightarrow B_1C_4, C_4 \rightarrow A_2A_1, \\ A_4 \rightarrow B_1A_2, A_4 \rightarrow B_1A_1, A_4 \rightarrow 1\}.$$

## A.6. Herstellen der Greibach-Normalform

Algorithmus 9 führt die Umformung einer CFG in Chomsky-Normalform in die Greibach-Normalform durch. Die geschachtelten für-Schleifen für  $i$  und  $j$  eliminieren Produktionen der Form  $A_i \rightarrow A_ju$  mit  $i \geq j$ : Für  $i > j$ , wird das Vorkommen von  $A_j$  am Anfang der rechten Seite durch Hineinkopieren der Regel für  $A_j$  (d. h. deren rechten Seite) ersetzt, wobei alle Möglichkeiten in Betracht gezogen werden müssen. Die Korrektheit dieser Ersetzung folgt aus Lemma A.4.1. Für den Fall  $A_i \rightarrow A_iu$  wird die Elimination der Links-Rekursion, wie wir sie in Lemma A.4.3 gesehen haben, angewendet. Nach Abschluss der geschachtelten für-Schleifen, sind Regeln der Form  $A_i \rightarrow A_ju$  nur noch möglich, wenn  $j > i$  gilt. Daher muss jede Regel mit  $A_n$  (der „größten Variablen“) als linker Seite von der Form  $A_n \rightarrow u$  sein, wobei  $u$  nicht mit einer Variablen (und daher mit einem Symbol aus  $\Sigma$ ) beginnt. Deshalb können wir sukzessive die Regeln für  $A_n$ , dann für  $A_{n-1}$ , u. s. w. hineinkopieren in Regel der Form  $A_i \rightarrow A_ju$  (und  $A_j$  durch alle Möglichkeiten ersetzen): Nach dem Behandeln von  $A_n$ , kommt kein  $A_n$  mehr in der Form  $A_i \rightarrow A_nu$  vor, nach dem Behandeln von  $A_{n-1}$  kommt kein  $A_{n-1}$  mehr in der Form  $A_i \rightarrow A_{n-1}u$  vor u. s. w., so dass am Ende keine Regeln mehr die Form  $A_i \rightarrow A_ju$  hat. Die Korrektheit der Ersetzungen folgt aus Lemma A.4.1.

Da die anfängliche Grammatik in Chomsky-Normalform, sind alle Regeln, die mit  $A_i$  beginnen, danach in Greibach-Normalform. Die im Rahmen der Elimination der Links-Rekursion eingeführten Regeln mit  $B_i$  als linker Seite, hingegen müssen noch einmal durch hineinkopieren der  $A_i$ -Regeln behandelt werden, was in der letzten für-Schleife geschieht. Die Korrektheit der Ersetzungen folgt erneut aus Lemma A.4.1.

Für die Korrektheit des gesamten Algorithmus muss man verifizieren, dass die in den Kommentaren angegebenen Invarianten tatsächlich erfüllt sind, und daher am Ende, sämtliche Produktionen entsprechend der Greibach-Normalform von der Form  $A \rightarrow au$  mit  $a \in \Sigma, u \in V^*$  sind.

Insgesamt lässt sich jede CFG (die nicht das leere Wort erzeugt) zunächst in Chomsky-Normalform und anschließend mit Algorithmus 9 in Greibach-Normalform transformieren. D. h. wir haben den folgenden Satz gezeigt:

**Satz A.6.1.** *Zu jeder CFG  $G$  mit  $\varepsilon \notin L(G)$  gibt es eine CFG  $G'$  in Greibach-Normalform, sodass  $L(G) = L(G')$  gilt.*

**Algorithmus 9** : Herstellen der Greibach-Normalform**Eingabe** : CFG  $G = (\{A_1, \dots, A_n\}, \Sigma, P, A_i)$  in Chomsky-Normalform mit  $\varepsilon \notin L(G)$ **Ausgabe** : CFG  $G'$  in Greibach-Normalform mit  $L(G) = L(G')$ **Beginn**/\* Ziel der geschachtelten Für-Schleife ist, dass es Regeln  $A_i \rightarrow A_j u$  nur gibt, wenn  $j > i$  gilt. \*/**für**  $i = 1$  bis  $n$  **tue**    **für**  $j = 1$  bis  $i - 1$  **tue**        /\* Ersetzen der Regeln  $A_i \rightarrow A_j u$  mit  $i > j$  \*/        **für alle**  $A_i \rightarrow A_j u \in P$  **tue**            Seien  $A_j \rightarrow w_1 \mid \dots \mid w_m$  alle Regeln in  $P$  mit  $A_j$  als linker Seite;            Ersetze  $A_i \rightarrow A_j u$  durch  $A_i \rightarrow w_1 u \mid \dots \mid w_m u$  in  $P$ ;        **Ende**    **Ende**/\* Ersetzen der Regeln  $A_i \rightarrow A_i u$  \*/    **wenn**  $A_i \rightarrow A_i u \in P$  **dann**        Wende Lemma A.4.3 an, um die links-rekursiven Regeln für  $A_i$  zu ersetzen;        Sei  $B_i$  die dabei neu erzeugte Variable;    **Ende****Ende**/\* Nun gibt es nur noch Regeln der Form  $A_i \rightarrow A_j u$  mit  $j > i$ . \*//\* Insbesondere gilt damit auch für alle Regeln  $A_n \rightarrow u$ :  $u$  beginnt mit einem Zeichen aus  $\Sigma$  \*//\* Ziel der nächsten Schleife: Ersetze alle Regeln  $A_i \rightarrow A_j u$  mit  $j > i$  \*/**für**  $i = n - 1$  bis  $1$  **tue**    **für alle**  $A_i \rightarrow A_j u \in P, j > i$  **tue**        Seien  $A_j \rightarrow w_1 \mid \dots \mid w_m$  alle Regeln mit  $A_j$  als linker Seite;        /\* Beachte: Alle  $w_1, \dots, w_m$  fangen mit Zeichen aus  $\Sigma$  an, da Schleife absteigend läuft \*/        Ersetze  $A_i \rightarrow A_j u$  durch  $A_i \rightarrow w_1 u \mid \dots \mid w_m u$  in  $P$ ;    **Ende****Ende**/\* Nun sind alle Regeln mit  $A_i, i = 1, \dots, n$  als linker Seite in Greibach-Normalform, behandle noch die eingeführten Regeln mit  $B_i$  als linker Seite \*/**für**  $i = 1$  bis  $n$  **tue**    **für alle**  $B_i \rightarrow A_j u \in P$  **tue**        Seien  $A_j \rightarrow w_1 \mid \dots \mid w_m$  alle Regeln mit  $A_j$  als linker Seite;        Ersetze  $B_i \rightarrow A_j u$  durch  $B_i \rightarrow w_1 u \mid \dots \mid w_m u$  in  $P$ ;    **Ende****Ende****Ende**

**Beispiel A.6.2.** Betrachte  $G = (\{A_1, A_2, A_3\}, \{a, b, c, d\}, P, A_1)$  mit

$$P = \{A_1 \rightarrow A_1A_2 \mid a, \\ A_2 \rightarrow A_1A_3 \mid A_3A_3, \\ A_3 \rightarrow A_3c \mid d\}.$$

Beim Durchlaufen der doppelt-geschachtelten „Für“-Schleifen, wird zunächst für  $i = 1$  und  $j = 1$  die Produktion  $A_1 \rightarrow A_1A_2$  ersetzt durch  $A_1 \rightarrow aB_1 \mid a$  und  $B_1 \rightarrow A_2 \mid A_2B_1$ , d. h. danach ist die Menge der Produktionen

$$P_1 = \{A_1 \rightarrow aB_1 \mid a, \\ A_2 \rightarrow A_1A_3 \mid A_3A_3, \\ A_3 \rightarrow A_3c \mid d, \\ B_1 \rightarrow A_2 \mid A_2B_1\}.$$

Für  $i = 2$  und  $j = 1$  wird die Produktion  $A_2 \rightarrow A_1A_3$  ersetzt durch  $A_2 \rightarrow aB_1A_3 \mid aA_3$ , d. h. die Menge der Produktionen ist danach

$$P_2 = \{A_1 \rightarrow aB_1 \mid a, \\ A_2 \rightarrow aB_1A_3 \mid aA_3 \mid A_3A_3, \\ A_3 \rightarrow A_3c \mid d, \\ B_1 \rightarrow A_2 \mid A_2B_1\}.$$

Für  $i = 3$  und  $j = 3$  wird  $A_3 \rightarrow A_3c$  ersetzt durch  $A_3 \rightarrow d \mid dB_3$ ,  $B_3 \rightarrow c \mid cB_3$ , d. h. die Menge der Produktionen ist danach

$$P_3 = \{A_1 \rightarrow aB_1 \mid a, \\ A_2 \rightarrow aB_1A_3 \mid aA_3 \mid A_3A_3, \\ A_3 \rightarrow d \mid dB_3, \\ B_1 \rightarrow A_2 \mid A_2B_1, \\ B_3 \rightarrow c \mid cB_3\}.$$

Durchlaufen der „Für  $i = n - 1$  bis 1“-Schleife ersetzt  $A_2 \rightarrow A_3A_3$  durch  $A_2 \rightarrow dA_3 \mid dB_3A_3$ , d. h. die Menge der Produktionen ist danach

$$P_4 = \{A_1 \rightarrow aB_1 \mid a, \\ A_2 \rightarrow aB_1A_3 \mid aA_3 \mid dA_3 \mid dB_3A_3, \\ A_3 \rightarrow d \mid dB_3, \\ B_1 \rightarrow A_2 \mid A_2B_1, \\ B_3 \rightarrow c \mid cB_3\}.$$

Durchlauf der letzten „Für“-Schleife ersetzt

- $B_1 \rightarrow A_2$  durch  $B_1 \rightarrow aB_1A_3 \mid aA_3 \mid dA_3 \mid dB_3A_3$ , und
- $B_1 \rightarrow A_2B_1$  durch  $B_1 \rightarrow aB_1A_3B_1 \mid aA_3B_1 \mid dA_3B_1 \mid dB_3A_3B_1$

was insgesamt die Grammatik  $G' = (\{A_1, A_2, A_3, A_4, B_1, B_3\}, \{a, b, c, d\}, P', A_1)$  mit

$$\begin{aligned}
 P' = \{ & A_1 \rightarrow aB_1 \mid a, \\
 & A_2 \rightarrow aB_1A_3 \mid aA_3 \mid dA_3 \mid dB_3A_3, \\
 & A_3 \rightarrow d \mid dB_3, \\
 & B_1 \rightarrow aB_1A_3 \mid aA_3 \mid dA_3 \mid dB_3A_3 \mid aB_1A_3B_1 \mid aA_3B_1 \mid dA_3B_1 \mid dB_3A_3B_1, \\
 & B_3 \rightarrow c \mid cB_3\}.
 \end{aligned}$$

als Ausgabe (in Greibach-Normalform) erzeugt.

## A.7. Die Kuroda-Normalform für kontextsensitive Grammatiken

Ähnlich zur Chomsky-Normalform für kontextfreie Grammatiken, gibt es die sogenannte Kuroda-Normalform (benannt nach dem japanischen Linguisten Sige-Yuki Kuroda) für kontextsensitive Grammatiken:

**Definition A.7.1.** Eine Typ 1-Grammatik  $G = (V, \Sigma, P, S)$  ist in Kuroda-Normalform, falls alle Produktionen in  $P$  einer der folgenden vier Formen entsprechen:

$$A \rightarrow a \quad A \rightarrow B \quad A \rightarrow BC \quad AB \rightarrow CD$$

wobei  $a \in \Sigma$  und  $A, B, C, D \in V$ .

Die ersten drei Formate sind auch für kontextfreie Sprachen erlaubt (wobei Regelformat 1 und 3 die erlaubten Formate für CFGs in Chomsky-Normal sind). Die Kuroda-Normalform „erweitert“ kontextfreie Grammatiken daher um Regeln der Form  $AB \rightarrow CD$ .

**Satz A.7.2.** Sei  $L$  eine kontextsensitive Sprache mit  $\varepsilon \notin L$ . Dann gibt es eine Grammatik in Kuroda-Normalform, die  $L$  erzeugt.

*Beweis.* Da  $L$  eine kontextsensitive Sprache ist, gibt es eine Typ 1-Grammatik  $G = (V, \Sigma, P, S)$  mit  $L(G) = L$ . Wende Algorithmus 10 mit  $G$  als Eingabe an. Die Ausgabegrammatik  $G'$  erfüllt die Aussage: Zunächst ist klar, dass die Ausgabe in Kuroda-Normalform ist, da nicht konforme Produktionen durch konforme ersetzt werden. Wir begründen, dass die einzelnen Modifikationen die erzeugte Sprache nicht verändern:

Das Einführen der Produktionen  $A_a \rightarrow a$  und das Entfalten der Produktionen  $A \rightarrow B_1 \cdots B_n$  kennen wir bereits von der Herstellung der Chomsky-Normalform für kontextfreie Grammatiken und es verändert dort wie auch für kontextsensitive Grammatiken die Menge der ableitbaren Worte nicht.

Wir betrachten eine einzelne Ersetzung einer Regel  $A_1 \cdots A_m \rightarrow B_1 \cdots B_n$  ( $n \geq m + 2$ ) in mehrere Regeln wie in Algorithmus 10 beschrieben. Sei  $G_0$  die Grammatik vor der Ersetzung und  $G_1$  die Grammatik danach. Ein Ableitungsschritt  $\alpha A_1 \cdots A_m \alpha' \Rightarrow_{G_0} \alpha B_1 \cdots B_n \alpha'$  kann mit

$G_1$  in mehreren Schritten durchgeführt werden:  $\alpha A_1 \cdots A_m \alpha' \Rightarrow_{G_1} \alpha B_1 D_2 A_3 \cdots A_m \alpha' \Rightarrow_{G_1}^{m-2} \alpha B_1 \cdots B_{m-1} D_m \alpha' \Rightarrow_{G_1}^* \alpha B_1 \cdots B_{n-2} D_{n-1} \alpha' \Rightarrow_{G_1} \alpha B_1 \cdots B_n \alpha'$

Da die Variablen  $D_i$  neu sind, und jede Variable  $D_i$  nur einmal in einer linken Seite einer Produktion vorkommt, muss jede Ableitung für ein Wort  $w \in \Sigma^*$  mit Grammatik  $G_1$  sämtliche neuen Produktionen in der gegebenen Art und Weise verwenden (anders können keine Terminalzeichen erzeugt werden), d. h. wenn  $\alpha A_1 A_2 \alpha' \Rightarrow_{G_1} \alpha B_1 D_2 \alpha' \Rightarrow_{G_1}^k w$ , dann lassen sich alle Ersetzungen von  $D_i$  für  $i = 2, \dots, n-1$  in der Ableitung identifizieren und es ergibt sich, dass  $\alpha' = A_3 \cdots A_n \alpha''$  gelten muss und sich die Ableitung umsortieren lässt zu  $\alpha A_1 \cdots A_n \alpha'' \Rightarrow_{G_1}^{k_1} \alpha B_1 \cdots B_n \alpha'' \Rightarrow_{G_1}^{k_2} w$  mit  $k = k_1 + k_2$  und  $k_1 > 0$ , was zeigt, dass  $\alpha A_1 \cdots A_n \alpha'' \Rightarrow_{G_0} \alpha B_1 \cdots B_n \alpha'' \Rightarrow_{G_1}^{k_2} w$ . Diese Rückübersetzung in eine Ableitung für  $G_0$  kann nun für die restlichen  $k_2$  Schritte  $\alpha B_1 \cdots B_n \alpha'' \Rightarrow_{G_1}^{k_2} w$  wieder angewendet werden. Wichtig ist hierbei, dass diese Ersetzung terminiert, was stimmt, da  $k_2 < k$  gelten muss.  $\square$

## A.8. Beweis des Satzes von Kuroda

**Satz A.8.1.** *Jede kontextsensitive Sprache wird von einem LBA erkannt.*

*Beweis.* Sei  $L$  eine Typ 1-Sprache. O.B.d.A. nehmen wir an, dass  $G = (V, \Sigma, P, S)$  eine Grammatik in Kuroda-Normalform ist. Wir konstruieren eine nichtdeterministische Turingmaschine, die als Bandalphabet  $\Gamma$  zumindest  $((\Sigma \cup V) \cup \overline{\Sigma \cup V} \cup \square) \subseteq \Gamma$  verwendet. In den folgenden Erläuterung erwähnen wir das speziell markierte letzte Zeichen auf dem Band nicht explizit, gehen aber davon aus, dass der LBA entsprechend Rücksicht darauf nimmt und notwendige Ersetzungen macht bzw. definierte  $\delta$ -Übergänge hat. Der LBA versucht (nichtdeterministisch) aus einem Wort  $w \in \Sigma^*$  das Startsymbol  $S$  der Grammatik rückwärts herzuleiten, indem er Produktionen  $\ell \rightarrow r \in P$  rückwärts auf das Wort anwendet, d. h. der LBA sucht nach Vorkommen von  $r$  in der aktuellen Eingabe und ersetzt diese durch die linke Seite  $\ell$ . Für Produktionen  $A \rightarrow a, A \rightarrow B, AB \rightarrow CD$  verändern diese Ersetzungen die Wortlänge nicht. Für den Fall  $A \rightarrow BC$  muss das Teilwort  $BC$  durch  $A$  ersetzt werden. Dies geschieht indem  $BC$  durch  $\square A$  ersetzt wird und anschließend werden alle Zeichen von links um 1 nach rechts verschoben. Wenn auf dem Band ausschließlich das Startsymbol  $S$  steht, dann geht der LBA in einen akzeptierenden Zustand. Beachte, dass der LBA nichtdeterministisch ist, denn er kann einerseits frei wählen, welche Produktion aus  $P$  rückwärts angewendet wird und andererseits für eine feste Produktion  $\ell \rightarrow r$  kann er eines von verschiedenen Vorkommen von  $r$  im aktuellen Bandinhalt auswählen.

Da  $G$  in Kuroda-Normalform ist, kann die Suche nach einer rechten Seite  $r$ , folgendermaßen „programmiert“ werden: Beginne links an der Eingabe und laufe diese durch. Speichere im aktuellen Zustand, das Symbol links vom Schreib-Lesekopf (d. h. für jedes Symbol in  $\Sigma$  gibt es einen eigenen Zustand) und entscheide mit dem aktuellen Symbol, ob es eine passende Produktion gibt (da rechte Seiten von Produktionen in Kuroda-Normalform aus maximal 2 Zeichen bestehen, genügt dies).

---

**Algorithmus 10** : Herstellung der Kuroda-Normalform

---

**Eingabe** : Eine Typ 1-Grammatik  $G = (V, \Sigma, P, S)$  mit  $\varepsilon \notin L(G)$

**Ausgabe** : Eine Typ 1-Grammatik in Kuroda-Normalform die  $L(G)$  erzeugt.

**Beginn**

**für alle**  $a \in \Sigma$  **tue**

    /\* Führe neue Variable  $A_a$  für  $a$  ein, und ersetze Vorkommen von  $a$  durch das Nichtterminal \*/  
     $G := (V \cup \{A_a\}, \Sigma, \{\ell[A_a/a] \rightarrow r[A_a/a] \mid \ell \rightarrow r \in P\}, S);$

**Ende**

/\* Nun sind alle Regeln von der Form  $A \rightarrow a$  oder  $A_1 \cdots A_m \rightarrow B_1 \cdots B_n$  mit  $A_i, B_j \in V$  \*/

**für alle**  $A \rightarrow B_1 \cdots B_n \in P$  mit  $n > 2$  **tue**

    Seien  $C_1, \dots, C_{n-2}$  neue Variablen;  
     $V := V \cup \{C_1, \dots, C_{n-2}\};$   
    /\* Ersetze in  $P$  die Produktion  $A \rightarrow B_1 \cdots B_n$  durch neue Regeln \*/  
     $P := (P \setminus \{A \rightarrow B_1 \cdots B_n\})$   
     $\cup \{A \rightarrow B_1 C_1\} \cup \{C_i \rightarrow B_{i+1} C_{i+1} \mid i = 1, \dots, n-3\} \cup \{C_{n-2} \rightarrow B_{n-1} B_n\};$

**Ende**

**für alle**  $A_1 \cdots A_m \rightarrow B_1 \cdots B_n \in P$  mit ( $m > 2$  oder  $n > 2$ ) und  $n \geq m + 2$  **tue**

    Seien  $D_2, \dots, D_{n-1}$  neue Variablen;  
     $V := V \cup \{D_2, \dots, D_{n-1}\};$   
    /\* Ersetze  $A_1 \cdots A_m \rightarrow B_1 \cdots B_n$  durch neue Regeln \*/  
     $P := (P \setminus \{A_1 \cdots A_m \rightarrow B_1 \cdots B_n\})$   
     $\cup \{A_1 A_2 \rightarrow B_1 D_2\} \cup \{D_i A_{i+1} \rightarrow B_i D_{i+1} \mid i = 2, \dots, m-1\}$   
     $\cup \{D_i \rightarrow B_i D_{i+1} \mid i = m, \dots, n-2\} \cup \{D_{n-1} \rightarrow B_{n-1} B_n\}$

**Ende**

**für alle**  $A_1 \cdots A_n \rightarrow B_1 \cdots B_{n+1} \in P$  mit  $n > 2$  **tue**

    Seien  $D_2, \dots, D_n$  neue Variablen;  
    /\* Ersetze in  $P$  die Produktion  $A_1 \cdots A_n \rightarrow B_1 \cdots B_{n+1}$  durch neue Regeln \*/  
     $V := V \cup \{D_2, \dots, D_n\};$   
     $P := (P \setminus \{A_1 \cdots A_n \rightarrow B_1 \cdots B_{n+1}\}) \cup \{A_1 A_2 \rightarrow B_1 D_2\}$   
     $\cup \{D_i A_{i+1} \rightarrow B_i D_{i+1} \mid i = 2, \dots, n-1\} \cup \{D_n \rightarrow B_n B_{n+1}\}$

**Ende**

**für alle**  $A_1 \cdots A_n \rightarrow B_1 \cdots B_n \in P$  mit  $n > 2$  **tue**

    Seien  $D_2, \dots, D_{n-1}$  neue Variablen;  
     $V := V \cup \{D_2, \dots, D_{n-1}\};$   
    /\* Ersetze in  $P$  die Produktion  $A_1 \cdots A_n \rightarrow B_1 \cdots B_n$  durch neue Regeln \*/  
     $P := (P \setminus \{A_1 \cdots A_n \rightarrow B_1 \cdots B_n\}) \cup \{A_1 A_2 \rightarrow B_1 D_2\}$   
     $\cup \{D_i A_{i+1} \rightarrow B_i D_{i+1} \mid i = 2, \dots, n-2\} \cup \{D_{n-1} A_n \rightarrow B_{n-1} B_n\}$

**Ende**

Gebe die so entstandene Grammatik aus;

**Ende**

---

Für Produktionen  $A \rightarrow a$  und  $A \rightarrow B$  wird das aktuelle Symbol durch  $A$  ersetzt, anschließend wird der nächste Schritt gestartet (d. h. es gibt einen Zustand, der den Schreib-Lesekopf nach links fährt). Für Produktionen  $AB \rightarrow CD$ , wird  $B$  geschrieben und der Kopf nach links wechseln, dann  $A$  geschrieben und der nächste Schritt gestartet. Für Produktionen  $A \rightarrow BC$  schreibe  $A$  und wechsele nach links, schreibe  $\square$ , fahre ganz nach links und starte Prozedur zum Verschieben der Zeichen nach rechts, solange bis die Lücke geschlossen ist. Diese Prozedur funktioniert so, dass der Zustand zunächst das linkeste Symbol speichert, und anschließend nach rechts über das Band läuft und das gespeicherte Symbol mit dem gelesenen Symbol vertauscht. Das Vertauschen wird beendet, nachdem  $\square$  mit einem anderen Symbol vertauscht wird.

Da für alle Produktionen  $\ell \rightarrow r \in P$  gilt  $|\ell| \leq |r|$  werden nur Teilworte  $r$  durch gleichlange oder kürzere Teilworte  $l$  ersetzt, daher kommt die Turingmaschine mit dem durch die Eingabe zur Verfügung gestellten Platz aus und kann als LBA programmiert werden.  $\square$

**Bemerkung A.8.2.** Die Konstruktion im Beweis des letzten Satzes kann auch für beliebige Typ 1-Grammatiken (die nicht notwendigerweise in Kuroda-Normalform sind) angepasst werden. Die Suche nach einem Teilwort  $r$  wird dann komplizierter, da innerhalb der Zustände der TM kodiert nicht nur ein Zeichen gespeichert werden muss, sondern maximal  $q - 1$  Zeichen, wobei  $q$  die Länge der längsten rechte Seite aller Produktionen ist. Trotzdem reichen endlich viele Zustände dafür aus und die TM kommt mit linear beschränktem Platz aus (und kann daher als LBA konstruiert werden).

Für Typ 0-Grammatiken kann es Produktionen der Form  $\ell \rightarrow r$  mit  $|r| < |\ell|$  geben, sodass einerseits mehr Platz benötigt wird und andererseits die Konstruktion der TM etwas aufwendiger ist, da sie Platz schaffen muss (was aber trotz allem möglich ist). Daher kann man auch eine NTM  $M$  (aber keinen LBA!) für Typ 0-Grammatiken  $G$  konstruieren, sodass  $L(M) = L(G)$  gilt.

**Satz A.8.3.** Jede Typ  $i$ -Sprache (für  $i=0,1,2,3$ ) wird von einer nichtdeterministischen Turing-Maschine akzeptiert.

Wir betrachten nun die andere Richtung:

**Satz A.8.4.** Sei  $M$  ein LBA. Dann ist  $L(M)$  eine kontextsensitive Sprache.

*Beweis.* Basierend auf dem LBA  $M = (Z, \Sigma \cup \widehat{\Sigma}, \Gamma, \delta, z_0, \square, E)$  konstruieren wir eine kontextsensitive Grammatik  $G$ , sodass  $L(G) = L(M)$  gilt. Die Idee für die Grammatik ist es, zunächst ein beliebiges Wort aus  $\Sigma^*$  zu generieren, dann den Ablauf der Turingmaschine auf diesem Wort mithilfe der Grammatik zu simulieren, und schließlich falls die Turingmaschine akzeptiert, das Wort endgültig herzustellen. Damit das Ganze gelingt, wird zunächst nicht nur das Wort aus  $\Sigma^*$ , sondern gleichzeitig die Startkonfiguration der Turingmaschine erzeugt. Die gleichzeitige Erzeugung funktioniert, indem wir ganz spezielle Symbole für die Menge der Variablen der Grammatik verwenden: Einerseits neue Variablen  $S$  und  $A$  und andererseits Tupel (die wir übereinander schreiben)  $\left\langle \begin{smallmatrix} u \\ v \end{smallmatrix} \right\rangle$  wobei  $u \in \Sigma$  und  $v \in \Gamma \cup (Z\Gamma)$ . Die unteren Komponenten der Tupel

sind entweder Bandinhalte oder ein Paar bestehend aus Zustand und Bandinhalt. Die Regeln zur gleichzeitigen Erzeugung eines Wortes  $w \in \Sigma^*$  und der Startkonfiguration von  $M$  für das Wort  $w$  sind:

$$P_1 := \left\{ S \rightarrow A \left\langle \begin{matrix} a \\ \widehat{a} \end{matrix} \right\rangle \mid a \in \Sigma \right\} \cup \left\{ A \rightarrow A \left\langle \begin{matrix} a \\ a \end{matrix} \right\rangle \mid a \in \Sigma \right\} \cup \left\{ A \rightarrow \left\langle \begin{matrix} a \\ z_0 a \end{matrix} \right\rangle \mid a \in \Sigma \right\}$$

Worte  $w \in L(M)$  mit  $|w| < 2$  können dadurch nicht erzeugt werden, diese werden durch extra Regeln direkt aus  $S$  erzeugt (das sind nur endlich viele): Sei  $P_0 = \{S \rightarrow w \mid |w| < 2, w \in L(M)\}$ .

Für alle anderen Worte  $w = a_1 \cdots a_n \in \Sigma^*$  gilt  $S \Rightarrow_{P_1} A \left\langle \begin{matrix} a_n \\ \widehat{a}_n \end{matrix} \right\rangle \Rightarrow_{P_1}^* \left\langle \begin{matrix} a_1 \\ z_0 a_1 \end{matrix} \right\rangle \left\langle \begin{matrix} a_2 \\ a_2 \end{matrix} \right\rangle \cdots \left\langle \begin{matrix} a_n \\ \widehat{a}_n \end{matrix} \right\rangle$

Lesen wir die obere Reihe des erzeugten Wortes, ist dies  $w$  und die untere Reihe entspricht der Startkonfiguration  $z_0 a_1 \cdots a_{n-1} \widehat{a}_n$  von  $M$  für  $w$ .

Der nächste Regelsatz  $P_2$  bildet die Transitionsrelation für  $M$  anhand der Definition von  $\delta$  nach:

Alle diese Regeln operieren auf den Tupelsequenzen  $\left\langle \begin{matrix} a_1 \\ v_1 \end{matrix} \right\rangle \cdots \left\langle \begin{matrix} a_n \\ v_n \end{matrix} \right\rangle$  und lassen die obere Folge von Buchstaben unverändert (da sie nur die Konfiguration des LBA verändern), d. h. wenn wir Regeln  $P_2^{\text{unten}}$  haben, dann können wir den gesamten Regelsatz definieren durch

$$P_2 := \left\{ \left\langle \begin{matrix} a \\ u \end{matrix} \right\rangle \left\langle \begin{matrix} b \\ v \end{matrix} \right\rangle \rightarrow \left\langle \begin{matrix} a \\ u' \end{matrix} \right\rangle \left\langle \begin{matrix} b \\ v' \end{matrix} \right\rangle \mid a, b \in \Sigma \text{ und } uv \rightarrow u'v' \in P_2^{\text{unten}} (\text{mit } u, v, u', v' \in \Gamma \cup (Z\Gamma)) \right\} \\ \cup \left\{ \left\langle \begin{matrix} a \\ u \end{matrix} \right\rangle \rightarrow \left\langle \begin{matrix} a \\ u' \end{matrix} \right\rangle \mid a \in \Sigma \text{ und } u \rightarrow u' \in P_2^{\text{unten}} (\text{mit } u, u' \in \Gamma \cup (Z\Gamma)) \right\}$$

Wir definieren nun  $P_2^{\text{unten}}$ :

$$P_2^{\text{unten}} := \{c z a \rightarrow z' c b \mid \text{für alle } c \in \Gamma \text{ und } (z', b, L) \in \delta(z, a)\} \\ \cup \{z a c \rightarrow b z' c \mid \text{für alle } c \in \Gamma \text{ und } (z', b, R) \in \delta(z, a)\} \\ \cup \{z a \rightarrow z b \mid \text{für alle } c \in \Gamma \text{ und } (z', b, N) \in \delta(z, a)\}$$

Es lässt sich prüfen, dass aus  $w z w' \vdash_M^* u z' u'$  auch  $w z w' \Rightarrow_{P_2^{\text{unten}}}^* u z' u'$  folgt, wobei die Darstellung von  $z, z'$  entsprechend geändert ist, d. h. sie tauchen in der Ableitung nur verbunden mit einem Zeichen aus  $\Gamma$  auf (hier mit dem ersten Zeichen von  $w'$  bzw.  $u'$ ).

Der dritte Regelsatz  $P_3$  hat die Aufgabe nach Akzeptieren des LBA, aus den Tupelsequenzen das Wort  $a_1 \cdots a_n$  zu erzeugen.

$$P_3 := \left\{ \left\langle \begin{matrix} b \\ z a \end{matrix} \right\rangle \rightarrow b \mid z \in E, a \in \Gamma, b \in \Sigma \right\} \cup \left\{ \left\langle \begin{matrix} b \\ a \end{matrix} \right\rangle \rightarrow b \mid a \in \Gamma, b \in \Sigma \right\}$$

Es gilt  $\left\langle \begin{matrix} a_1 \\ b_1 \end{matrix} \right\rangle \cdots \left\langle \begin{matrix} a_m \\ b_m \end{matrix} \right\rangle \left\langle \begin{matrix} a_{m+1} \\ z b_{m+1} \end{matrix} \right\rangle \left\langle \begin{matrix} a_{m+2} \\ b_{m+2} \end{matrix} \right\rangle \cdots \left\langle \begin{matrix} a_n \\ b_n \end{matrix} \right\rangle \Rightarrow_{P_3}^* a_1 \cdots a_n$ . Beachte, dass die Regeln aus  $P_3$

auch in Nicht-Zielzuständen einzelne Tupel  $\langle a_i \rangle$  in  $a_i$  umformen können, aber niemals ein Tupel  $\langle z b_i \rangle$  mit  $z \notin E$ . Da die  $\langle a_i \rangle$  Variablen der Grammatik sind, lässt sich daher kein Wort aus  $\Sigma^*$  herleiten.

Schließlich sei  $G = \left( \{S, A\} \cup \left\{ \langle u \rangle \mid u \in \Sigma, v \in \Gamma \cup (Z\Gamma) \right\}, \Sigma, P_0 \cup P_1 \cup P_2 \cup P_3, S \right)$ . Dann gilt für alle  $w \in \Sigma^*$ :  $S \Rightarrow_G^* w$  genau dann, wenn  $w \in L(M)$ .

Des weiteren gilt, dass  $G$  eine kontextsensitive Grammatik ist, da es keine verkürzenden Regeln gibt.  $\square$

## A.9. Der Satz von Rice

Der folgende Satz (von Henry Gordon Rice im Jahr 1953 veröffentlicht, (Ric53)) zeigt, dass quasi jede interessante Eigenschaft von Turingmaschinen algorithmisch nicht entscheidbar ist. Z.B. zeigt der Satz, dass die Sprache  $L = \{w \mid M_w \text{ berechnet eine konstante Funktion}\}$  nicht entscheidbar ist.

**Satz A.9.1** (Satz von Rice). *Sei  $\mathcal{R}$  die Klasse aller Turingberechenbaren Funktionen. Sei  $\mathcal{S}$  eine beliebige Teilmenge, sodass  $\emptyset \subset \mathcal{S} \subset \mathcal{R}$ . Dann ist die Sprache*

$$C(\mathcal{S}) = \{w \mid \text{die von } M_w \text{ berechnete Funktion liegt in } \mathcal{S}\}$$

*unentscheidbar.*

*Beweis.* Sei  $\Omega$  die überall undefinierte Funktion. Offensichtlich gilt  $\Omega \in \mathcal{R}$ . Nehme an, dass  $\Omega \notin \mathcal{S}$  gilt. Da  $\emptyset \subset \mathcal{S}$ , gibt es eine Funktion  $q \in \mathcal{S}$ , die von einer Turingmaschine  $Q$  berechnet wird.

Sei  $M^*$  eine Turingmaschine, die bei Eingabe  $y$  und einer weiteren Turingmaschine  $M$  das folgende macht:

1. Simuliere  $M$  auf leerer Eingabe.
2. Wenn  $M$  anhält, dann simuliere  $Q$  mit Eingabe  $y$ .

Sei  $f$  die Funktion, die aus einer Beschreibung  $w$  für TM  $M_w$ , die Beschreibung der TM  $M_w^*$  erstellt.

Dann gilt:  $w \in H_0 \implies M_w$  hält auf leerer Eingabe  
 $\implies M_w^*$  berechnet  $q$   
 $\implies$  die von  $M_w^*$  berechnete Funktion liegt  $\mathcal{S}$   
 $\implies f(w) \in C(\mathcal{S})$

und ebenso:  $w \notin H_0 \implies M_w$  hält nicht auf leerer Eingabe  
 $\implies M_w^*$  berechnet  $\Omega$   
 $\implies$  die von  $M_w^*$  berechnete Funktion liegt nicht in  $\mathcal{S}$   
 $\implies f(w) \notin C(\mathcal{S})$

Damit haben wir gezeigt:  $w \in H_0 \iff f(w) \in C(\mathcal{S})$  und somit  $H_0 \leq C(\mathcal{S})$  und da  $H_0$  unentscheidbar ist, ist damit auch  $C(\mathcal{S})$  unentscheidbar.

Wenn  $\Omega \in \mathcal{S}$  gilt, dann zeige mit obigem Beweis, dass  $H_0 \leq C(\mathcal{R} \setminus \mathcal{S})$ . Dann folgt aus der Unentscheidbarkeit von  $C(\mathcal{R} \setminus \mathcal{S})$  auch die Unentscheidbarkeit von  $\overline{C(\mathcal{R} \setminus \mathcal{S})} = C(\mathcal{S})$  (dies ist eine Konsequenz aus Korollar 7.3.3).  $\square$

## A.10. Unentscheidbarkeit des Postschen Korrespondenzproblems

Wir zeigen die Unentscheidbarkeit des PCP durch zwei Reduktionen: Wir zeigen sowohl  $H \leq \text{MPCP}$  als auch  $\text{MPCP} \leq \text{PCP}$ . Dabei ist MPCP das *modifizierte* PCP, welches nur Lösungen erlaubt, die mit dem Spielstein der ersten Bauart beginnen.

### A.10.1. MPCP $\leq$ PCP

**Definition A.10.1** (Modifiziertes Postsches Korrespondenzproblem). *Gegeben sei ein Alphabet  $\Sigma$  und eine Folge von Wortpaaren  $K = ((x_1, y_1), \dots, (x_k, y_k))$  mit  $x_i, y_i \in \Sigma^+$ . Das Modifizierte Postsche Korrespondenzproblem (MPCP) ist die Frage, ob es für die gegebene Folge  $K$  eine Folge von Indizes  $1, i_2, \dots, i_m$  mit  $i_j \in \{1, \dots, k\}$  gibt, sodass  $x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$  gilt.*

**Lemma A.10.2.**  $\text{MPCP} \leq \text{PCP}$

*Beweis.* Wir müssen eine Funktion  $f$  angeben, sodass jede Instanz  $K$  des MPCP-Problems eine Lösung hat genau dann wenn Instanz  $f(K)$  eine Lösung für das PCP hat.

Sei  $K = ((x_1, y_1), \dots, (x_k, y_k))$  eine MPCP-Instanz über Alphabet  $\Sigma$  und seien  $\$$  und  $\#$  Symbole, die nicht in  $\Sigma$  vorkommen. Für  $w = a_1 \cdots a_n \in \Sigma^+$  definieren wir:

$$\begin{aligned} \bar{w} &= \#a_1\#a_2\#\cdots\#a_n\# \\ \acute{w} &= a_1\#a_2\#\cdots\#a_n\# \\ \grave{w} &= \#a_1\#a_2\#\cdots\#a_n \end{aligned}$$

Schließlich sei  $f(K) = ((\bar{x}_1, \grave{y}_1), (\acute{x}_1, \grave{y}_1), \dots, (\acute{x}_k, \grave{y}_k), (\$, \#\$)) = (x'_1, y'_1), \dots, (x'_{k+2}, y'_{k+2})$ .

Wenn  $1, i_2, \dots, i_m$  eine Lösung für das MPCP ist, dann sind die oberen und die unteren Worte der Spielsteinfole  $(\bar{x}_1, \bar{y}_1)(x'_{i_2}, y'_{i_2}) \dots, (x'_{i_m}, y'_{i_m})(\$, \#\$)$  ebenfalls gleich. D.h.  $1, i_2 + 1, \dots, i_m + 1, \dots, k + 2$  ist eine Lösung für die PCP-Instanz  $f(K)$ . Umgekehrt gilt: Wenn  $f(K)$  eine Lösung  $(i_1, \dots, i_m)$  hat, dann muss  $i_1 = 1$  gelten, da sonst schon beim ersten Spielstein  $\begin{bmatrix} x'_{i_1} \\ y'_{i_1} \end{bmatrix}$  die beiden Worte unterschiedlich anfangen. Ebenso muss  $i_m = k + 2$  gelten und alle Steine zwischendrin ( $i_j$  mit  $2 \leq j < m$ ) müssen von der Form  $(x'_u, y'_u)$  sein. Damit ist klar, dass aus der Lösung für  $f(K)$  auch eine MPCP-Lösung für  $K$  konstruiert werden kann:  $i_1, i_2 - 1, \dots, i_{m-1} - 1$ .  $\square$

### A.10.2. $H \leq \text{MPCP}$

#### Lemma A.10.3. $H \leq \text{MPCP}$ .

*Beweis.* Sei  $m\#w$  ein Wort bestehend aus einer Turingmaschinenbeschreibung  $m$  und einer Eingabe  $w$ . Wir müssen daraus eine Instanz für MPCP erstellen, so dass diese genau dann eine Lösung hat, wenn Turingmaschine  $M_m$  auf Eingabe  $w$  anhält. Der Beweis konstruiert ein MPCP, das nur dann eine Lösung hat, wenn es einen akzeptierenden Lauf der Turingmaschine  $M_m$  gibt. Sei  $M_m = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ . Das Alphabet für das MPCP sei  $\Gamma \cup Z \cup \{\#\}$ . Das erste Wortpaar ist  $\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \# \\ \#z_0w\# \end{bmatrix}$ . Lösungsfolgen müssen daher mit diesem Wortpaar beginnen. Die weiteren Paare  $\begin{bmatrix} x_2 \\ y_2 \end{bmatrix}, \dots, \begin{bmatrix} x_k \\ y_k \end{bmatrix}$  können durch vier Gruppen von Paaren beschrieben werden:

#### Kopierregeln:

- $\begin{bmatrix} a \\ a \end{bmatrix}$  für alle  $a \in \Gamma \cup \{\#\}$

#### Transitionsregeln:

- $\begin{bmatrix} za \\ z'c \end{bmatrix}$  falls  $\delta(z, a) = (z', c, N)$
- $\begin{bmatrix} za \\ cz' \end{bmatrix}$  falls  $\delta(z, a) = (z', c, R)$
- $\begin{bmatrix} bza \\ z'bc \end{bmatrix}$  falls  $\delta(z, a) = (z', c, L)$  für alle  $b \in \Gamma$
- $\begin{bmatrix} \#za \\ \#z'\square c \end{bmatrix}$  falls  $\delta(z, a) = (z', c, L)$
- $\begin{bmatrix} z\# \\ z'c\# \end{bmatrix}$  falls  $\delta(z, \square) = (z', c, N)$
- $\begin{bmatrix} z\# \\ cz'\# \end{bmatrix}$  falls  $\delta(z, \square) = (z', c, R)$

- $\left[ \begin{array}{l} bz\# \\ z'bc\# \end{array} \right]$  falls  $\delta(z, \square) = (z', c, L)$  für alle  $b \in \Gamma$

**Löschregeln:**

- $\left[ \begin{array}{l} az_e \\ z_e \end{array} \right]$  für alle  $a \in \Gamma, z_e \in E$
- $\left[ \begin{array}{l} z_e a \\ z_e \end{array} \right]$  für alle  $a \in \Gamma, z_e \in E$

**Abschlussregeln:**

- $\left[ \begin{array}{l} z_e\#\# \\ \# \end{array} \right]$  für alle  $z_e \in E$

Wenn die Turingmaschine einen akzeptierenden Lauf hat, dann gibt es eine Folge von Konfigurationen  $K_0 \vdash K_1 \vdash \dots \vdash K_n$ , wobei  $K_0 = z_0 w$  und  $K_n = u z_e v$  für ein  $z_e \in E$ . Dann hat das MPCP eine Lösung, die oben und unten das Wort  $\#K_0\#K_1\#\dots\#K_n\#K_{n+1}\#\dots\#K_m\#\#$  erzeugt, wobei  $K_m = z_e$  und jedes  $K_i$  mit  $n+1 \leq i \leq m$  jeweils aus  $K_{i-1}$  entsteht durch Löschen eines der benachbarten Zeichen von  $z_e$  in  $u'z_e v'$  entsteht.

Bei den Spielsteinfolgen hinkt die obere Folge immer der unteren Folge um eine Konfiguration hinterher, d.h. beim Erstellen sind Zwischenzustände der Spielsteinfolge so, dass oben  $\#K_1\#K_2\#\dots\#K_i\#$  und unten  $\#K_1\#K_2\#\dots\#K_i\#K_{i+1}\#$  abzulesen ist. Um diese beiden Folgen um die nächste Konfiguration zu verlängern, werden zunächst die Kopierregeln angewendet, um in die Nähe des Zustandes zu kommen. Dann wird eine der Überführungsregeln angewendet, und schließlich weitere Kopierregeln um den Rest der Konfiguration zu erzeugen. Ist man bei  $K_n$  angekommen, so werden die Löschregeln angewendet, um die Symbole auf dem Band zu löschen, bis man in der unteren Folge nur noch  $z_e\#$  stehen hat. Schließlich wird die Abschlussregel angewendet.

Umgekehrt lässt sich aus jeder Lösung für das MPCP, (welches ja mit dem ersten Spielstein beginnen muss) eine Konfigurationsfolge ablesen, die bezeugt, dass die Turingmaschine bei Eingabe  $w$  hält.

Schließlich kann die MPCP-Instanz aus der Beschreibung  $m\#w$  berechnet werden, d.h. es gibt eine berechenbare Funktion  $f$  mit  $m\#w \in H \iff f(m\#w) \in \text{MPCP}$ . Damit folgt  $H \leq \text{MPCP}$ . □

**Satz A.10.4.** *Das Postsche Korrespondenzproblem (sowie das modifizierte Postsche Korrespondenzproblem) ist unentscheidbar.*

*Beweis.* Da  $H$  unentscheidbar ist und  $H \leq \text{MPCP} \leq \text{PCP}$  gilt, folgt, dass MPCP als auch PCP unentscheidbar sind. □

## A.11. NP-Schwere von SAT

Bevor wir die NP-Schwere von SAT betrachten, beweisen wir einen Hilfssatz:

**Lemma A.11.1.** Für aussagenlogische Variablen  $\{x_1, \dots, x_n\}$  gibt es eine aussagenlogische Formel  $\text{exactlyOne}(x_1, \dots, x_n)$ , sodass  $I(\text{exactlyOne}(x_1, \dots, x_n)) = 1$  g.d.w.  $I$  genau eine der Variablen  $x_i$  auf 1 setzt und alle anderen auf 0.

Dabei ist die Größe der Formel  $\text{exactlyOne}(x_1, \dots, x_n)$  in  $O(n^2)$ .

*Beweis.* Die Formel

$$\text{exactlyOne}(x_1, \dots, x_n) := (x_1 \vee \dots \vee x_n) \wedge \bigwedge_{1 \leq i < j \leq n} \neg(x_i \wedge x_j)$$

leistet das verlangte: Die Disjunktion  $(x_1 \vee \dots \vee x_n)$  sichert zu, dass mindestens eine der Variablen  $x_1, \dots, x_n$  wahr sein muss, die Konjunktionen  $\neg(x_i \wedge x_j)$  sichern zu, dass zwei verschiedene Variablen nicht gleichzeitig wahr sein können (und daher höchstens eine der Variablen  $x_1, \dots, x_n$  wahr sein darf). Die Größe der Formel ist  $O(n^2)$ .  $\square$

Wir betrachten nun den zweiten Teil, um den Nachweis der NP-Vollständigkeit von SAT zu vervollständigen – die NP-Schwere von SAT. Dieser Beweis ist schwierig, da wir zeigen müssen: Wenn wir SAT lösen können, dann können wir *jedes* andere Problem  $L$  aus NP auch lösen, indem wir es in Polynomialzeit auf SAT reduzieren. Das bedeutet: Wir müssen für jedes  $x$ -beliebige Problem  $L$  aus NP nachweisen, dass  $L \leq_p$  SAT gilt.

Da  $L$  ein Problem aus NP ist, wissen wir, dass eine NTM existiert, die  $L$  in (nichtdeterministischer) Polynomialzeit entscheidet. Dies machen wir uns zunutze, indem wir die Berechnungen jeder NTM  $M$  bei gegebener Eingabe  $w$  in eine aussagenlogische Formel kodieren, sodass gilt:  $M$  akzeptiert g.d.w.  $F$  erfüllbar ist. Da die Laufzeit von  $M$  durch ein Polynom beschränkt ist, muss die Formel auch nur solche Berechnungen abdecken, welche diese Länge haben. Dies wird die Größe der Formel passend beschränken.

**Lemma A.11.2.** SAT ist NP-schwer.

*Beweis.* Sei  $L \in \text{NP}$  beliebig. Sei  $M$  die nichtdeterministische Turingmaschine mit  $L(M) = L$ , sodass alle Berechnungen von  $M$  nach polynomiell vielen Schritten halten. Sei  $p$  das Polynom, welches die Rechenzeit von  $M$  beschränkt. Sei  $w = a_1 \cdots a_n \in \Sigma^*$  eine Eingabe für  $M$ . Wir konstruieren eine aussagenlogische Formel  $F$ , sodass gilt

$$w \in L \iff F \text{ ist erfüllbar}$$

Sei  $\Gamma = \{b_1, \dots, b_l\}$ ,  $Z = \{z_0, \dots, z_k\}$  und  $z_0$  der Startzustand. Die Formel  $F$  enthält die folgenden aussagenlogischen Variablen, die wir zunächst erläutern:

- $\text{State}_{t,z}$  für  $t = 0, 1, \dots, p(n)$  und  $z \in Z$ . Dabei soll gelten:  $\text{State}_{t,z} = 1$  g.d.w. nach  $t$  Schritten ist TM  $M$  im Zustand  $z$ .

- $Pos_{t,i}$  für  $t = 0, 1, \dots, p(n), i = -p(n), \dots, p(n)$ . Dabei soll gelten:  $Pos_{t,z} = 1$  g.d.w. nach  $t$  Schritten von  $M$  befindet sich der Schreib-Lesekopf von  $M$  auf Position  $i$
- $Tape_{t,i,b}$  für  $t = 0, 1, \dots, p(n), i = -p(n), \dots, p(n), b = b_1, \dots, b_l$ . Dabei soll gelten:  $Tape_{t,i,b} = 1$  g.d.w. nach  $t$  Schritten steht auf dem Band in Position  $i$  das Zeichen  $b$ .

Bandposition 0 ist dabei die Position des Kopfes am Anfang und negative Positionen sind links davon, positive rechts davon. Da in  $p(n)$  Schritten nicht mehr als  $p(n)$  Kopfbewegungen gemacht werden, reicht es aus, die Positionen  $-p(n), \dots, p(n)$  des Bands zu betrachten.

Die Formel  $F$  lässt sich in vier Unterformeln aufteilen:

$$F = Rand \wedge Anfang \wedge Transition \wedge Ende$$

Wir beschreiben die Unterformeln:

- *Rand* legt verschiedene Randbedingungen fest:
  - Zu jedem Zeitpunkt  $t$  befindet sich  $M$  in genau einem Zustand  $z$ . Das sichert die Formel  $\bigwedge_{t \in \{0, \dots, p(n)\}} exactlyOne(State_{t,z_0}, \dots, State_{t,z_k})$  zu.
  - Zu jedem Zeitpunkt  $t$  befindet sich der Kopf von  $M$  in genau einer Position auf dem Band. Das sichert die Formel  $\bigwedge_{t \in \{0, \dots, p(n)\}} exactlyOne(Pos_{t,-p(n)}, \dots, Pos_{t,p(n)})$  zu.
  - Zu jedem Zeitpunkt  $t$  befindet sich in jeder Bandzelle genau ein Symbol aus  $\Gamma$ . Das sichert die Formel  $\bigwedge_{t \in \{0, \dots, p(n)\}} \bigwedge_{i \in \{-p(n), \dots, p(n)\}} exactlyOne(Tape_{t,i,b_1}, \dots, Tape_{t,i,b_l})$  zu.

Daher setzen wir:

$$Rand := \bigwedge_{t \in \{0, \dots, p(n)\}} \left( \begin{array}{l} exactlyOne(State_{t,z_0}, \dots, State_{t,z_k}) \\ \wedge exactlyOne(Pos_{t,-p(n)}, \dots, Pos_{t,p(n)}) \\ \wedge \bigwedge_{i \in \{-p(n), \dots, p(n)\}} exactlyOne(Tape_{t,i,b_1}, \dots, Tape_{t,i,b_l}) \end{array} \right)$$

- *Anfang* beschreibt die Anfangsbedingungen, d.h. die Formel fixiert die Variablenbelegung zum Zeitpunkt  $t = 0$ :
  - Die Maschine ist im Startzustand, d.h.  $State_{0,z_0}$  muss gelten
  - Der Schreib-Lesekopf ist auf Position 0, d.h.  $Pos_{0,0}$  muss gelten.
  - Die Eingabe  $w = a_1 \cdot \dots \cdot a_n$  steht auf dem Band und alle anderen Zellen enthalten das Blank-Symbol. Dies leistet die Formel

$$\left( \bigwedge_{i \in \{0, \dots, n-1\}} Tape_{0,i,a_{i+1}} \right) \wedge \left( \bigwedge_{i \in \{-p(n), \dots, -1\}} Tape_{0,i,\square} \right) \wedge \left( \bigwedge_{i \in \{n, \dots, p(n)\}} Tape_{0,i,\square} \right).$$

Daher setzen wir

$$Anfang := State_{0,z_0} \wedge Pos_{0,0} \wedge \left( \bigwedge_{i \in \{0, \dots, n-1\}} Tape_{0,i,a_{i+1}} \right) \wedge \left( \bigwedge_{i \in \{-p(n), \dots, -1\}} Tape_{0,i,\square} \right) \wedge \left( \bigwedge_{i \in \{n, \dots, p(n)\}} Tape_{0,i,\square} \right)$$

- *Transition* beschreibt die Formeln, die den Zustandsübergang der TM fixieren:
  - Für den Übergang von  $t$  zu  $t + 1$  muss der Zustand, der Bandinhalt und die Kopfposition geändert werden. Sei  $dir(N) = 0, dir(L) = -1, dir(R) = 1$ . Dann kann der Zustandsübergang durch die folgende Formel beschrieben werden:

$$\bigwedge_{\substack{t \in \{0, \dots, p(n)-1\}, \\ z \in Z, \\ i \in \{-p(n)+1, \dots, p(n)-1\}, \\ b \in \Gamma}} \left( \begin{array}{c} (State_{t,z} \wedge Pos_{t,i} \wedge Tape_{t,i,b}) \\ \implies \bigvee_{(z',b',y) \in \delta(z,b)} (State_{t+1,z'} \wedge Pos_{t+1,i+dir(y)} \wedge Tape_{t+1,i,b'}) \end{array} \right)$$

- Zellen des Bandes auf denen der Kopf nicht steht, bleiben unverändert. Das leistet die Formel:

$$\bigwedge_{\substack{t \in \{0, \dots, p(n)-1\}, \\ i \in \{-p(n), \dots, p(n)\}, \\ b \in \Gamma}} ((\neg Pos_{t,i} \wedge Tape_{t,i,b}) \implies Tape_{t+1,i,b})$$

Insgesamt ergibt sich daher

$$\begin{aligned} Transition := & \left( \bigwedge_{\substack{t \in \{0, \dots, p(n)-1\}, \\ z \in Z, \\ i \in \{-p(n)+1, \dots, p(n)-1\}, \\ b \in \Gamma}} \left( \begin{array}{c} (State_{t,z} \wedge Pos_{t,i} \wedge Tape_{t,i,b}) \\ \implies \bigvee_{(z',b',y) \in \delta(z,b)} (State_{t+1,z'} \wedge Pos_{t+1,i+dir(y)} \wedge Tape_{t+1,i,b'}) \end{array} \right) \right) \\ & \wedge \left( \bigwedge_{\substack{t \in \{0, \dots, p(n)-1\}, \\ i \in \{-p(n), \dots, p(n)\}, \\ b \in \Gamma}} ((\neg Pos_{t,i} \wedge Tape_{t,i,b}) \implies Tape_{t+1,i,b}) \right) \end{aligned}$$

- *Ende* beschreibt die Endbedingung, dass ein akzeptierender Zustand erreicht wird:

$$Ende := \bigvee_{z \in E, t \in \{0, \dots, p(n)\}} State_{t,z}$$

Falls  $w \in L$ , dann gibt es einen Lauf von  $M$   $z_0 w \vdash^r u z_e v$  mit  $z_e \in E$  und  $r \leq p(n)$ . Dann liefert der Lauf eine Belegung  $I$  der Variablen von  $F$ , welche die Formel wahr macht: In der Unterformel *Rand* belege die besuchten Zustände, Positionen und Bandinhalte mit 1, welche durch die Konfigurationsfolge gegeben sind. Falls die Folge nach  $t_e < p(n)$  Schritten endet, so setze die Variablen für  $t > t_e$  auf die Werte für den Zeitpunkt  $t_e$ . Die *Anfang*-Formel liefert immer die Belegung für die dort vorkommenden Variablen. Diese Belegung passt zur Anfangskonfiguration des Laufs. Für die *Transition*-Formel, setze  $I(State_{t,z}) = 1, I(Pos_{t,i}) = 1, I(Tape_{t,i,b}) = 1$  entsprechend der besuchten Zustände und alle anderen auf 0. Dann ist  $I(Transition) = 1$ , da alle gemachten Übergänge auch genau in den Formeln abgebildet sind. Schließlich wurde durch

die erzeugte Belegung für *Transition* auch die Variable  $State_{t,z_e}$  für  $z_e \in E$  auf 1 gesetzt (da die Konfigurationsfolge so endet). Insgesamt zeigt dies, dass die Formel erfüllbar ist.

Umgekehrt, wenn es eine erfüllende Belegung  $I$  gibt, d.h.  $I(F) = 1$ , dann kann daraus eine Konfigurationsfolge konstruiert werden, die einen Lauf der Turingmaschine auf der Eingabe  $w$  darstellt und akzeptierend endet.

Damit gilt  $w \in L \iff F$  ist erfüllbar. Schließlich ist noch zu zeigen, dass die Formel  $F$  in (deterministischer) Polynomialzeit berechnet werden kann. Zunächst ist klar, dass die Formel berechenbar ist. Für die Laufzeit genügt es, die Größe der Formel zu betrachten. Wir nehmen als Maß die Anzahl an Variablenvorkommen in Bezug auf  $p(n)$ . Wir nehmen das Alphabet und die Zustandsanzahl von konstanter Größe an.

Dann hat die Formel *Rand* die Größe  $O(p(n)^3)$ . Die erste Unterformel ist von der Größe  $O(p(n))$ , da wir die Zustandsanzahl als konstant annehmen. Für die zweite Teilformel von *Rand* wenden wir Lemma A.11.1 an: Daher hat jede der *exactlyOne*(...)-Formeln in der Konjunktion die Größe  $O(p(n)^2)$  und davon gibt es  $O(p(n))$  viele in der Konjunktion, was insgesamt  $O(p(n)^3)$  ergibt. Die dritte Formel von *Rand* hat Größe  $O(p(n)^2)$ , da die inneren *exactlyOne*(...)-Formeln als konstant angesehen werden (da  $|\Gamma|$  als konstant angenommen ist) und die Konjunktionen  $O(p(n)^2)$  als Faktor hinzufügen.

Die Größe der *Anfang*-Formel ist  $O(p(n))$ . Die Größe beider Formeln der *Transition*-Formeln ist  $O(p(n)^2)$  und die Größe der *Ende*-Formel ist  $O(p(n))$ . Insgesamt zeigt dies, dass die Größe der Formel  $F$  polynomielle Länge hat und daher in Polynomialzeit berechnet werden kann.  $\square$